The present work was submitted to
the Research Group
Software Construction

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

Master Thesis

Test Case Prioritization for Combinatorial Testing

Priorisierung von Testfällen für Kombinatorisches Testen

presented by

Joshua Bonn

Aachen, September 28, 2020

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

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Master's thesis entitled

Test Case Prioritization for Combinatorial Testing

independently and without illegitimate assistance from third parties. I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

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

Test Case Prioritization 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 28, 2020 (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 28, 2020 (Joshua Bonn)

Acknowledgment

This thesis would not have been possible without the help from some other people. Therefore, I want to use this short paragraph to thank them.

First, I would like to thank Konrad Fögen for supervising this thesis. It was a pleasure to work with him on the topic of combinatorial testing throughout two seminars, my bachelor thesis, and finally, my master thesis. Whenever I had any questions while writing this thesis he always had an answer and provided valuable input.

Next, I would like to thank Professor 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 Professor Dr. rer. nat. Bernhard Rumpe for agreeing to be the second examiner for my thesis.

Last, but not least, I thank my colleagues at work, friends, and family who supported me during my bachelor and master studies.

Joshua Bonn

Abstract

Combinatorial testing is a model-based black-box testing technique which aims to reduce the number of required test cases while still providing high failure detection capabilities. To this end, it divides the input and environment of the system under test into a finite number of parameters and values. The combinatorial coverage criterion then requires a test suite to only cover all value combinations of up to t parameter. Since one test case can cover multiple t value combinations this produces fewer test cases than exhaustive testing.

Even though combinatorial test suites are comparatively small, they can still contain hundreds or thousands of test cases. Especially in the light of modern development methods such as continuous integration where fast feedback cycles become more and more important, test suites with this many test cases should fail as fast as possible. The field of combinatorial test prioritization addresses this problem and attempts to order the generated test suite so that the failing test cases appear as early as possible. While researchers developed multiple algorithms and techniques for prioritization, no framework which integrates all of them for automated comparison currently exists.

To this end, this thesis extends coffee4j, an automated combinatorial testing framework, to support different prioritization techniques. After an overview of the current state of research, it first develops a concept for integrating the techniques into a general combinatorial testing framework. Additionally, this thesis introduces new techniques for test case prioritization based on previously failing test cases and their failure-inducing combinations. It then evaluates these techniques to show how researchers can use the integration into coffee4j for comparing different prioritization algorithms in future work.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Professional software development without automated testing is almost impossible, and due to a steady increase of software complexity the accompanying test development also grows more and more complex. Additionally, even for simple programs it is already impossible to automatically test every corner case, thus requiring a selection of tests to keep the failure detection rate high, while keeping the number of tests as small as possible. This is especially important in scenarios such as continuous testing and integration, where automated tests run every time a developer adds code to the project [SE04; SAZ17]. Shorter testing times result in faster feedback to the developer for quicker error detection and correction.

One testing technique to reduce the number of required test cases is *combinatorial testing (CT)*. It defines an abstraction above the *system under test (SUT)* by describing the environment and input using a finite number of parameters and values, also called an *input parameter model (IPM)* [NL11]. In exhaustive testing, a tester needs to execute one test case for each possible combination of values, thus requiring millions to billions of test cases for real world systems. CT reduces the number of required test cases. The combinatorial test coverage criterion states that a test suite only has to test all combinations of values from up to t parameters, where t is often a value between one and six [NL11]. Since one test case contains multiple smaller combinations, CT requires fewer test cases — hundreds to thousands instead of millions to billions. While a reduction in test suite size also means that test suites may no longer detect some possible failures, early studies concluded that combinations consisting of up to six parameters detect all bugs [KWAMG04]. Therefore, testing with $t > 6$ is not economical due to diminishing returns.

CT is an active area of research. A large percentage of this research focuses on finding ways to generate smaller combinatorial test suites or making the generation process faster [NL11]. Other topics include the consideration of constraints on parameters and their values, studying the effectiveness of CT with real world systems, and negative testing with error constraints [NL11; JST20; FL20; Hu+20; FL19]. Another important area of research is *fault characterization (FC)*, where additional test cases after the initial test suite narrow down the sub-combination of values which causes the system to fail. This combination is then called a *failure-inducing combination (FIC)* and helps developers with finding the defect causing the failure faster [NL11; Gha+12]. To this end, earlier work introduced coffee4j as an automated framework for combinatorial testing with extension points for different FC algorithms, thus reducing the amount of manual work in the CT process [Fö+20; Bon18; BFL19; Ber19].

In light of the fast feedback cycles necessary for continuous testing and integration, *combinatorial test case prioritization (CTCP)* is also an important topic in research [NL11;

Hua+13; Cho+16; QC13; BC05]. While thousands of tests run faster than millions of them, they still take too long if one execution is slow, or if multiple combinatorial tests need to run. CTCP orders each generated test case so that earlier test cases are more likely to fail than later ones. This reduces the amount of time developers have to wait for feedback from automated test suites. For example, Qu et al. developed an algorithm to prioritize test cases based on code coverage information collected from previous runs and evaluated it according to the effectiveness for higher strength CT [QC13]. In addition to the used information, the point of prioritization is another distinction between different techniques. Here, it is either possible to already generate the test cases in a prioritized way or to prioritize them afterwards in a separate step [QC13].

Currently, no framework exists which enables the comparison of all different ways to prioritize a combinatorial test suite regarding their effectiveness and potential deficiencies.

Contributions

This thesis extends the coffee4j framework so that it supports the automated execution of third-party CTCP algorithms, therefore enabling easy comparison. As a proof of concept, it also introduces a new prioritization approach which uses failure information from previous runs. All in all, the thesis answers the following three research questions:

RQ1: How can combinatorial test case prioritization during or after test suite generation based on (additional) information extracted from the input parameter model or supplied from external sources be integrated into the coffee4j architecture and process model?

RQ2: Does failure-based combinatorial test prioritization or non-prioritized combinatorial testing require fewer test cases to detect failures?

RQ3: Does failure-based combinatorial test prioritization during or after test suite generation require fewer test cases to detect failures?

Structure of this Thesis

This thesis consists of seven chapters, the first of which is the Introduction. Next, Chapter 2 introduces background information on combinatorial testing, regression test optimization, and technologies used during the development of this thesis and Chapter 3 discusses current research related to the topic of CTCP. With the necessary background knowledge, Chapter 4 defines requirements which the integration of CTCP into coffee4j has to meet. Additionally, it presents the integration into a general CT process and coffee4j's architecture and explains how failure-based test case prioritization can work. Chapter 5 then shows how coffee4j implements the developed concepts. Finally, Chapter 6 evaluates the integration of CTCP into coffee4j and the failure-based prioritization, and Chapter 7 summarizes the findings.

2. Background

Contents

This chapter introduces background knowledge required for understanding the main chapters of the thesis. First, Section 2.1 covers the basics of general testing methods like regression testing and combinatorial testing. Later chapters of the thesis extend these basic methods. Next, Section 2.2 introduces central technologies used during the concept and realization chapter.

2.1. Testing Techniques

An important field in the area of software engineering is software testing. Without testing, one could never know if a system meets important functional or non-functional requirements, and only end-users would detect errors. Since early studies have already shown a large increase in correction costs if one finds errors in later stages of the software development process, modern processes should try to detect them as early as possible [Ste+04]. Therefore, software testing moved closer — in time and place — to the actual implementation of code, for example, to unit tests.

In an ideal world, testing would cover every possible flow through an application. The name of this method is exhaustive testing [Sul+04]. Since it tests all user-actions, users can never reach an unexpected situation, thus never running into errors or even system failures. However, this ideal scenario does not survive the contact with reality in even the simplest of situations [Sul+04]. Imagine a function which computes the maximum in a list of integers. Since each integer contains 32 bits, there are 2^{32} possible numbers. If we were to only test this function exhaustively with ten input numbers, there would be $(2^{32})^{10} \approx 2.136 \times 10^{96}$ test cases. The observable universe contains only 10^{80} atoms.

It therefore becomes obvious that software testing includes the hard problem of filtering these 2.136×10^{96} test cases for the important ones and even defining what makes a test case important. To solve parts of this problem, researchers developed a myriad of

software testing techniques. Well-known examples include boundary value testing, which attaches a special importance to edge case input values, and equivalence class testing, which partitions the input space into categories of equivalent or similar inputs [ND12]. During the development of unit tests, most developers use the basic idea behind these techniques subconsciously (e.g. testing edge case inputs), but often not in a formalized way, thus missing potentially important tests.

Another method to limit the number of tests is combinatorial testing, which Subsection 2.1.2 will explain.

2.1.1. Regression Testing

Orthogonal to the problem of general test selection, software testing also has to answer the question of when and how often to test [JPR00]. Since early error detection reduces the correction costs, testers should test each feature as early as possible, but often it is not enough to test a feature once and then assume that it will continue to work from there on [Ste+04; BRO13].

General

Regression testing attempts to solve this problem by executing old tests even after finishing the original feature. [BRO13] This is necessary because the implementation of a new feature is seldom completely independent of old features and may therefore introduce defects in unexpected places. Since a test suite for those older features already exists, an initial idea could be to execute those test again. Ideally that means the exiting tests would detect an error if a developer introduces some defect for an old feature later on — also called a *regression* — and therefore allow for fast defect correction [BRO13].

A development organization should always specify how often to run regression tests. As always, earlier testing leads to less correction costs, and therefore the discipline of continuous integration and testing defines that automatic pipelines execute the complete regression test suite after every commit [SE04; SAZ17]. In modern development practices like *test driven development (TDD)* developers execute regression tests even more often during the implementation of new code [Bec02].

Since regression testing repeats the same tests multiple times without any changes, an automated process is very cost effective. Especially with approaches like continuous testing where regression test suites run tens of times a day, manual testing would become too expensive [RW06]. Therefore, a best practice is to write automated tests when developing a feature and then execute those same tests as part of the regression test suite.

Regression Test Optimization

While large regression test suites may find errors earlier than other approaches, they also come with their own set of problems. Since automatic pipelines execute each test on every commit, the costs for the required number of machines which execute those tests

can outweigh the benefit of early error detection. Additionally, regression test suites may take multiple hours to run. Especially in a continuous testing and TDD where quick feedback is important to the development process this is unacceptable. Since regression test suites also grow with each new feature, the problem becomes worse over time.

There are multiple ways to overcome this problem. Locally, developers often intuitively know which subset of tests to execute to receive a relatively fast and reliable feedback on whether they introduced big defects [Gli+14]. The automated regression test pipelines will then (hopefully) find any smaller defects. While often applied in practice, this technique can be very unreliable and only exists in the heads of individual developers and is therefore not reproducible in general. Another way to reduce the time required for testing is removing or disabling old tests. However, this has the obvious disadvantage of reducing test coverage and due to the continuous growth of the regression test suite it cannot solve the problem once and for all. Developers would therefore have to adjust the test suite continuously, which often takes too much time.

In contrast to these manual techniques for reducing the time spent on regression testing there is also the automated variant called *regression test optimization (RTO)*. RTO itself has two important subfields, *regression test selection (RTS)* and *regression test prioritization (RTP)* [DS08].

Regression Test Selection RTS is the formalization and automation of one of the manual techniques described above. Instead of relying on developer intuition to decide which tests are important, algorithms make the decision automatically before each regression test suite run [RH96]. Ideally, they only execute those tests which actually detect a regression, but this is almost always impossible to predict. The challenge of RTS is therefore to find a small subset of tests that still detects all defects [Ple15].

Since it is not possible for test selection algorithms to know which ones will fail, most approaches use heuristics based on information gathered in previous runs [RH96]. For example, one approach would be to detect which tests recently failed and always include them in the selection since this could reveal that developers currently work on that part of the code under test [MGS13]. Similarly, code coverage information could lead to a selection where each line of code or branch is covered by at least one test [QC13]. A combination of multiple heuristics is also possible.

Even though RTS executes fewer tests, selected test suites can still sometimes take longer than the original test suite. This is due to the time spent on selecting the subset. Therefore, in practice testers have to carefully check whether the chosen selection algorithm runs quick enough to actually decrease the testing time [Ple15; OSH04].

Regression Test Prioritization The other area of RTO is RTP. It does not focus on selecting a specific subset of all tests but instead orders/prioritizes them to run tests which are likely to fail first [Rot+01]. Figure 2.1 shows an example of this. Here, assuming all tests take one second, test $t5$ is very efficient at detecting failures, so it should run first to provide this efficiency at the beginning of the test suite. The general idea of prioritization is that developers can now already start to find the defect when the first

test failed. Therefore, early failures become important.

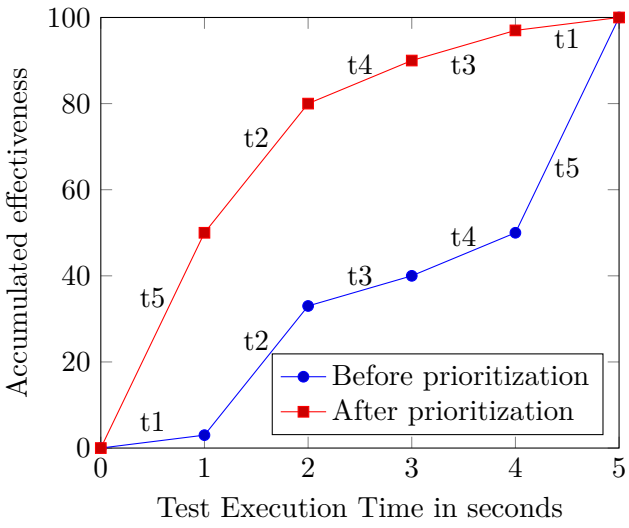

Figure 2.1.: Regression test prioritization effectiveness [Ple15]

As with RTS it is not possible for the prioritization algorithm to know which test will fail. RTP therefore also uses heuristics to assign value to individual tests [Rot+99]. Similarly, it also should not take too much computation time since this will increase the already high testing time.

Formally, one can define the effectiveness as a function f which maps every test t to an effectiveness value $f(t)$ [Rot+01]. RTP must then find an order $t_{i_1}, ..., t_{i_n}$ such that the following holds:

- The order includes all regression tests.
- For every other order $t_{j_1}, ..., t_{j_n}$ and every prefix $t_{i_1}, ..., t_{i_k}$ and $t_{j_1}, ..., t_{j_k}$ with $1 \leq k \leq n$ it holds that $\sum_{h=1}^{k} f(t_{i_h}) \geq \sum_{h=1}^{k} f(t_{j_h})$. In other words: there is no better order.

For practical purposes prioritization techniques then replace the effectiveness function f with a heuristic h which attempts an approximation.

2.1.2. Combinatorial Testing

It is possible to apply regression testing regardless of the technique originally used to construct the underlying tests. They could even be comprised of tests generated using different techniques. In general, one can categorize all testing techniques into two areas. The first one, white-box testing, creates tests according to the actual implementation [ND12]. One example for such a technique is constructing tests so that they cover each branch. While white-box testing is good for checking whether the existing code is correct,

it might not catch missed cases defined in the requirements. On the other hand, black-box testing constructs tests without caring about the actual implementation [ND12]. Tests then only cover the interface and behavior specified in the requirements. Examples for black-box testing are boundary value, equivalence class, and, the topic of this thesis, combinatorial testing [NL11]. This section will first explain the basics of combinatorial testing, such as the input parameter model, and then go more in-depth on the topics of fault characterization and combinatorial test case prioritization. The general structure and topics discussed are similar to those of the background chapter in the bachelor thesis which developed coffee4j [Bon18].

General

In boundary value testing, a tester focuses on a single parameter or variable at a time and defines the values which represent certain edge cases in the specification. When extending the concept to multiple variables, there are two possible approaches: either test each variable one at a time or test the Cartesian product of all variables. As with traditional exhaustive testing, this so called worst case boundary value testing can quickly run into the problem of exponential test suite growth. For example, a system with ten parameters, each having the four boundary values and one nominal value, would require 5^{10} test inputs. While this is considerably less than testing all possible input values, nine million test cases still take too much time to run, even if a single test case is relatively fast. Combinatorial testing aims to solve this problem.

Interaction Faults Before understanding how CT solves the problem of testing combinations of multiple parameters one first has to understand why it is necessary. Consider the following Java method:

```
1  public void method(String operatingSystem, String browser, ...) {
2    if (operatingSystem.equals("Windows")) {
3      // working code
4    } else if (operatingSystem.equals("Linux")) {
5      if (browser.equals("Edge")) {
6        // non-working code
7      }
8    }
9    // working code
10 }
```

Source Code 2.1: InteractionFault.java

This method has two important input parameters, each with a discrete number of possible values, and will only fail if the operating system is Linux and the browser is Edge. In all other cases the method will execute correctly. If a tester were to only test the values of each parameter by itself, it is very likely that no test case will actually check the combination of Linux and Edge. Therefore the defect would stay undetected until an end-user experiences the problem.

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

To solve this problem, testers needs to not only test the individual parameter values, but also at least all possible combinations between any two parameters. This is exactly what the combinatorial test coverage criterion defines [NL11; KWAMG04].

Parameters and Values A central concept of combinatorial testing are parameters and values [NL11; KWAMG04; QC13; Bon18]. In general, sources for parameters are either its environment or inputs of the system under test. For example, a list of numbers for a function which determines the maximum is an input parameter and browser or operating system are examples for environment parameters. Although the parameter sources are different, CT treats both types of parameters exactly the same during test case generation.

Each parameter must then have a list of a finite number of discrete values. Therefore, CT cannot test continuous input variables without some form of discretization. For example, testers can use techniques from boundary value or equivalence class testing to select a small number of representatives. With parameters that are already discrete, like the browser or operating system, values usually come from the requirements specification.

Since CT is a very general technique, testers can use it across different levels of the testing pyramid. A unit test would most likely consist of only input parameters for a method, and a higher-level end-to-end test would then also include environment parameters. In each case, CT does not care about the actual parameters or values, since the test case generation process is independent.

Formally, a so called input parameter model then consists of n parameters $p_1, ..., p_n$ each having values $v_{i,1}, ..., v_{i,s_{p_i}}$ for every $1 \leq i \leq n$. s_{p_i} denotes the number of values of parameter p_i, which can, and most likely will be, different for every parameter. An IPM must always contain at least one parameter, and every parameter must have a minimum of two variables. Parameters with just one variable are constant and therefore not interesting for combinatorial test case generation.

If the specific parameters and values are not important, a multiplication of the number of values can abbreviate an IPM. For example, $2 \times 2 \times 2 \times 3 \times 4 \times 4$ denotes a model with six parameters, three of which have two values, one with three values, and two with four values. To make this even shorter, one can combine parameters with the same size using power notation: $2^3 \times 3 \times 4^2$.

Table 2.1 contains a list of parameters for a browser-based video game. The abbreviated

notation introduced above would describe the model as $3 \times 4^2 \times 5$. It will serve as a running example for the remainder of this thesis.

Test Cases and Combinations A combinatorial test case assigns every parameter to one of its values [NL11]. For example, a valid test case in the example above could be assigning the operating system to be Windows, the browser to be Chrome, defining a ping of 10 ms, and a connection speed of 100 KB/s. An abbreviated form would be the tuple (Windows, Chrome, 10 ms, 100 KB/s) if one assumes the parameter order is the same way as in Table 2.1.

CT also places a large emphasis on incomplete test cases, which are generally called *combinations* [NL11]. A combination may not assign a value to some parameters. This means that the aforementioned test case would be a combination, but (Windows, –, –, 10 KB/s) or even (–, –, –, 1000 KB/s) would also be valid combinations. A combination is a t-way combination if it assigns a value to exactly t parameters. The examples before were a four-way combination, two-way-combination, and one-way-combination.

If a combination c_1 has the same values as c_2 for all parameters which have a value in c_1, and c_2 possibly also has values for some other parameters which do not have a value in c_1, c_2 contains c_1. An example would be (Windows, –, –, 10 KB/s), which contains the combination (–, –, –, 10 KB/s). One can combine two combinations c_1 and c_2 to form a new combination if none of their parameters are in conflict, i.e. if there is no parameter p for which the combinations c_1 and c_2 both define a value with $c_1(p) \neq c_2(p)$.

Sometimes, parameters in an IPM may not be a direct environment configuration or input like, for example, the browser and operating system. In such cases a translation from abstract test cases to concrete test cases is necessary [NMT12; Gha+13]. One example for this is testing regular expressions. In the model developed by Ghandehari et al. there are several parameters which describe what meta-characters or special characters a regular expression should include and at which position [Gha+13]. While these parameters still describe the input of the program, a test needs to transform them before the actual execution. In the regular expression example, the test needs to transform an abstract testcase into a concrete input which satisfies all constraints given by the semantics of the abstract parameters and values.

In the regular expression example, some program has to generate a concrete expression to use as an input from all abstract parameters included in the model so that it satisfies all constrains given by abstract values in the current test case.

Constraints Usually, it is not possible to test all combinations of values. In the running example form above, (Windows, Safari, 10 ms, 100 KB/s) is not testable since there is no version of Safari for Windows. A trivial idea of ensuring that invalid combinations do not appear in the finial test suite would be to remove test cases which contain them after the generation process. However, this may violate the idea of combinatorial testing as even those invalid test cases may also contain necessary t-way combinations not included in any other test cases [GOA05; FL20]. Therefore, researchers came up with different techniques for excluding certain combinations [NL11; GOM07].

While some approaches do modify the test suite after generation so that it does not contain invalid combinations without losing test coverage, the most widely used ones already avoid those combinations during generation [NL11; GOM07]. To this end, the user needs to specify which combinations are not possible. One way of doing so is specifying the constraints in some form of zeroth-order logic. To avoid the test case mentioned above, a constraint could be $OS = Windows \Rightarrow Browser \neq Safari$.

Generation algorithms usually delegate the task of dealing with constraints to so-called constraint solvers which can, for every combination, compute whether an extension to a complete test case would still be possible without violating any constraints [Lei+07; Yu+13]. Constraint solving is an important discipline but this thesis will not discuss it in detail. It suffices to know that such techniques exist, and that they are fast enough so that test suite generation can use them.

Since constraints, together with parameters and values, also describe the possible input space of a combinatorial test, they are also part of the IPM.

Testing Strength The central concept of CT is the testing strength, denoted as t. It is a natural number between one and the number of parameters (inclusively). The CT coverage criterion specifies that a combinatorial test suite with testing strength t must contain all possible t-way combinations in at least one test case [NL11]. Such a test suite is a t-way test suite and the process of executing the test suite is t-way testing. A special case is $t = 2$ which is called pairwise testing.

There are three central advantages to testing only all t-way combinations. First, the number of required test cases is much smaller than with exhaustive (or n-way) testing. Since every test case for n parameters contains multiple t way combinations (for $t < n$), t-way testing results in much smaller test suites. For example, an exhaustive test suite for the IPM defined in Table 2.1 contains 240 test cases, two-way testing only requires 20. While 240 test cases may not seem like much, the advantage is even bigger for IPMs with more parameters and values where the number of exhaustive test cases may be in the millions or billions. As specified above, combinatorial testing achieves this reduction in the number of test cases due to the fact that the test case (Windows, Chrome, 10 ms, 1 KB/s) only contains one n-way combination, but six t-way combinations: (Windows, Chrome, –, –), (Windows, –, 10 ms, –), (Windows, –, –, 1 KB/s), (–, Chrome, 10 ms, –), and so on.

The second advantage of combinatorial testing is that the number of required test cases does not rise exponentially when adding a new parameter. Instead the growth is logarithmic [Col04]. Therefore, two-way test suites even for large IPMs remain small. The two-way test suite for a $2 \times 3^8 \times 4^5 \times 6$ model only contains 176 test cases.

Last but not least, the third and final advantage of CT is that testing larger combinations would not even have that much of a benefit. This is also why CT is an acceptable way to reduce test suite size. A study of Kuhn et al. which classified bugs in bug databases by the number of involved parameters found that all bugs involve a maximum of six parameters [KWAMG04]. As a result, six-way testing would have the same effectiveness as exhaustive testing for these programs. Looking at the percentage of bugs per number

of involved parameters in Figure 2.2 also shows that testing with higher testing strengths has diminishing returns. Most bugs either involve one or two parameters, while only 5-20 percent involve three parameters. For non-critical systems it may therefore be effective enough to test with a testing strength of only two or three.

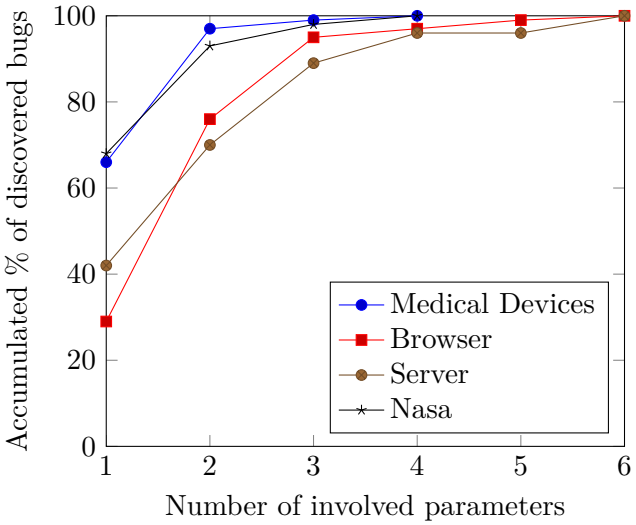

Figure 2.2.: Combinatorial testing effectiveness [KWAMG04; Bon18]

In some situations, defining one testing strength for all parameters may not be ideal [Cze06]. For example, consider a situation where the parameters describe two systems which interact in the test. If you know that failures are more likely to happen inside a system than in their communication, it would make sense to test the parameters of one system with a high strength but test combinations of parameters across systems with a lower strength. When looking at the running example, one can image that all parameters are just duplicated to OS1, OS2, Browser1, Browser2, and so on. Mixed-strength testing allows the user to define that the test suite tests all parameters inside the set {OS1, Browser1, Ping1, Speed1} at strength three while the default testing strength is two [Cze06]. As a result, the test suite must contain all value combinations between OS1, Browser1, and Ping1, but not those between OS1, Browser1, and Ping2. This reduces the number of necessary test cases when comparing to just testing everything at strength three.

Test Suite Generation A difficult, in fact NP-complete, problem in CT is finding a t-way test suite with the smallest number of test cases [YT98]. It has to combine each test case in exactly the right way so that it contains the most t-way sub-combinations not covered before. Since the problem is NP-complete, most generation approaches do not attempt to find an optimal solution for every possible IPM [NL11]. The three most common techniques are:

- Algebraic construction. This is only possible for a small subset of all IPMs. The

generation process is usually fast and generated test suites are as small as possible, but the requirements on the model are so specific that real combinatorial tests can seldom use them in practice [NL11; Col04]. A way to still use them is to extend the IPM by additional dummy values or parameters and later replace them with real values. While this method remains fast, the test suite size will no longer be optimal. Additionally, constraint handling is not easily possible [GOM07; NL11].

- Use of heuristics. Researcher successfully applied techniques such as simulated annealing to CT test suite generation using special heuristics for test suite construction [AG+12; NL11]. A problem with this approach is that most of them are rather slow, but on the other hand they produce a small number of test cases. Some heuristic construction methods also support features like constraints [GOA05].

- Greedy construction. These algorithms iteratively construct the test suite by always choosing a next value/test case so that it covers the highest number of t-way combinations out of a list of alternative values/test cases [GOA05; NL11]. Examples include *In-Parameter-Order-General (IPOG)* and the *Deterministic Density Algorithm (DDA)* [YT98; QC13; BC05; BC07]. An advantage of greedy algorithms is the fast generation. Especially for pairwise and three-way testing it usually takes about a second to construct a test suite for a medium sized model with about ten parameters. However, these test suites have a higher number of test cases than those constructed with other approaches [NL11; GOA05]. In the end, testers have to make a trade-off between fast generation and small test suites. Greedy algorithms nearly always include support for constraints and other CT features, or it is possible to extend them.

Another category of distinction between generation algorithms is repeatability. Some approaches, like IPOG and DDA, will always generate the same test suite when given the same IPM (deterministic algorithms) while others, like Simulated Annealing or the *Automatic Efficient Test Generator (AETG)*, will not (non-deterministic algorithms) [GOA05].

Fault Characterization

After a generation algorithm computes a combinatorial test suite for the given testing strength, a tester needs to execute those tests. The result is usually a list of combinatorial test case results such as in Table 2.2. Since some of the test cases have failed, a developer would now get a notification which contains the tested parameter values and the concrete exception message and then start to debug the application with an exact replication of the environment and input. However, in most cases the debugging time could be shorter if the developer knew in advance which sub-combination actually causes a failure [NL11; Gha+15]. With the example results in Table 2.2, one could theorize that every test case which contains the combination (–, Firefox, 1000 ms, –) fails. This additional information could lead to a faster discovery of the defect since the developer now knows that the problem is probably a timing bug in Firefox specific parts of the code which appears

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

Table 2.2.: Example test case results

in environments with a high ping. In such a case (–, Firefox, 1000 ms, –) is called the *failure-inducing combination (FIC)* since its presence in a test case will always cause a failure [Gha+12].

A naive way of finding FICs is to take the set of all sub-combinations contained in a failing test case and remove those which also appear as part of a successful test case since those cannot be failure-inducing. In case of the example from above, the test case (Android, Firefox, 1000 ms, 1000 KB/s) (the last test case in Table 2.2) contains 16 sub-combinations (including the full test case and the empty one). Since Table 2.2 already contains a successful test case the empty combination cannot be failure-inducing, and the success of (Android, Firefox, 10 ms, 1 KB/s) also rules out (Android, –, –, –), (–, Firefox, –, –), and (Android, Firefox, –, –). With the complete test suite results only six combinations remain. (–, Firefox, 1000 ms, –) is the smallest of those. However, the problem is that a FIC must **always** cause a failure, and CT does not test exhaustively by design. Therefore, the combination (iOS, Firefox, 1000 ms, 100 KB/s) could be successful, but it is not part of any test case since all contained three-way combinations are already part of other test cases. It is therefore possible, and even likely, for the naive algorithm to return a wrong FIC.

To address this problem, a combinatorial test needs to generate and execute test cases which go beyond the CT coverage criterion to cover additional combinations which ease the detection of FICs. Such techniques can either generate those additional test cases before execution (non-adaptive) or afterwards when they know the initial test case results (adaptive) [BFL19]. The latter onces have the advantage that they know which test cases have failed [Zhe+16]. Therefore, they can specifically target potentially failure-inducing combinations in their additionally generated test cases and thus may require fewer test cases to discover FICs. Additionally, adaptive algorithms have the advantage that they need to execute fewer test cases if no test case fails, since non-adaptive approaches always need to generate a larger test suite beforehand. The algorithms of both approaches are called *fault characterization (FC)* algorithms.

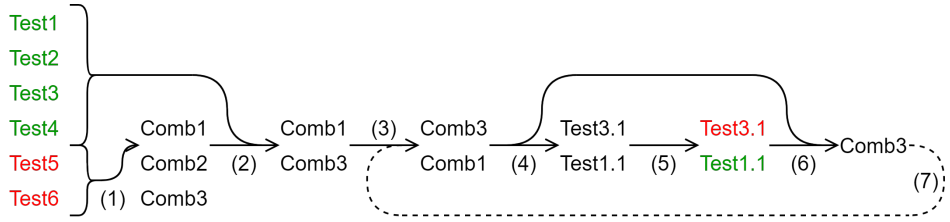

Figure 2.3.: Basic algorithm of BEN as depicted in a previous bachelor thesis [Gha+15; Bon18]

An especially hard topic for FC algorithms is constraint handling. With constraints, certain combinations become untestable and it is harder to construct test cases containing specific combinations since very few extensions to full test cases may be valid. Therefore, only few FC algorithms actually support constraints, one of them being the coffee4j implementation of Mixtgt [AGR19].

BEN BEN is a fault localization tool [Gha+12; Gha+15]. As such, it does not only perform FC to find a failure-inducing combination, but also collects code coverage information during the execution of special test cases containing those FICs to narrow down the piece of code which actually causes the failure. This paragraph will take a more in-depth look at how BEN performs adaptive FC, because later chapters will use the approach in an adjusted form. From now on, BEN will refer to the included FC algorithm.

BEN works by keeping an internal list of suspicious combinations of size t. A combination is suspicious if it appears in at least one failed test case and in no successful test case, similar to how the naive FC approach worked. To be relatively sure that a combination is actually failure-inducing, BEN will then generate test cases which contain suspicious combinations together with values that are not likely to be in a FIC. If this test case fails it is therefore very likely that the suspicious combination is actually failure-inducing. In the example from above, BEN could generate the test case (Linux, Firefox, 1000 ms, 10 KB/s) for the suspicious combination (Linux, Firefox, 1000 ms, –) if 10 KB/s appears in few failing test cases. BEN repeats this process until it does not generate any new test cases. At this point it reports the most suspicious combinations as failure-inducing. If the FIC is a r-way combination with $r < t$ BEN can also detect it since all possible t-way combinations which contain the FIC are necessarily suspicious. Since the whole concept of BEN builds upon the likelihood of a combination being failure-inducing it does not work as reliable as other FC algorithms, because it can report combinations that are not actually failure-inducing. However, early tests with BEN show that the results are generally reliable [Gha+15].

Figure 2.3 shows this process in more detail. First, in step (1), BEN extracts all combinations contained in failing test cases to form the list of suspicious combinations. Next, it refines this list in step (2) by removing all combinations which occur in successful test cases. Step (3) then orders the suspicious combinations according to the internal

likelihood of them being failure-inducing, and step (4) creates additional test cases around the top x suspicious combinations. Some external system then needs to executes them in step (5), since BEN cannot automatically execute test cases. If a test is successful — such as the one for combination 1 — the algorithm removes it from the list of suspicious combinations in step (6), and then uses the new test results to update the combination suspiciousness in step (7), which also starts a new iteration if necessary. Once BEN is sure that the remaining combinations are actually failure-inducing, it returns them to the user.

