The present work was submitted to
the Research Group
Software Construction

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

Bachelor Thesis

Evaluation of Regression Test Optimization Techniques

Evaluierung von Techniken zur Optimierung von Regressionstests

presented by

Lukas Schade

Aachen, September 4, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Christian Plewnia, 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 Bachelor's thesis entitled

Evaluation of Regression Test Optimization Techniques

independently and without illegitimate assistance from third parties. I have use 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 Bachelorarbeit mit dem Titel

Evaluation of Regression Test Optimization Techniques

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 4, 2018 (Lukas Schade)

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 4, 2018 (Lukas Schade)

Acknowledgment

First of all, I want to thank Christian Plewnia for being a great supervisor. He always found time for my questions and provided valuable feedback.

Furthermore, I want to thank Prof. Dr. rer. nat. Horst Lichter for giving me the opportunity to write this thesis at the Research Group Software Construction. Additionally, I want to thank Prof. Dr. rer. nat. Bernhard Rumpe for agreeing to review my thesis.

I want to thank my friends who proofread this thesis and provided helpful feedback. Last but not least, I want to thank my parents for supporting me.

Thank you!

Lukas Schade

Abstract

Regression testing is an important part of software development. Though it helps to uncover bugs, long running test suites prevent short feedback loops and developers get distracted by other tasks in the meantime. The consequences are increasing costs for software development. This is why the regression testing process needs to be optimized. One possibility to do so are *Regression Test Optimization* (RTO) techniques. These select or prioritize tests by some criterion. In general the goal is to find failing tests faster. However, RTO is rarely applied in practice. On the one hand this may be related to the lack of tool support, but on the other hand research so far mostly evaluated RTO techniques in artificial setups. Thus, it is unclear whether RTO provides a benefit in real-world projects.

This thesis presents an evaluation of a selection and a prioritization technique on open-source projects with faults that were detected in the past. The selection technique selects tests that cover changed parts of the source code. As prioritization technique the well studied additional coverage technique is chosen. It sorts tests by the additionally covered code. Additionally, a technique that uses information from the last test run to prioritize tests is implemented. This technique is not evaluated due to a lack of appropriate data.

Three open-source projects with in total 165 versions are used for the evaluation. The selection technique excluded on average 91.9% of the tests, which lead to a time saving of 89% compared to running all tests. All selections contained the fault revealing tests. However, the evaluation of the prioritization technique yielded mixed results. These results are compared to evaluations found in the literature. Finally, possible improvements of the evaluation process are suggested.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Contents

Regression Testing is a time and resource consuming part of software development. By re-running tests frequently the confidence is increased that changes do not introduce bugs. Thus, regression testing is an important task. But with an ongoing development more and more tests are created by which the regression testing process takes up more and more time. In consequence, Developers have to wait a long time to make sure their changes did not introduce a new bug. This prevents short feedback cycles and slows down the development process. The build process of continuous integration tools also takes much longer. It can also increase costs for the machines to run the tests.

Regression Test Optimization (RTO) aims to lower the costs of regression testing. This can either be done by running only a subset of tests (*regression test selection*) or prioritizing specific tests (*regression test prioritization*) with the goal to detect failing tests faster. A combination of both is also possible. According to Gligoric et al. in practice regression test optimization is often done by hand [Gli+14]. They compared manual to automated test selection. By this they found out that manual selection often misses relevant tests and adds irrelevant tests. Their conclusion is that better automated selection techniques that are well integrated in the development process are needed.

There are several optimization techniques presented in the literature. Papers like the systematic literature review on regression test prioritization by Yogesh Singh et al. [Sin+12] or the survey about minimization, selection and prioritization by Shin Yoo and Mark Harman [YH07] provide a good overview. Probably one of the earliest research about regression test selection was done by Harrold and Souffa in 1988 [HS88] and Taha et al. in 1989 [TTL89]. They proposed to use data flow analysis to find tests that cover changed parts of the code. This concept is very similar to the selection techniques found in more recent literature.