To construct new test cases, BEN needs to define which values are likely to be in failure-inducing combinations and which are not. It does so with a concept of component suspiciousness, where a component is the assignment of one parameter to a value, or a one-way combination. The following four formulas are of central importance [Gha+12]:

$$\rho(o) = \frac{1}{3}(u(o) + v(o) + w(o)) \tag{2.1}$$

$$u(0) = \frac{|\{t \in T | result(t) = failure \wedge o \in t\}|}{|\{t \in T | result(t) = failure\}|} \tag{2.2}$$

$$v(o) = \frac{|\{t \in T | result(t) = failure \wedge o \in t\}|}{|\{t \in T | o \in t\}|} \tag{2.3}$$

$$w(o) = \frac{|\{c \in S | o \in c\}|}{|S|} \tag{2.4}$$

Formula 2.1 defines that the suspiciousness ρ of component o includes three equally weighted parts u, v, and w. Formula 2.2 defines that u is the ratio of failed test cases which contain o and the number of failed test cases, while v is the ratio of the failed test cases with o and the number of test cases in which o also appears, as defined in Formula 2.3. $w(o)$ deals with the list of suspicious combinations S. Formula 2.4 defines it as the ratio of the number of suspicious combinations in which component o appears and the total number of suspicious combinations. All in all, the component suspiciousness rises if a component appears in many failing test cases, if most test cases in which it appears fail, and if it is part of many suspicious combinations.

For an example, one can look at the limited number of test case results in Table 2.2. Here, component (–, Firefox, –, –) appears in two successful test cases and four failing ones. There are 16 suspicious three-way combinations, and Firefox appears in 12 of them. Consequently, $u(Firefox) = \frac{4}{4} = 1$, $v(Firefox) = \frac{4}{6} = 0.\overline{6}$, and $w(Firefox) = \frac{12}{16} = 0.75$. All in all, the suspiciousness is $\rho(Firefox) = \frac{1}{3}(1 + 0.\overline{6} + 0.75) = 0.80\overline{5}$. For component 1 KB/s, which is not part of the FIC, the result is $\rho(1 \text{ KB/s}) = \frac{1}{3}(\frac{1}{4} + \frac{1}{5} + \frac{3}{16}) = 0.2125$. All in all, the result for components in FICs should be close to one, while the result for other components should be near zero.

Combinatorial Test Prioritization

While CT reduces the number of necessary test cases, it can still require hundreds or thousands of them. Depending on the individual test case execution time, it may take anything from some seconds to multiple hours to completely execute a combinatorial test suite. In a continuous testing scenario this is not acceptable. As with all other forms of regression testing one needs some form of prioritization so that developers get quick feedback and can start debugging test failures as early as possible.

Paragraph "Regression Test Prioritization" of Subsection 2.1.1 already introduced the concept of using information from previous runs to construct a near optimal order in which to run test cases. If one combines this with CT the combinatorial test selection criteria could be a form of RTS, since it aims to execute a small subset of all possible test cases while still maintaining good test coverage. The application of RTP to CT would then cover the t-way combinations in a specific order [QC13; NL11].

If a generator would always generate the exact same combinatorial test suite, test case prioritization would work just like with all other regression tests. Every technique build for traditional RTP becomes directly applicable. However, in CT there is an advantage which is not present in other testing methods: it is possible to adjust the actual test cases. In addition to collecting information about test cases as a whole, it is also possible to collect it about individual values. In the generation step special algorithms could then generate a test suite in such a way that early test cases combine the most important values — thus optimizing the test suite further than with traditional RTO [NL11; QC13]. For example, consider the two test cases (Android, Edge, 100 ms, 1 KB/s) and (iOS, Chrome, 10 ms, 10 KB/s). If the values Android, Edge, 10 ms, and 10 KB/s are often in FICs then traditional prioritization would have to execute the two test cases one after the other and may catch a failure only with the second one. A CT generation algorithms could use the additional information to generate a new test case (Android, Edge, 10 ms, 10 KB/s) as the first test case.

One can classify existing prioritization approaches in two categories, the point of prioritization, and if/how it uses additional information.

Point of Prioritization There are two alternative points of prioritization which the paragraph above already shortly presented. One option is to generate a normal combinatorial test suite and prioritize the test cases afterwards [QC13]. While this seems to be like traditional RTP, is can become difficult if the used generation algorithm, like AETG, always generates different test suites. The prioritization technique can then no longer order previously executed test cases. Even if the test suite is always the same, there can still be differences to normal RTP if the additional prioritization information is only available on individual values or incomplete combinations. Prioritization techniques then need to combine the information for an effectiveness rating of a complete test case.

The other option is already using the available information when generating the test suite [QC13]. Depending on the information this means that the test suite may change even if the algorithm is deterministic and the IPM always stays the same. However, such optimizations always have a trade-off. If the generation considers the importance

of values, it may run into conflicts with the goal of covering all t-way combinations as fast as possible, therefore making the resulting test suite larger [QC13]. Studies on real programs have to show whether the earlier fault detection is worth the larger test suites.

Available Information The second category of classification is the available information. Some prioritization algorithms, such as incremental interaction coverage, do not use any additional information at all [Hua+13]. Instead, they prioritize a combinatorial test suite using the testing strength. The main idea is to first test all possible one-way combinations, then two-way combinations, and so on up to all possible t-way combinations. Since Figure 2.2 already showed that most failure are caused by a small number of parameters, this technique of iteratively increased testing strength could find failures faster.

Other techniques encode additional information as value weights in the IPM [QC13]. Here, a higher weight means that a test suite should either contain a value more often or it should appear in an earlier test case, depending on the prioritization technique. Testers can also use negative weights to model costs associated with certain values. For example, it could be costly to test on a MacOS system and therefore this value should not appear as often as other values while still maintaining the combinatorial test coverage criterion. With these techniques an important point is the determination of value weights from external information.

A third category of prioritization approaches uses the additional information at the actual point of prioritization, i.e. during the generation or afterwards. Since the information is only available to the algorithm which retrieves it, other prioritization or fault characterization algorithms are not able to use it.

2.2. Technologies

This thesis requires a certain understanding of some technologies. For example, the realization in Chapter 5 uses the Java programming language, and requires features such as annotations. Since the realization integrates into coffee4j, an automated CT framework developed in earlier theses, it also requires background knowledge about the inner workings.

2.2.1. Java

Sun Microsystems first developed the Java programming language in 1995. Currently, it is under the ownership of Oracle after the company acquired Sun in 2010. Although often referred to as an object-oriented programming language, Java combines multiple paradigms such as the aforementioned object-oriented, functional, concurrent, and generic programming. It also supports reflection, the examination and modification of internal programming properties of a program itself [HC02].

Java is a general purpose language. Due to the rich ecosystem of third-party libraries and frameworks it is possible to program graphical user interfaces, big data analysis engines, search engines, databases, and many more applications. Additionally, the

Spring framework and Jakarta EE also enable its use in back-end server applications or microservices in companies such as Netflix and Twitter [OC20].

One popular Java feature often used by frameworks is annotations. They can add static information to annotatable elements such as classes, methods, fields, and parameters. Code generation frameworks can use annotations to decide what code to generate, and other frameworks like the testing framework JUnit use them to distinguish between certain kinds of methods. For example, the following could be an annotation for a test framework:

```
1 @Target(ElementType.METHOD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface Test {
4   String name() default "";
5 }
```

Source Code 2.2: TestAnnotationDefinition.java

An annotation is a special interface (`@interface`) which may only have non-void no-argument functions that return other annotations, strings, primitive data types, or arrays of any of those. It may also specify a default value for every function using the **`default`** keyword. Often, annotations are themselves annotated with annotations. In this example, `@Target` specifies the element types on which developers can put the annotation, and `@Retention(RetentionPolicy.RUNTIME)` defines that the information will be accessible via reflection at runtime. Otherwise only code generation during compilation could access the information. Developers can then use the `@Test` annotation in the following way:

```
1 @Test(name = "some test name")
2 public void testMethod() {
3   // perform some test steps
4 }
```

Source Code 2.3: TestAnnotationUse.java

2.2.2. JUnit

One of the most often used Java frameworks is JUnit. As part of the xUnit family of testing frameworks, it inherits the popular way of writing tests form Kent Beck's original SUnit for testing with Smalltalk [Bec97]. Since its original inception JUnit underwent several key changes. The most recent one was a complete rewrite and redesign of the architecture with JUnit5. Since then, the framework developers split it into a general testing platform called JUnit Platform, and the default test engine named JUnit Jupiter [Tud20; Bec+20].

Figure 2.4 depicts this general architecture. The platform is responsible for providing a general interface to all build tools, e.g. maven, and development platforms, e.g. IntelliJ through which they can discover and execute tests. However, it does not handle discovery

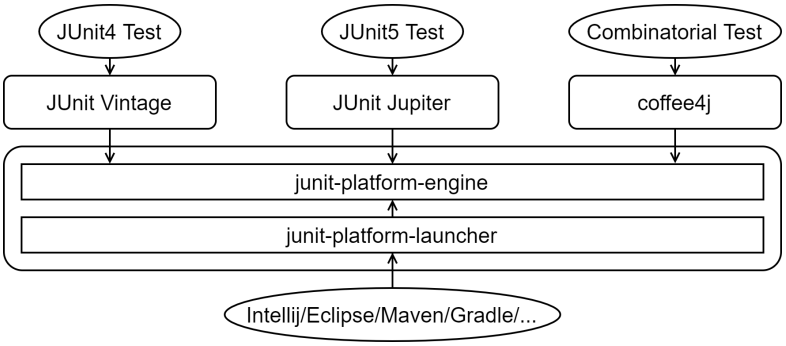

Figure 2.4.: JUnit5 Modular Architecture [Tud20]

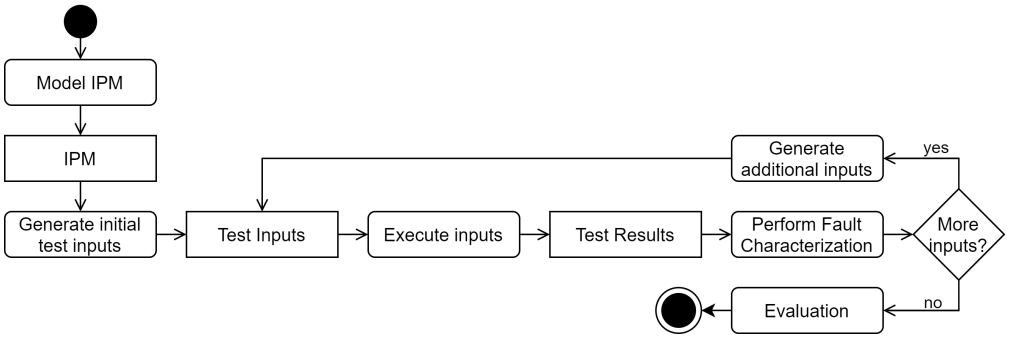

Figure 2.5.: Process of old coffee4j version [BFL19]

and execution by itself, but rather delegates this to so called *Test Engines* [Bec+20]. Each engine is responsible for handling its own kind of test. For example, there is an engine which can integrate old JUnit4 tests into JUnit platform, and one engine for the new JUnit5 way of writing tests. Additionally, this architecture also allows for the integration of third-party engines which implement the `TestEngine` interface. coffee4j is one example.

2.2.3. coffee4j

A central part of this thesis deals with the coffee4j framework [Bon18; BFL19; Ber19]. Therefore, this sub-section will roughly explain the realized CT process, architecture, and how to use the framework in its current state.

Process

Figure 2.5 shows the basic process which coffee4j follows during execution. First, the user has to define an IPM in the internal domain specific language. This is necessary so that coffee4j can know which parameters and values to consider during the generation. However, due to the extensible architecture of coffee4j it would also be possible to automatically convert the IPM from any other popular format.

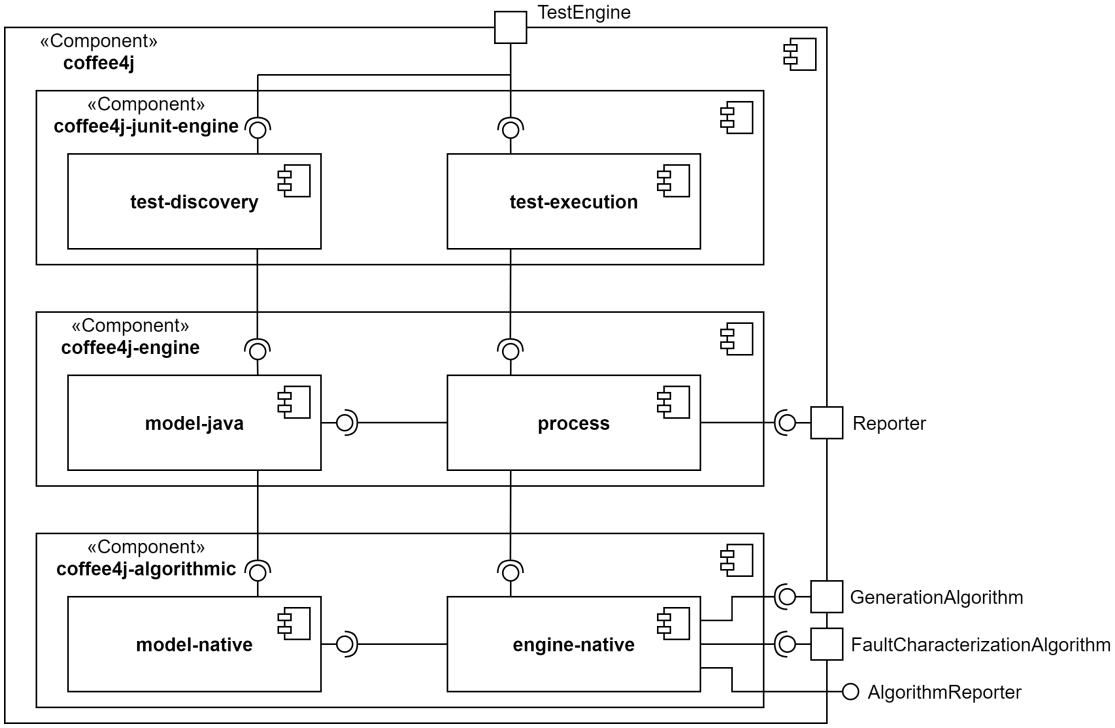

Figure 2.6.: Architecture of old coffee4j version [BFL19; Ber19]

The next step contains the initial generation of test inputs. For this task, coffee4j just takes on a management role and delegates the actual generation to another algorithm. Such a clear separation allows for the user to decide which algorithms the framework should use and, if necessary, also write a custom one. The result of the generation step is a combinatorial test suite, which contains a number of test cases that together cover all t-way combinations which are possible in the user-defined IPM. Next, coffee4j executes those test inputs. For this task the user defines a `TestInputExecutor` which takes a test case and evaluates whether the SUT behaves correctly or not.

With the test case results, it is then possible to evaluate whether fault characterization is necessary. If this is the case, a user-supplied fault characterization algorithm can generate additional test cases until it knows which combinations are failure-inducing. In that case it returns no more test inputs for execution and instead provides a list of discovered FICs.

To extend coffee4j and modify the process, there are also callbacks before and after each important phase [Ber19].

Architecture

coffee4j uses a layered architecture as Figure 2.6 depicts. `coffee4j-junit-engine`, the topmost layer, contains an implementation of JUnit5's `TestEngine` interface [Ber19].

It is therefore responsible for discovering all combinatorial test methods in a specific target directory or class, and relaying a structured test plan to the JUnit platform. To recognize test methods, it uses a custom `@CombinatorialTest` annotation. The test discovery mostly uses the `EngineDiscoveryRequestResolver` made available by JUnit5 to reduce the amount of code which needs to be developed specifically for coffee4j. Therefore, nearly everything works like in JUnit Jupiter, the default test engine. For example, test developers can execute all test methods in a class with the development environment just like with any other test class.

Separated form test discovery, the `test-execution` component is responsible for loading the user-defined configuration of a combinatorial test, including configured algorithms and the IPM. Subsection "Usage" will explain how the configuration discovery works. The component then executes each combinatorial test method one at a time with a completely new instance of `coffee4j-engine` and `coffee4j-algorithmic`. Therefore, only the JUnit layer has to deal with multiple test methods, the lower levels can each focus on one CT. Additionally, as known from JUnit Jupiter, the coffee4j engine also supports lifecycle callbacks. Test developers can use those to manage test setup and tear-down [Bec+20; Ber19].

Under the JUnit layer there are two layers for coffee4j itself. Theoretically, only one layer is necessary, but due to performance constraints we decided to split the framework into two parts early on [Bon18; BFL19]. The lower level, `coffee4j-algorithmic`, only deals with primitive data types. For example, it defines a parameter as one integer, the number of its values, and references values only via their indexes. An array like `[0, 2, 0, 1]` can therefore represent the test case (Windows, Firefox, 10 ms, 10 KB/s). Due to the internal memory layout of the Java heap, this is a significantly better representation due to memory locality, and lead to a high performance gain in comparison with directly using `Parameter`, `Value`, and `Map`-based `Combination` classes.

To avoid confusing the user with index-based reporting and definition of combinatorial test, `coffee4j-engine` contains the actual `Value` and `Parameter` classes for an easy to use interface. As a result, implementations of the `Reporter` interface deal with the easy to use classes. The definition of IPMs is located in `model-java`, and `process` contains the management of the CT process as defined previously.

All computationally intense parts, like the generation of an initial test suite or fault characterization, happen in the `coffee4j-algorithmic` component. This always requires a conversion between the two internal representations.

Usage

There are two different ways to use coffee4j. The first one is through its Java *application programming interface (API)* which configures test cases using class instances constructed using the builder pattern and the second one is through the JUnit `TestEngine` [Bon18; Ber19]. This section will focus on the second one.

```
1 private static InputParameterModel.Builder model() {
2   return inputParameterModel("browserGame")
3     .positiveTestingStrength(2)
```

```
4       .parameters(
5         parameter("OS").values(
6           "Windows", "Linux", "MacOS", "Android", "iOS"),
7         parameter("Browser").values(
8           "Chrome", "Edge", "Firefox", "Safari"),
9         parameter("Ping").values(10, 100, 1000),
10        parameter("Speed").values(1, 10, 100, 1000));
11  }
12
13  @CombinatorialTest
14  @EnableGeneration(algorithms = Aetg.class)
15  @EnableFaultCharacterization(algorithm = Ben.class)
16  void executeTest(
17      @InputParameter("OS") String operatingSystem,
18      @InputParameter("Browser") String browser,
19      @InputParameter("Ping") int ping,
20      @InputParameter("Speed") int speed) {
21    // some test code which uses the parameters
22  }
```

Source Code 2.4: Coffee4jUsageOldVersion.java

In the example code, the first method defines an IPM using the coffee4j API. Every model has a name, and any number of parameters which each contain an arbitrary number of values. In contrast to other CT frameworks, coffee4j does not limit parameter types. Values can have primitive types (e.g. `int`), build in reference type like `String`, and even custom user supplied types. If a value type does not match the type as given in the test method, coffee4j will throw an exception.

For the configuration of algorithms, coffee4j relies heavily on annotations. As mentioned before, the custom `@CombinatorialTest` annotation marks a method as a combinatorial test method so that the JUnit layer of coffee4j can discover it. Through this annotation it is also possible to change some default values like the model method name and display name in the JUnit tool windows in an integrated development environment.

Every configurable feature of coffee4j then has an additional annotation which mostly sets the class to use. For example, `@EnableGeneration` configures the algorithms used for the initial test suite generation, and `@EnableFaultCharacterization` defines the algorithm to use for FIC discovery. The test method itself then has the same input parameters as the ones defined in the IPM and references parameters using the `@InputParameter` annotation. From the user perspective, this will then behave like a JUnit Jupiter parameterized test, where the framework calls the test method once per parameter value assignment. The only difference is that the tester does not define combinations beforehand but instead lets coffee4j calculate them to cover the CT coverage criterion.

Of course there are many more features with which developers can configure a combinatorial test. coffee4j therefore provides an `coffee4j-example` module which contains at least one explaining example for each of them.

3. Related Work

Contents

Currently no approach which combines both failure-based regression test prioritization and combinatorial testing exists. However, there is existing work on both individual topics. This chapter will give an overview of both to help understand how the approach presented in this thesis works, and what it does which has not been done before. Additionally, it presents an existing tool which integrates combinatorial test prioritization.

3.1. Failure-based Regression Test Prioritization

As briefly mentioned before, some existing techniques prioritize test suites based on historical information about test failures. The common reason for that is the notion that there are some test cases which fail more often then others. In such a case it would make sense to run them first.

One approach for failure-based test prioritization for regression testing developed by Marijan et al. is ROCKET [MGS13]. ROCKET examines the historical execution data of each test before performing a prioritization step. To avoid old failure information from interfering with the prioritization, it weights each test run, with the latest one receiving the highest weight. Therefore, if a continuous integration pipeline detects a failure, there is a high probability that the next pipeline run will execute the failing test case, leading to quick failures if no one fixed the bug, or fast reassurance that the fix was successful if the pipeline does not fail during the first few tests. Similarly, the tool will prioritize test cases lower if they previously revealed bugs, but someone already fixed them a long time ago and they did not resurface. In addition to failure-based prioritization, ROCKET also takes into account the total test execution time when creating a prioritized test suite. Very fast test cases may therefore run earlier even if they have a lower error detection rate. In a case study, ROCKET detected regressions faster than random and manual testing.

Cho et al. extended ROCKET using statistical analysis of the failure history [CKL16]. This reduces large variations in the priority values of tests in subsequent runs.

Similarly, Kim et al. developed an RTS technique which prioritizes test cases based on recent failures and the time passed since a test last ran [JP02]. As a result, their

algorithm attempts to cycle through the entire available test suite over multiple runs, while simultaneously executing those tests more often which recently caused failures. The parameter α can control the trade-off between the two prioritization goals. In a study on several programs of the Siemens Suite, they concluded that history-based RTS has an advantage over non-history based RTS but the computational costs for creating such a test suite were very high.

Ali et al. developed yet another approach which uses historical failure data called the CTFF model [Ali+20], This prioritization and selection technique focuses on agile development processes and uses user stories to identify relevant test cases. It then groups them into clusters according to their change frequency and prioritizes the whole cluster higher if one test case has a high failure-frequency.

All in all, there are multiple approaches which prioritize test suites based on historical failure-information, and the techniques generally improve failure detection times. However, all of the approaches presented above do not explicitly deal with CT. While it would be possible to integrate them when using a deterministic generation algorithms, they would fail as soon as the IPM changes in any way. With agile development, which needs to react to fast changing requirements, model changes can occur very frequently. As those model changes completely change the generated test suite, regular RTP and RTS approaches would have no historical information on which they could build their analysis. It is therefore necessary to specifically adapt these approaches to CT. However, traditional RTO algorithms may still work as an inspiration for CTCP.

3.2. Combinatorial Test Prioritization

Equivalently to how there are many failure-based prioritization approaches which do not work with CT, there are also some CT prioritization approaches which do not take failure-information into account. Nevertheless, the way in which they perform CTCP is relevant and related to this work.

One of the simpler CT prioritization techniques is incremental interaction coverage. This technique, developed by Huang et al., first covers all one-way combinations, then all two-way combinations, and continues to incrementally increase the coverage until the test suite covers all t-value combinations [Hua+13]. Since most faults involve few parameters, this technique can detect failure in earlier test cases. However, this technique does not use additional historical information. While this makes it easy to apply without an infrastructure to store these information, this also means that it may prioritize test cases unnecessarily high. This can happen due to the fact that the approach does not prioritize test cases for the same t-value combinations against each other, but instead generated like in normal combinatorial testing.

As mentioned in Paragraph “Available Information” of Subsection 2.1.2, many other approaches also use weights for prioritization [Cho+16; QC13; BC05]. Higher value weights mean that the value should appear earlier in a test suite. Qu et al. use this concept to assign weights based on previously collected code coverage information. To this end, they present three weight calculation approaches. In all of them, the first step is

to create an ordered subset of all test cases based on the cumulative coverage information by iteratively adding the next test case with the highest remaining branch coverage. The first weighting algorithm then assigns each value a weight based on the number of value occurrences in the subset divided by the total subset size. Since the subset contains the test cases which ensure complete branch coverage with the lowest number of test cases, this weighting should in theory lead to fast branch coverage. The second weighting scheme multiplies the result of the first one with a parameter weight based on the maximum value weight in each parameter. Instead of changing the calculation of the weight based on the created subset, the third method changes the subset creation by adding multiple test cases with the same additionaly branch coverage in each step.

While all three techniques use historical data for prioritization, they do not take any failure information into account. For example, if a test case covers just very few lines, but these lines often fail, the techniques will still execute it towards the end of a prioritized test suite. The approach presented in this thesis would instead execute it up front.

When assigning weights to individual values, an important part of the prioritization technique is the prioritized generation of combinatorial test suites. Most papers do this by using an adjusted version of DDA.

```
1  RemainingCombinations = AllCombinationsAtStrengthT;
2  while (RemainingCombinations is not empty) {
3    Order parameters according to t-way weights
4    Initialize next test with all parameters assigned to no values
5    for (parameter = nextHighestFactorWithoutValue) {
6      Compute t-way value interaction weights
7      select value with highest weight and set it in the test
8    }
9    add test to test suite
10   remove covered t-way combinations from RemainingCombinations
11 }
```

Source Code 3.1: DeterministicDensityAlgorithm

At the highest level, it generates a test suite one test case at a time. For each test case, the algorithm orders all parameters by their accumulated possible interaction weight, i.e. it adds the weights for every uncovered combination which contains the current parameter together. The weight of a single combination is the product of all individual value weights. Afterwards, the for loop initializes the current test case one parameter at a time. For each parameter, it selects the values with the highest weight, where the weight of one value is the sum of the weights of all uncovered combinations in which it appears. This is done until the test suite covers all t-way combinations [QC13; BC05; BC07].

This thesis extends the basic algorithm to also support constraints. While Bryce et. al. already developed a technique for handling constraints, this is not general enough to be of use [BC06]. Their approach assigns negative weights between -1 and 0 to interactions or values that should not appear often (soft constraints) or at all (hard constraints). The algorithms then automatically avoid selecting values contained in those

interactions. While this approach works for avoiding specific two-way combinations, it does not work if the avoidance depends on other additional values. For example, one could think of a constraint that involves three parameters even when testing only all two-way combinations. If all two-way combinations involved in this invalid three-way combination should appear at the beginning of the test suite, they need a high weight, which means the algorithm could potentially select them in one test case. Our extension avoids this and instead delegates the constraint handling to an external SAT solver.

3.3. Combinatorial Test Prioritization Framework

In addition to developing a failure-based combinatorial test case prioritization technique and extending DDA, this thesis also extends coffee4j to support test case prioritization. The only somewhat related work in this area are tools researchers implemented to test the previously mentioned algorithms, and CPUT, a tool which uses combinatorial testing for prioritization [Sam+11]. However, none of those support general, user-defined prioritization algorithms. It would therefore always be necessary to adjust the source code of the program to include a new algorithm. In contrast, the extension of coffee4j to test case prioritization needs to support all of the above.

4. Concept

Contents

After the last two chapters gave the required background knowledge, the next three chapters will now present the contributions of this thesis. This chapter begins by introducing the concepts which went into the development of the contribution. Section 4.1 first explains how to extend coffee4j to support CTCP. To this end, it collects general requirements of different user groups, and then uses them to extend coffee4j's underlying CT process and architecture. Next, Section 4.2 develops concept for test case prioritization using previously failed test cases, and Section 4.3 develops an algorithm to enable prioritized test suite generation with constraints based on value weights.

4.1. Framework

Currently, coffee4j only implicitly supports two possible approaches for prioritization based on the categorization in part "Combinatorial Test Prioritization" of Subsection 2.1.2. With regard to the point of prioritization it is only possible to prioritize in the generation algorithm itself, since no extension point for later prioritization exists. Similarly, those generation algorithms can only either use no additionally information or load it themselves since there is no extension point for loading it into the IPM. For example, this means algorithms which construct test suites based on incremental interaction coverage are already possible in coffee4j without any changes.

Ideally however, coffee4j, should support all current prioritization approaches. This means that this thesis must at least extend it to support prioritization after generation,

and it should also support certain approaches which encode additional prioritization information in the IPM. To reach this goal, the next few subsections will first collect a formalized list of requirements and then transform them into a generalized CT process and coffee4j architecture.

4.1.1. Requirements

To evaluate whether a software architecture is well suited for a certain task it is important to know all information which it has to consider. In software engineering, these information are usually called requirements. They consist of two distinct groups: functional and non-functional requirements.

Functional requirements describe the actual function of a software system. They are the observable results and options present in the system and in general what the system should accomplish. On the other hand, non-functional requirements define *how* the system should accomplish its functional requirements. This can include objectives observable by the user (e.g. performance and usability) or concepts important to the continued development of the system (e.g. testability, extensibility, and maintainability).

User Groups

Before defining the individual requirements, it is often advantageous to first define different possible users of the system. In a real software system one would also look at other stakeholders than just the users, but in the scope of this thesis focusing on the users is enough. There are four main groups of users, partially taken from previous work [Bon18; Ber19]:

Test Developer Test developers are the ones who actually write combinatorial tests using coffee4j. They only interact with coffee4j using the top-level interface, usually in form of the JUnit test engine. Consequently, testers usually do not care for how coffee4j actually works internally as long as it supports the needed CT features and follows the defined process. Since they work quite extensively with the user interface, testers also want the interface to be easily usable and consistent.

Product Developer Product developers are people who work on the SUT. Often, product developers also write the test cases, but to create better boundaries between the user group this section assume that they only execute test cases. Since they have to execute the combinatorial tests multiple times a day, either on their own machine or in a automated build pipeline, they care about the execution speed and are the prime target for prioritization measures.

Algorithm Developer Additionally, there are some users who write implementations for the framework's extension points. Those are called algorithm developers. Often, they are researchers in the field of CT who experiment with new generation, prioritization, or fault characterization algorithms. To develop the algorithms they have to know

more about the inner workings of coffee4j and understand the complete CT process. Additionally, algorithm developers often want to compare different implementations of the same algorithm, or other algorithms for the same task with each other. Due to easier configuration and repeated execution it is better to do this on the lower engine level API from 2.6.

Framework Developer In addition to people developing the specific algorithm implementations, there are also those who further continue the framework itself, like Konrad Fögen and other researchers. This user group has a focus on the evolution of the framework and therefore prioritizes properties such as maintainability and testability.

The next few sections will list the functional (**F<id>**) and non-functional (**N<id>**) requirements of the different user groups. Some of them may fit into multiple groups. If this is the case this section only lists for the user group with the most interest in the requirement. Inside each user group, the requirements have a common order. The first ones deal with the generator point of prioritization and then those for the after-generation point of prioritization follow. A third group of requirements then deals with integrating additional information into an IPM. Finally, group four contains all requirements which do not fit into either of those categories.

Test Developer Requirements

This section contains the functional and non-functional requirements which mainly deal with the definition of combinatorial test cases.

F1 Specification of Prioritizing Generator Users must have the ability to specify that coffee4j should use a generator which incorporates prioritization. This is required to support the generator point of prioritization from Paragraph "Point of Prioritization" of Subsection 2.1.2. As in previous works on coffee4j, it should again be possible to specify multiple generators, and also mix those which use prioritization information with those that do not. Additionally, users should be able to specify no prioritizing generators and instead use generators which construct non-prioritized test suites.

N1 No Special Specification of Prioritizing Generator Currently, coffee4j already allows for configuration of multiple generation algorithms using the `@EnableGeneration` annotation. Specifying a generator which uses prioritization information should not require a different annotation, but rather use the existing one. This avoids confusing the user and makes the coffee4j framework agnostic of the inner workings of a generator. While this requirement may seem obvious and trivial, it could also be possible to provide prioritization information separated from the IPM. In this case algorithms using prioritization information may have a different interface and therefore also different configuration options.

F2 Specification of After-Generation Prioritization Algorithm Instead of prioritizing the test suite during generation, Paragraph "Point of Prioritization" of Subsection 2.1.2 also mentioned the option of prioritizing in a later stage. There can be many different algorithms to order combinatorial test cases after an initial generation. The user should be able to configure which one the framework actually uses in the test configuration. coffee4j cannot handle prioritization according to different criteria by itself. Therefore, it should only be possible to specify at most one prioritization algorithm, although it is possible that the single algorithm delegates to multiple other algorithms. It must always be a valid configuration option to specify no prioritization algorithm.

N2 Uniform Specification of After-Generation Prioritization Algorithm The coffee4j framework already allows for the configuration of many algorithms for each combinatorial test, for example the initial generation and fault characterization algorithm. The configuration of the test case prioritization algorithm should use a similar configuration format so that the test developer programs to a uniform interface. This leads to better usability and explorability of configuration options. Additionally, it must be possible to use custom configuration options.

F3 Specification of Integrators for Prioritization Information into IPM As mentioned in Paragraph "Available Information" of Subsection 2.1.2, some prioritization approaches add additional information into the IPM. To support them, coffee4j must therefore allow the test developer to specify the concrete integrator which adds relevant information. In contrast with the prioritization algorithm from Requirement F2, it is possible to consider multiple different sources for prioritization information, such as historical data and information gathered from static code analysis. A test developer must therefore be able to specify as many information integrators as s/he likes. Again, it must also be possible to specify no information integrator at all if a prioritization approach requires no additional information.

N3 Uniform Specification of Prioritization Information Integrator Similar to Requirement N2, this requirement states that the configuration of information integrators must also be user friendly and use the same concept as other configuration options. This is done to ensure a uniform interface and good usability.

F4 Manual Specification of Value Weights As mentioned in Section 3.2, many existing prioritization approaches require the use of value weights. Since this is such a central concept, the IPM should directly support them. In addition to information integrators, users must then also have the ability to specify weights. This is an instance of manual prioritization, where tester can use her/his domain knowledge to manually supply information which automatic algorithms can then use for the prioritization.

F5 Manual Specification of Mixed Testing Strengths Mixed testing strengths is another approach to reduce the number of generated test cases. Since fewer test cases

lead to quicker failures, coffee4j should also get this feature as part of this thesis. The tester must be able to specify which parameter groups of the IPM coffee4j should test together at what strength, and also give a default base strength. In addition to the manual specification, automated regression test approaches could potentially also use this by testing parameters often involved in failures at a higher strength, or the other way around.

F6 Manual Specification of Seed Test Cases Testers often have an intuitive understanding of which combinations may cause errors. If such a combination includes more than t parameters, the only options to guarantee its inclusion in a combinatorial test suite is to either raise the testing strength to the size of the combination, or define a (partial) seed test case which includes the suspicious combination [Cze06]. Since seed test cases could also be automatically generated to re-test certain combinations in regression test scenarios, coffee4j must be extended to allow the manual and automatic addition of seed test cases to the IPM as part of this thesis.

N4 Uniform Specification of New Model Features The user interface should include the new IPM features mentioned in Requirements F4, F5, and F6 in such a way that it is uniform with the existing definition of the IPM. For example, this means that the fluent API could include them as part of the build pattern.

N5 Documentation via Examples To make it easier for test developers to use the new prioritization features, a form of documentation is necessary. Therefore, coffee4j must document the features via several examples.

Product Developer Requirements

After the requirements of test developers, this section will now take a closer look at the requirements a product developer has for test case prioritization in coffee4j.

F7 Use After-Generation Prioritization If Specified If the test developer used the options specified by Requirement F2, coffee4j must use the given algorithm to prioritize the initially generated test cases. This must only happen if the user configures a prioritizer. Otherwise, coffee4j should behave in the same way as before, and execute the initial test cases without any reordering. If activated, the reordering must occur immediately after the initial generation, before the framework calls any currently existing reporting endpoint. Therefore, the prioritization is transparent to the user, in the form that a user does not know whether prioritization happened during generation or afterwards.

F8 Use Prioritization Information Integrator If Specified Similarly, coffee4j must use the specified information integrators from Requirement F3 if at least one is specified. Otherwise, execution continues as normal. In the case that a test developer specified multiple integrators, coffee4j must execute them in the specified order until one integrator

fails or all were successful. The use of information integrators must never influence whether coffee4j executes prioritizing generators or after-generation prioritization algorithms.

F9 Reporting of Modified Model The integrators specified in Requirements F3 and F8 can change any aspect of the IPM. To avoid confusion, there should be a reporting endpoint or callback which reports each model modification to the user. Otherwise, the user may not understand why the prioritization did not work as expected, or why the test suite does/does not contain certain test cases.

F10 Specification of Fail-Fast Execution Mode To get fast feedback, especially in automated testing pipelines, coffee4j must support a so called fail-fast mode in which it stops test execution after the first failing test case. Product developers and test developers must be able to specify whether coffee4j should use this fail-fast mode, or whether coffee4j executes all test cases. In particular if the fail-fast mode is enabled, product developers must easily be able to disable it in their local environment because additional test executions may allow the developer to gain more insight into what causes the failure. To this end, an automatic determination of the execution mode must also be possible. For example, one can think of an extension which automatically determines whether the tests are currently executed in a automated testing pipeline or in a local development environment. In the pipeline, tests could then run in fail-fast mode, while the local environment executes all test cases.

N6 Uniform Specification of Fail-Fast Execution Mode Of course, the specification of the fail-fast mode must also adhere to the uniform specification interface defined in the current version of coffee4j. It must also require only a one line change to get back to normal execution mode.

F11 Stop Initial Test Suite Execution After First Failure in Fail-Fast Execution Mode As explained in Requirement F10, coffee4j must stop the execution of the initial test suite directly after the first test case fails if it is running in fail-fast mode. This must hold for all test cases, even across generation algorithm boundaries. For example, if the test developer specified two generation algorithms A and B, coffee4j must also stop executing the initial test cases generated by algorithm B if one generated by algorithm A failed.

F12 Perform Fault Characterization After First Failure in Fail-Fast Execution Mode coffee4j must only stop executing the initial test cases, not any other test cases. In particular, this means that fault characterization must still run after the first failed test case in fail-fast mode if the test developer specified a fault characterization algorithm. This is necessary so that the product developer still knows which parameters and values were involved in the failure.

F13 Report Failed Test Cases During Execution Especially in non-fail-fast mode, coffee4j must always report all failed test cases to the user immediately after the test

case failed. This is important so that the developer can always directly see that a test case has failed even if the rest of the tests are still running. Due to the shorter time to notice a failure, the developer can also start debugging earlier.

Algorithm Developer Requirements

Next, this section presents the requirements for coffee4j by the developers of prioritization approaches.

F14 Extension Point for Custom Prioritizing Generator Similarly, algorithm developers must also be able to develop their own generation algorithm which uses prioritization information. This additional information can either come from an external source directly referenced in the generator, or be embedded in the IPM via the new modeling capabilities from Requirements F4, F5, and F6. The extension point for developing a prioritizing generator must also be the same as the one for normal generators. This makes it easier for algorithm developers to create new algorithms. It also eases the integration of prioritization information into existing algorithms.

F15 Specification of Prioritizing Generator in Lower Layer API Experience gained in previous theses concerning coffee4j tells us that comparison between different algorithms is easier on the `coffee4j-engine` level than through the `coffee4j-junit-engine` API. It does not use the specific way with which JUnit executes individual test cases, and removes an unnecessary layer not important to the actual aspect under test. Therefore, it must be possible to specify a prioritizing generation not only on the JUnit-layer as specified by Requirement F1, but also on the `coffee4j-engine` layer. Here, the same conditions as in Requirement F1 apply; An extension developer can specify an arbitrary number of prioritizing generators but it is also possible to specify non at all.

F16 Extension Point for Custom After-Generation Prioritization Algorithm In addition to users being able to choose the used algorithm for test case prioritization after generation (Requirement F2), they should also have the ability to develop completely new algorithms. Therefore, coffee4j must have an extension point which it calls when it needs to prioritize the initially generated test cases. Algorithm developers must then be able to program to the interface of this extension point to create new algorithms.

F17 Specification of After-Generation Prioritization Algorithm in Lower Layer API Of course, extension developers must not only be able to specify necessary prioritizing generators on the lower layer API, but also after-generation prioritization algorithms. The same constraints as in Requirement F2 apply: Users can only specify one algorithm, and it is also possible to specify none.

F18 Availability of Model-Based Prioritization Information in Algorithms Prioritization information stored in the IPM, for example as value weights, must not only be

available to the generation and test case prioritization algorithm, but also to other extension points in the `coffee4j-algorithmic` layer. This includes the fault characterization algorithm and the algorithms involved with interleaving generation (not actually part of this thesis).

F19 Extension Point for Loading and Integrating Prioritization Information into IPM One place where many CTCP approaches differ is not the actual prioritization in the extension points of Requirements F16 and F14 but the calculation/loading of additional information for prioritization [QC13]. For example, multiple approaches can calculate the value weights (Requirement F4) differently, but still use the same weight-based algorithm for prioritized test case generation. coffee4j must therefore definitely have an extension point where algorithm developer can specify this. The extension point must allow for the implementation to load arbitrary data, since one cannot yet think of all possible data that can be used for prioritization. Additionally, it must be able to adjust at least the IPM properties for prioritization presented in Requirements F4, F5, and F6.

An important part of this extension point is also that it has to be able to identify parameters across multiple runs. In the topmost layers of coffee4j, `coffee4j-junit-engine` and `coffee4j-engine`, this is not a problem since each parameter has an explicit name, and values identify themselves through equality. However, the lower layer, `coffee4j-algorithmic`, represents parameters and values as indexes. Assigning, for example, value weights only based on indexes can lead to wrong prioritization. Consider the running example form Table 2.1. `coffee4j-algorithmic` represents it as `[5, 4, 3, 4]` via the order of parameters and number of values. However, if the user decides to exchange the order of Browser and Speed, the model would still look the same, as both parameters have the same number of values (4). An information integrator could not distinguish the two models and therefore also not assign value weights correctly. Consequently, the extension point for loading additional information for prioritization cannot be on the `coffee4j-algorithmic` layer of coffee4j.

F20 Specification of Integrators for Prioritization Information in Lower Layer API Similarly, the integrators for additional information used during the prioritization must also be configurable in the `coffee4j-engine` layer to allow for easier comparison of different prioritization approaches. Here, the specifications from Requirement F3 apply: the user can specify no integrator, or an arbitrary number of them.

N7 Good testability of extension points The algorithms behind common prioritization approaches can be very complex [QC13; BC05; BC07]. Especially prioritized combinatorial test suite generation (Requirement F14) may often fail only in certain edge cases. Therefore, testing the testing system is necessary in this case. To aid algorithm developers with this process, the extension points must be as simple as possible and should not require the definition of many dependencies for successful test execution. Ideally, the extension point allows for the algorithms to be a stateless function which maps from certain inputs, like an IPM to a defined and easily verifiable output. An algorithm

developer then only needs to write test cases in the form of inputs and expected output. Using combinatorial testing would also be possible.

Framework Developer Requirements

The last user group, framework developers, also has some requirements regarding the integration of test case prioritization capabilities into coffee4j. This section lists those requirements.

N8 Test Framework Independence of Prioritization As mentioned by Bernwald, the lower levels of coffee4j (`coffee4j-engine` and `coffee4j-algorithmic`) must be independent of the used testing framework [Ber19]. This must of course stay the same when coffee4j integrates prioritization. Even if, for example, TestNG is used as a user-interface for coffee4j, it should still be possible to prioritize the individual test cases. Requirements F17 and F20 already partly ensure this, but it is still important to consider this requirement explicitly during the development of architecture and implementation.

N9 Testability of Prioritization Requirement N7 already specified that implementations of the extension points should be easily testable. On the opposite end, framework developers also need the integration of the extension points into the framework to be easily testable. Low testability normally results in fewer automated tests, which hinders the integration of new features. Therefore, good testability of the prioritization feature also increases the maintainability of coffee4j. Since it is not enough to have easily testable code, this requirement also includes that automated tests should cover the new features of coffee4j.

N10 Use of Established Patterns The existing code in coffee4j follows some patterns or standards. For example, users can either specify all configuration options via the default `@Enable...` annotations or program custom configuration annotations which coffee4j automatically detects. New code should also employ those patterns. This makes maintaining coffee4j and extending it with new algorithms for further research much easier and more consistent.

4.1.2. Combinatorial Testing Process

Section 4.2.2 already presented the old CT process which coffee4j currently implements. Due to the changing requirements this process needs some adjustment. For example, the old process does not consider the integration of additional information into the IPM (Requirements F3, F8, and F19).

Figure 4.1 shows the adjusted process as an *event-driven process chain (EPC)* diagram. The colors differentiate the most important areas of the process. Red is the integration of additional information for prioritization into the IPM, blue the test case prioritization after generation, green the execution with optional fail-fast mode, and yellow the fault characterization. On can divide the process into six basic phases:

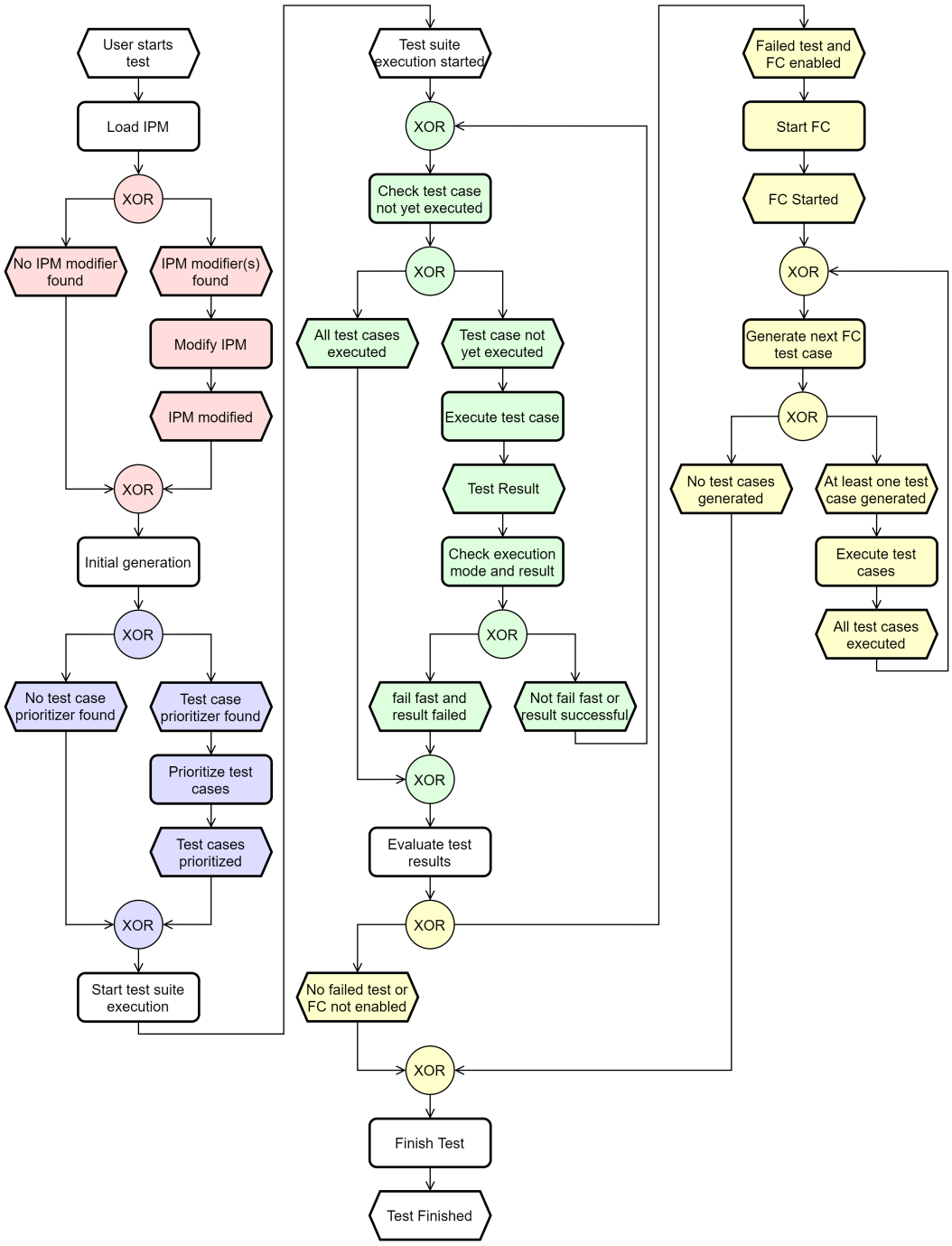

Figure 4.1.: EPC diagram of the combinatorial testing process

1. The first phase begins once the user starts the combinatorial testing process. From this moment on, the combinatorial testing framework takes control and performs the rest of the process automatically. First, it loads the IPM from the location specified by the user just as in the old process definition. The next part is new and based on Requirements F3, F8, F19. It integrates additional information into the IPM and algorithm developers can therefore use it to load necessary information for their prioritization approaches. The process (and later also the architecture) includes this as a general automatic modification of the user-supplied model. Therefore, it is not only possible to adjust parts of the model for prioritization, but rather everything defined in the model such as parameters, their values, and constraints. This makes the extension point general purpose. Implementations can therefore not only use it for prioritization, but also for loading additional parameters or values from external sources like csv files. If the user supplied any IPM modifiers, the process executes them on the loaded model, and every operation afterwards uses this modified model. Otherwise, the model is left as is. At this point the process reports every modification to the user, which is not present in the EPC diagram due to size constraints.

2. After the model has reached its final form, the generation of initial test cases can start. Here, the user has two options. S/he can either specify a normal generator like IPOG as in previous framework versions or use the new functionality specified in Requirement F1 to define a prioritizing one like DDA. Depending on this choice, the result of this step is an un-prioritized test suite, which just guarantees the CT coverage criterion, or a prioritized one, which also orders the test cases according to an generator internal criterion (e.g. value weights).

3. Phase three is a new addition to the process and optionally allows for a test case prioritization algorithm to order the test cases generated in the previous phase (Requirements F2, F7, and F16). The decision of whether the process executes a prioritization algorithm only depends on whether the user configured one, not on whether the generation algorithm already prioritized the test cases. Therefore, it is possible to first generate the test cases in a prioritized way and then also order them according to a different prioritization criterion.

4. After the test cases have been generated and prioritized, the actual execution can start. Here, the important and new part is the fail-fast mode specified by Requirements F10, F11, and F12. Therefore, the process executes each test case individually, for as long as there still are non-executed test cases. Once it knows the test case result, it checks whether the user enabled the fail-fast execution mode and if the test case failed. If this is the case, it directly halts the execution and does no longer execute all remaining test cases. Otherwise, the per test case execution continues as before.

5. The next phase contains the fault characterization. Here, everything works as before. It only runs if there is at least one failed test case, and the user supplied a

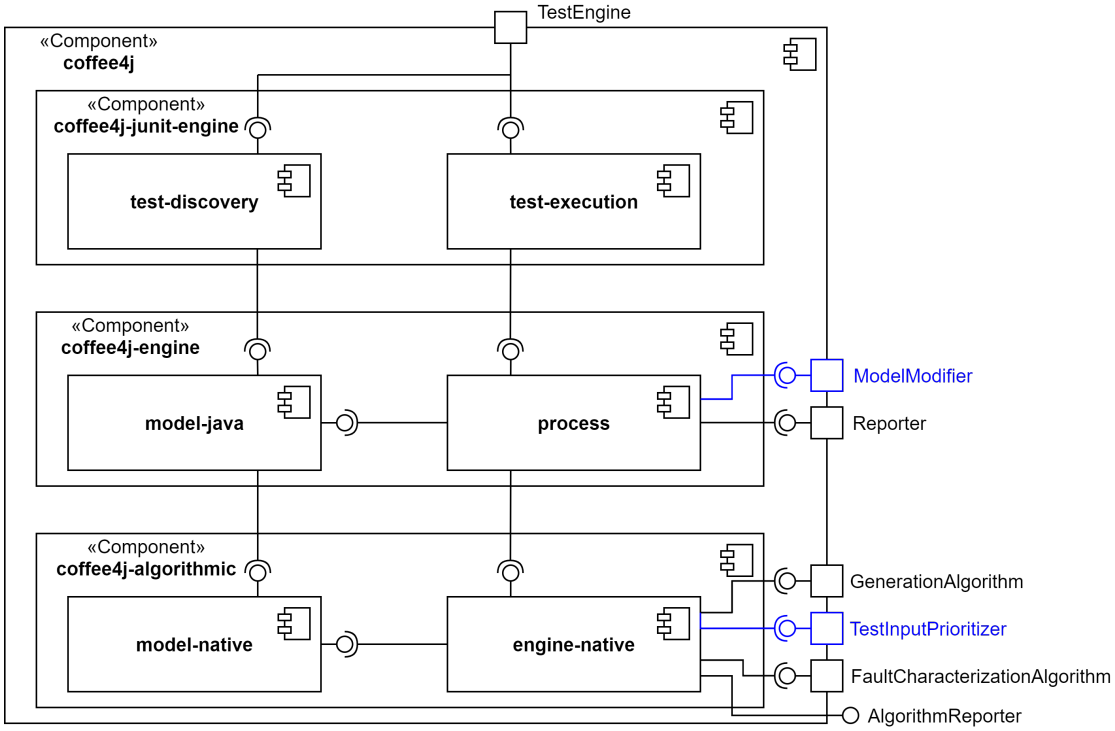

Figure 4.2.: Architecture of coffee4j

fault characterization algorithm to the combinatorial testing framework. Due to Requirement F12, fault characterization is still performed regardless of whether the execution mode is set to fail-fast or not. The fault characterization itself is an iterative process. In each iteration, the FC algorithm gets the execution results of the previously executed set of test cases, and can then decide whether additional test cases are necessary to further narrow down the failure-inducing combination. Once this is no longer the case, the FC phase finishes.

6. Finally, the sixth phase finishes the test. This involves cleanup of the test environment and a final evaluation of the test run. In case of history-based test case prioritization this may involve persisting relevant execution data, like code-coverage statistics, test results, or discovered FICs.

4.1.3. Architecture

In addition to the necessary adjustments to the CT process, the requirements from Section 4.1.1 also necessitate changes to the component architecture, in particular the external extension points. Figure 4.2 shows the adjusted component diagram. The existing extension points stay the same as before. Reporting, fault characterization and initial generation are still necessary in the new CT process. The only change in those

extension points is that the internal representation of the IPM now includes prioritization information like value weights and new features such as mixed-strength testing and (partial) seed test cases. Additionally, prioritizing generation algorithms are now possible, but this does not require any changes in the existing interface. However, there are two new extension points, `ModelModifier` and `TestInputPrioritizer`, to satisfy Requirements F2, F3, F16, and F19.

ModelModifier The `ModelModifier` extension point is the realization of Requirement F19. It consists of one interface which converts a given IPM into a new model. Due to its general nature, it can not only load prioritization information for currently realized prioritization features, like value weights, but any information at all which pertains the modeling of a SUT. For example, it can load completely new parameters and values and integrate them into the IPM. The general extension point also has other advantages with regards to test case prioritization. While most current approaches represent the information in forms of value weights, this could change in the future towards other forms of representation. If the extension point would only allowed for the modification of weights, framework developers would have to adjust it to support those new forms of representation. Alternatively, it would also be possible to add one extension point per model feature. However, this would not allow for extensions which modify multiple parts of the IPM. Therefore, an extension point which lets the user modify every aspect of the IPM is the most general form and also aids in testability and maintainability. It is easy for algorithm developers to test their prioritization approaches since they simply need to test a function which maps from one IPM to another with different examples. With regard to maintainability, the argument from before holds: A general extension point is more likely to still fit the need of future iterations of coffee4j.

An important part of Requirement F19 was that implementations of `ModelModifer` need to be able to track parameters and values across multiple runs of coffee4j. Therefore, the extension point is in the `coffee4j-engine` layer where it has access to parameter names and the concrete values.

TestInputPrioritizer The second new extension point is the `TestInputPrioritizer`, which realizes Requirement F16. It consists of one interface that has a single method which maps form a given IPM and a possibly unordered collection of test cases to a new, ordered list of test cases. The interface name (`TestInput...` and not `TestCase...`) is deliberate since coffee4j calls test cases test inputs to avoid confusion with JUnit test cases.

The interface is again as general as possible. Developers can not only use it to prioritize the initial test suite, but any collection of test cases. Theoretically, this means that fault characterization algorithms could use a `TestInputPrioritizer` to prioritize the FC test cases.

For the prioritization of test cases it is not important to know the actually names of the parameters and their values since all necessary prioritization information is in the IPM. Therefore, the extension point is in the `coffee4j-algorithmic` layer. This

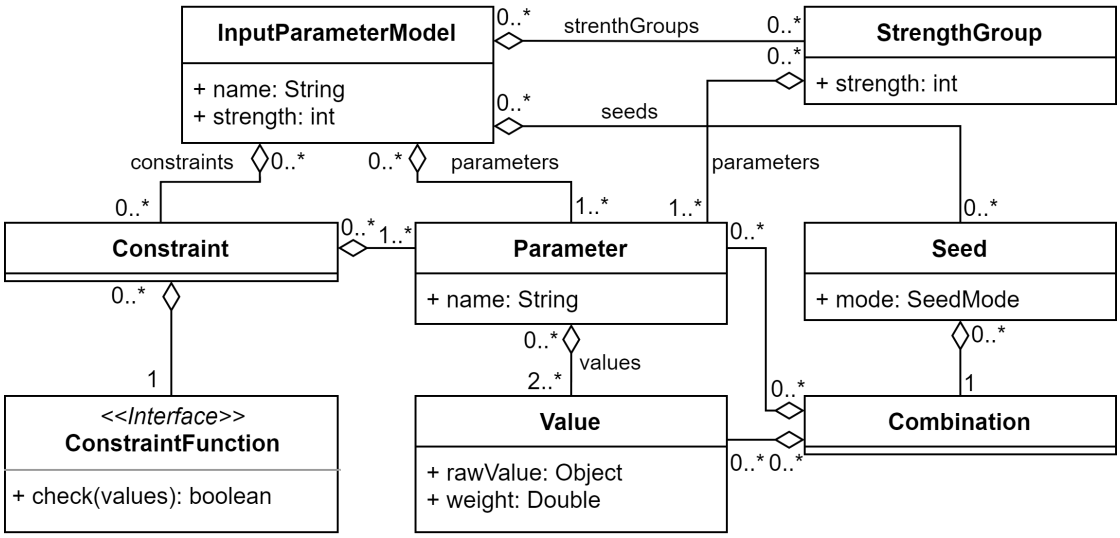

Figure 4.3.: Class diagram of coffee4j's domain model

allows for faster computation based on Java primitive data types instead of actual objects as in the `coffee4j-engine` layer.

4.1.4. Domain Model

Due to the new requirements, mainly F4, F5, and F6, the domain model needed some adjustment. Figure 4.3 shows the new domain model of the IPM.

A large part of the diagram stayed the same as it is in the current version of coffee4j [Bon18]. An `InputParameterModel` still contains a basic testing strength and an arbitrary number of `Constraints` and `Parameters`. Each `Parameter` has two or more `Values`. Together, `Parameters` and `Values` can form a `Combination`, which also represents a test case if it assigns a value to all parameters.

The first change is the added weight in the `Value` class. It is the result of Requirement F4. Since literature has many different ways of using weights, a **`double`** was appropriate so that different prioritization approaches can all work using the same `Value` class. In most approaches, a weight of zero means that the value is not weighted in any special way, positive weights mark important values, and negative weights create soft or hard constraints. Another possible case it that a value does not have any weight at all. In this case, the prioritization algorithm itself needs to decide how to handle those values. To differentiate between "no value" and "value of zero", the weight is not a primitive (**`double`**), but an object (`Double`), which can be **`null`**.

Next, the `StrengthGroup` class deals with Requirement F5 to enable mixed-strength testing. Here, the strength from the `InputParameterModel` represents the basic or default testing strength. It holds for all possible parameter combinations not contained in a `StrengthGroup`. The `StrengthGroup` itself then overwrites the default strength

for specific sets of parameters. It consists of at least one parameter, and a specified overwriting testing strength. Therefore, it can also be considered to be a sub-model of the original `InputParameterModel` since it focuses on some specific parameters with a given testing strength. The combinatorial test suite must cover each `strength`-way combination of parameters in every `StrengthGroup`. For example, if one takes all parameters of the running example (Table 2.1) and there is a strength group with parameters OS, Browser, and Speed at strength two, then the test suite needs to cover all value combinations between the following parameters: {OS, Browser}, {OS, Speed}, {Browser, Speed}, {Ping}. The important part is that the Ping parameter is tested at strength one, so the only requirement for the final test suite is that each value of Ping occurs at least once. Since Ping is not part of the `StrengthGroup`, a test suite does not need to cover all value combinations in {OS, Ping} (although it is allowed).

The last change was the addition of the `Seed` class for Requirement F5. It represents the concept of a (partial) seed test case, where some parameters already have an assigned value. Since the `Combination` already represents the concept of a partial assignment of parameters to value, `Seed` delegates this to an instance of `Combination`. The only other part the `Seed` class contains is a variable for the `SeedMode`. `SeedMode` is an enumeration which has the values `NON_EXCLUSIVE` and `EXCLUSIVE`. This thesis added the concept of different modes for seeds to ease the fault characterization in regression testing scenarios. If two failure-inducing combinations of size $> t$ often occur when testing a SUT, those two combinations should probably be tested in later iterations as well. Since they have size $> t$ the generated test suite may not contain them, therefore one needs to explicitly add it as partial seed test cases. However, if it is possible to combine the two FICs into a single test (for example with (Windows, –, –, 1 KB/s) and (–, Chrome, –, 1 KB/s)) and both fail, many fault characterization algorithms may have a hard time to distinguish between them and report wrong results [AGR19]. Additionally, the failure from the first FIC may mask the second one because it occurs earlier in the execution. Then, the exception for the second FIC may first appear in a test case generated for fault characterization, thus making it even harder for the FC algorithm to find the correct FICs. Therefore, `SeedMode.EXCLUSIVE` guarantees that two partial seeds do not appear in the same test case, while `SeedMode.NON_EXCLUSIVE` makes it possible to combine seeds and therefore possibly make the test suite smaller.

4.2. Failure-based Test Case Prioritization

At this point one now knows how the CT process, coffee4j's architecture, and its domain model need to change to support different types of prioritization approaches. Building onto this knowledge, this section presents a new prioritization approach based on the failure history of individual test cases.

Subsection 4.2.2 will first present the general idea and process. Next, Subsection 4.2.3 explains how one can integrate the prioritization approach into coffee4j using the presented process and architecture changes from Section 4.1. Since the failure-based test case prioritization approach is a very general one, Subsection 4.2.4 then presents a

few concrete ways to use it for calculating value weights so that it is possible to carry out prioritized generation using a general value-weight-based prioritizing generation algorithm.

4.2.1. General Idea

The idea of failure-based combinatorial test case prioritization came from the combination of traditional regression test prioritization and value weight based CT prioritization.

Traditional RTP approaches such as the ones presented in Section 3.1 already use the failure history of individual test cases to decide which ones they tests first [MGS13; JP02; Ali+20]. Those techniques prioritize tests which failed often over a long period of time or in the recent m runs higher than other test cases. This increases the chance of fast failures, and therefore developer productivity.

A straightforward way of combining this approach with CT would be to generate the same combinatorial test suite each time and then change the order of the test cases per run. However, the problem lies in the extension of the combinatorial test suite. If the model always stays the same over multiple runs, a deterministic generation algorithm will indeed always generate the same combinatorial test suite. However, when introducing a change in the model, this does not lead to the deletion of a few irrelevant test case and the addition of some new test cases, but to a completely new test suite. Consequently, one would have to throw away all historical information. Since incremental and iterative development approaches like agile software development may require frequent changes in the IPM, this is unacceptable.

All other combinatorial regression test approaches face the same problem. Therefore, techniques such as the one developed by Qu et al. use a different smallest unit for prioritization — not the test case but an individual value [QC13]. While parameter values may change in practices like agile software development, this usually means that the historical information about the value are no longer important. This is the same as with a traditional test case in RTP. If the tester removes a test case, s/he only needs to remove the historical information about this particular test case from the database as it is no longer relevant. When removing a value from an IPM, its historical information is also no longer needed, but it will not influence the gathered information of other values.

Therefore, a CT prioritization approach based on failing test cases would have to break down the responsibility of a failing test case to its individual values. Values which often appear in failing test cases should have a higher priority than other ones. This is the basic idea of the presented approach for prioritizing combinatorial test cases based on historical data about test failure: calculate the weight of values based on their appearance in older failing test cases.

Fault characterization already does something similar to the presented prioritization approach: it attempts to find individual values or small combinations responsible for test failures. It therefore makes sense to also incorporate the FC part of the CT process and base the value weights not only on failed test cases, but also on identified FICs.

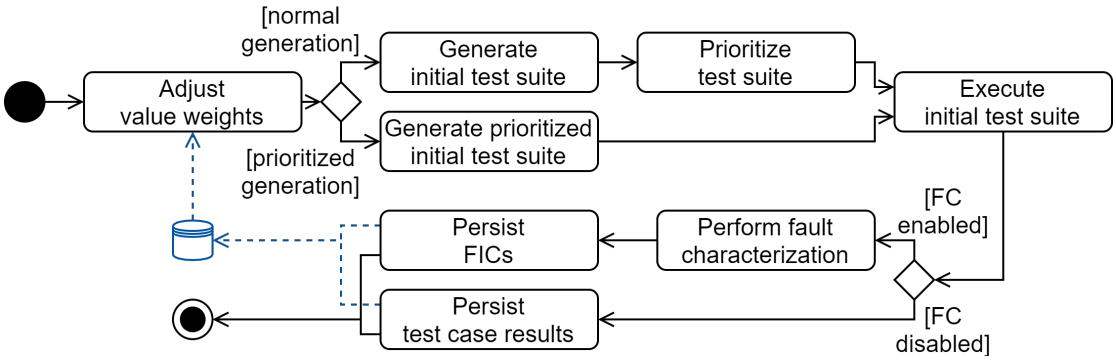

Figure 4.4.: Activity diagram of failure-based combinatorial test prioritization

4.2.2. Process

Figure 4.4 shows the process used for prioritizing combinatorial test cases based on historical failure information. It consists of the following steps:

1. First, it adjusts (or adds) the weights of all values based on the persisted information about failing test cases. There are two general ways to store the information. Either as a list of test cases for each run with the execution result, or as a list of FICs of every run. The step then determines value weights for each previous run. For example, with a very simple way for calculating the weight per run, in a first run five test cases which contained OS=Windows failed (weight=5), and in the second run two failed (weight=2). The overall value weight is seven, the sum of the value weights from each run. Optionally, one can also normalize the value weights for each parameter. Subsection 4.2.4 will explain different techniques for calculating the weights in more detail. The results of the step is an IPM, where the value weight ideally reflect the probability of a value being involved in a failing test case.