The evaluation of the RTO techniques is a problem in research about RTO. A lot of initial work has to be done until an optimization technique can be tested. Additionally, faulty versions of programs are needed to find out if a technique is able to find failing tests faster. For selection techniques it has also to be checked if the failing tests are found at all. Such faulty versions of programs are in general not publicly available. In consequence, techniques are often only evaluated in an artificial setup and by comparing it against optimal and random selection respectively ordering.

An often used method to create faulty versions of programs is mutation seeding. By this, changes are randomly spread into the source. Then, mutations that cause test cases to fail are selected. For example Yafeng Lu et al. evaluated several techniques with this approach [Lu+16]. Other evaluation approaches have been pursued by Rothermel et al. [RUC01] and Elbaum et al. [EMR02]. They implemented different versions of prioritization techniques that are based on code coverage and compared the results with random and optimal ordering. To do so, they defined a metric called APFD, "which measures the weighted average of the percentage of faults detected over the life of the suite" [EMR02]. Programs with failing tests were created by manually and randomly adding faults. These evaluations only allow limited predictions to the usage of regression test optimization in practice. It is not clear if such artificial faults appear like this in practice.

1.1 Contributions

Although many techniques were proposed in the literature, RTO is rarely used in practice. One reason for this may be a lack of evidence that RTO does yield a benefit in the real world. This thesis aims to help to solve this issue. Instead of testing techniques on projects with artificial faults, this thesis presents an evaluation of a selection and a prioritization technique on open-source projects with faults that were detected in the past. This should give further indications about the effectiveness of the techniques in practice. The contributions of this thesis are as follows:

1. Regression test optimization techniques are implemented. The selected techniques are *Cover Changes Selection*, *Additional Coverage Prioritization* and *Fail-History Prioritization*. The source code of relevant parts of the implementation is presented.

2. The *Cover Changes Selection* and *Additional Coverage Prioritization* are evaluated on open-source projects with real-world bugs. Additionally, the correlation between the number of changed files and the number of selected tests in the *Cover Changes Selection* technique is analyzed.

1.2 Structure of this Thesis

At first, chapter 2 presents some existing implementations of RTO techniques and their usage in practice. These are for example the selection tool Ekstazi and Microsoft's tool THEO. Additionally, other techniques that were found in the literature are presented.

Then the concepts of the selected techniques are presented in chapter 4. At first, the required background is given. This includes the regression test optimization framework Lazzer and code coverage in general. Then, the three techniques that were selected for this thesis are presented and it is explained why these were selected.

Chapter 5 is about how the selected RTO techniques were realized. First, it shortly describes how a new optimization technique can be added to Lazzer. Then, two code

coverage tools are compared and one of them is selected for the implementation. After this, it is explained how the code coverage tool is implemented and how a list of changed files is retrieved. Thereon, the source code of relevant parts is given and explained. At the end of the chapter, it is described what was changed in Lazzer in order to evaluate these techniques.

In chapter 6, two of the implemented techniques are evaluated with the help of three open source projects with in total 165 real bugs. First, the evaluation methodology is explained in detail. Then, it is described how the evaluation process was implemented and which problems needed to be solved. Afterwards, the results are presented and discussed. Furthermore, the threats to validity are explained.

Finally, a conclusion is given in chapter 7. Possible improvements of the implementation and topics for future research are proposed. This includes for example prioritization on test method level and the evaluation of history based techniques.

2 Related Work

Many regression test optimization techniques have been proposed in the literature. But there are only few implementations of techniques publicly available. One of them is Ekstazi [Gli] [GEM15] [Gli15]. It is a regression test selection tool developed by Milos Gligoric and can be integrated in build systems like `Maven` or `Ant`. Ekstazi selects tests that cover modified source files. Some projects use Ekstazi in practice, e.g Apache Commons Math and Apache Camel. Another publicly available tool is Infinitest [Inf]. It also selects tests that cover modified code and is available as plugin for the development environments IntelliJ and Eclipse. Furthermore, the code coverage tool OpenClover [Oped] also provides a mechanism to run only tests that cover modified code. Florian Dreier describes in his master thesis "Obtaining Coverage per Test Case" [Dre17] the implementation of a regression test selection and prioritization technique in the commercial software Teamscale. The optimization technique selects all tests that cover changes and sorts the selected tests according to the additional coverage per time.