2. Next, a generation algorithm generates the initial test cases. There are two general possibilities, as presented in Section 4.1. Either the generation algorithm already considers the value weights, or a prioritizer changes the test case order afterwards. Theoretically, it would also be possible to perform a first prioritization attempt during generation and then prioritize more thoroughly afterwards. Since the presented prioritization approach uses value weights, prioritization algorithms already exist for both scenarios: DDA for prioritized generation, and an unnamed adaption of DDA for later prioritization [QC13].

3. After the initial generation, the process executes the test suite one test case at a time. At this point, it would be possible to use the fail-fast mode of coffee4j (Requirements F10, F11, F12). While this would potentially lead to fewer information about successful and failing test cases, the weight calculation algorithms should work in such a way that they can deal with only partially executed test suites. For

example, the weight calculation cannot assume that every run includes all t-way combinations.

4. Once the process executed all test cases in the initial combinatorial test suite, the next important step is persisting the gathered data for future use. In the presented prioritization approach, there are two scenarios. One can either persist the complete test case results or perform fault characterization after the initial test suite to only persist a reduced number of combinations each run — the failure-inducing ones. While FC does reduce the size of the persisted data, it also introduces a weak link in the prioritization process. Since fault characterization algorithms can sometimes report wrong FICs, this could result in wrong value weights, and therefore also wrong prioritization. Persisting all test results can never be wrong in itself, since it requires no additional computation.

   This problem is also relevant for the other scenario — persisting all test case results. Here, the algorithm could theoretically compute the value weights once after each test run. The value adjustment step at the beginning could then focus on different ways to combine the weights of multiple runs into one weight. However, this splits the computation over multiple extension points and also introduces issues when the tester wants to change the weight calculation approach; If the prioritization approach does not persist the actual results, it cannot recalculate the weights. Therefore, the easiest and most adjustable way of persisting the failure information is just persisting all test case results so computations are possible afterwards. Only in the fault characterization case this is not possible, since it may need to execute additional test cases.

In theory, this process can include an indefinite number of iterations. Each run adds new information to the central data story, thus refining the value weights. In practice, it may however also be necessary to remove old failure-data at some point, since each run could potentially save thousands of test cases and results.

4.2.3. Integration into coffee4j

The presented process already gives a good idea at how to integrate the phases of failure-based prioritization approach into coffee4j. This section will now assign each of them to one extension point of coffee4j — either an old one or a newly introduced one from this thesis — and explain how the extension point enables the function of the process step.

Adjust value weights The adjustment of value weights was the first step in the process and responsible for loading persisted data about the failure-history of a combinatorial test. Based on this data, it calculates weights for each value in the IPM. Therefore, it is a prime candidate for the `ModelModifier` extension point presented in Paragraph "ModelModifier" of Subsection 4.1.3. coffee4j executes this extension point at the beginning of its process, just like the adjustment of value weights is the first step

for failure-based prioritization. Additionally, it allows modification of any part of the user-defined IPM. This includes the modification of value weights.

Consequently, an implementation of the `ModelModifier` extension point can realize the creation of value weights by first loading the persisted data from an external source. Since coffee4j's architecture does not place any requirements on what a `ModelModifier` can do, no other extension point is necessary. Afterwards, the implementation can use the loaded data to calculate a weight for each value, and create a new IPM where everything is the same as in the original except for the value weights. At this point, it is possible to integrate different weight calculation strategies as an extension point of the concrete `ModelModifier` implementation.

Initial test suite generation Generating the initial test suite — either already prioritized or not — is a largely solved problem [QC13; BC05; BC07]. coffee4j already includes an extension point for custom test suite generation called the `GenerationAlgorithm` in Figure 4.2 and Requirement F14. Since the algorithm from Qu et al. already creates prioritized test suites based on value weights, it would also be possible to reuse the existing algorithm instead of creating a new one for this extension point. A new algorithm would simply have to use the value weights which were added by the beforementioned `ModelModifier` and consider them during the generation process.

Test suite prioritization Regardless of which algorithm generates the initial test suite, the prioritization step can always stay the same. In coffee4j, algorithm developers can use the newly introduced `TestInputPrioritizer` extension point for this task. An implementation would get the initial test suite generated by an implementation of the `GenerationAlgorithm` extension point, and the value weights loaded into the IPM using the `ModelModifier` extension point. An implementation of `TestInputPrioritizer` could then reorder the test cases based on the weights of the included values. Additionally, it would also be possible to exchange some values to create test cases with higher overall priority while still maintaining the coverage of all t-way combinations.

Persist results A very important part of the process is persisting information across multiple runs. As Figure 4.4 depicts, this can either be in the form of characterized FICs, or test case results for the complete initial test suite. In each case, one can use the reporting part of coffee4j. Figure 4.2 already showed that coffee4j has a `Reporter` extension point in the `coffee4j-engine` layer [Bon18]. Here, every important event results in the execution of a corresponding callback function. For example, callback functions for starting the initial generation, finishing the execution of a test case, and stopping the execution of initial test cases exist.

The approach which persists all test case results can use these callbacks to first get a list of all test cases in the initial test suite, and then internally save their results onces reported through the appropriate callback. After coffee4j executes the callback signaling the end of the test suite execution, the reporter can persist the internal list of execution results to external sources. Since coffee4j executes this callback regardless of whether it

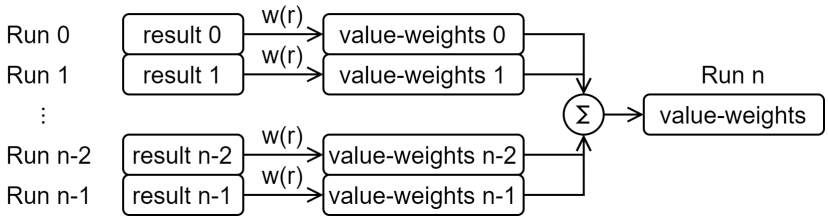

Figure 4.5.: Common framework for value weight calculation methods

executed the test suite completely or just until the first failure, this will always result in a persisted list of all test case results.

In case one chooses the approach which only persists located FICs, one can use another callback function which the framework calls once it finished fault characterization. The reporter can then again persist only the located FICs. As before, coffee4j calls this function regardless of whether the user enabled failure-fast mode or not. All in all, coffee4j's `Reporting` extension point offers all callback functions necessary to persist the information for failure-based combinatorial test case prioritization.

4.2.4. Weight Calculation

The most important part of all weight-based prioritization approaches is how they actually calculate the weights. In case of the failure-based prioritization, this thesis presents three different calculation approaches, two of which work with all test case results and one of which uses FICs.

Common Framework All three approaches share a common calculation framework, which future work could also adjust. Figure 4.5 shows how the calculation across multiple runs works. If the current run is run n, and the data of runs 0 to $n-1$ are available in an external data source, the calculation framework first treats the results of those runs individually. It loads the data for each run, either as test case results or a list of discovered FICs. If no test case failed, the data for a run should be empty. Next, it applies the weighting function $w(r)$ to every run result r individually. That means that the calculated weights only depend on the results of the respective run, not on any other runs. Another requirement for the weighting function $w(r)$ is that the resulting weights must all be zero if r is empty. Consequently, the absence of any data does not prioritize any values above others. In future work one could change this schema so that a weighting function w includes all runs. This would, for example, allow a weighting function to ignore FICs which do not appear more than x times across all runs.

Once the calculation framework knows the results of w for every run, it can combine them to form a single set of value weights which a `ModelModifier` will then integrate into the IPM. Currently, the combination of value weights uses simple addition. Optionally, it is also possible to normalize the weights of every parameter to one. For example, if the weights are Windows=3, Linux=2, MacOS=1, Android=0, iOS=0 for the OS parameter

and 10ms=0, 100ms=2, 1000ms=98 for the Ping parameter, the final weights would then be Windows=$\frac{1}{2}$, Linux=$\frac{1}{3}$, MacOS=$\frac{1}{6}$, Android=0, iOS=0, 10ms=0, 100ms=$\frac{1}{50}$, 1000ms=$\frac{49}{50}$. Future versions of failure-based prioritization could also use different combination approaches, for example weighting recent runs higher than old runs.

Since the basic calculation framework is the same for all presented approaches, the next paragraphs will mostly focus on defining the function $w(r)$ in different ways.

Simple FIC weighting The first weighting technique uses failure-inducing combinations. For this method, the combinatorial test must execute fault characterization in every run, and the user must register a `Reporter` extension point implementation with coffee4j to persist all identified FICs.

The simple weighting schema now adds a weight of one for each value in the failure inducing combination. Consequently, the result of $w(r)$ is a value-weighting function $w_v(v)$ where each value v gets assigned the value $w_v(v) = \sum_{c \in FIC_r} containsValue(c, v)$ where FIC_r is the set of all FICs identified in run r and $containsValue(c, v)$ returns one if combination f contains value v and zero otherwise. For example, consider run r with $FIC_r = \{$(Windows, –, 10 ms , –), (–, Chrome, 10 ms, –), (Linux, –, –, 10 KB/s)$\}$. $w(r)$ has the result $w_v(v)$ where w_v(Windows) = 1, w_v(Linux) = 1, w_v(Chrome) = 1, and w_v(10 KB/s) = 2.

The idea behind this weighting is that values which often appear in failure-inducing combinations should appear earlier in the test suite. During generation or prioritization, the algorithms do not consider value weights independently but instead calculate them up to the unit of a t-way combination (often by multiplication). Therefore, if both Windows and 10ms get a weight higher than zero, the combination (Windows, –, 10 ms , –) will have a higher weight and therefore appear much earlier. If test cases fail repeatedly across multiple runs with the same FIC, this leads to faster failures.

Simple test case result weighting One can also apply the FIC-based approach for calculating value weights to complete test case results. This changes the formula for value weight calculation to $w_v(v) = \sum_{c \in F_r} containsValue(c, v)$, where F_r is the list of all failed test cases in run r. For example, if $F_r = \{$(Windows, Chrome, 10 ms, 10 KB/s), (Windows, Firefox, 100 ms, 10 KB/s)$\}$ resulting weights are w_v(Windows) = 2, w_v(Chrome) = 1, w_v(Firefox) = 1, w_v(10 ms) = 1, w_v(100 ms) = 1, and w_v(10 KB/s) = 2.

Even if only Windows and 10 KB/s were responsible for the test failures (failure-inducing), the other values in the failing test cases would still have positive weights. However, if the generation algorithm always generates different test suite, this will still also prioritize the failure-inducing combinations. While the prioritization may therefore be a little bit worse since it also prioritizes values unrelated to failure, it can also be faster and more reliable, because there is no need for fault characterization. This saves time in the test process, and also avoids problems arising from wrongly identified FICs, which could lead to worse results than overprioritizing values unrelated to the failures.

If the generation algorithm itself does not perform prioritization, and instead prioritization is performed afterwards, this behaves a lot like traditional, non-CT regression

test prioritization. The test suite always contains the same test cases, so the approach prioritizes those which previously lead to failures higher. However, the key different for CT is that the prioritization would still work if the IPM, and therefore the generated test suite changes, since prioritization still uses weights on individual values. One would not have to throw away all historical information after adding one new value.

Suspiciousness-based test case result weighting The last weight calculation approach of this thesis bases its weight calculation on the notation of suspiciousness of a component introduced by the fault characterization algorithm BEN, which Paragraph "BEN" of Subsection 2.1.2 previously presented.

BEN works by assigning a notion of suspiciousness to each value, which represents how likely it is for the value to be in a failure-inducing combination [Gha+12; Gha+15]. It then uses this suspiciousness to generate test cases which include one suspicious (probably failure-inducing) combinations and assign all other parameters to non-suspicious values. A concept for a calculation algorithm could directly use the notation of value suspiciousness to assign a weight to every value. Test suites should then contain t-way combinations which include many suspicious values near the beginning.

It is possible to use all formulas originally developed for BEN without any adjustment. The only difference is that the prioritization approach needs to construct all referenced sets from information collected in the respective previous run. $w(r)$ would then return $w_v(v)$ where $w_v(v) = \rho(o)$, with ρ being the component suspiciousness function, and o the unique component which contains value v.

The advantage of this approach over the previous one is the consideration of successful test cases. Combinations in a failing test case which also appear in successful test cases are not suspicious, and therefore the contained values will receive a lower weight than the one computed by the previous approach. At the same time, this third calculation approach retains the advantage of approach two. It performs no fault characterization since the value weight calculation only uses test case results and not FICs. Additionally, previous studies showed that the notion of suspiciousness works in BEN for real world programs when performing fault characterization [Gha+15]. Since the weight calculation also attempts to find values which are likely to cause failure, it would be sensible for it to work in this case, too.

4.3. Weight-based Prioritization with Constraints

Now that the last sections presented a concept for the integration of CTCP into coffee4j and a new failure-based prioritization approach which uses coffee4j's new architecture, this section will focus on the more low-level task of actually generating combinatorial test suite which prioritize t-way combinations based on value weights.

4.3.1. Algorithm

Section 3.2 in the Related Work chapter already introduced the DDA algorithm for this task. While it does work well for generating prioritized combinatorial test suites, it can only do so if there are no constraints. However, in practice, nearly all relevant SUT have at least some constraints on their input values or environment. For example, even the simple running example of Table 2.1 needs constraints regarding the possible combinations of OS and browser since Safari is not available on Android, Windows, and Linux. As mentioned in Section 3.2, Bryce et al. already extended DDA for forbidden t-way combinations which would receive the weight of -1, this approach is not sufficient if constraints span more than t values [BC06]. Constraints such as `(parameter1=value2 && parameter2=value3) => parameter3!=value1` are not possible if the testing strength t is two. The problem in the algorithm is in multiple places. First, it is not possible to cover all combinations of strength t. This is the case because some of those (like combining Windows and Safari) are simply no longer possible. Additionally, backtracking might be necessary if the algorithm enters a loop in which it always constructs the same test case without covering any remaining t-way combination since the constraints limit the values which cover these combinations. To enable constraint handling, this thesis extends the algorithm in the following way:

```
1  RemainingCombinations = AllCombinationsAtStrengthT;
2  remove all c ∈ ReminingCombinations where !valid(c);
3  startWithLoopBreakingTestCase = false;
4  while (RemainingCombinations is not empty) {
5    if (startWithLoopBreakingTestCase) {
6      initialize nextTestCase with highest uncovered combination;
7    } else {
8      initialize nextTestCase with empty values;
9    }
10
11   Order parameters according to t-way weights;
12   for (parameter = nextHighestFactorWithoutValue) {
13     Compute t-way value interaction weights;
14     select value with highest weight so that valid(nextTestCase);
15   }
16
17   if (nextTestCase covers at least one ReminingCombination) {
18     startWithLoopBreakingTestCase = false;
19     add nextTestCase to test suite;
20     remove covered t-way combinations from RemainingCombinations;
21   } else {
22     if (startWithLoopBreakingTestCase) {
23       failure;
24     } else {
25       startWithLoopBreakingTestCase = true;
26     }
27   }
```

```
28 }
```

Source Code 4.1: ConstraintAwareDeterministicDensityAlgorithm

One can immediately see that constraint handling makes the algorithm much longer and more complicated, mostly due to special handling of loop-breaking test cases. The constraint handling in the constraint-aware DDA variant uses *satisfiability (SAT)* solving. A central part, the `valid(combination)` function, checks whether it is possible to extend a potentially uncompleted test case to a complete test case which does not violate any user-defined constraints. This is in line with coffee4j's general constraint handling, so that it is easy to integrate the algorithm. Even when using it outside of coffee4j, this should not be difficult to accomplish since many SAT solvers exists to which the algorithm can delegate this task. The algorithm itself can therefore focus on building a test suite, and not on checking whether certain value assignments make a test case impossible.

The adjusted DDA variant starts exactly the same as before: it constructs a set containing all possible t-way combinations to notify the algorithm once the test suite reached the combinatorial coverage criterion. Since constraints can make some t-value combinations impossible, the algorithm needs to remove invalid combinations — those which can never appear in a valid test case — beforehand. Otherwise the algorithm would enter an infinite loop since it tries to cover combinations which are never valid.

Additionally, this variant introduces a new variable called `startWithLoopBreaking-TestCase`. This is necessary due to the greedy nature of DDA. With constraints, it can sometimes happen that the algorithm creates a new test case which does not cover any uncovered t-value combination. Since DDA is deterministic, it will then generate the same test case over and over again since the starting conditions stay the same. The main reason for the necessary introduction of this loop-breaking variable is ordering the parameters according to the combined weight of all uncovered t-value combinations, where the weight of one combination is the product of its value weights. It can — speaking from experience — happen that, for example with testing strength two, there is no uncovered two-way combination between the two parameters with the highest weight. They simply have a high weight because both have many uncovered combinations with other parameters. In this case, DDA will select the two values with the highest individual weight. Since the two parameters have no uncovered combination, this creates a partial test case that does not yet remove any remaining combination. Without constraints this would be no problem, since the algorithm would set the last two parameters which still have uncovered combinations in such a way that the next test case covers at least one remaining combination. However, if constraints forbid such a value assignment based on the first two assigned values, it is possible to create a test case which does not cover any combination. This makes loop-breaking necessary and is the reason for the new variable to keep track of whether the last test case covered any remaining combination.

If the variable is set to **`true`**, lines five to nine of the code above will initialize the new test case with the highest still uncovered combination. This guarantees that a test case covers at least one remaining combination, and the algorithm must therefore terminate

since only a finite number of t-way combinations exists. If the variable is `false`, the algorithm instead initializes the next test case with an empty combination as in the normal DDA algorithm.

Next, the loop for assigning a value to each currently uncovered parameter of the next test case starts. This works the same as before — the algorithm orders the parameters by their weights, which it calculates as the sum of the weights of all still uncovered combinations which include the relevant parameter. Inside the loop, special attention has to be paid to the assignment of the next value. Here, it is only possible to assign a valid value that does not lead to a test case which violates a constraint. Therefore, the algorithm will not necessarily assign the value with the highest weight, but instead the value with the highest weight which does not lead to an invalid test case. Again, it calculates the weight by summing up uncovered combinations, but in the case of value weights it only considers the uncovered combinations which contain the relevant value and are compatible with the current test case, i.e. one can add the combination to the test case under construction.

After the next test case is complete, the next step is to check whether the test case covers any of the remaining combinations. If this is the case, one can proceed as before and add it to the test suite and remove all covered combinations, otherwise line 25 of the code above sets the `startWithLoopBreakingTestCase` variable to `true` so that the algorithm does not enter an infinite loop. In that case the algorithm also does not add the next test case to the test suite since it offers no advantage. If the algorithm generates two test cases that do not cover any of the remaining combinations directly after one another, something is very wrong — either in the algorithm implementation or the SAT checker – and it terminates with a failure.

All in all, this thesis adjusts DDA in such a way that it will never create a test case which is not valid, always terminates, covers all valid t-way combinations, and still prioritizes the ones which have a high value weight. Due to the necessary loop-breaking and other parts of the constraint handling, it may however sometimes happen that test suites are larger, and sometimes combinations with high priority may appear later than expected due to constraints.

4.3.2. Example

The previous section contained a very abstract description of the constraint-aware DDA variant. To make some concepts clearer, this section will go through a small example which requires a loop-breaking case. Since they are only necessary under very few circumstances, the example does not use the IPM from the running example introduced in Table 2.1, but instead uses a custom one from Table 4.1. It consists of five parameters: Dot, Star, Exact, Plus, and Range. Dot and Star each have the values **M**iddle and **E**nd while parameter Exact also includes a **S**tart value. Parameters Plus and Range only contain the **S**tart and **M**iddle value. For a shorter notation, all tables abbreviate values by their first character. All values have a weight which the table shows in parentheses behind the respective value. The given constraints ensure that at most one parameter can have value End or Start at a time, so that (Middle, Middle, Middle, Middle, Middle)

Name	Values			Constraints
Dot	M (0.6)	E (0.4)		Star=E ⇒ Dot≠E, Exact=E ⇒ Dot≠E
Star	M (0.8)	E (0.2)		Dot=E ⇒ Star≠E, Exact=E ⇒ Star≠E
Exact	S (0.6)	E (0.3)	M (0.1)	
Plus	S (0.75)	M (0.25)		Exact=S ⇒ Plus≠S, Range=S ⇒ Plus≠S
Range	S (0.3)	M (0.7)		Exact=S ⇒ Range≠S, Plus=S ⇒ Range≠S

Table 4.1.: IPM for the DDA loop-breaking example

Name	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
Dot	M	E	M	M	E	M	M	M	E
Star	M	M	E	M	M	E	E	M	M
Exact	E	S	S	M	M	M	M	E	M
Plus	S	M	M	M	S	M	S	M	M
Range	M	M	M	S	M	S	M	S	S

Table 4.2.: Test cases (TC) generated by DDA

and (Middle, End, Middle, Middle, Start) are examples of valid test cases while (End, Middle, End, Start, Middle) is not valid since it contains End two times.

This is very reduced and slightly modified version of a model by Ghandehari et al. for testing a program which parses regular expressions [Gha+13]. Each parameter represents one feature in the regular expression language and their values decide the position in the generated input string. For example, one could translate (Middle, End, Middle, Middle, Start) with `[a-e].b{5}z+a*`. This example only reduces the model by removing some parameters and values so that the number of test case which CT generates becomes manageable for a small example.

The rest of this example assumes a testing strength of two, and at every place in the algorithm where two weights are the same, it selects the first parameter/value as ordered in the IPM. For example, if parameters Exact and Dot have the same weight, the algorithm will select parameter Dot first. Similarly, in case of a loop-breaking test case, the combinations are first prioritized by their weight, and then by their order. With those additions, the algorithm becomes completely deterministic.

When retracing the example from the next few paragraphs, one needs to keep track of all two-way combinations to be covered, their weights, and whether they have already been covered in a previous test case. To this end, the Appendix includes Table A.1, which presents all of those two-way combinations and the necessary information.

In a first step, DDA creates the list of all possible two-way combinations. Next it removes combinations like (End, End, –, –, –) or (–, –, Start, –, Start) which a test suite cannot possibly cover because of constraints, resulting in a set of 42 combinations.

Additionally, it initializes `startWithLoopBreakingTestCase` to **`false`**. The next few steps will now go through the algorithm for some important test cases. For an overview over all test cases, consult Table 4.2.

TC1 The initial weights of the parameters are (3.8, 3.86, 3.19, 3.325, 3.595). Those numbers are the results of calculating the sum of all uncovered and valid combinations which contain the respective parameter as described in Section 3.2. DDA now orders the parameters according to their weight — Star, Dot, Range, Plus, Exact — and assigns them a value in this exact order. Since the value of variable `startWithLoopBreakingTestCase` is **`false`**, the initial next test case is an empty combination which contains no value. For the first parameter, Star, the calculated remaining weights of its values are 3.2 for value Middle and 0.66 for value End. One can calculate those numbers by summing up all uncovered and valid combinations from Table A.1 where the value of Star is Middle or End respectively. Possible explanations for the concrete weights are the constraints on value End which makes the number of valid two-way combinations smaller, and the fact that value Middle has a higher weight. Since selecting value Middle does not violate any constraint, the algorithm will select it and set it in the next test case. Next, parameter Dot needs a value. Again, the algorithms calculates the value weights, but since the test case already assigns Star to Middle, it only counts those remaining combinations which allow Star to be Middle. For example, the combination (Middle, End, –, –, –) does not contribute to the weight, however (Middle, Middle, –, –, –) and (Middle, –, Middle, –, –) do count since they both allow for Star to be Middle. The resulting weights are 2.28 for value Middle and 1.4 for value End, so the algorithm selects value Middle again, making our partial test case (Middle, Middle, –, –, –). After one applies the process to all parameters, the resulting test case is (Middle, Middle, End, Start, Middle). Since this covers at least on previously uncovered combination, the algorithm adds it to the test suite and removes all covered two-way combinations from the internal list. In Table A.1 one can see this in those rows where "Covering test cases" column contains a "1".

TC3 Test case three is the next interesting test case. The parameter weights are (1.33, 0.98, 1.045, 0.9, 1.005), and therefore the order is Dot, Exact, Range, Star, Plus. First, Dot gets value Middle and Exact value Start based on their weights. Consequently, the current test case is (Middle, –, Start, –, –) when starting to calculate the value weights for parameter Range. However, even though value Start has weight 0.555 and value Middle only has a weight of 0.14, the algorithm still selects value Start. This is due to the constraint in line 14 of the algorithm, where it states that a value selection must lead to a valid test case. Selecting Start for Range would violate constraint Exact=Start $\Rightarrow$ Range$\neq$Start, and therefore, Middle is the highest weighted valid value. All other parameter assignments are possible by selecting the value with the highest weight, leading to a test case of (Middle, End, Start, Middle, Middle).

TC7 After the algorithm has already computed six test cases, the first loop-breaking test

case is necessary. However, the algorithm first tries to generate test case seven the normal way, since it cannot know that loop-breaking is necessary without generate a test case to prove it. At first, it orders all parameters according to their weight. In this iteration, those are (0.12, 0.15, 0.165, 0.225, 0.21), resulting in the order Plus, Range, Exact, Star, Dot. When looking at Table A.1 one can see that these weights are the result of only four remaining combinations: (End, –, –, –, Start), (–, End, –, Start, –), (–, –, End, Middle, –), and (–, –, End, –, Start).

Now, the assignment of values to parameters in the given order begins. For parameter Plus, the only uncovered combinations are (–, End, –, Start, –) and (–, –, End, Middle, –), leading to the selection of value Start since the first combination has a higher weight ($0.15 > 0.075$). Next, (End, –, –, –, Start) and (–, –, End, –, Start) are the only remaining combinations for parameter Range. Since they only contain value Start, it has a higher weight than value Middle. However, the current test case already assigns value Start to parameter Plus, which makes it impossible to select value Start since the constraints only allow for one value to be Start. Therefore, the algorithm needs to select value Middle even tough it has a weight of zero. For parameter Exact, both remaining combinations (–, –, End, Middle, –) and (–, –, End, –, Start) are already impossible, so all values have a weight of zero. Since our variant of DDA takes the first value which creates a valid test case, it selected value End for Exact. As a result, the next test case is currently (–, –, End, Start, Middle). The constraints then only allow for Dot and Start to be middle, since Exact is already End, resulting in test case (Middle, Middle, End, Start, Middle) which does not cover any of the beforementioned remaining combinations. Since the algorithm is deterministic and no base conditions have been changed, it would always generate the same test case and therefore get stuck in an infinite loop.

At this point the loop-breaking test case comes into play, and line 25 sets the corresponding variable `startWithLoopBreakingTestCase` to **`true`**. Therefore, the algorithm does not add the generated test case to the test suite, and the next iteration will start with a test case which covers at least one remaining combination. Since the algorithm selects the remaining combination with the highest weight, it selects (–, End, –, Start, –) and then fills all parameters without a value assignment according to the normal procedure. At the end, test case seven is (Middle, End, Middle, Start, Middle), and the algorithm can continue as normal.

Those were only three example test cases for the three important cases: generation without a selection driven by constraints, generation with some value selections driven by constraints, and loop-breaking test cases. To get a better understanding of the algorithm, one can go through it one test case at a time, or debug this very same example in coffee4j.

5. Realization

Contents

Chapter 4 introduced the basic concepts of combinatorial test case prioritization and presented an integration of different prioritization approaches into coffee4j's architecture and process model. This chapter focuses on transforming the abstract architecture changes into concrete implementation changes. Like Chapter 4, this chapter starts with the general implementation of the necessary extension points in coffee4j with Section 5.1 and then moves towards progressively more specific topics. Section 5.2 explains how this implementation helps to realize the concept of failure-based test case prioritization and how users can easily configure it in the test description. Finally, Section 5.3 describes certain implementation choices for the constraint-aware DDA variant. Algorithm developers could then use those last two sections as a basis for designing other prioritization approaches and integrating them into coffee4j.

This chapter often refers to the new version of coffee4j which realizes all prioritization features. It is located in repository `https://git.rwth-aachen.de/joshua.bonn/thesis-evaluation`, and all implementations refer to the state of the repository at commit `7cc0f68a8be57b9dc8d81296d7d8ba62520ec69c`.

5.1. Framework

The basis for all CTCP approaches is the adjustment of coffee4j's process model and architecture. For this, Section 4.1.1 introduced several functional and non-functional requirements, which the concepts presented in the following sections partly realized. However, while the developed concepts are a good step towards comparing different prioritization approaches, an important missing part is the adjustment of coffee4j's actual Java implementation.

This section will therefore transform the abstract concepts into concrete changes in coffee4j's implementation. Subsection 5.1.1 starts by integrating all changes in the CT process from Subsection 4.1.2 into coffee4j's `coffee4j-engine` layer, where the rest of the process definition resides. Next, Subsection 5.1.2 presents all new extension points and how they can help with realizing different kinds of prioritization approaches. Subsection 5.1.3 realizes changes to the domain model in all layers. Finally, Subsection 5.1.4 then finishes this section by showing how end users can configure coffee4j's new features and therefore use them for combinatorial test case prioritization. This includes configuration on the API and JUnit level.

5.1.1. Combinatorial Test Process

Subsection 4.1.2 introduced an extended CT process which includes all necessary steps for currently known prioritization approaches. Figure 4.1 depicts this process. It consists of four main parts: adjusting the IPM, generating prioritized test cases, executing them — possibly in a fail-fast mode — and performing fault characterization if necessary.

coffee4j models the CT process using phases [Ber19]. Each phase has a defined input and output type, and requires a context containing additional information necessary for executing a phase. For example, the phase for executing test cases is called the `ExecutionPhase` and takes a list of combinations as an input, and returns a map from those combinations to their respective test results as an output. Its `ExecutionContext` contains additional information like the IPM, callback methods for individual test results, and the actual test method.

Before this thesis, there were three phases: One for generating an initial test suite, a second one for executing the generated test cases, and the fault characterization phase for discovering FICs [Ber19]. This thesis adjusts two of thoses phases, and adds a fourth one to implement the process from Figure 4.1. The next few points will discuss each phase and present all changes:

1. **`ModelModificationPhase`**: This first phase did not exist before this thesis and handles everything concerning the modification of IPMs. Similar to a basic implementation of the `ModelModifier` extension point, the phase has the same input and output, `InputParameterModel`. Its context, the `ModelModificationContext`, only includes all configured `ModelModifiers` and a reporter for model modifications. During execution, the phase checks whether any modifier has been registered. If this is the case, it executes all registered modifiers in the given order, with the output of modifier n as the input of modifier $n + 1$, and it returns the result of the last modifier as the IPM for all other phases. Otherwise, it simply returns the given IPM. For every modification made the phase calls the corresponding reporting endpoint to that any user can keep track of all changes. All in all, the `ModelModificationPhase` realizes Requirements F3, F7, and F9.

2. **`SequentialGenerationPhase`**: Next comes the phase which includes the generation of the initial test suite. This phase already existed in previous versions

of the framework, but this version modifies it to support different prioritization approaches. Most of the phase definition can stay the same. It still requires an `InputParameterModel` as an input and returns a list of test cases as an output. However, the context needs to change to include an optional implementation of the `TestInputPrioritizer` endpoint. If such an implementation is present during the execution, the `SequentialGenerationPhase` executes it after generating the initial test suite as before. It then returns the prioritized list instead of the initially generated one. If no implementation is present, everything stays the same and it returns the initially generated list of test cases. This realizes Requirements F2 and F7. Contrary to the `ModelModificationPhase`, the generation phase does not call any new reporting endpoints so that the user does not know whether prioritization occurred inside a generation or dedicated prioritization algorithm.

3. **`ExecutionPhase`**: The next part of the process diagram from Figure 4.1 deals with test suite execution. coffee4j realizes it in the `ExecutionPhase` which maps the generated list of test cases to a map from test cases to their respective results. To support the execution of test cases, `ExecutionPhase` needs some contextual information. In previous versions, this included reporters, lifecycle callbacks, and the actual test methods to get test results for arbitrary test cases. While these information are still necessary, Requirements F10, F11, and F12 introduced an additional feature for the execution part: the fail-fast mode. Therefore, the `ExecutionPhase` now also requires an `ExecutionMode`, which has a default of `EXECUTE_ALL`. In this case, coffee4j executes all test cases in the list in the given order regardless of whether one test case fails. However, if a user specifies the mode `FAIL_FAST`, the execution stops directly after the first test case failure, and the phase returns all test results — including the failed one — as its output. The code calling our `ExecutionPhase` does therefore not know whether only a part, or all of the test cases were executed except if it compares the test cases in the result set with the input set. However, this is not necessary, since the EPC diagram clearly states that the later parts of the CT process do not depend on this information.

4. **`FaultCharacterizationPhase`**: Requirement F12 even explicitly states that coffee4j should perform fault characterization regardless of whether the failing test case occurred in fail-fast, or the normal execution mode. Therefore, no changes to the `FaultCharacterizationPhase` were necessary.

All in all, those phases preserve the conditions set in Bernwald's thesis: it is possible to chain them via their inputs and outputs to form a complete CT process [Ber19]. Figure 5.1 shows how the output of one phase is always the input of the next one. Therefore, the so called `ProcessManager` can incorporate the new `ModelModificationPhase` into coffee4j's process by executing it once before the generation phase. All other phases stay the same, at least from an outside perspective, and only require changes in their configuration. This validates Bernwald's design in that developers can change the underlying functionality of the phases without big adjustments in the code which uses them [Ber19].

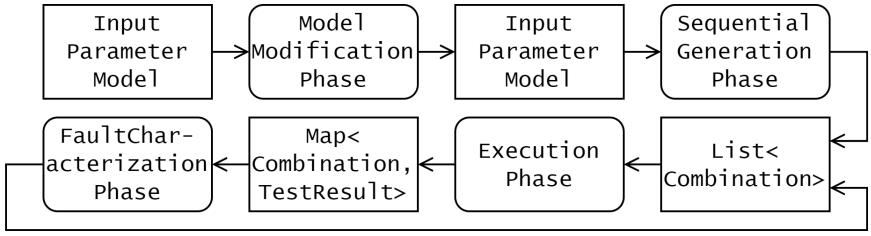

Figure 5.1.: Inputs and outputs of coffee4j's phases

5.1.2. Architecture

With regards to the actual architecture, Subsection 4.1.3 introduced two additional extension points. coffee4j's implementation realizes both of them as a single interface, which makes creating an additional implementation relatively easy. The next paragraphs will explain the semantics and contracts of the interface more in depth.

ModelModifier The first extension point added for this thesis in the `ModelModifier`. As explained in Paragraph "ModelModifier" of Subsection 4.1.3, algorithm developers can use this extension point to add external information to the IPM, for example weights to control prioritization algorithms.

```
1 public interface ModelModifier extends Extension {
2
3   InputParameterModel modify(InputParameterModel original);
4
5 }
```

Source Code 5.1: ModelModifier.java

To this end, it provides a very small interface with just one method. The framework calls it to modify some original `InputParameterModel` according to the semantics of the class implementing the interface. For example, an `AddParametersFromCsv-ModelModifier` would always add parameters from a csv file to the given model. An important point of the interface is the requirement for statelessness. The modification of one model should not depend on whether the same instance of the `ModelModifier` previously modified some other model. As a consequence, testing those implementations also becomes much easier. One can just regard a `ModelModifier` as a function which maps one IPM to another one. While statelessness is a hard requirement, determinism is not explicitly stated but should be upheld if possible. For example, implementations which add weights to a model might calculate these weights in a nondeterministic way, or even assign random weights in the beginning. This must always be possible.

coffee4j will always execute implementations of `ModelModifier` one after another in the `ModelModificationPhase`. Therefore, the framework guarantees that it will always call modifier n with the result of modifier $n-1$ and will again pass its result

to modifier $n + 1$. This guaranteed execution order might later be relevant if, for example, a first modifier adds parameters to a model and the next one loads weighting information. Only the stated execution order would make sense, as otherwise the values of the additional parameters would not have weights.

Additionally, coffee4j guarantees that it will only call the `modify` method with a non-`null` value, as long as each modifier never returns `null`. Instead, if an exception which the `ModelModifier` cannot handle occurs, it should throw an exception or, if the information added by the modifier is not critically important, return the original model.

An important part of the interface definition is also the fact that it extends the `Extension` marker interface added by Bernwald. In general, an extension in coffee4j is a callback in the `coffee4j-engine` layer that the framework calls at defined extension points in the process phases and for which a user can optionally supply implementations. This allows the coffee4j's JUnit layer to discover modifiers by declaring them with the general `@EnableExtension(xy.class)` annotation.

TestInputPrioritizer The second extension point is the `TestInputPrioritizer` from Paragraph "TestInputPrioritizer" of Subsection 4.1.3. It realizes the requirement of an additional algorithm which is only responsible for ordering test cases, and not generating them. Just as the `ModelModifier`, coffee4j realizes the `TestInputPrioritizer` extension point as a small Java interface.

```
1 public interface TestInputPrioritizer {
2
3   List<int[]> prioritize(
4     Collection<int[]> testCases, TestModel model);
5
6 }
```

Source Code 5.2: TestInputPrioritizer.java

The interface has only one method, which is responsible for prioritizing a collection of test cases using information provided by a `TestModel`. One can clearly see that this extension point is in the `coffee4j-algorithmic` and not the `coffee4j-engine` layer such as `ModelModifier`, since it uses `int[]` to represent a test case, and the `TestModel` class, which is the low-level equivalent of `InputParameterModel`. The following subsections will explain the `TestModel` more in depth. Just as the `ModelModifier` interface, implementations of this one have to be stateless, and should try to be deterministic. This means that a `TestInputPrioritizer` should not attempt to learn new information across multiple runs on its own, but instead delegate this task either to an external service, or to the `ModelModifier`, depending on the prioritization approach.

coffee4j only allows for one `TestInputPrioritizer` implementation per run. This is done to avoid the complex task of prioritizing a test suite according to multiple different criteria in coffee4j itself. Should this be an requirement, the extension developer

must implement a custom `TestInputPrioritizer` which takes multiple criteria into account and register it as the sole prioritizer implementation.

Of course, coffee4j makes some guarantees regarding the context in which is uses a registered `TestInputPrioritizer`. The test cases passed to a prioritizer will always be the direct output of a generation algorithm, and not changed by coffee4j. However, since the test cases may not have a clearly defined order after generation, coffee4j passes them as a general collection and not as a list of test cases. As before, coffee4j also guarantees that none of the parameters will ever be `null`. Instead, the collection of test cases can be at most empty. At the same time, all implementations of the `TestInputPrioritizer` interface must guarantee that they will always either throw an exception, or return a list which contains exactly the same test cases as given in the argument. This explicitly forbids modification of the test cases.

Since `TestInputPrioritizer` is not in the `coffee4j-engine` layer but in the `coffee4j-algorithmic` one, it does not extend the `Extension` interface but must instead use a custom registration mechanism. The "Usage" subsection will explain that in more detail.

5.1.3. Domain Model

As explained in Subsection 4.1.4, the domain model of coffee4j also needed some adjustment. In particular, Requirements F4, F5, and F6 mandated the addition of value weights, mixed strength testing, and seed test cases. To this end, Figure 4.3 adds the `StrengthGroup` class, `Seed` class, and the weight property to the `Value` class. However, the realization of the new modeling features is not as straight forward as adding a few classes.

`coffee4j-engine` Layer In particular, negative testing makes everything a bit harder. With negative testing, coffee4j not only generates all t-way combinations which lead to successful SUT behavior but also some which result in exceptional behavior [Bon18; FL19]. For example, in a calculator program, the tests should check that a division by zero produces the correct error message. The normal, positive test cases must not include this corner case since division by zero should be forbidden by a constraint. To avoid extra test cases for those error cases, coffee4j allows user to model them explicitly as constraints. In addition to normal *exclusion* constraints which can never occur, like testing Safari on Windows, one can also specify *error* constraints which mark part of the input space as invalid for normal testing, but testable with special negative test cases. In the example from before, $divident \neq 0$ could be such an error constraint. For positive testing, coffee4j treats error constraints the same way as normal constraints. With negative testing, it instead generates additional error test cases by inverting one error constraint at the time and using this inverted constraint to ensure that the additional test cases always violate the error constraint, i.e. will always lead to the expected error.

While negative testing is generally not part of this thesis, one has to consider it for seed test cases. For example, it could be that one wants to specify seed test cases for positive

and negative testing. Therefore, coffee4j allows the user to specify seeds for positive testing, and for every error constraint. The framework then uses the second kind of seeds when it inverts the respective error constraint. Therefore, the `InputParameterModel` class contains two new fields: `positiveSeeds`, which contains the seeds for positive test cases, and `negativeSeeds`, which is a map from unique error constraint names to a list of seeds.

coffee4j's `coffee4j-engine` layer does not consider the two other features, mixed strength testing and weights, for negative testing. This is the case because constraints cannot invalidate weights and mixed strength groups. The rest of the class diagram in Figure 4.3 depicts exactly how coffee4j actually implements the classes, with the exception that fields are generally private and have dedicated getters.

`coffee4j-algorithmic` Layer Due to the split between the `engine` and `algorithmic` layer, one needs to do every change with the IPM twice. Once for the actual `InputParameterModel` class which the user directly employs to define the test model, and once for the `TestModel` class with which algorithm developers deal when implementing their algorithms.

Before this thesis, there were only two classes which together modeled the IPM in the algorithmic layer: The `TestModel` class containing positive and negative testing strength and all parameters and values as indices, and the `TupleList` class which models constraints as a list of forbidden tuples (`int[]`) [Bon18]. An actual algorithm would therefore always have to know whether it generated test cases for positive or negative testing. Based on this fact, it needed to access different constraints. In most algorithm implementations, this led to the introduction of a custom abstraction layer, so that the actual algorithm implementation only depended on the abstraction, and not on the actual `TestModel`. Therefore, it no longer had to care whether it generates positive or negative test cases, as the abstraction did not make any differentiation.

With seed test cases and mixed group strength, this thesis introduced two additional factors which could be different depending on whether the algorithm was generating a positive or negative test suite. The seed test cases could be different because the user is able to specify them for specific negative test groups (via the error constraints), and the mixed strength groups differ because coffee4j automatically generates a strength group for the parameters involved in the respective error constraint. This thesis therefore makes `TestModel` a general interface which is useful for positive and negative testing, so that the algorithms can directly work on the necessary abstraction layer. As an added bonus, it is now very easy to take an algorithm originally developed for positive CT, and directly use it for negative CT without adjustment.

Figure 5.2 shows the new way `coffee4j-algorithmic` implements `TestModel`. All algorithms like the prioritization or generation algorithm should only depend on the `TestModel` interface. It defines what a model includes via declared getter methods. For example, the `getDefaultTestingStrength` method returns the default testing strength, and the algorithm does not have to concern itself with whether the result value actually comes from the `positiveTestingStrength` or `negativeTestingStrength`.

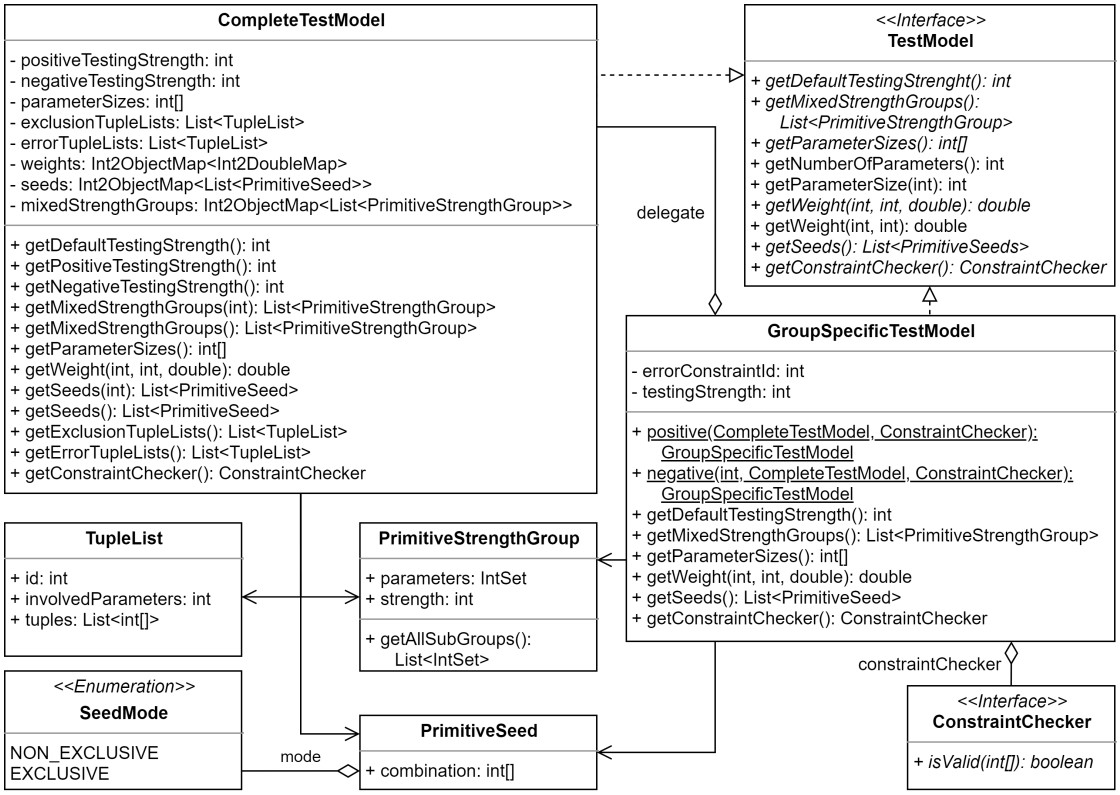

Figure 5.2.: Class diagram of `coffee4j-algorithmic`'s domain model

In addition to some abstract methods which concrete implementations have to implement, `TestModel` also includes default methods for often used requests like `getNumberOfParameters` or `getParameterSize` which depend on the implementation of `getParameterSizes`.

Currently, there are two concrete implementations of `TestModel`: `CompleteTestModel` and `GroupSpecificTestModel`. The first one contains everything which previously was part of the `TestModel` class. Therefore, it contains the testing strength, seeds, mixed strength groups for both positive and negative CT, and all constraints (in form of `TupleLists`). It internally realizes seeds and mixed strength groups with a map from an ID to a list of the concrete type, i.e. `PrimitiveStrengthGroup` or `PrimitiveSeed`. Per convention, the key in this map is either the ID of the corresponding `TupleList` if it is for negative tests, or `-1` if it is for positive tests. For example, if the model has one error constraint (a constraint for negative testing) with the ID "1", then the map $\{-1 \mapsto [[0,1],[0,0]], 1 \mapsto [[0,2]]\}$ means that the test case $[0,2]$ should be a seed test case when generating a test suite for the error constraint, and both $[0,1]$ and $[0,0]$ are seeds for positive test cases. `CompleteTestModel` then realizes the methods defined in `TestModel` for positive testing. This means that `CompleteTestModel.getDefaultTestingStrength()` will always return the pos-

itive testing strength.

The second implementation of `TestModel`, `GroupSpecificTestModel`, realizes the interface's function for one specific group ID which can either point to the positive test cases if it is "-1", or to the error constraint for negative testing. It will then delegate all calls to an instance of `CompleteTestModel`. For example, if one creates an `GroupSpecificTestModel` with an `errorConstraintId` of "1", then a call to `GroupSpecificTestModel.getSeeds()` delegates to `delegate.getSeeds(1)`. All in all the group specific model therefore realizes a `TestModel` view on the `CompleteTestModel` for one specific error constraint.

The remaining classes, `PrimitiveStrengthGroup` and `PrimitiveSeed` are mirrors of the classes `StrengthGroup` and `Seed`. The only difference is that they use primitive types. coffee4j can therefore translate a `InputParameterModel` directly to a `CompleteTestModel` and internally create group specific views of this class which still adhere to the `TestModel` interface to support both positive and negative testing with the same algorithms. All algorithms can therefore use the prioritization features like seeds and mixed strength groups without caring about the actual implementation. In future work one could even make the value weights different for negative testing, and all algorithms would continue to work without any modification, since one only needs to change the `GroupSpecificTestModel`.

5.1.4. Usage

The last three subsections explained how coffee4j implements the process, extension points, and adjusted domain model. This already gives a good idea as to how algorithm/extension developers can use coffee4j to test and compare new prioritization approaches. However, it is not yet clear how a user can actually use these new features. To this end, this section will explain how one can use the new model features and configure the implementations of the extension points with the API on the `coffee4j-engine` layer and the more user-friendly `coffee4j-junit-engine` layer. This will cover Requirements F1, F2, F3, F4, F5, F6, F10, F17, F20, N1, N2, N3, N4, and N6.

Input Parameter Model

```
1  inputParameterModel("BrowserGame")
2    .positiveTestingStrength(2)
3    .parameters(
4      parameter("OS").values(
5        "Windows", "Linux", "MacOS", "Android", "iOS"),
6      parameter("Browser").values(
7        "Chrome", "Edge", "Firefox", "Safari"),
8      parameter("Ping").values(10, 100, weighted(1000, 0.5)),
9      parameter("Speed").values(1, 10, 100, 1000))
10   .errorConstraint(constrain("OS", "Browser")
11     .withName("Safari on Windows")
12     .by((String os, String browser) ->
```

```
13        !(os.equals("Windows") && browser.equals("Safari"))))
14    .seed(
15      seed(entry("OS", "iOS"), entry("Ping", 10), entry("Speed", 1)))
16    .seed("Safari on Windows",
17      seed(entry("Ping", 10), entry("Speed", 10)))
18    .mixedStrengthGroup(
19      mixedStrengthGroup("OS", "Browser", "Ping").ofStrength(3))
20    .build();
```

Source Code 5.3: NewInputParameterModelFeatures.java

Requirements F4, F5, and F6 introduced many new model features which Subsections 4.1.4 and 5.1.3 integrated into coffee4j. Of course, the user also needs to be able to configure the features using the same interface as before. Therefore, this thesis extends the interface by using the same builder pattern as before.

The code above demonstrates the use of all new features. First, value 1000 ms of parameter Ping uses a value weight to assign a higher priority. This is done using the `weighted(Object, double)` static factory method, which creates a new `ValueBuilder` instance with the configured raw value (1000) and the weight (0.5). The `values` and `value` method in the `Parameter.Builder` class then explicitly check whether the passed object is an instance of `ValueBuilder` or not. It is necessary to pass the intermediate `ValueBuilder` object since all values of a parameter should have consecutive IDs starting at zero and those IDs are assigned inside the parameters's builder class.

In addition to weighted parameter values, this thesis introduced the seeding feature. For this, it added a new `seed(Entry<String, Object>...)` static factory method, which allows a user to specify a seed with instances of the `Map.Entry` class. One entry always maps one parameter to a specific variable. To allow for partial seeds, it is also possible to not have one entry for every parameter, but instead leave some out if the specific value is not important to the seed test case. Generation algorithms can then decide which concrete value to assign to those parameters to optimize test suite size. In addition to normal seeds, Subsection 5.1.3 also introduced seeds for negative testing, which lines 16 and 17 of the above example show. Here, the error constraint defining the negative test cases has an explicit name using `withName`, and the `seed` method then gets this same name as a reference to the constraint. All other parts of the seed definition stay the same as in the positive case to create a uniform interface.

In both cases, the factory `seed` method (the inner one) only creates instances of `Seed.Builder` and not directly of `Seed`. This additional indirection is necessary as the `Seed` class contains actual instances of `Parameter` and `Value` with their unique IDs, so it cannot directly create the respective instances as those are only stored in the `InputParameterModel.Builder` instance. The `seed` method (the outer one) therefore always calls `Seed.Builder.build(List<Parameter>)` with all currently registered parameters to create an actual instance of the `Seed` class which uses correct `Parameter` and `Value` instances. While this makes the code more complex, it creates an easier user interface which this thesis deemed to be more important.

`coffee4j-engine` Layer

Once the user defined a model, s/he still has to configure the actual algorithms and test method. In coffee4j this is possible on the `coffee4j-engine` and `coffee4j-junit-engine` layer [Bon18; Ber19]. The first one works similar to the previous definition of `InputParameterModel`:

```
new DefaultTestingSequentialPhaseManager(
  phaseManagerConfiguration()
    .testMethodConfiguration(testMethodConfiguration()
      .inputParameterModel(model())
      .testExecutor(
        (Combination combination) -> TestResult.success())
      .build())
    .executionConfiguration(executionConfiguration()
      .executionMode(ExecutionMode.FAIL_FAST)
      .generator(new Ipog())
      .faultCharacterizationAlgorithmFactory(Mixtgte::new)
      .prioritizer(new WeightBasedPrioritizer())
      .executionReporter(new PersistentFailingTestCasesReporter())
      .build())
    .extensions(List.of(new TestResultBasedModelModifier()))
    .build())
  .run();
```

Source Code 5.4: Coffee4jUsageEngineLayer.java

This consists of three main parts:

1. A `TestMethodConfiguration` which includes the IPM and the actual code to execute for each test case — this layer realizes it as a lambda. Here, the only changes this thesis made were the ones inside the `InputParameterModel`.

2. The `SequentialExecutionConfiguration` contains the definition of all algorithm specific to sequential combinatorial testing. This is in contrast to interleaving combinatorial testing which coffee4j also supports, but which is not part of this thesis. Here, most of the changes outside of the IPM occurred. The fluent API now has two new methods, `executionMode(ExecutionMode)` and `prioritizer(TestInputPrioritizer)` used to configure the execution mode of the combinatorial test (default is `EXECUTE_ALL`), and an optional prioritizer to use after the initial test suite generation.

3. A list of `Extensions`. This list can include all kinds of extensions, for example an `AfterGenerationCallback`, but also implementations of the new `ModelModifier`. It is even possible for a class to implement multiple extension points.

Some additional configuration is also possible, for example to specify a different `ExecutionPhase`, but those generally always stay on their default value. With the given options, it is now possible to configure everything necessary for the two points of prioritization and three different uses of available information. If the prioritization is done inside the generator, an algorithm developer can use the existing `generator()` method to register any number of generators, prioritizing or not. Otherwise, s/he can pass a `TestInputPrioritizer` to the `prioritizer()` method. For the available information, algorithm developers can either use information already present in the IPM, add new information using a `ModelModifier` in the `extensions()` method, or load the information inside the generator/prioritizer itself. The prioritization approaches can then use the old `Reporter` interface to persist new prioritization information over multiple runs. Additionally, all added features adhere to the same fluent API concepts used in previous versions of coffee4j and therefore present a uniform interface to the user. Since developers can configure coffee4j programmatically, it is also easily possible to execute the same combinatorial test with different prioritization configurations and thus compare them to one another.

`coffee4j-junit-engine` Layer

Of course it is also possible to configure the same features trough the JUnit API. A difference to the normal API is that algorithm developers can also define custom configuration options using the extension points in coffee4j's JUnit API. This subsection will therefore first present the default way to configure the new algorithms in Paragraph "Default Configuration" and then explain how algorithm developer can build their custom configuration options in Paragraph "Custom Configuration".

Default Configuration The default configuration options are nearly the same as for all other algorithms:

```
1  @CombinatorialTest(executionMode = ExecutionMode.FAIL_FAST)
2  @EnableGeneration(algorithms = Ipog.class)
3  @EnableFaultCharacterization(algorithm = Mixtgte.class)
4  @EnableTestInputPrioritization(WeightBasedPrioritizer.class)
5  @EnableExtension(TestResultBasedModelModifier.class)
6  @Reporter(PersistentFailingTestCasesReporter.class)
7  void testGreetingsPositive(
8      @InputParameter("OS") String operatingSystem,
9      @InputParameter("Browser") String browser,
10     @InputParameter("Ping") int ping,
11     @InputParameter("Speed") int speed) {
12   // arrange - act - assert
13 }
```

Source Code 5.5: Coffee4jUsageJunitLayer.java

It is now possible to use the new `@EnableTestInputPrioritization` annotation to configure exactly one `TestInputPrioritizer` to use after the initial generation. This annotation is optional, so it is also possible to leave it out and instead perform prioritization inside the configured generator. Similarly, `@EnableExtension` allows for the configuration of multiple extensions, including the new `ModelModifier` one. In both cases, the user only has to specify the class of the extension/prioritizer, and the framework will automatically create one instance by accessing the default constructor using reflection. However, automatically using the default (no-argument) constructor has the disadvantage that it is not possible to configure the instance via constructor parameters. Instead, the only option is to use Java system properties or environment variables. Since this is not very user friendly and also makes different configurations for multiple tests impossible, algorithms which require configuration by the user should always provide a custom configuration option as explained in the next paragraph.

Additionally, it is possible to configure the execution mode in the `@CombinatorialTest` annotation. While it would also be possible to create another annotation for this, the aim was to reduce the necessary knowledge of possible configuration annotations, and instead use a parameter in the annotation which will always be there. The execution mode is also one feature which is usable in both the sequential and interleaving mode.

Custom Configuration As mentioned before, it is also possible to provide custom configuration options. To this end, every configuration option like `Reporter`, `TestInputPrioritizer`, `Extension` and `InputParameterModel` follow the same general structure. Let's assume that one wants to add a new configuration option for instances of X. We now need three to four different classes. The X`Provider` interface defines one method which returns one (or if allowed multiple) instance(s) of X based upon the actual test `Method`. For example, it can use the given method to discover additional configuration annotations to create a custom instance of X which passes the configured options on to a non-default constructor. The next classes are `@`X`Source` and `@`X`Sources`, both annotations. If it is only possible to configure a single X instance, like with the `TestInputPrioritizer`, only `@`X`Source` is necessary, otherwise the other annotation class represents the repeatable annotation — a special construction in Java which allows for one annotation to occur multiple times on the same annotatable element. `@`X`Source` only has one configurable parameter, which is the X`Provider` class used to load actual X instances. The final class, X`Loader`, then searches a Java method for all `@`X`Source` annotations, gets the X`Provider` classes from their parameters, creates one instance per configured class, and finally passes the `Method` in question on towards the providers to get X instances. A user therefore only has to use the X`Loader` class and can load all kinds of differently configured X instances without knowing the implementation details [Bon18].

This would make it possible to annotate the CT method with many `@`X`Source` annotations to configure the instances. However, an even better and more user friendly way to configure X instances is to make `@`X`Source` a meta-annotation and annotate custom annotation with `@`X`Source`. Due to the way Java handles annotations, it then

seems as if the method is annotated with `@XSource` even though it is annotated with a more readable annotation name.

A practical example of this is the new `@EnableTestInputPrioritizer` annotation. It is itself annotated with `@TestInputPrioritizerSource(ConstructorBasedTestInputPrioritizerProvider.class)`. The `TestInputPrioritizerLoader` will therefore discover the source annotation, get the declared provider class, create an instance of the provider and then call the provider to create the actual `TestInputPrioritizer` instances.

This construction also makes it possible to, for example, set the `ExecutionMode` depending on whether a continuous integration pipeline currently executes the tests.

```
1 @ExecutionModelSource(CiBasedExecutionModelProvider.class)
2 public @interface EnableFailFastExecutionOnCi {
3 }
4
5 public class CiBasedExecutionModeProvider implements
      ExecutionModeProvider {
6   public ExecutionMode provide(Method m) {
7     return isOnCi() ? ExecutionMode.FAIL_FAST :
          ExecutionModel.EXEUTE_ALL;
8   }
9 }
```

Source Code 5.6: CustomExecutionModeSetting.java

When a test developer then annotates a method with `@EnableFailFastExecutionOnCi` the `ExecutionModeLoader` will discover the `@ExecutionModeSource` annotation and therefore use the `CiBasedExecutionModeProvider` to load the execution mode. As a result, the test will run in fail-fast mode when executed on CI, and in execute-all mode otherwise. This dynamic decision at runtime would not be possible with any other configuration option except of course creating the configuration programmatically as in the `coffee4j-engine` level API.

All new features support this `Loader` → `@Source/@Sources` → `Provider` construction. It is therefore possible to create a single annotation which completely configures one prioritization approach and registers the necessary algorithms, including those for loading model weights and integrating them into the IPM, the generation/prioritization algorithm, and the reporter for persisting collected information.

5.2. Failure-based Test Case Prioritization

The previous section presented the general implementation of all extension points and configuration options necessary to support different prioritization approaches. To get a better understanding of those concepts, this section will describe the realization of the failure-based test case prioritization approach presented in Section 4.2. First, Subsection 5.2.1 describes how to adjust the process to ease the implementation, and Section 5.2.2

then shows the actual classes created for the extension points of coffee4j and how they work together. At the end, Section 5.2.3 presents an easy way to use the prioritization approach based on the custom configuration option from Paragraph "Custom Configuration" of Subsection 5.1.4.

5.2.1. Process

Subsection 4.2.2 described the general process for failure-based CTCP. To quickly recapitulate the main points: The process starts with the adjustment of value weights, goes on towards generation, prioritization, and execution of the initial test suite, and then optionally performs fault characterization. Afterwards, it saves the necessary data, like FICs or failing test cases. All in all, this general process describes six possible instances. There are two ways to perform the prioritization, during the generation or afterwards, and three ways to calculate the weights, FIC-based, only negative test cases, and negative and positive test cases with suspiciousness of components, which one can combine with each other.

For the actual implementation, it would be of advantage to reuse as many components as possible between the six process instances. For the different prioritization points this is already possible, as the prioritization only depends on the the weights defined in the IPM and not on how the prioritization approach calculates them — one big advantage of IPM-centered approaches. However, the three ways to calculate weights all require different information. While the first one needs FICs, the latter ones both require complete test cases, with the second one needing only negative and the third one also positive test case results.

One possible way to unify those three approaches is to not differentiate between FICs and test cases, but instead focus on combinations. For each run, the approaches only save a map of combinations to a result, which is either successful or failed. With this idea, they can persist FICs and failing test cases with failed results, and passed test cases with successful ones. This common abstraction also simplifies weight calculation. Both approaches in Paragraph 4.2.4 and 4.2.4 sum up value occurrences in FICs and failing test cases and with the common abstraction of failing combinations both approaches can use the same weight calculator. The only thing that needs to change is what combinations an approach persists.

The new process therefore works the following way: first is loads a list of failed and successful combinations for every previous run. For each run, it then performs the calculation independently based on some approach specific calculator and later integrates the weights into the IPM. After the old CT process based on this model finishes, the new process then persists some failing and successful combinations. The source of those combinations can be FICs or test cases.

In this process, it is now possible to create new prioritization approaches by specifying what combinations to persist and how to transform them into value weights in later runs.

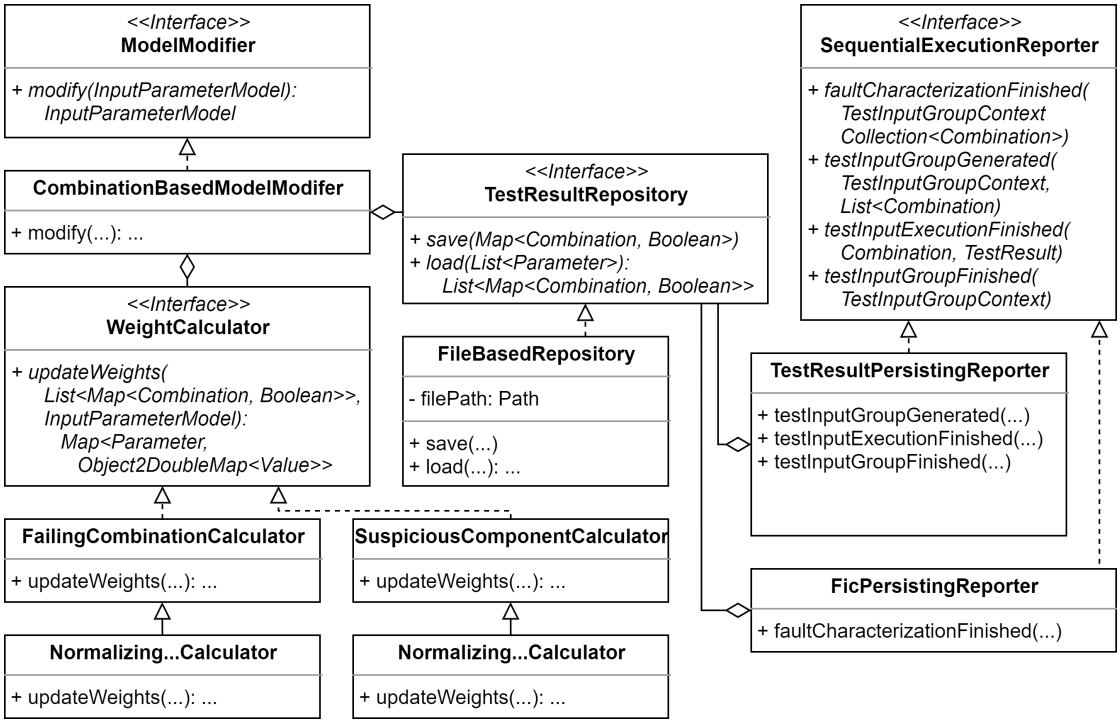

Figure 5.3.: Class diagram of failure-based prioritization approach

5.2.2. Integration into coffee4j

Now that the general process with which the implementation handles different approaches when prioritizing based on previous test cases or FICs is clear, it is time to look at the actual classes involved, and how users can register them with coffee4j.

For the middle part dealing with the generation of a prioritized test suite or the prioritization after generation one can use standard algorithms since the approaches transform all custom prioritization information into value weights. Therefore, this subsection will focus on the start and end of the process, namely how to load persisted information to integrate it into the IPM and how to save the corresponding information for later runs. Section 5.3 will discuss how to efficiently implement a prioritizing generator based on DDA.

Figure 5.3 depicts all relevant classes for persisting combinations and their results in a *Unified Modeling Language (UML)* class diagram. To make the diagram as small and still as meaningful as possible, it abbreviates some method definitions in subclasses by replacing parameters and returns types with three dots. In those cases one can assume that the method's signature is equal to the one its superclass/interface. Additionally, if a subclass does not explicitly implement a non-concrete method, one can assume that it is always an empty implementation since the subclass does not use it.

Repository In general, the class's diagram center is the `TestResultRepository`. It defines a general interface for a repository which can save one map of combinations to test results per test run and load the saved maps for all runs which were ever saved. This realizes the persistence layer of the prioritization approach. For example, if the prioritization uses FICs, one would save all discovered FICs of one run with the value `false`, or an empty map if there are no FICs. The next runs would then load the discovered FICs of all previous runs.

Currently, only one prototype implementation exists, the `FileBasedRepository`. It persists all saved combinations and their results in a file. The coffee4j user then has to specify the same file for all runs of the test and can then automatically load them.

ModelModifier `CombinationBasedModelModifier` is the only implementation of the `ModelModifier` interface, and consequently all prioritization approaches use it. The variability comes into play with different `WeightCalculator` implementations.

In general, the modifier has a very short implementation and few responsibilities. First, it loads the failed and successful combinations from all previous runs using a `TestResultRepository`. In reality this will always be a file-based one, but the class only depends on the `TestResultRepository` to allow for easier modifiability — for example, if one were to decide that a database should instead store all combinations. Next, it passes those loaded combinations to the `WeightCalculator`, which returns a map that assigns a weight to each `Value`. At the end, the modifier then integrates those weights into the given IPM.