There are some publications that describe the use of regression test optimization in an industrial environment. Edward Dunn Ekelund and Emelie Engström, for example, present a regression test selection technique that is based on test history and coarse test coverage [EE15]. It is used in practice at their employer Axis Communications. The technique uses coverage data on package level. They state that running selection techniques based on finer code coverage would be too costly, due to the size and complexity of the code. The presented technique needs a lot of historical data to deliver precise results, but by running only a subset of tests a lot of data is lost. Thus, they suggest running the full test suite periodically, e.g. every night. By this, there is more historical data available for the technique, which results in a better selection. They did not clearly state if the technique is used on the developer's machines or in a continuous integration environment. It is only said that the historical data is stored in a database. This would theoretically allow to share data between multiple machines. Another example for the use of regression test selection in practice is Microsoft's tool THEO, presented by Herzig et al. [Her+15]. It is a selection technique that is based on a cost analysis. The technique compares the expected costs of running a test with the expected costs of skipping a test. If running the test is more expensive than not running the test and probably missing a bug, the test is skipped. Various costs were considered for the cost analysis. These are for example costs for shared infrastructure on which the tests are run, the costs of inspecting a test failure and the costs of missing a test failure. They state that the technique can save "millions of dollars per year, while maintaining product quality" [Her+15].

There are also many prioritization techniques presented in the literature. Prioritization techniques are often used in combination with selection techniques, but apart from that

their practical use seems to be rare. Rothermel et al. proposed the additional coverage prioritization technique [Rot+99] [RUC01]. This technique sorts tests by how much additional not-yet covered code is covered by a test. It starts with the test that covers the most elements. Then comes the test that covers the most elements that are not-yet covered. This is continued until all tests are sorted. If all elements are covered, it starts again with the test that covers the most elements and is not-yet selected. The meaning of element is dependent on the granularity and can for example be a file, method or statement. Similar to the additional coverage prioritization techniques, there are techniques that are based on interaction coverage [BM07] [SOZ11].

Furthermore, Rothermel et al. proposed a technique that uses mutation analysis to compute a fault exposing probability for each test case [Rot+99]. By mutation analysis multiple faults are seeded into the source code and it is checked for each test how many faults it can detect. These information are then used to prioritize tests that are more likely to detect a test.

Before Microsoft developed THEO, Amitabh Srivastava and Jay Thiagarajan presented a test prioritization system, called Echelon [ST02]. It orders tests "based on changes between two program versions" [ST02]. They reported that it is possible to integrate regression test prioritization successfully in large scale software projects.

Wong et al. proposed a combination of a selection and prioritization technique. It selects tests that cover modified code and sorts tests by their additional coverage per execution time. Additionally, the optimized suite is minimized by removing tests as long as the minimized set has still the same coverage.

Jung-Min Kim and Adam Porter presented a technique for resource constrained environments [KP02]. First it prioritizes tests based on historical execution data. Then it selects the first tests that can be run in a given timespan. They proposed three different variants for the prioritization. One variant sorts tests based on whether the test was executed in the last run. Another variant prioritizes tests that failed often in the past. The third variant assigns a higher priority to tests that cover parts of the code that was infrequently covered in previous testing sessions.

3 Regression Test Optimization

Contents

This chapter explains **Regression Test Optimization**, or short RTO, in more detail. Furthermore, a framework for the evaluation of selection techniques and a metric to assess the result of prioritization techniques is described. Finally, this chapter presents existing evaluations of RTO techniques.

3.1 Regression Test Selection

Regression Test Selection techniques select tests by some criterion. Ideally the selection contains exactly the failing tests. But in practice it is not possible to predict which tests will fail. Thus, selection techniques have to use a different criterion to select tests. The most common approach is to select tests that execute changed code [HS88] [TTL89] [YH07] [Har+01] [Gra+01] [RH96] [RH97]. There are also selection techniques that use a different criterion to select tests, for example Microsoft's selection technique THEO that selects tests based on a cost analysis [Her+15].

Rothermel et al. introduced a framework to evaluate regression test selection techniques [RH94]. The goal is to provide a common base for the evaluation and comparison of selection techniques. This framework is presented in the following. First two definitions for test cases are given, which are later used to define criteria for the evaluation of regression test selection techniques.