Currently, there are four implementations of the `WeightCalculator` interface. The first one, `FailingCombinationCalculator`, implements the calculation algorithm described in Paragraphs 4.2.4 and 4.2.4. Its subclass `NormalizingFailingCombinationCalculator` delegates the computation to the superclass and normalizes the weights afterwards. Similarly, `SuspicousComponentCalculator` bases its calculation on the suspiciousness of components like the weight calculation method from Paragraph 4.2.4, and its subclass `NormalizingSuspiciousComponentCalculator` additionally normalizes the weights per parameter.

SequentialExecutionReporter The last missing link in the presented process is the persistence of information used for prioritization. For failure-based prioritization this is done using the `SequentialExecutionReporter` interface. It defines several callbacks throughout all phases of the CT process like for the execution result of a single test case, the results of FC and whether all tests have finished their execution [Ber19].

For the three different weight calculation methods one needs two different ways to store information. `FicPersistingReporter` is responsible for persisting all discovered FICs as failed combinations using an instance of the `TestResultRepository`. If the FC algorithm could not find any FICs or if no test case failed, it saves an empty map. On the other hand `TestResultPersistingReporter` persists all test results. To this end, it uses the `testInputGroupGenerated` callback to get a collection of all test cases which are part of the initial test suite and persists their results internally whenever

coffee4j calls `testInputExecutionFinished`.

All these weight calculators and reporters allow us to perform the three calculation approaches as Subsection 4.2.4 defined by using certain combinations. For the simple FIC weighting (Paragraph 4.2.4) one has to combine `FicPersistingReporter` with `FailingCombinationCalculator`. `TestResultPersistingReporter` and `FailingCombinationCalculator` can achive simple test case result weighting (Paragraph 4.2.4) while suspiciousness-based test case result weighting (Paragraph 4.2.4) also requires `TestResultPersistingReporter` as a reporter but `SuspiciousComponentCalculator` as a weight calculator. The last remaining combination of `FicPersistingReporter` and `SuspiciousComponentCalculator` does not make much sense since the notion of suspiciousness implies having successful test cases, which cannot be the case if the reporter only persists FICs which are inherently failing combinations.

5.2.3. Usage

For actually configuring the `WeightCalculator` and `SequentialExecutionReporter` one must employ a custom configuration as presented in Paragraph 5.1.4. This is necessary due to the fact that `@EnableExtension` and `@Reporter` only support classes which have a no-argument constructor. However, the constructor is the only place where one could inject the necessary `TestResultRepository` and `WeightCalculator` into the `CombinationBasedModelModifier`. A custom configuration option would be able to define how a modifier instance is created and therefore inject a calculator and repository into the constructor of the model modifier.

```
1 @CombinatorialTest
2 ...
3 @EnableFailureBasedPrioritization(
4     filePath = "path/to/file.txt",
5     weightCalculator = SuspiciousComponentCalculator.class,
6     reporter = TestResultPersistingReporter.class)
7 void testMethod(...) {
8   // ...
9 }
```

Source Code 5.7: FailureBasedPrioritizationUsage.java

The actual configuration happens with the custom `@EnableFailureBasedPrioritization` annotation. It is itself annotated with an `@ExtensionSource` and `@ReporterSource` annotation, which point towards a custom extension and reporter provider. The extension provider extracts the path argument from the `@EnableFailureBasedPrioritization` annotation and creates a `FileBasedRepository` using that path. Next, it creates a new instance of the given calculator class using a no-arguments constructor. It then uses the reporter and calculator instances to create a new `CombinationBasedModelModifier` and returns it. On the other hand, the custom reporter provider also constructs a `FileBasedRepository` the same way as the

extension provider, but then creates an instance of the reporter class with the repository as a single constructor argument. All in all, this means that the given weight calculator always needs to have a no-arguments constructor, and the reporter needs to have a one with exactly one argument of type `TestResultRepository`.

A user can therefore just annotate a test method with the `@EnableFailureBased-Prioritization` annotation and configure the file path to integrate failure-based prioritization into an existing combinatorial test. The annotation will take care of configuring the test in such a way that it persists failure-based information and automatically loads it into the IPM at test startup. To make the configuration even easier, the `weightCalculator` and `reporter` field already have default assignments so a user only needs to define the file path.

5.3. Weight-based Prioritization with Constraints

The previous section described how to implement a concrete prioritization approach in coffee4j which uses value weights. An important part of its process is generating test suites based on those value weights. To this end, Section 4.3 already presented a modified variant of DDA which can handle constraints in addition to prioritizing values based on weights. This section will now deal with the actual implementation.

One possible way to implement the algorithm is to follow the general structure of the pseudo code in Section 4.3. However, this can and will lead to serious performance problems. The pseudo code uses a set `RemainingCombinations` to stores all yet uncovered t-way combinations. It uses this to check which combinations it still needs to cover, and what parameters and values have a higher weight and should therefore come first. However, with a higher testing strength, the number of possible t-way combinations is large. For example, if the IPM contains twenty parameters with four values each, there are 3040 two-way combinations, 18240 three-way combinations, and already 77520 four-way combinations. Some operations in the algorithm like calculating the parameter weight and removing covered combinations would then require us to look through the complete set of all remaining combinations even if only part of them are actually relevant.

Lei et al. faced the same problem when creating a variant of IPOG which could create test suites of strengths higher than two [Lei+07]. Their solution used a two-level data structure called a coverage map. On the first level it stores one instance of the second-level data structure for every possible t-way **parameter** combination. In the running example, this would contain the combinations {OS, Browser}, {OS, Ping}, {OS, Speed}, {Browser, Ping}, {Browser, Speed}, and {Ping, Speed}. At the second level, an instance of the data structure stores all **value** combinations of its respective parameter combination. For the parameter combination {Browser, Ping} this would be (–, Chrome, 10 ms, –), (–, Chrome, 100 ms, –), (–, Chrome, 1000 ms, –), (–, Edge, 10 ms, –), and so on. Whenever the algorithm needs to know some information about the remaining combinations, it asks the coverage map, which then delegates the request to all relevant second-level data structure instances. This is more efficient as it only consider those second-level instances which contain a parameter that is in the request. For example, if the DDA algorithm needs to

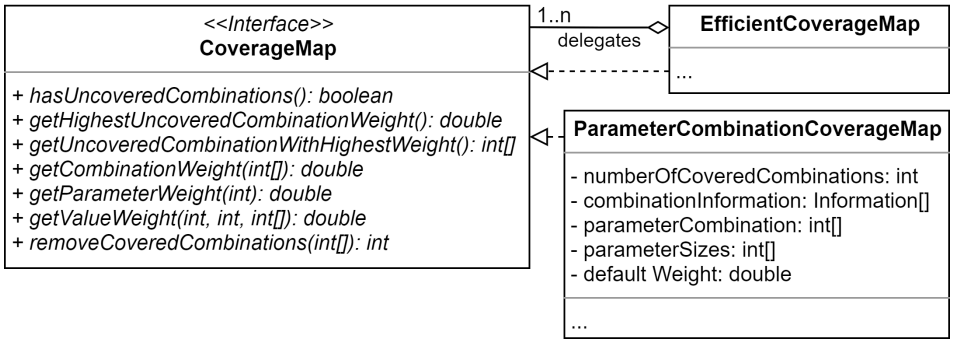

Figure 5.4.: Class diagram of coverage map

know the weight of a parameter, the coverage map only needs to consider second-level instances which have this parameter in their respective combination.

In addition to the separation between parameter combination, one can also optimize the second-level data structure even further. By ordering uncovered combinations in an array, it can create a direct and easily calculable mapping between a combination and its position [Lei+07]. For this, it can use a function which takes into account the index of the parameter value and the size of all parameters. As an example, the Ping parameter has three values. When looking at the parameter combination {Browser, Ping}, the value combination (–, Edge, 1000 ms, –) would have index $1 \times 3 + 2 = 5$, since Edge is at index 1 in parameter Browser, and 1000 ms is at index 2 in parameter Ping. More generally, if one has a three-way parameter combination of parameters i, j, and k and the sizes of those parameter are s_i, s_j, and s_k that the index of combination (v_i, v_j, v_k) is $v_i \times (s_j \times s_k) + v_j \times s_k + v_k$. This generalizes to any number of parameters. The data structure can also recalculate a combination via its index by using the modulo (%) operator so that it never has to store the actual combinations but instead can just use a bitmap which stores whether a coverage map covers a combination or not on the respective index.

Figure 5.4 shows how coffee4j implements this schema for DDA. Here, `CoverageMap` defines the general interface with all requests that DDA needs to make to a coverage map during the test suite generation. `EfficientCoverageMap` is the first-level data structure, which contains one delegate for each parameter combination. Similarly, `ParameterCombinationCoverageMap` realizes the coverage second-level data structure for one single parameter combination. All together, the classes follow the composite pattern and therefore allow for easy evolution in case someone discovers a more efficient implementation for the first or second level.

To better understand the relationship between the classes one can look at an example. Lets say the algorithm needs to order all parameters and therefore request the weight of one specific parameter. `EfficientCoverageMap` forwards this request to all delegates and then sums up the returned values. Those delegates are `ParameterCombinationCoverageMaps`, one per parameter combination. To answer the request they can first look whether their parameter combination contains the parameter in question, and then either

return a weight of zero if this is not the case, or the cumulative weight of all uncovered combinations if they contain the parameter. `ParameterCombinationCoverageMaps` which do not contain the parameter will therefore skip the request.

Due to the optimized index in `ParameterCombinationCoverageMap` it is also easy to remove all combinations covered by a new test case. Each instance needs to check the values in the test case, calculate the index and set the `Information` at this index to covered. It is not necessary to search through every remaining combination to cover all those which the test case contains.

All in all, those optimizations allow for an efficient implementation of the constraint-aware DDA variant presented in this thesis. In most common use cases it remains nearly as fast as IPOG while also generating prioritized test cases.

6. Evaluation

Contents

Until now this thesis presented a concept and implementation of a general framework for CTCP, an approach for test case prioritization based on historical failure information, and an algorithm for generating prioritized CT test suites based on weights under the presence of arbitrary constraints. However, an important part is still missing: the evaluation of all presented concepts. It would be useless if we now had build a framework for CT prioritization but it did not work as expected.

This chapter will evaluate the presented concepts in two parts. First, Section 6.1 will evaluate the integration of CTCP into coffee4j. This includes checking the requirements and discussing whether coffee4j now supports all prioritization approaches presented in part "Combinatorial Test Prioritization" of Subsection 2.1.2. Next, Section 6.2 includes several practical experiments made with coffee4j using the new prioritization approaches from Section 4.2.

During the evaluation, this thesis will also answer its research questions, which were presented in the introduction. For a better overview, they are printed here again:

RQ1: How can combinatorial test case prioritization during or after test suite generation based on (additional) information extracted from the input parameter model or supplied from external sources be integrated into the coffee4j architecture and process model?

RQ2: Does failure-based combinatorial test prioritization or non-prioritized combinatorial testing require fewer test cases to detect failures?

RQ3: Does failure-based combinatorial test prioritization during or after test suite generation require fewer test cases to detect failures?

Chapter 4 already partially gave the answer to **RQ1**, but the actual evaluation of whether the presented concepts reached their goal will be topic of this chapter. Section 6.2 will then answer **RQ2** and **RQ3** by performing several experiments.

6.1. Framework

Before checking if failure-based test case prioritization detects failures earlier than traditional CT, one first needs to evaluate whether the coffee4j framework works as expected. Otherwise the experiment results would not have any significance since the framework could negatively or positively influence the results. Therefore, this section will evaluate whether the integration of CTCP into coffee4j works as expected. To this end, Subsection 6.1.1 first checks whether the framework correctly realizes the different requirements, while Subsection 6.1.2 evaluates whether it can actually support all categories of prioritization approaches.

6.1.1. Requirements

A good approach for checking whether a software does what it is supposed to do is checking it against the defined functional and non-functional requirements. If it does not fulfill them there are two possible explanations: Either the software does not do what the user wants and therefore also does not fulfill the requirements, or the requirements did not represent what the user wants but the software somehow does. The latter case could occur if changing requirements were not documented during the development process. To avoid the much worse situation of not satisfying the user needs due to not fulfilling the requirements, it is always important that the requirements are defined so that one can check them in the finished software product.

For most requirements one can do this automatically via automated acceptance tests. Especially for functional requirements this is a good practice and it also allows for regression testing to reuse those automated test. Therefore, they automatically guarantee that the tested parts of the software still work in later iterations even after adding new features in other parts.

coffee4j's functional requirements are all verified via automated tests. To this end, the package `de.rwth.swc.coffee4j.junit.engine.it.requirements.prioritization` in the `coffee4j-junit-engine` module contains at least one automated test per functional requirement. For easier identification, the class-name always contains the tested requirement.

In addition to the functional requirements, one can also automatically verify some non-functional ones. For example, Requirement N1 stated that it should not be necessary to specify a generator which uses prioritization information any differently than a normal generation algorithm. Therefore, the same package also contains some automated tests focused on non-functional requirements dealing with usability questions.

For some other non-function requirements, automatic verification is not possible. Here, the next paragraphs take a more argumentative approach.

Requirements N2, N3, N4, N6 These requirements state that the registration of after-generation prioritizers, model modifiers, new model features, and the fail-fast mode should all happen in a way that is uniform to the existing configuration options. For

most of them, one has to look at two possible configuration points: The API on the `coffee4j-engine` and `coffee4j-junit-engine` level.

In both cases, all algorithm configuration options (prioritizer and model modifier) are configured the same way as existing algorithms like the generator. In JUnit users use the `@Enable...` annotation and pass the algorithm class and on the lower-level API they pass one instance to the respective method in the builder. Additionally, they both support the `Source-Provider-Loader` pattern on the JUnit layer. This allows for custom instantiation options such as injecting necessary constructor parameters.

It is also possible to configure the execution mode via the same pattern in the JUnit layer which makes it nicely integrated into the existing information. Additionally, the default option for configuration is the `@CombinatorialTest` annotation. This is consistent with definition of other features present in both the sequential and interleaving CT approach such as the IPM and test name.

Last but not least, a user can also configure all new features available for the IPM through the usual API consisting of several builder classes. For seeds and mixed strength groups, parameters are referenced by their name, just as in the existing constraints concept. Value weights also use common concepts. One can give values a weight by using the `weighted` factory method, and this also only adds an argument for the weight and does not change the definition of the value itself. It is still possible to define every kind of object as a value. Additionally, it is easily possible to mix both weighted and non-weighted values and not necessary to only given one type. This would have been the case with alternating method parameters — always specifying one value and then one weight — which was also an option. A user therefore only has to define those weights, mixed strength groups, and seeds that s/he actually wants to use, just as with all other configuration options of the IPM. Since the `coffee4j-engine` and `coffee4j-junit-engine` layers configure IPMs the same way, one does not need to look at both of them explicitly in this evaluation.

All in all, one can say that all added configuration options come with sensible defaults and seemlesly integrate into the existing API. Due the common use of a `@Enable...` annotation-prefix, all new features can also easily be discovered by users.

Requirement N5 This requirement mandated a documentation of all new prioritization features via examples. Therefore, coffee4j now includes the new `coffee4j-examples` project which contains one example test case per feature. In contrast to the automated acceptance tests for the functional requirements, the maven build does not automatically execute these tests and they also do not assert any execution results. They only focus on creating an easy to follow example which demonstrates how developers can use certain features alone and in combination with other features. In addition to the code itself the examples also include explanatory comments.

Requirement N7 Another important requirement was the testability of the new extension points. This is especially important for algorithm developers, as they need to be sure that their algorithms actually perform as expected before using them in real-world

scenarios. Subsection 5.1.2 already explained that coffee4j realizes all new extension points as functions which map a set of input parameters to one output. Additionally, they are all stateless. This makes testing them relatively effortless, as a test only needs to create an instance of the algorithm itself and can then test the relevant method via a parameterized test with some inputs and expected output conditions. Tests for the implementations of algorithms for failure-based test case prioritization demonstrate this principle.

Requirement N8 One can verify the requirement for test framework independence by looking at the `coffee4j-engine` layer. Since it is possible to use all features which this thesis introduces on that level, the features obviously cannot depend on any concrete testing framework. Even the `Source-Provider-Loader` pattern for configuring a combinatorial test does not directly depend on JUnit even though it is located in the `coffee4j-junit-engine` layer. However, since no integration of coffee4j into other framework currently exists, this cannot easily be demonstrated.

Requirement N9 This requirement focused on the user group of framework developers and stated that all parts of the integration of prioritization into coffee4j must be testable. Of course it is difficult to evaluate whether a piece of code is testable without attempting to write tests for it. Therefore, the evaluation of this requirement are all the test cases in different parts and layers of coffee4j, which check that the added functionality works as expected.

6.1.2. Prioritization Approaches

As stated in Section 4.1, the original goal for integration of test case prioritization techniques into coffee4j was to support all current points of prioritization and means of using additional information. This subsection will look at all of those categories and assess whether the current implementation of coffee4j supports them.

Point of Prioritization

First, this subsection takes a look at the point of prioritization. As described in Paragraph "Point of Prioritization" of Subsection 2.1.2, the prioritization of individual test cases must either happen directly during their generation or in a separate step after the generation.

During Generation Prioritization during generation was technically already possible before this thesis. However, this only included prioritization without any additional information. Now, it is possible to use the new model features like value weights and seeds to prioritize certain combinations over others. coffee4j supports this point of prioritization via the `TestInputGroupGenerator` interface. Here, an algorithm developer can return a new `TestInputGroup` which includes any number of test cases and now it is possible to use more model information to also prioritize them. `DeterministicDensitlyAlgorithm` is one example of such a prioritizing generator.

After Generation If the generation itself does not prioritize test cases, it is also possible to do so afterwards in a separate step. To this end, coffee4j includes the `TestInputPrioritizer` extension points. Implementations of this extension point can use all information from the IPM to prioritize a set of initially generated test cases in any way they want as long as they do not change the test cases themselves. An example implementation of this is the `WeightBasedTestInputPrioritizer` which prioritizes test cases by always picking a next test case where the contained not-currently covered t-way combinations have the highest cumulative weight.

Available Information

The second category is the information which is available for prioritization. Here, the possibilities are that an approach either uses no additional information, encodes additional information inside the model, or loads information directly in the algorithms themselves from an external source without going through coffee4j's IPM.

No Information One option for prioritization is to use no additional information. coffee4j does not need to handle this case explicitly since it is automatically possible for an algorithm to only use the information supplied by the old IPM. Currently, coffee4j contains no example implementation for this case, but one could easily add an algorithm which creates test cases for incremental combinatorial coverage.

Information in Model coffee4j now explicitly supports the widespread practice of loading prioritization information into the IPM. To this end, it is now possible to add weights, manually or automatically. Additionally, the `ModelModifier` extension point allows for explicit integration of prioritization information directly after the start of a combinatorial test, for example by calculating weights based on historical execution information.

While coffee4j now supports weights, seeds, and mixed-strength testing in the new IPM, one could still think of other modeling constructs to describe prioritization information. For example, it could be necessary to explicitly put weights on t-way combinations instead of individual values to avoid accidentally weighting combinations very high when all contained values are part of important combinations. Similarly, it is possible, that new prioritization approaches use even different means to represent prioritization information. Since it is not possible to anticipate the needed structure in the IPM, one continuously needs to adapt it to serve the needs of algorithm designers if they decide to use other ways of expressing the necessary information. While it would always be possible for them to just use the option of loading prioritization information only inside the generation algorithm, this has the disadvantage that only custom algorithms will be able to use the information. coffee4j's IPM should therefore be extended if the new features are not enough for some prioritization algorithms.

Currently, coffee4j contains three examples of this way of handling prioritization. Section 5.2 presented all of them, and they use failure information from previous runs for prioritization.

External Information If it is not possible to integrate the necessary information into the IPM, it is always possible to load it directly inside the generation or prioritization algorithm. This is also explicitly supported by coffee4j due to the use of the `Source-Provider-Loader` pattern with all new algorithms. Here, specific instances of algorithms can be created with constructors that take arguments for configuration. It is therefore possible to pass all necessary information for accessing external information for prioritization to an instance of the algorithm, and make the configuration seamlessly integrate into coffee4j by defining appropriate custom annotation. While coffee4j contains no explicit example of an algorithm which loads its custom prioritization information, it does show how to configure algorithms using custom annotation. The implementation of such an algorithm would then be relatively straight forward.

6.1.3. SonarQube

The last sections evaluated whether the integration of CTCP into coffee4j satisfies the requirement and therefore also supports all currently known CT test case prioritization approaches. However, another important way to evaluate a software system is static analysis. Therefore, SonarQube was used to examine coffee4j and check the quality of the code. This is especially important for future extensions of coffee4j, since it is much easier to extend code which does not contain many bugs and is well tested.

SonarQube currently puts coffee4j at about 7700 lines of code, with an additional 2091 lines of comments. Most of those lines are inside actual algorithm implementation in the `coffee4j-algorithmic` algorithmic package, with IPOG requiring 358 lines and DDA consisting of 249. The test coverage in the complete project is at about 80%, with 70% condition coverage. In many classes the low coverage is a result of not explicitly testing code generated by the development environment like the `equals` or `hashCode` methods, which can also contain many conditions. The parts of the code actually implementation prioritization algorithms or strategies are at 95-100% code and condition coverage, which enables safe evolution in those areas.

In addition to those statistics, SonarQube also reports some issues with the code, for example blocks of duplicated lines. However, nearly all of those are not part of the code written for this thesis, or not relevant due to being false positives. All in all, SonarQube gives the coffee4j project a rating of “A”, which is a good indicator that coffee4j remains easy to extend and maintain.

6.2. Experiments

Until now this chapter evaluated the integration of CTCP into coffee4j and whether it supports all current prioritization approaches. This section will instead focus on evaluating the specific failure-based prioritization technique presented in Sections 4.2 and 5.2 using different experiments. Those will also implicitly evaluate the constraint-aware extension of DDA.

The next section first presents the general setup used for executing the experiments.

Subsection 6.2.2 then presents the results and how to interpret them and the chapter finishes with a discussion of the results in Subsection 6.2.3.

All scenarios, configurations, and measurements are available in the `https://git.rwth-aachen.de/joshua.bonn/thesis-evaluation` repository for every logged in user. Commit `7e8547c068fa9132c4900198fa50729191ac56cb` is the bases for this evaluation.

6.2.1. Setup

An experiment setup always consists of three parts: algorithm configurations, scenarios (with IPMs), and metrics captured during the experiment execution. This subsection will explain all of those in depth to give an understanding of what those experiments actually test.

Scenarios

First, one needs to define the concept of a test scenario for these experiments. Since they execute failure-based test case prioritization, which is a history-based prioritization technique, there are two general ways to perform testing:

1. Explicitly define a given starting point in form of a database, or in this case a file, of previously failed test case/discovered FICs and then execute one or more runs in coffee4j to see how the approaches prioritize combinations. This has the advantage that its easy to setup complex scenarios which would require hundreds of runs in coffee4j without actually executing them. One only has to define the expected outcome in form of a list of FICs or test case results. Then, one only executes the "interesting" runs and compares them across different algorithms and calculation approaches. However, in case of test case result based techniques one does actually need to generate the test cases for all those runs and assign them a suitable result to make the techniques work. Additionally, this way of performing experiments assumes that previous iterations all worked as the author expects. It may also be hard to test subsequent versions of algorithms since each of them may generate a different set of test case. Therefore, one would have to adjust all static test results, which is much work.

2. The second option is to always start at "zero", i.e. with no previous test history. One then needs to execute all runs on the actual system until one gets to the relevant ones. While this may require more execution time, this is a more dynamic and realistic form of testing, since it is easy to later change small parts of the configuration or scenario and run the whole experiment again. It is not necessary to adjust some data as in the first option.

Due to the higher realism and quicker iteration times, this evaluation uses the second form of scenarios. Therefore, one scenario basically consists of an IPM and an arbitrary list of `TestInputExecutors` which define what test cases should pass in one particular

run. For example, one can define that the IPM is the one from the running example in Table 2.1, and in the first six runs all test cases containing the combination (Windows, –, 10 ms, –) fail, and in the next five runs all those containing (Linux, –, 100 ms, –) fail. This scenario therefore simulates a system which may be tested as part of an automated testing pipeline and has one cause of failure for the first six versions which a developer then fixes, but this fix introduces a defect in another part of the software.

Since the evaluation uses coffee4j itself, it is possible to specify any arbitrary `TestInputExecutor`. It just has to be a piece of code which transforms one given `Combination` into a `TestResult` — the part normally done by the static test method calling a SUT. While some implementations may simply fail if the test case contains a specific FIC, others could even call versions of real programs. This makes it possible to define a scenario as an IPM of a real program which then tests multiple consecutive versions of the program to show how the prioritization approaches actually fare in real life scenarios. One can divide the scenarios tested during this evaluation into two categories: synthetic scenarios and scenarios working with actual programs.

Synthetic Scenarios Synthetic scenarios do not use any real world program and, in case of this evaluation, always consist of a static IPM and a list of (failure-inducing) combinations which will cause the program to fail for each run. Since there is no formal method behind the selection of the scenarios, they often include a bias and one should only use them to evaluate whether the algorithms work correctly in basic scenarios. However, to make them a little bit more realistic, they use a different model than the running example. One can see it in the `CommonConstants` class. It has a $2^2 \times 3^7 \times 4^5 \times 6$ parameter-value configuration and contains no constraints, seeds, weights, or mixed-strength groups. For this evaluation, the following synthetic scenario were constructed and executed based on the model:

1. `SameFicMultipleTimesScenario`: This first scenario consists of five runs, each with the same single FIC. As a result, one can easily see whether a weight calculation approach using failure-based prioritization technically works for very simple scenarios. The scenario exists in three different variants: first, both the model strength and size of the FIC are two, then there is one variant which increases the model strength to three while still having a FIC of size two, and the final variant has both model and FIC at strength three. Therefore, the scenario also validates that the weight calculation works for higher-strength CT.

2. `SameFicsMultipleTimesScenario`: As a next step, this scenario contains the same three FICs per run. This checks that the weighting method can work with multiple FICs and does not just prioritize one FIC to much. In contrast to the previous scenario, this one only contains two variants, one with both model and FICs at strength two, and one with both at strength three. All following synthetic scenarios contain the same variants, though they do not state so explicitly.

3. `SameFicsWithOtherFicsInBetweenScenario`: The third scenario checks whether a weighting method only considers the results of the last run, or also

those of previous ones. To this end, it consists of eight runs with changing and repeating FICs. The first one contains a single relevant FIC which is the focus of this scenario. Next, two runs do not contain any FIC at all and let all test cases pass. Runs four and six both contain another fixed FIC, while runs five and seven contain a different fixed FIC. At the end, run eight again consists of the relevant failure-inducing combination of the first run. If the weighting method did not place to much weight on the FICs executed in runs two through seven, it should still weight the FIC from run one higher than at the beginning.

4. `FicsWithSameValueScenario`: Scenario number four deals with errors that occur in situations similar, but not equal to past error situations. It consists of four runs. The first two each have two independent (no shared values) FICs, and the third and fourth one both have the same FIC which is a combination of the values from the first two FICs. This models a situation where an error occurs in a location similar to where errors previously occurred. If the tested prioritization approach can generalize from the first two FICs, it should be able to catch the third one faster than normally.

5. `ManyIterationsWithOccurrenceProbabilitiesScenario`: The last scenario is a bit more complex. It consists of 500 runs were the individual FICs are sometimes fixed combinations and sometimes random. This checks whether the algorithms can deal with random data and still prioritize the test cases which fail more often. All in all, 20% of all runs have random failure-inducing combinations, another 20% use the first fixed FIC, 17% the second one, 15% the third, 10% the fourth, again 10% the fifth and finally 8 % use the sixth fixed FIC. One can assume that failure-based prioritization approaches order the FICs which occur more often higher even though some runs fail due to random causes.

To find the actual combinations used for all FICs one can take a look at the evaluation module in coffee4j.

Actual Program Scenarios The synthetic scenarios give a good first impression on whether a failure-based prioritization approach works in basic situations. However, to test whether they will work in real-world scenarios, one inevitably needs to use real-world programs. In contrast to synthetic scenarios, real world programs contain more complex failure scenarios. For example, a synthetic scenario just simulates the failure by giving a list of FICs and failing all test cases which contain it, while a real program execution can be much more complex in that only a subset of those combinations actually always cause failures. Additionally, real programs may contain multiple defects which interact with each other and therefore make it impossible or at least very difficult to determine an actual combination responsible for the failure. Therefore, the actual program scenarios use real programs with either carefully seeded or actually discovered defects.

The actual programs used in this thesis come from *Software-artifact Infrastructure Repository (SIR)* [DER05]. SIR contains a collection of seeded and non-seeded faulty

versions of multiple programs. An often used subset of those is the Siemens Suite of programs, which contains `tcas`, `schedule2`, `schedule`, `totinfo`, `printtokens`, `printtokens2`, and `replace`. For all of those programs, Ghandehari et al. developed abstract IPMs and programs which can convert an abstract test case into concrete input data [Gha+13].

This thesis only uses the `schedule`, `replace`, `printtokens`, and `printtokens2` program during the evaluation, since the other programs had too few failing versions at testing strengths two and three to actually test the failure-based prioritization. It tests each program both at strength two and three, and uses the models and input generators developed by Ghandehari et al. to execute the actual programs. Another important part is the test oracle which determines whether the program behaved correctly for the current input. Here, the oracle just compares the execution of the faulty version to the execution of a program version without the defects and determines an execution to be successful if the outputs and result code match.

Until now this subsection looked at how one can perform a single run of coffee4j with a program of the Siemens Suite. However, as with the synthetic scenarios, one needs to perform multiple runs to actually evaluate the failure-based prioritization approaches. To this end, each scenario with an actual program consists of 25 runs, where each run uses a random version of the program with faults that can be detected by CT with strength two or three. Of course, this does not simulate an actual program evolution over multiple versions, but it does include real faulty programs with real failure scenarios.

Configurations

The evaluation executes each of the presented scenarios with different algorithm configurations. Those define which prioritization, generation and model modification algorithms to use, and how the prioritization actually calculates value weights. All in all there are three categories with multiple actual configurations: the used generators, the point of prioritization, and the actual method used for prioritization. The evaluation framework then creates the Cartesian product of those options and executes each relevant entry once for every scenario.

Generators To check how well the weight-based prioritization approaches and the constraint-aware DDA variant work, one needs to test multiple different generation algorithms. It this case, the three algorithms **AETG**, **IPOG**, and **DDA** were selected due to their generation characteristics.

AETG uses randomization during the generation and therefore nearly always generates different test suites for the same model. While it does not perform prioritization itself, it could be interesting to see how well an after-generation prioritizer can deal with test suites that do not always stay the same. However, the coffee4j implementation of AETG is very slow (above one minute for the model in the synthetic scenarios), so this evaluation only uses it for the first few synthetic scenarios with relatively few runs.

In contrast to AETG, IPOG always deterministically generates the same test suite for the same input. However, it also does not perform any form of prioritization. An

interesting combination is IPOG with the weighting method based on failing test cases. Since the test cases are always the same, the after-generation prioritization may just prioritize the test cases which previously failed very high without caring about the actual failure-inducing combination.

The last algorithm, the constraint-aware DDA variant, is also deterministic just like IPOG, but it can also generate a prioritized test suite based on value weights. This is therefore the only algorithm which can perform prioritization by itself.

Point of Prioritization The second configuration category is the point of prioritization. Here, there are four options:

1. **None**: Does not use any prioritization. This is the base case to which one can compare all other points of prioritization. Since the prioritization method does not make any difference if prioritization is not even enabled, there is only one entry for each generator with the point of prioritization set to None.

2. **Generator**: Only the generators themselves perform prioritization. Since only DDA supports prioritization, this setting only makes sense if the generator is DDA. Otherwise it will result is the same measurements as setting the point of prioritization to None.

3. **After(0)**: This configuration performs prioritization after the generator using the `WeightBasedTestInputPrioritizer` presented in previous sections. In case of the DDA generator it is not possible to disable prioritization during generation, so the test suite is prioritized at two points to see if it makes any difference. The zero in parentheses means that the default weight for all values which do not have one is zero. This can make a difference when comparing the weights of two combinations to each other. Lets assume that value Windows has a wight of 0.5, Linux has one of 0.2, and Chrome does not have any weight. Since the algorithm calculates the weight of a combination by computing the product of the contained value weights, both combinations (Windows, Chrome, –, –) and (Linux, Chrome, –, –) have an equal weight of zero. This may go against the intuition that (Windows, Chrome, –, –) should be weighted higher since Windows has a higher weight than Linux.

4. **After(ϵ)**: Everything is the same as in After(0), but instead of a default weight of zero the prioritizer uses a very small default weight, for example 0.000000001. In this case (Windows, Chrome, –, –) has a weight of 0.0000000005 and the weight of (Linux, Chrome, –, –) has one of 0.0000000002. Therefore, the first combination has a higher weight, as probably expected by most users.

Prioritization Method The prioritization method contains the actual data which a `Reporter` implementation saves and the weight calculation method inside a `ModelModifier`. To this end, it contains all relevant combinations presented in Figure 5.3:

1. **FailResult**: `FailingCombinationCalculator` + `TestResultPersistingReporter`.
2. **FailResultN**: `NormalizingFailingCombinationCalculator` + `TestResultPersistingReporter`.
3. **FailFic**: `FailingCombinationCalculator` + `FicPersistingReporter`. Additionally, this uses the `Mixtgte` algorithm for identifying failure-inducing combinations.
4. **FailFicN**: `NormalizingFailingCombinationCalculator` + `FicPersistingReporter`. Additionally, this uses the `Mixtgte` algorithm for identifying failure-inducing combinations.
5. **SuspResult**: `SuspiciousComponentCalculator` + `TestResultPersistingReporter`.
6. **SuspResultN**: `NormalizingSuspiciousComponentCalculator` + `TestResultPersistingReporter`

Sometimes, the `Mixtgte` algorithm takes too long to find the relevant FICs and runs for multiple hours at a time until it finally results in an `OutOfMemoryException` since the internal representation of all test cases and the combinations saved in the FC algorithm is too large. To avoid this, the affected scenario (replace at strength three) was not executed with the two FIC-based prioritization methods.

Identification Each generator, point of prioritization, and prioritization method has a short identifier. If one combines the identifiers, one can identify every combination with a short name. For example, **AETG-After(0)-SuspResultN** means that the experiment uses the AETG generator together with a prioritizer which runs after the generation and has a default weight of zero. The prioritization approach calculates the weights themselves with the suspicious component method for weight calculation based on individual test case results and then normalizes them afterwards.

Measurements

To actually evaluate the prioritization approaches after executing all scenarios with all configurations, it is important to measure relevant information in the individual runs. For the actual prioritization it is important to measure which test case actually fails. Therefore, the first measurement of the evaluation is always the index (starting at zero) of the first failed test case and the first occurrence of all currently injects FICs for the synthetic scenarios. Additionally, it can be an issue that prioritized test suites get to large. This happens if the prioritization algorithm puts to much emphasis on covering important combinations and not on covering remaining t-value combinations. A second measurement is therefore always the initial test suite size. The third and last measurement is the time (in milliseconds) it took to generate a prioritized test suite. This is always the sum of the initial generation time and the time spent in the `TestInputPrioritizer`.

While also an interesting part of the runs, the evaluation does not include any measurements of the FC process, like the time spent computing additional test cases or the

number of test cases generated during fault characterization. This would only make sense if the evaluation also compared different FC algorithms and not only used `Mixtgte` to support the FIC-based weighting method.

6.2.2. Results

All in all, this evaluation includes 19 scenarios with about 25 runs each, up to 45 relevant configurations per scenario, and 3 measurements per run leading to (roughly) about $19 \times 25 \times 45 \times 3 = 64125$ individual data points. It is obviously not feasible to discuss and compare all of them to each other or even to present them all in this printable version of the thesis. Therefore, it does not include the synthetic example, except for the `ManyIterationsWithOccurenceProbability` as diagrams, but interested readers can find all data points in an excel file in the evaluation repository referenced at the beginning of this section under `src/main/results`. For all other scenarios, Chapter B in the appendix includes box blots for the index of the first failing test case and test suite size.

6.2.3. Discussion

With the synthetic scenarios, the outcome is largely as expected. In all cases, prioritizing for a specific combinations results in that combination being near the beginning of the test suite. Often, it even reaches index zero. Additionally, prioritizing according to multiple FICs works for all approaches, and every FIC is at a low index in a prioritized test suite. However, the evaluation also shows some problems in the prioritization approach. Often, the prioritization oversteers in one direction, and while previously failing combinations appear near the start, some other combinations get pushed too far towards the end of the test suite. This may make it necessary to revisit the test suite generation and change the focus more towards the CT coverage criterion and only perform prioritization if it does not do too much damage.

Regarding the generation time, there is no clear difference between DDA and IPOG. Both usually take less than 100 ms to generate a test suite with strength three. However, AETG is the clear exception. Even for the test suite at strength two, it takes over one second, and for strength three this even increase to over one minute. This is much too long for testing multiple configurations which include AETG for about 25 runs in later scenarios, which is why those scenarios no longer include it. With a more efficient implementation like the one for DDA from Section 5.3 it should however be possible to make AETG much faster than it currently is. If this is done, it is again possible to compare the algorithm to IPOG and DDA.

AETG also performs worse for all prioritization approaches in the first few scenarios which use the same FIC across multiple runs. This could be the case because the other algorithms either generate according to the weights (DDA) or always generate the same test cases, which makes this a normal regression test prioritization problem (IPOG), while AETG always generates different test cases and therefore "confuses" approaches based on failing test cases. In those approaches, the actually failure-inducing combination is not

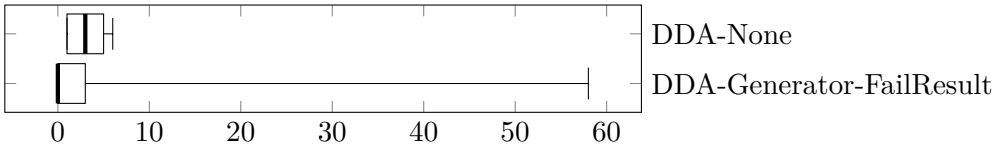

Figure 6.1.: Boxplot of first failure in prioritized and non-prioritized configurations from the schedule scenario at strength three

clear, therefore the prioritization algorithm running after AETG must treat all values in a failing test case with the same importance. However, if more values appear in a test case without a failure-inducing combination, this test case will receive a higher priority even though it does not cause a failure. With IPOG this does not happen because the test cases are always the same and therefore also contains the FIC, and DDA is also likely to re-generate the same test case if all values in it are weighted equally high. However, AETG does appear to be very successful in combination with FIC-based weighting method (**FailFic** or **FailFicN**) since this only weights the values actually responsible for the previous failure. In other scenarios which also contains multiple different FICs over many runs, FIC-based approaches are also better at prioritization, probably because they assign high weights to fewer values and therefore make the prioritization more specific to those few, often failure-inducing, values.

When looking at the test suite size, AETG is always the clear winner. For strength three, it usually requires about 168 test cases, while IPOG initially generates 184 and DDA without prioritization requires 181. However, when performing prioritization during the generation, DDA requires more test cases, sometimes even above 200 for the same model. This is due to the fact that prioritizing the combination is more important than quickly covering all combinations. Such a significant increase in test cases also makes prioritizing with DDA more unfeasible, since even if it catches previous FICs early on with a prioritized test suite it would require more test cases over multiple runs if some of those runs do not have any failure. Then, one needs to execute the complete (larger) test suite completely.

The results of evaluating the scenarios based on real programs largely support the preliminary findings. In general, the configurations which enable prioritization nearly always have a lower median index of first failure than the non-prioritized ones. Figure 6.1 shows how a typical box diagram of a non-prioritized and prioritized configuration look for the schedule scenario at strength three, but Figures B.4, B.7, B.8, and B.9 in the appendix also make this particularly clear. However, at the same time, the prioritized configurations also usually have a higher maximal index of first failure. This often happens in the first few runs of one experiment when one run already discovered a failure and prioritized too much in the direction of the first failure so the next different failing combination is then at the end of the test suite. Changing the weight calculation or prioritization algorithm to include a higher default weight may revise this issue.

What this also means, is that it is mostly the first few runs which produce the high failure indexes as shown in Figure 6.2. If the box plots were made without, for example,

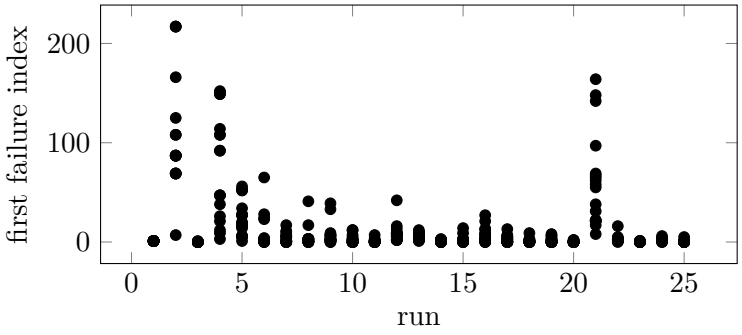

Figure 6.2.: Scatter diagram of the first failure index per run over all prioritizing configurations with DDA from the printtokens2 scenario at strength three

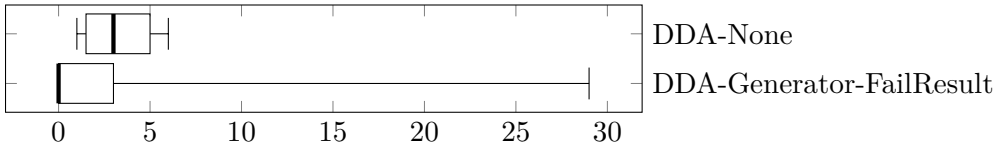

Figure 6.3.: Boxplot of first failure in prioritized and non-prioritized configurations from the schedule scenario at strength three without the first ten runs

the first ten runs, the prioritized version would be even better than the non-prioritized one. Figure 6.3 shows boxplots for the same configurations as Figure 6.1, but the first five runs were left out. The prioritized configuration immediately looks better. In a real world scenario, the question is how often the tests discover a combination which did not fail before. If this occurs fairly often, the prioritized variant may often require a high number of test cases to detect this new combination.

One prioritization method which suffers more from high maximas and upper quantile is the FIC-based method. Figure 6.4 demonstrates this for six configurations tested with the printtokens2 scenario. This may have two different explanations. First, this could be the case because the FIC-based prioritization method naturally prioritizes for a much more specific combination than the test result based ones since it only focuses on the values actually responsible for a previous failure. Therefore, fewer values have high weights, which may cause the algorithm to prioritize too much in one specific direction. The other explanation is that FC can also fail. If this is the case, they identify a wrong combination as failure-inducing, and later runs will prioritize the wrong values. The prioritization methods based directly on test results do not have this problem since they would prioritize all values in the failing test case and therefore automatically also prioritize the ones which form a failure-inducing combination. For this reason it is very important to have a good FC algorithm when performing prioritization based on its results to avoid a case of garbage in $\Rightarrow$ garbage out.

Regarding the other two prioritization methods, the results do not show that one method is clearly better than the other when used with DDA, but for IPOG the method

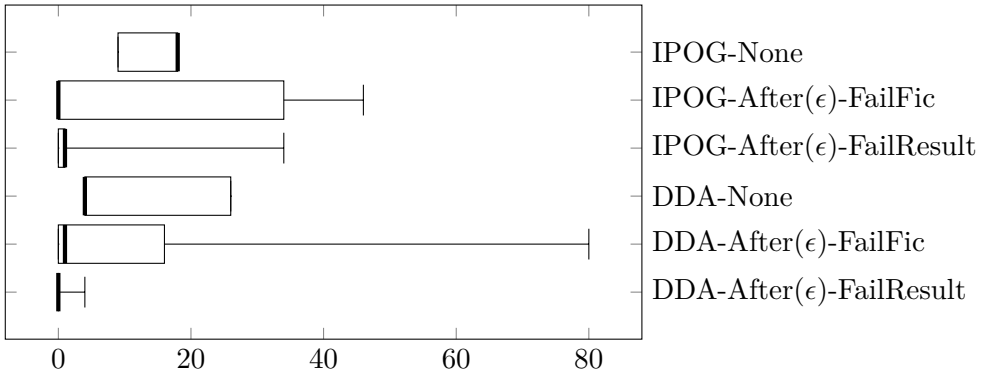

Figure 6.4.: Boxplot of first failure of FIC-based prioritization methods from the schedule printtokens at strength two

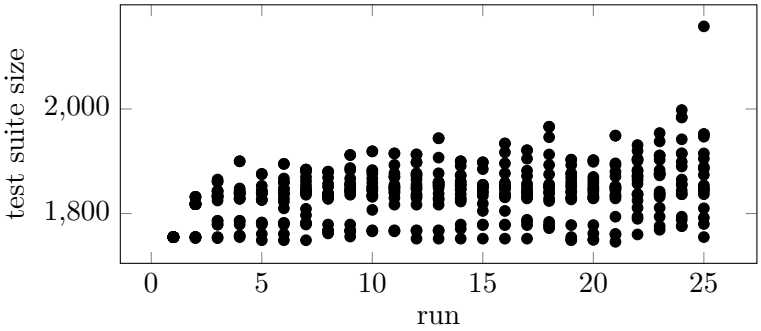

Figure 6.5.: Scatter diagram of the test suite size per run over all prioritizing configurations with DDA from the printtokens2 scenario at strength three

based solely on failing test cases sometimes performs better than the one which rates components regarding their suspiciousness. A possible explanation for this observation is the fact that IPOG always deterministically generates the same test suite. The FailResult or FailResultN method therefore perform better as they only assign values to failing test cases which the prioritization step can then just reorder. On the other hand, a prioritization method based on suspicious components may break down the responsibility of the failure to specific values and then, if it comes to the wrong conclusion, prioritize values not responsible for the failure. With DDA these factors are not as apparent as the whole test suite is seldom the same between two runs.

Normalizing the weights does not seem to make much of a difference. While there are scenarios where the normalizing variant is better than the non-normalized one, there are also equally many where it is the other way around.

When looking at the point of prioritization, prioritizing during or after the generation does not make a consistent difference, though DDA often has a lower median than IPOG. This fact has to be taken with a grain of salt though, since DDA also generates much more test cases and may therefore be worse when no test case fails. Figure 6.5 shows

the number of test cases per run generated by DDA during runs where configuration was enabled. Increases by 10% are not seldom, and with test suites including over one thousand test cases this results in significantly longer test suite execution time. However, prioritizing afterwards with a default weight of ϵ is often slightly better than prioritizing with a default of zero. Figures B.3, B.4, and B.6 in the appendix show this.

All in all, the evaluation shows that the methods presented in Section 4.2 work and lead to faster error detection in the evaluated scenarios. However, the prioritization also leads to larger test suites for DDA which partly negates the effect. For IPOG, the results show the prioritization method based on failing test cases as the best one in combination with prioritizing after generation with a default weight of ϵ. Currently, the recommendation would therefore be to use either **IPOG-After(ϵ)-FailResult** or **IPOG-After(ϵ)-FailResultN**. However, actually using those methods in production requires more studies on real world systems to better understand their failure characteristics.

7. Conclusion

To conclude the thesis, this chapter will first give a summary of what the last chapters presented and how it affects current research. Afterwards, it gives an outlook to future areas of research in the field of combinatorial test case prioritization based on the contributions of this thesis.

Summary

This thesis added several contributions to the field of CTCP. First, Chapter 2 gave an overview of the current state of research in related topics such as regression test optimization and traditional combinatorial testing. In addition to the basics of CT it also introduced advanced topics such as fault characterization with the example of BEN, an algorithm which assesses the probability of values being involved in a failure-inducing combination. Finally, the chapter gave a brief overview of the current state of coffee4j, an automatic CT framework written in Java.

Chapter 3 then continued with an overview of current research specific to failure-based regression test optimization and combinatorial test case prioritization. This identified several gaps in research, such as a combination of both approaches in the form of failure-based CTCP and a framework for performing and comparing different prioritization approaches. Furthermore, it identified a possible improvement to DDA, an algorithm for generating prioritized combinatorial test suites, by making it aware of constraints.

A concept for filling those research gaps was the topic of Chapter 4. First, it collected requirements which a framework for combinatorial testing must fulfill in order for it to be useful in use and research of CT. Next, it integrated those requirements into the concepts behind coffee4j, such as the combinatorial testing process, the general architecture with its possible extension points, and its domain model. The next section then focused on presenting a new failure-based combinatorial test case prioritization approach. This approach saves either test case results or FICs and then calculates value weights based on how likely it is for a value to cause a failure, the idea being that failures are likely to be in areas related to earlier ones. For calculating the value weights, this thesis presented three approaches, one of them relying on BEN's notion of failure probability of values. Finally, the last section of Chapter 4 then extended DDA to support arbitrary constraints on the input space.

At this point, all presented ideas were theoretical concepts. To ensure their practicability, Chapter 5 integrated these concepts into coffee4j. Additionally, it gave an overview of the necessary steps to develop new prioritization approaches and implement them using the coffee4j API.

Chapter 6 then performed an evaluation of the concepts to answer the three research

questions. To this end, the first section verified that the coffee4j framework now supports all different kinds of prioritization approaches currently in use (**RQ3**). For the evaluation of failure-based CTCP the next section performed several experiments — some based on real world programs — to ensure realistic failure scenarios. All in all, the results of the evaluation shows that failure-based prioritization detects changes faster (**RQ2**) but prioritizing during or after generation does not make much of a difference (**RQ3**). The best configuration seems to be using the IPOG algorithm, prioritizing after generation, and basing value weights only on previously failed test cases.

Future Work

As nearly all research contributions, this one can also be the basis of further research. Therefore, this section presents a few areas requiring future work.

Evaluation with real consecutive program versions While the evaluation in Chapter 6 did use real program versions and therefore also had realistic failure scenarios, it selected those versions randomly since they were not real consecutive versions. Therefore, it is necessary to re-evaluate the failure-based prioritization approach using actual programs with consecutive versions. The challenge for this lies in finding a suitable existing program and developing combinatorial tests for it including a test oracle.

Combination of different weight-based prioritization techniques Chapter 3 referenced a prioritization approach by Qu et al. based on code coverage information [QC13]. To integrate this information into the CT process, it also uses value weights. Similarly, other information could potentially also employ value weights, thereby giving multiple prioritization approaches a common base. An interesting question is then how one can integrate those approaches with one another. For example, one can prioritize with both failure-based and code coverage information.

Comparison of different prioritization approaches in coffee4j coffee4j now technically supports all different kinds of prioritization approaches. A next step referenced by previous chapters would be to implement existing ones and then compare them in different scenarios to see which one performs better under real-world conditions. This helps with evaluating approaches and checking in which direction future work in the whole field could go. To get those scenarios, it would again be necessary to find a fitting program which can be tested using combinatorial testing.

A. Weight-based Prioritization with Constraints Example

Combination					Weight	Covering test cases
Dot	Star	Exact	Plus	Range		
M	M				0.48	1, 4, 8
M	E				0.12	3, 6, 7
E	M				0.32	2, 5, 9
E	E				0.08	Not valid
M		S			0.36	3
M		E			0.18	1, 8
M		M			0.06	4, 6, 7
E		S			0.24	2
E		E			0.12	Not valid
E		M			0.04	5, 9
M			S		0.45	1, 7
M			M		0.15	3, 4, 6, 8
E			S		0.3	5
E			M		0.1	2, 9
M				S	0.18	4, 6, 8
M				M	0.42	1, 3, 7
E				S	0.12	9
E				M	0.28	2, 5
	M	S			0.48	2
	M	E			0.24	1, 8
	M	M			0.08	4, 5, 9
	E	S			0.12	3
	E	E			0.06	Not valid
	E	M			0.02	6, 7
	M		S		0.6	1, 5
	M		M		0.2	2, 4, 8, 9
	E		S		0.15	7
	E		M		0.05	3, 6

Combination					Weight	Covering test cases
Dot	Star	Exact	Plus	Range		
	M			S	0.24	4, 8, 9
	M			M	0.56	1, 2, 5
	E			S	0.06	6
	E			M	0.14	3, 7
		S	S		0.45	Not valid
		S	M		0.15	2, 3
		E	S		0.225	1
		E	M		0.075	8
		M	S		0.075	5, 7
		M	M		0.025	4, 6, 9
		S		S	0.18	Not valid
		S		M	0.42	2, 3
		E		S	0.09	8
		E		M	0.21	1
		M		S	0.03	4, 6, 9
		M		M	0.07	5, 7
			S	S	0.225	Not valid
			S	M	0.525	1, 5, 7
			M	S	0.075	4, 6, 8, 9
			M	M	0.175	2, 3

Table A.1.: All two-way combinations for the example in Section 4.3.2 and a list of test cases which cover this combination

B. Evaluation Results

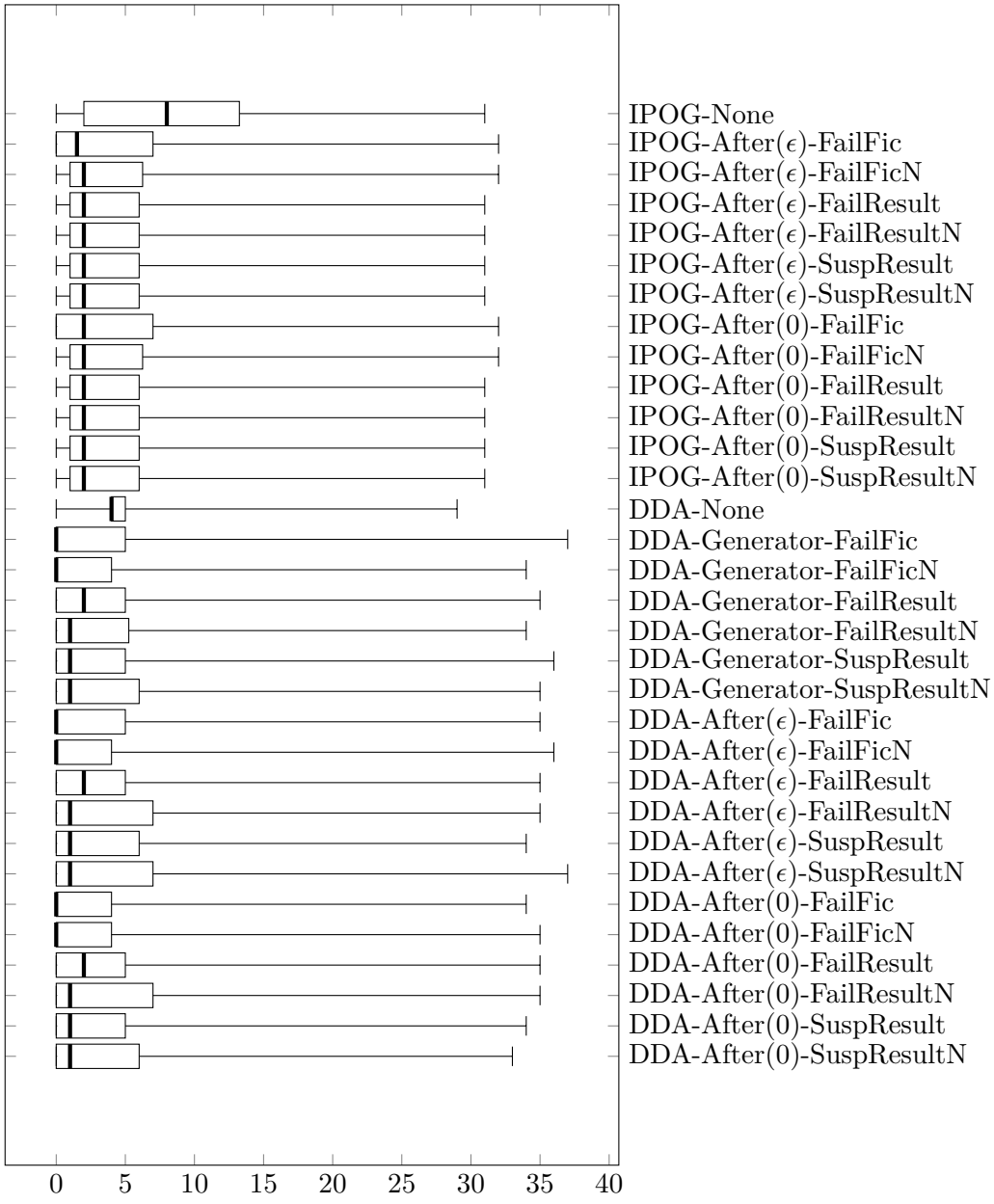

Figure B.1.: Boxplot of first failure for probability-based scenario strength two

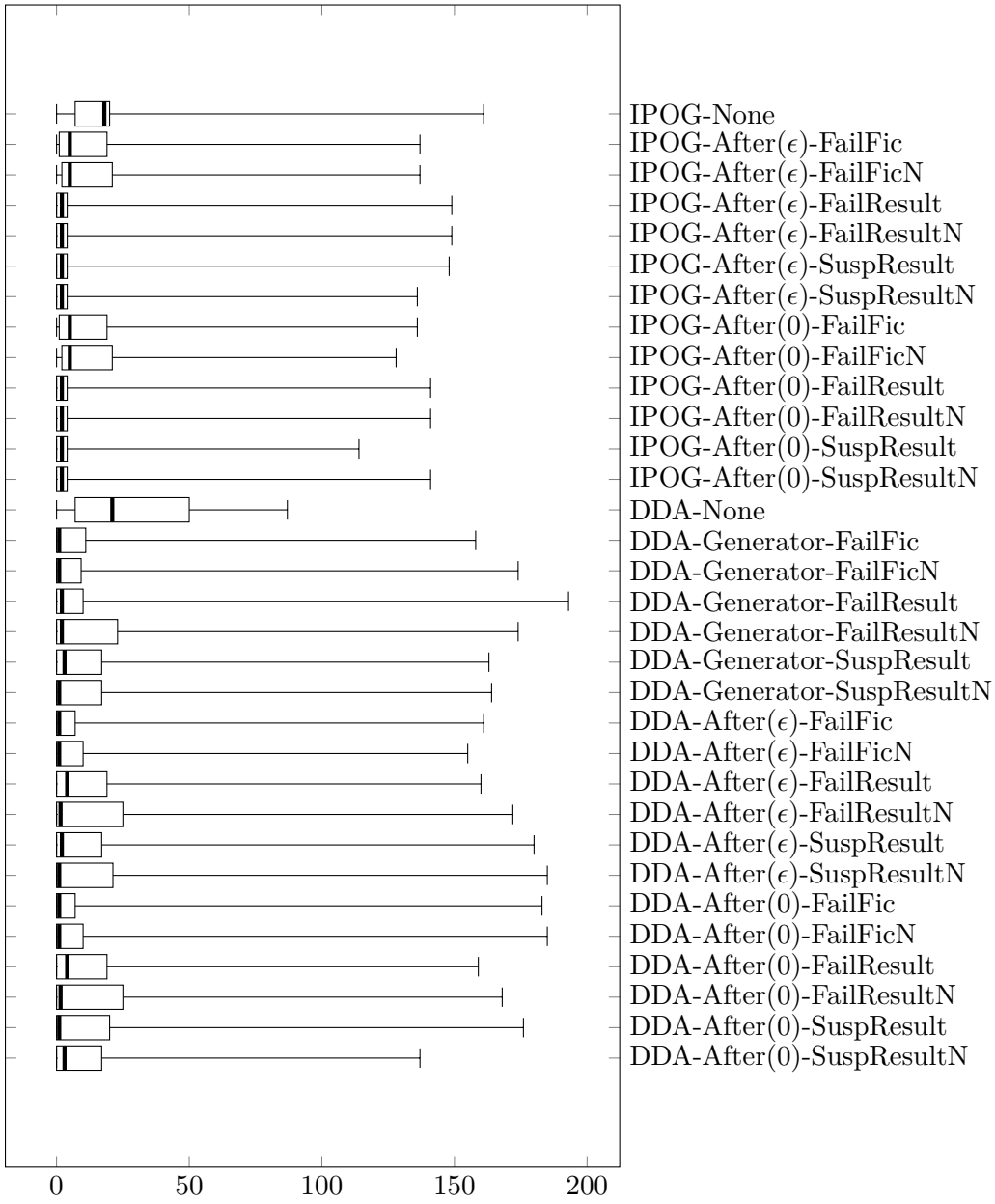

Figure B.2.: Boxplot of first failure for probability-based scenario strength three