Definition 1 *A test case $t \in T$ is **modification-revealing** if it produces different output in P and P'.*

Definition 2 *A test case $t \in T$ is **modification-traversing** if t covers new, modified or deleted code.*

Every test that is modification-revealing is also modification-traversing, because the output can only be different if some part of the covered code was changed. If only

non-modified code is covered, the output must be the same. But not every modification-traversing test is also modification-revealing. If a modified part of code behaves exactly like before, the output is the same. Hence, the set of modification-revealing tests is a subset of the set of modification-traversing tests. These definitions are now used to define some qualities of regression test selection techniques.

Inclusiveness The *inclusiveness* of a selection technique is the percentage of how many modification-revealing tests are selected out of all modification-revealing tests. For example, a technique that selects 2 out of 10 modification-revealing tests is 20% inclusive, with respect to the test suite and the modified program version. If a technique has an inclusiveness of 100%, it is called *safe*. That means such a technique selects all modification-revealing tests.

Definition 3 *A regression test selection technique is* ***safe*** *if it includes all modification-revealing tests, i.e., it has an inclusiveness of 100%.*

Precision The *precision* of a selection technique is the percentage of how many non-modification-revealing tests are excluded out of all non-modification-revealing tests. For example, if a technique selects 3 out of 10 non-modification-revealing tests, then it is 70% precise, with respect to the test suite and the modified program version. If a technique selects only modification-revealing tests, it has a precision of 100% and is called *precise*.

Definition 4 *A regression test selection technique is* ***precise*** *if it selects only modification-revealing tests and no non-modification-revealing tests, i.e., it has an precision of 100%.*

Efficiency The efficiency of a selection techniques is defined by the space and time requirements. Efficiency in terms of time includes how much time can be saved by the technique in total, i.e., the time saved by running only a subset of tests minus the time required to compute this subset. This also includes the time that is needed to gather the information the technique needs to compute the subset, for example the generation of test coverage information. Efficiency in terms of space requirements for example contains the space that is needed to store the required information.

Generality The generality of a techniques defines the ability to run the techniques on arbitrary situations. This contains what information are needed to use the technique.

For example, a technique that needs information that cannot be generated from the code is less general than a techniques that does not need any additional information.

3.2 Regression Test Prioritization

Regression Test Prioritization techniques sort tests by some criterion. In the following, a formal definition of the regression test prioritization problem is given. The prioritization criterion is represented by some function f that assigns every test case $t \in T$ a priority. Then, the bijection p assigns every test case a position. Given this, the prioritization is defined as follows:

Definition 5 (Test Case Prioritization) *For a given function $f : T \to N$, find a bijection $p : T \to \{1, \ldots, n\}, n = |T|$ of tests such that:*

$$\forall t_1 \forall t_2 : f(t_1) > f(t_2) \iff p(t_1) < p(t_2)$$

with $t_1, t_2 \in T$

3.2.1 APFD Metric

In general, the goal of regression test prioritization is to detect failing tests faster. How fast a test suite detects failing tests is often referred to as the *rate of fault detection* [EMR02]. Rothermel et al. introduced a metric for this, the average percentage of faults detected, or short APFD [Rot+99]. A high APFD value implies a faster (better) rate of fault detection. The APFD for a test suite $T_1 ... T_n$ and faults $F_1 ... F_m$ is given by the formula:

$$APFD = (1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}) * 100$$

Where n is the number of test cases, m the number of faults and TF_i the position of the first test that reveals the fault F_i.

In the following the APFD is computed for two examples. These examples are illustrated in figure 3.1. In the first example (left side of figure 3.1), the test suite consists of ten test cases. There are four faults which are detected by the tests at the positions 2, 5, 6 and 10. The APFD is then computed by $(1 - \frac{2+5+6+10}{4*10} + \frac{1}{2*10}) * 100 = 47.5$. In the second example (right side of figure 3.1) there are ten test cases and two faults. The faults are detected by the test cases at the positions 2 and 3. This results in an APFD of $(1 - \frac{2+3}{2*10} + \frac{1}{2*10}) * 100 = 80$.