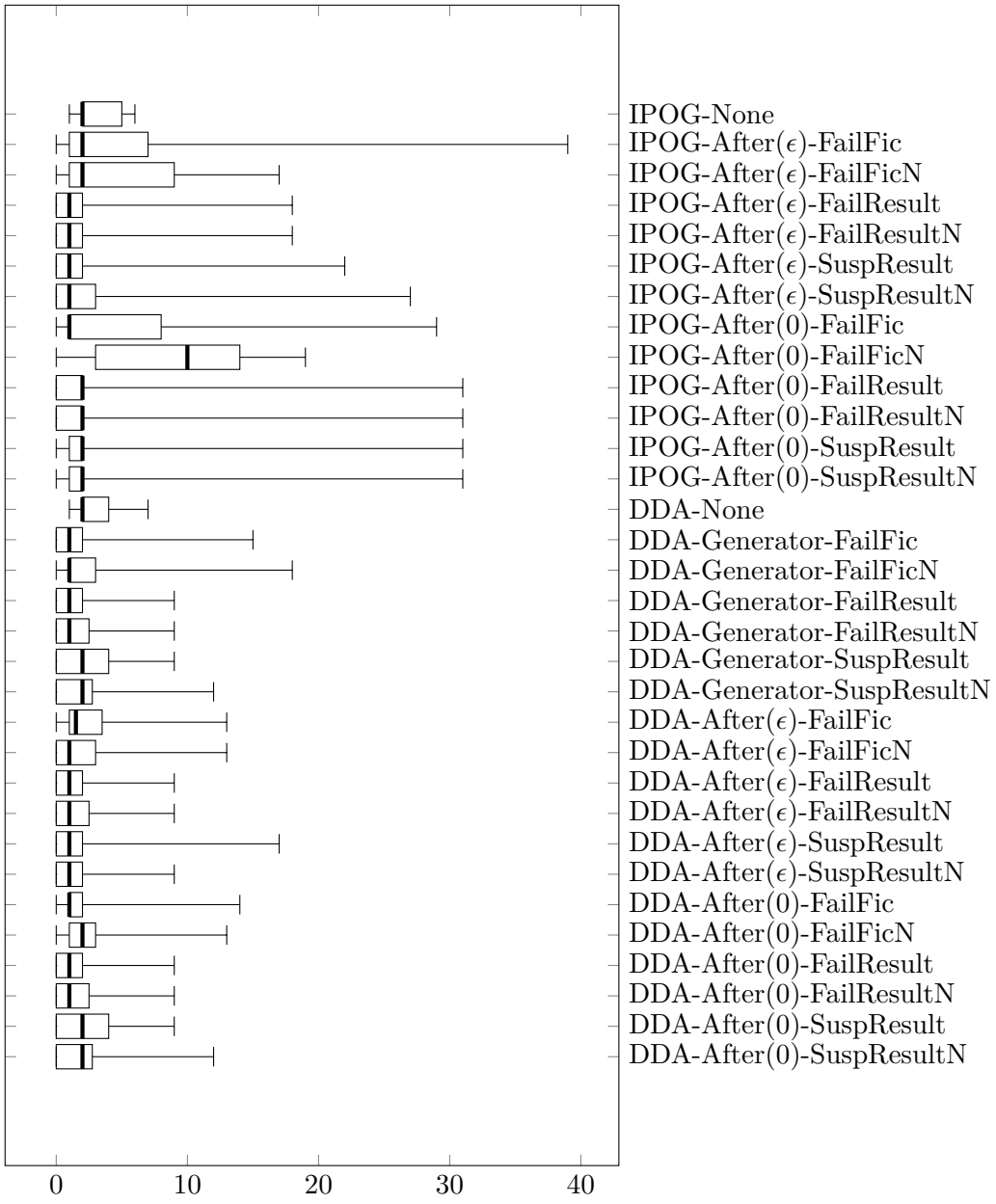

Figure B.3.: Boxplot of first failure for schedule scenario strength two

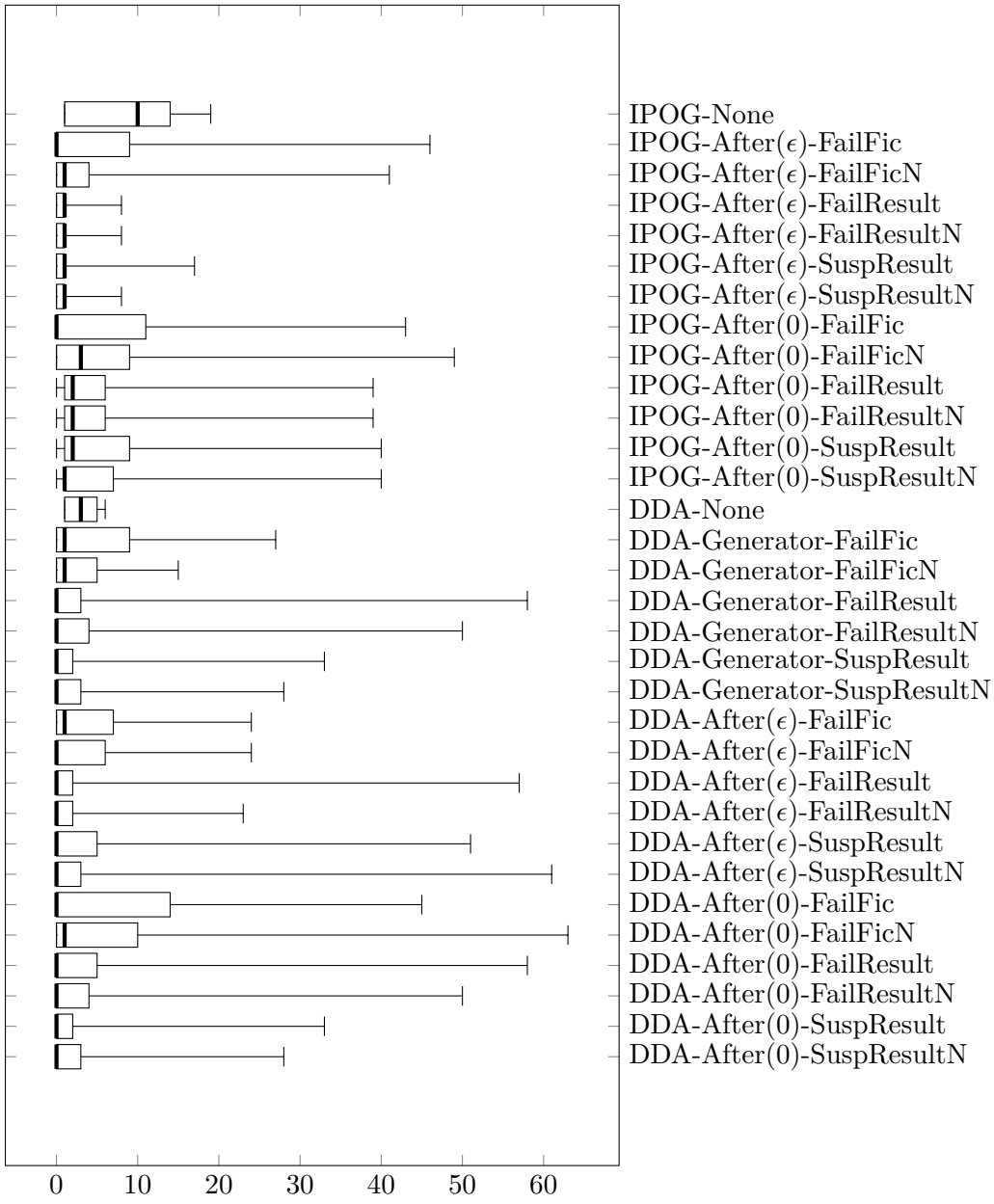

Figure B.4.: Boxplot of first failure for schedule scenario strength three

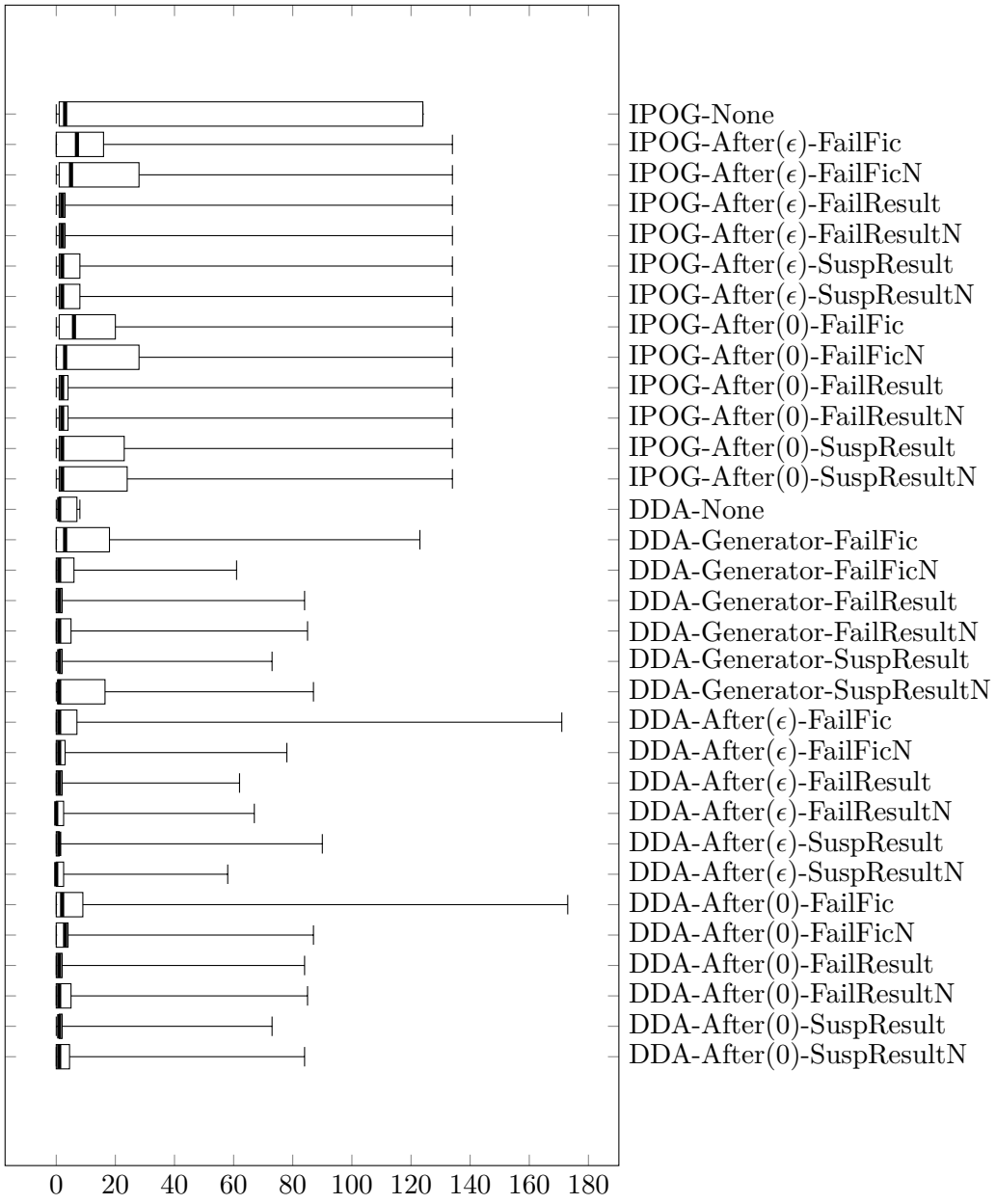

Figure B.5.: Boxplot of first failure for replace scenario strength two

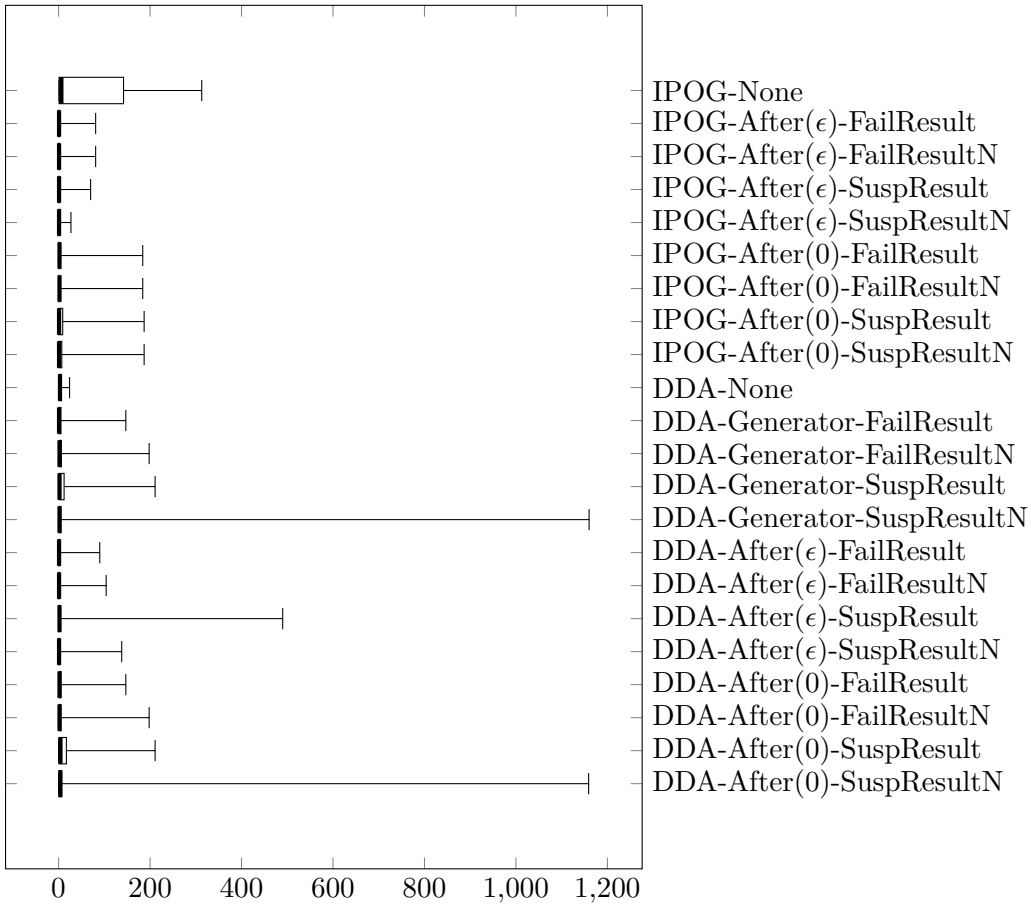

Figure B.6.: Boxplot of first failure for replace scenario strength three. This does not include techniques based on fault characterization since they ran into `OutOfMemoryErrors`

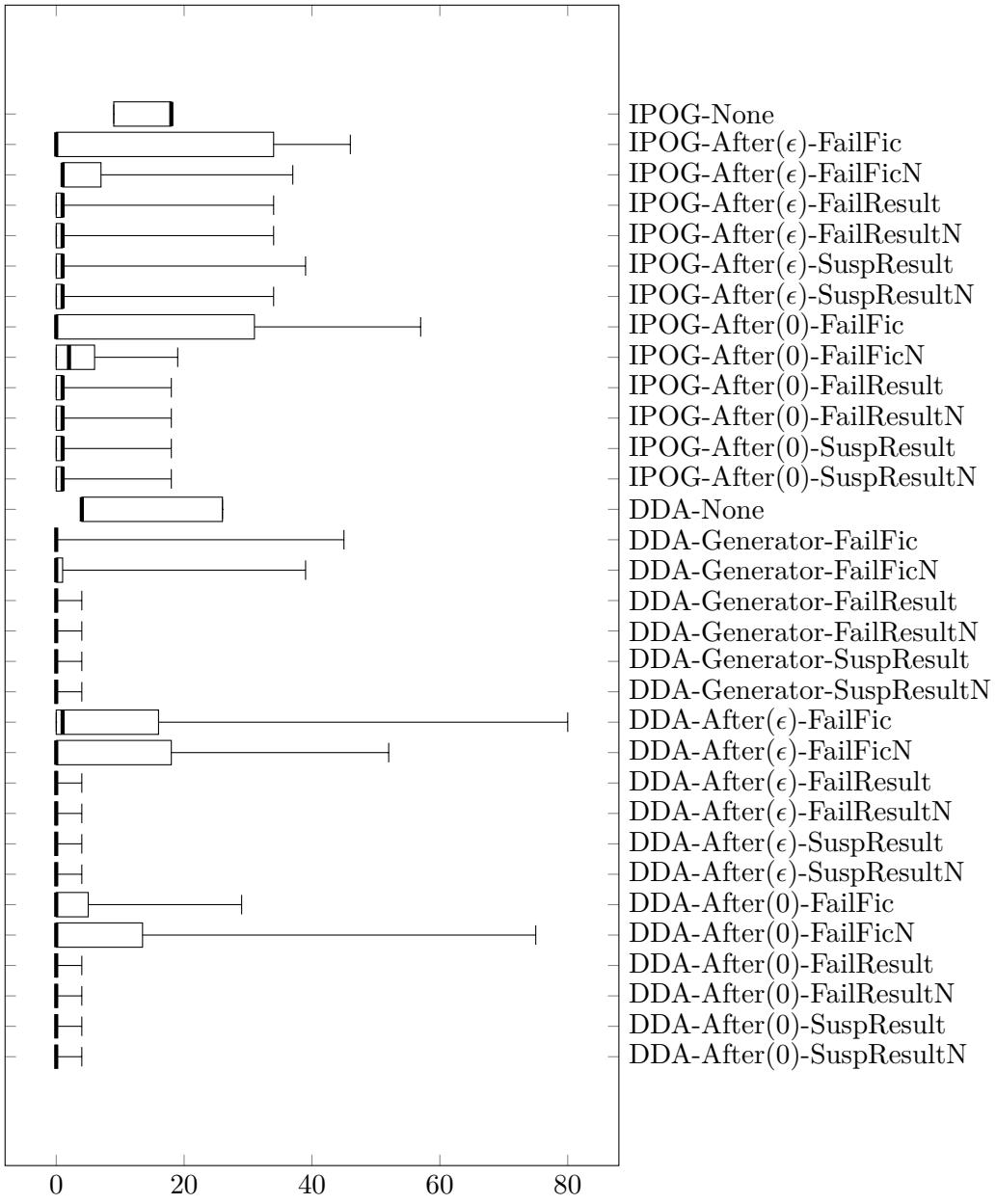

Figure B.7.: Boxplot of first failure for printtokens scenario strength two

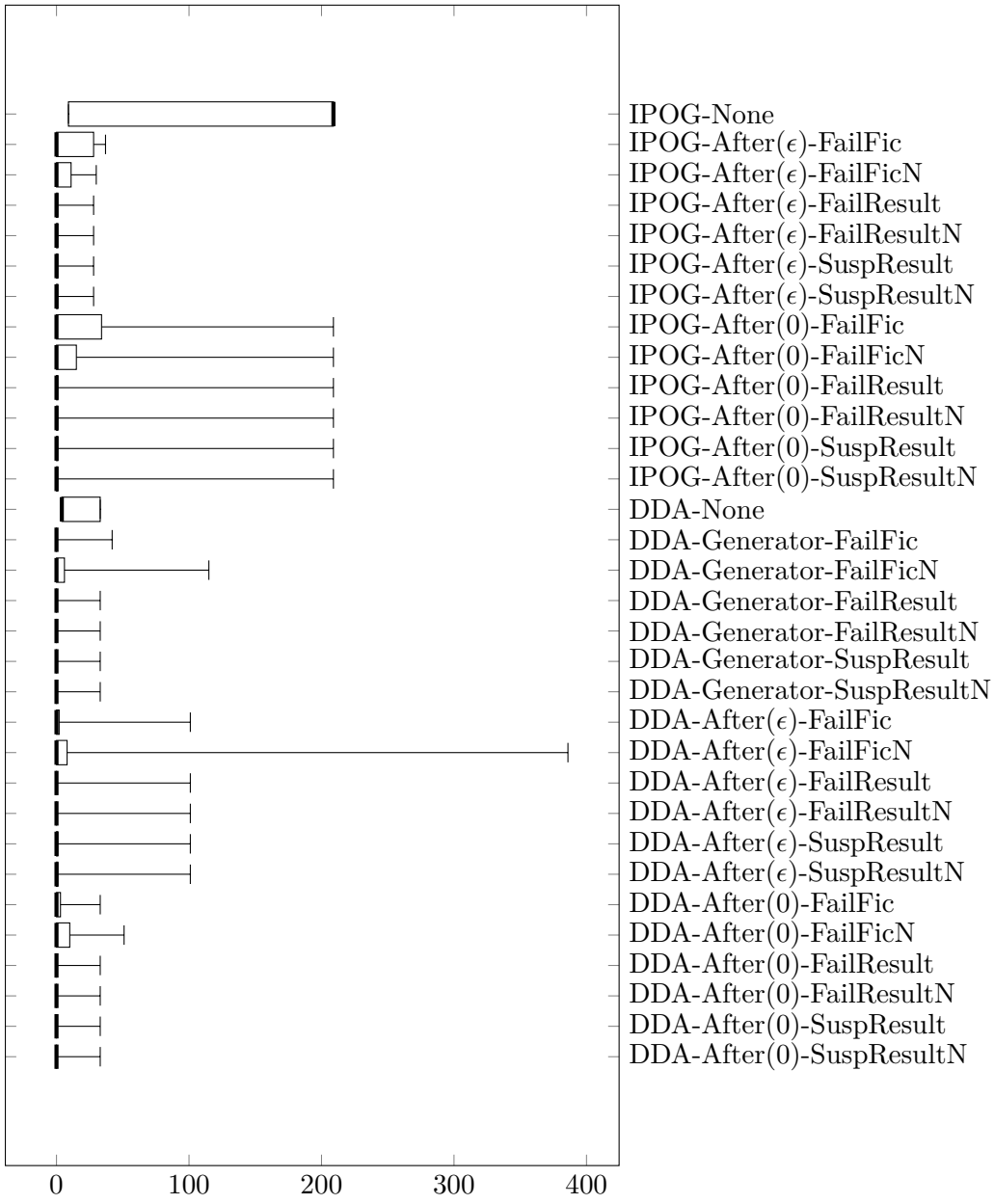

Figure B.8.: Boxplot of first failure for printtokens scenario strength three

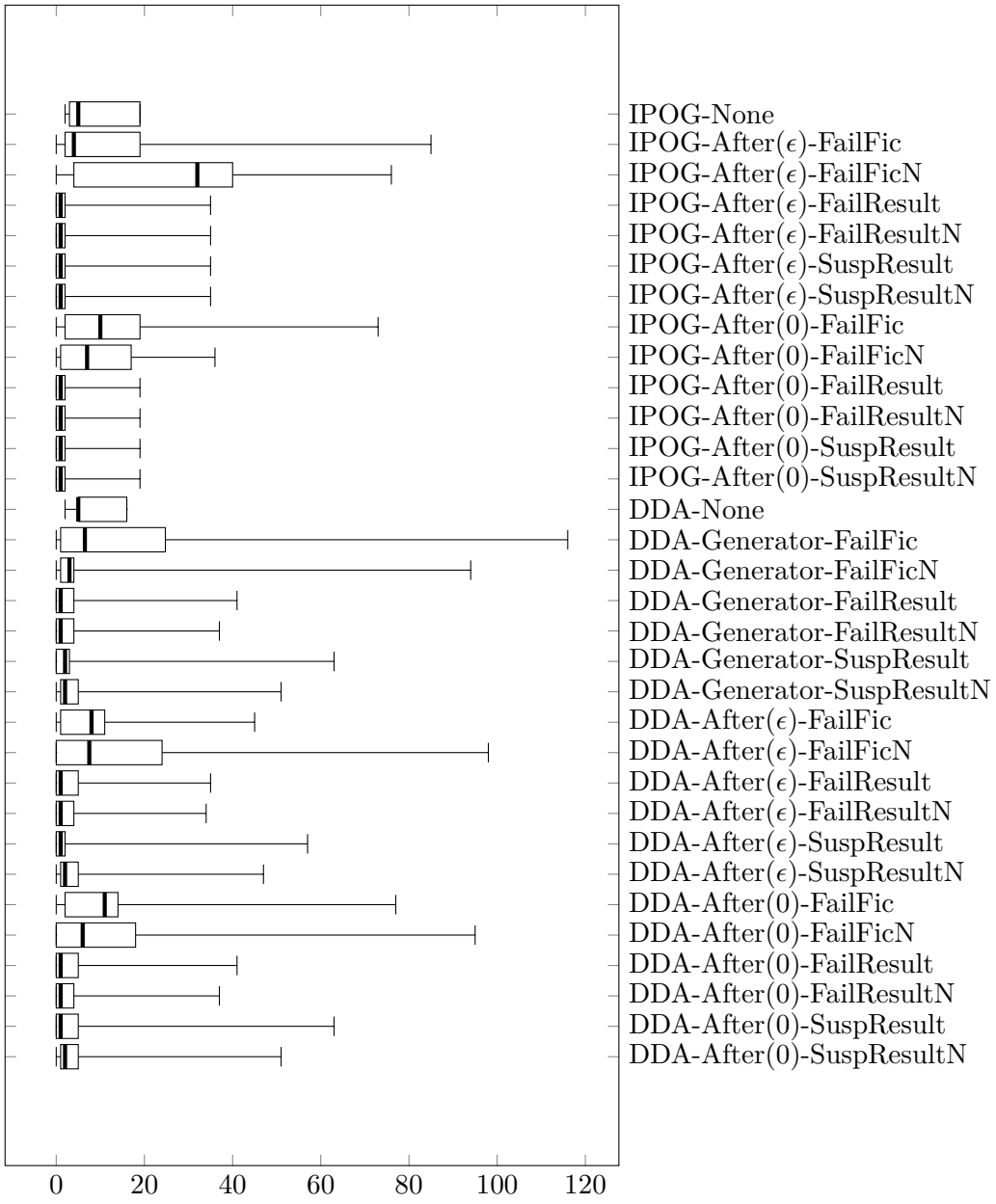

Figure B.9.: Boxplot of first failure for printtokens2 scenario strength two

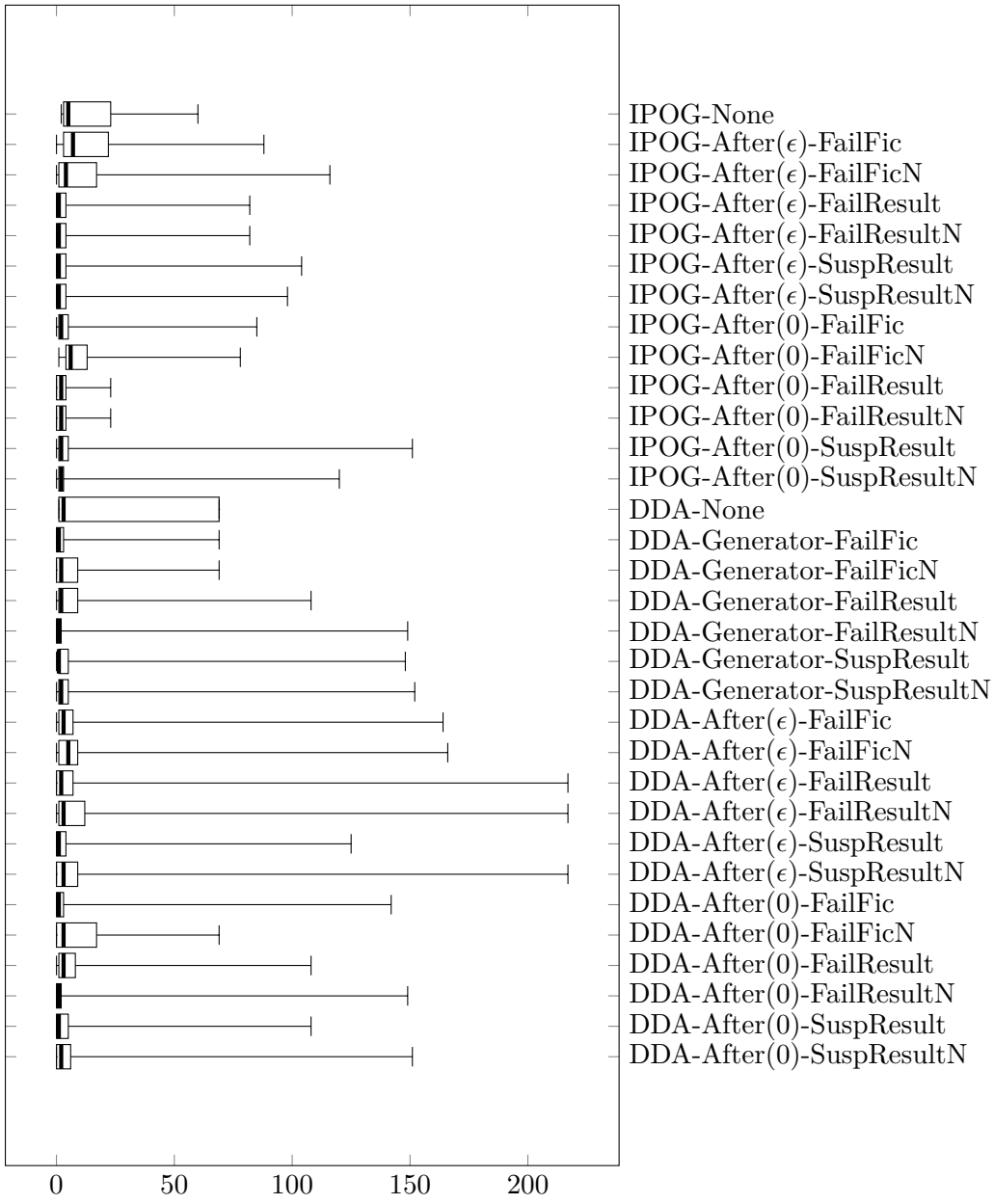

Figure B.10.: Boxplot of first failure for printtokens2 scenario strength three

Bibliography

[AG+12] H. Avila-George et al. “Simulated Annealing for Constructing Mixed Covering Arrays”. In: *Distributed Computing and Artificial Intelligence*. Ed. by S. Omatu et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 657–664 (cited on page 12).

[AGR19] P. Arcaini, A. Gargantini, and M. Radavelli. “Efficient and Guaranteed Detection of t-Way Failure-Inducing Combinations”. In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 200–209 (cited on pages 14, 41).

[Ali+20] S. Ali et al. “Enhanced regression testing technique for agile software development and continuous integration strategies”. In: *Software Quality Journal* 28 (2020), pp. 397–423 (cited on pages 24, 42).

[BC05] R. C. Bryce and C. J. Colbourn. “Test Prioritization for Pairwise Interaction Coverage”. In: *Proceedings of the 1st International Workshop on Advances in Model-Based Testing*. A-MOST ’05. St. Louis, Missouri: Association for Computing Machinery, 2005, 1–7 (cited on pages 2, 12, 24, 25, 34, 45).

[BC06] R. C. Bryce and C. J. Colbourn. “Prioritized interaction testing for pairwise coverage with seeding and constraints”. In: *Information and Software Technology* 48.10 (2006), pp. 960–970 (cited on pages 25, 49).

[BC07] R. C. Bryce and C. J. Colbourn. “The density algorithm for pairwise interaction testing”. In: *Software Testing, Verification and Reliability* 17.3 (2007), pp. 159–182 (cited on pages 12, 25, 34, 45).

[Bec02] K. Beck. *Test Driven Development. By Example*. Pearson, 2002 (cited on page 4).

[Bec+20] S. Bechtold et al. *JUnit 5 User Guide*. `https://junit.org/junit5/docs/current/user-guide/`. Last retrieved 2020-09-21. 2020 (cited on pages 18, 19, 21).

[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 (cited on page 18).

[Ber19] L. Bernwald. “Development of an Automated Combinatorial Testing Framework”. Bachelor Thesis. RWTH Aachen University, 2019 (cited on pages 1, 19–21, 28, 35, 56, 57, 65, 71).

[BFL19] J. Bonn, K. Foegen, and H. Lichter. “A Framework for Automated Combinatorial Test Generation, Execution, and Fault Characterization”. In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 224–233 (cited on pages 1, 13, 19–21).

[Bon18] J. Bonn. “Automated Fault Localization for Combinatorial Testing”. Bachelor Thesis. RWTH Aachen University, 2018 (cited on pages 1, 7, 8, 11, 14, 19, 21, 28, 40, 45, 60, 61, 65, 67).

[BRO13] M. Böhme, A. Roychoudhury, and B. Oliveira. *Regression Testing of Evolving Programs*. 2013 (cited on page 4).

[Cho+16] E. Choi et al. “Test Effectiveness Evaluation of Prioritized Combinatorial Testing: A Case Study”. In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2016, pp. 61–68 (cited on pages 2, 24).

[CKL16] Y. Cho, J. Kim, and E. Lee. “History-Based Test Case Prioritization for Failure Information”. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 2016, pp. 385–388 (cited on page 23).

[Col04] C. J. Colbourn. “Combinatorial aspects of covering arrays”. In: *Le Matematiche* 59.1, 2 (2004), pp. 125–172 (cited on pages 10, 12).

[Cze06] J. Czerwonka. “Pairwise Testing in Real World. Practical Extensions to Test Case Generators”. In: 2006 (cited on pages 11, 31).

[DER05] H. Do, S. Elbaum, and G. Rothermel. “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact”. In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435 (cited on page 85).

[DS08] G. Duggal and B. Suri. “Understanding Regression Testing Techniques”. In: (Jan. 2008) (cited on page 5).

[Fö+20] K. Fögen et al. *Coffee4j*. `https://coffee4j.github.io/`. Last retrieved 2020-09-21. 2020 (cited on page 1).

[FL19] K. Fögen and H. Lichter. “Combinatorial Robustness Testing with Negative Test Cases”. In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019, pp. 34–45 (cited on pages 1, 60).

[FL20] K. Fögen and H. Lichter. “Generation of Invalid Test Inputs from Over-Constrained Test Models for Combinatorial Robustness Testing”. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 171–180 (cited on pages 1, 9).

[Gha+12] L. S. G. Ghandehari et al. “Identifying Failure-Inducing Combinations in a Combinatorial Test Set”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 370–379 (cited on pages 1, 13–15, 48).

[Gha+13] L. S. G. Ghandehari et al. “Applying Combinatorial Testing to the Siemens Suite”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 362–371 (cited on pages 9, 52, 86).

[Gha+15] L. S. G. Ghandehari et al. “BEN: A combinatorial testing-based fault localization tool”. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015, pp. 1–4 (cited on pages 12, 14, 48).

[Gli+14] M. Gligoric et al. “An Empirical Evaluation and Comparison of Manual and Automated Test Selection”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: Association for Computing Machinery, 2014, 361–372 (cited on page 5).

[GOA05] M. Grindal, J. Offutt, and S. F. Andler. “Combination testing strategies: a survey”. In: *Software Testing, Verification and Reliability* 15.3 (2005), pp. 167–199 (cited on pages 9, 12).

[GOM07] M. Grindal, J. Offutt, and J. Mellin. “Managing Conflicts When Using Combination Strategies to Test Software”. In: *2007 Australian Software Engineering Conference (ASWEC’07)*. 2007, pp. 255–264 (cited on pages 9, 10, 12).

[HC02] C. S. Horstmann and G. Cornell. *Core Java 2: Volume I, Fundamentals, Sixth Edition*. 6th. Pearson Education, 2002. ISBN: 0130471771 (cited on page 17).

[Hu+20] L. Hu et al. “How does combinatorial testing perform in the real world: an empirical study”. In: *Empirical Software Engineering* 25.4 (July 2020), pp. 2661–2693 (cited on page 1).

[Hua+13] R. Huang et al. “Prioritization of Combinatorial Test Cases by Incremental Interaction Coverage”. In: *International Journal of Software Engineering and Knowledge Engineering* 23.10 (2013), pp. 1427–1457 (cited on pages 2, 17, 24).

[JP02] Jung-Min Kim and A. Porter. “A history-based test prioritization technique for regression testing in resource constrained environments”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, pp. 119–129 (cited on pages 23, 42).

[JPR00] Jung-Min Kim, A. Porter, and G. Rothermel. “An empirical study of regression test application frequency”. In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. 2000, pp. 126–135 (cited on page 4).

[JST20] H. Jin, C. Shi, and T. Tsuchiya. “Constrained Detecting Arrays for Fault Localization in Combinatorial Testing”. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC ’20. Brno, Czech Republic: Association for Computing Machinery, 2020, 1971–1978 (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 (cited on pages 1, 8, 10, 11).

[Lei+07] Y. Lei et al. “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 pages 10, 73, 74).

[MGS13] D. Marijan, A. Gotlieb, and S. Sen. “Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study”. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 540–543 (cited on pages 5, 23, 42).

[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 pages 4, 6, 7).

[NL11] C. Nie and H. Leung. “A Survey of Combinatorial Testing”. In: *ACM Comput. Surv.* 43.2 (Feb. 2011), 11:1–11:29 (cited on pages 1, 7–12, 16).

[NMT12] C. D. Nguyen, A. Marchetto, and P. Tonella. “Combining Model-Based and Combinatorial Testing for Effective Test Case Generation”. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, 100–110 (cited on page 9).

[OC20] Oracle Corporation. *Go Java*. `https://go.java/index.html`. Last retrieved 2020-09-21. 2020 (cited on page 18).

[OSH04] A. Orso, N. Shi, and M. J. Harrold. “Scaling Regression Testing to Large Software Systems”. In: *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. SIGSOFT ’04/FSE-12. Newport Beach, CA, USA: Association for Computing Machinery, 2004, 241–251 (cited on page 5).

[Ple15] C. Plewnia. “A Framework for Regression Test Prioritization and Selection”. Master Thesis. RWTH Aachen University, 2015 (cited on pages 5, 6).

[QC13] X. Qu and M. B. Cohen. "A Study in Prioritization for Higher Strength Combinatorial Testing". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 285–294 (cited on pages 2, 5, 8, 12, 16, 17, 24, 25, 34, 42, 43, 45, 96).

[RH96] G. Rothermel and M. J. Harrold. "Analyzing regression test selection techniques". In: *IEEE Transactions on Software Engineering* 22.8 (1996), pp. 529–551 (cited on page 5).

[Rot+01] G. Rothermel et al. "Prioritizing test cases for regression testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948 (cited on pages 5, 6).

[Rot+99] G. Rothermel et al. "Test case prioritization: an empirical study". In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 1999, pp. 179–188 (cited on page 6).

[RW06] R. Ramler and K. Wolfmaier. "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost". In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST '06. Shanghai, China: Association for Computing Machinery, 2006, 85–91 (cited on page 4).

[Sam+11] S. Sampath et al. "A tool for combination-based prioritization and reduction of user-session-based test suites". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 574–577 (cited on page 26).

[SAZ17] M. Shahin, M. Ali Babar, and L. Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943 (cited on pages 1, 4).

[SE04] D. Saff and M. D. Ernst. "An Experimental Evaluation of Continuous Testing during Development". In: *SIGSOFT Softw. Eng. Notes* 29.4 (July 2004), 76–85 (cited on pages 1, 4).

[Ste+04] J. M. Stecklein et al. "Error Cost Escalation Through the Project Life Cycle". In: (Toulouse, France, June 19, 2004–June 24, 2004). INCOSE Foundation, 2004 (cited on pages 3, 4).

[Sul+04] K. Sullivan et al. "Software Assurance by Bounded Exhaustive Testing". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: Association for Computing Machinery, 2004, 133–142 (cited on page 3).

[Tud20] C. Tudose. *JUnit in Action. Third Edition*. Manning Publications, 2020 (cited on pages 18, 19).

[YT98] Yu Lei and K. C. Tai. "In-parameter-order: a test generation strategy for pairwise testing". In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*. 1998, pp. 254–261 (cited on pages 11, 12).

[Yu+13] L. Yu et al. "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 242–251 (cited on page 10).

[Zhe+16] W. Zheng et al. "Locating Minimal Fault Interaction in Combinatorial Testing". In: *Advances in Software Engineering* 2016 (2016) (cited on page 13).

Glossary

Notation	Description
AETG	Automatic Efficient Test Generator
API	application programming interface
CT	combinatorial testing
CTCP	combinatorial test case prioritization
DDA	Deterministic Density Algorithm
EPC	event-driven process chain
FC	fault characterization
FIC	failure-inducing combination
IPM	input parameter model
IPOG	In-Parameter-Order-General
RTO	regression test optimization
RTP	regression test prioritization
RTS	regression test selection
SAT	satisfiability
SIR	Software-artifact Infrastructure Repository
SUT	system under test
TDD	test driven development
UML	Unified Modeling Language