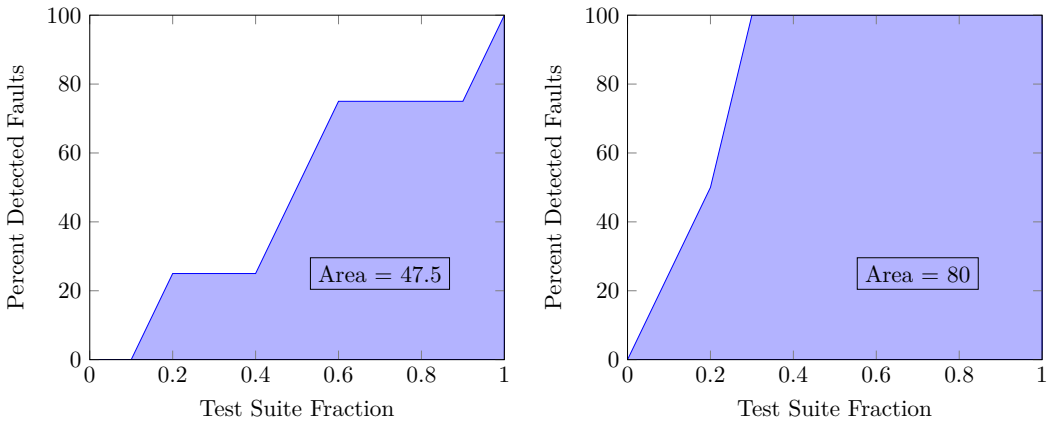


Figure 3.1: Two examples to show how the APFD metric can be determined graphically. The APFD corresponds to the blue area. Bot examples have a test suite containing ten tests. The example on the left side has four failing tests, which are revealed from the tests at the positions 2, 5, 6 and 10. The example on the right side has two failing tests. These are revealed from the tests at the positions 2 and 3.

3.3 State of Research

This section presents some evaluations of RTO techniques found in the literature. The primary focus is on evaluations of regression test selection techniques and additional coverage prioritization techniques, because these techniques are also evaluated in this thesis.

The evaluations of the selection techniques by Gligoric [GEM15] [Gli15], Herzig et al. [Her+15], Dreier [Dre17] and Ekelund an Engström [EE15] all indicate a time or cost saving potential. Ekelund an Engström constructed faulty version out of historical test execution data from their employers software product. This should give realistic results, but for a limited context. They report that on average 80% of the modification-traversing tests, i.e., tests that cover changed code, were selected and the optimized test set was 4% of the original size. Their conclusion is that the risk of missing some failing tests is low in comparison to the cost saving potential of the technique. Herzig et al. tested their selection technique THEO by replaying "past development periods of three major Microsoft products" [Her+15]. The evaluation revealed that 50% of the tests are excluded and that the technique can save "millions of dollars per year" [Her+15] without sacrificing quality. Since both of these techniques are highly tailored to the infrastructure and their software product, they are only limitedly comparable to other selection techniques. The evaluation by Gligoric has not been done with faulty versions of programs. He selected several open-source projects and tested the selection technique on multiple Git revisions. Dreier evaluated his implementation with open-source projects and created faulty version by mutation seeding. Gligoric reported that the testing time could be reduced by 32%

on average. He measured the time from the start of the optimization tool until the test run is finished. Dreier did not clearly state how much time could be saved or how many tests were selected, but he reports that 99.2% of the faults were detected and that 90% of the test fails appeared within the first 10% of the total execution time.

The additional coverage prioritization techniques have been object of multiple evaluations by Rothermel et al. [Rot+99] [RUC01], Elbaum et al. [EMR00] [EMR02] and Do et al. [DRK04]. The evaluations by Elbaum et al. and Do et al. list Rothermel as co-author. These techniques sort tests by how much code a test covers that is not yet covered by previous tests. Out of these evaluations, the one by Do et al. is the most similar to the evaluation presented in this thesis. They evaluated additional coverage prioritization techniques on open-source Java projects. Faulty versions were created by manually adding changes that lead to test failures. They state that they tried to insert faults that are as realistic as possible. The technique was implemented on basic block and method coverage level, in respect to Java bytecode. Additionally, these two versions were implemented with two test suite granularities, test class and test method. The test suite granularity defines on which level the technique sorts the test suite. With test class level granularity, whole test classes are sorted. The test methods within the test class can only be sorted inside the test class, but are not detached from its test class. At test method level, each test method is handled independently from its test class. The results show an increase of the APFD metric for both coverage granularities. Class level test suite granularity, however, resulted in significantly lower APFD values. In two out of five cases, the technique on class level technique was not able to increase the APFD value. The technique on method level increased the average APFD value in all projects.

4 Concept

Contents

This chapter explains the concepts of the RTO techniques that are implemented for this thesis. First, the required background is provided. This includes an overview of the RTO framework Lazzer, in which the techniques are embedded, and a brief explanation of test coverage in general. Finally, the RTO techniques that were selected for the evaluation are presented and it is explained why these were selected.

4.1 Background

This section presents the architecture of Lazzer and points out relevant components for the implementation of new RTO techniques. Furthermore, the concepts of test coverage are explained and important aspects regarding RTO techniques are located.

4.1.1 Lazzer

Lazzer is a framework for regression test optimization developed by Christian Plewnia as part of his master thesis [Ple15]. It allows the implementation and execution of regression test optimization techniques. Lazzer can be integrated as a Maven plugin or used as command line tool. The following shortly summarizes the relevant parts of Lazzer's architecture like it is described in the corresponding thesis by Plewnia [Ple15]. Figure 4.1 provides an overview of Lazzer's components. There are four components connected to the core of the framework. The *Client Adapter* builds the bridge to the project in which Lazzer is used. Data, for example coverage information, is shared with the optimization strategies by *Data Store*s. Optimization strategies implement the *Strategy* interface and can use multiple *Data Store*s. The *Test Framework* interface is implemented to connect with the client's test framework, e.g. JUnit for Java projects.

The components that are of interest here are *Data Store* and *Strategy*. The *Strategy* interface is used to implement new techniques and the *Data Store* interface is used to supply the strategies with coverage data.

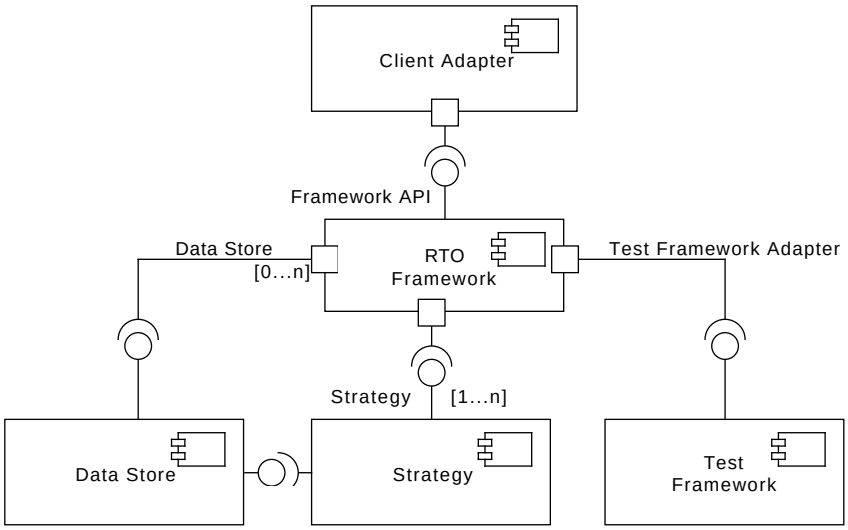


Figure 4.1: Lazzer UML Component Diagram [Ple15]

The optimization process is split into tasks which are organized in a pipeline system. This pipeline consists of different stages which are presented in figure 4.2.

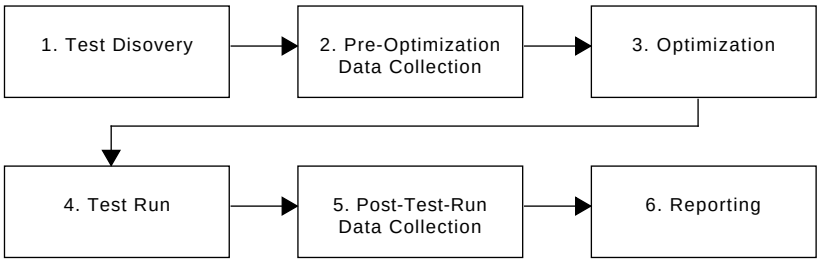


Figure 4.2: Lazzer task pipeline [Ple15]

The first task, *Test Discovery*, searches the given software project for tests. *Pre-Optimization Data Collection* initializes all data stores. In this phase the data stores load the data that is needed for the optimization. Then, in *Optimization*, the test suite is optimized with the optimization techniques. After this, the *Test Run* stage runs the optimized test suite. In the *Post-Test-Run Data Collection*, the data stores again collect information. This time they have the test results available to update their data. The last stage, *Reporting*, creates a test report.

4.1.2 Test Coverage

Test coverage measures which parts of the source code are executed (covered) by a test. There are two types of test coverage, global and per test. Global test coverage measure which parts of the source code are covered in total. Per test coverage measures the coverage for each individual test. Regression test optimization techniques need per test coverage. Global test coverage is not useful, because the techniques need coverage information for each individual test.

Furthermore test coverage can be distinguished by its granularity. That means how precisely the coverage is measured. Possible granularities are, for example, statement, method or file. Coverage on statement level, for example, measures which statements are executed, i.e. covered, by a test. In general, more precise coverage information is more costly. It needs more time to generate and more space to store the information. Furthermore, the complexity of an optimization technique is higher with finer coverage granularity, because there are more covered elements that need to be considered.

4.2 Regression Test Optimization Strategies

There are several regression test optimization techniques presented in the literature. They vary in the information they use to perform the optimization. For example, a required information can be the assigned importance of a test or the cost of running a test. Not all information is available in every software project, making some RTO techniques highly tailored to specific kinds of projects. For this research, I selected only techniques that require general information, such that they can be applied in any software project. These techniques do not have any requirements regarding the project or need data that cannot be generated from the source code.

4.2.1 Regression Test Selection

Regression Test Selection techniques aim to select a subset of tests such that the selection contains, at best, exactly the failing tests. Unfortunately it is in general not possible to predict which tests are going to fail. Therefore an approximation is needed, but the technique should still be safe, i.e., the selection should contain all failing tests. To find an approximation fulfilling that criterion we take a look at the reasons why tests fail. Basically, tests fail because some modification of the source code breaks a functionality that is checked by some test case. Thus, one approach is to select all tests covering modified code. Tests covering only non-modified code cannot fail, assuming they did not fail in the previous test run. The definition for the selected subset T' of all tests P is as follows:

$$\forall t \in T : t \in T' \iff t \text{ covers modified code}$$

This technique is safe because it selects all failing tests. But it is not precise, because tests that cover changed code do not always fail.

This is by far the most common approach for selection techniques. It can be seen as

the basis for many selection techniques. There are also techniques that are a combination of this technique with some sort of prioritization [Dre17] [Gli15] [Won+97].

4.2.2 Regression Test Prioritization

Additional Coverage Prioritization The Additional Coverage Prioritization technique sorts tests according to the additionally covered code. It iteratively selects the test that covers the most code that is not-yet covered by the already selected tests. This technique was presented for multiple coverage granularities by Rothermel et al. in "Test Case Prioritization: An Empirical Study" [Rot+99]. In this thesis the coverage granularities file, method and statement are considered. The additional coverage prioritization technique was selected because the results from Rothermel et al. are quite promising. Additionally, it is rather easy to implement, once test coverage data collection is implemented.

History-Based Prioritization This approach is described in the paper "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments" by Jung-Min Kim and Adam Porter [KP02]. It was later used by Park et al. to create a cost cognizant variant [PRB08].

The technique computes a value called *selection probability* P_k for every test case. It takes the previous selection probability and the last test result to compute a new selection probability. The impact of the last test result can be controlled by a parameter α. The general formula for the selection probability P_k is

$$P_k = \alpha h_k + (1 - \alpha) P_{k-1}$$

with $0 \leq \alpha \leq 1, k \geq 1$. If a test failed in the last run, the corresponding history variable h_k is set to 1, otherwise it is set to 0. The variable α can be chosen from 0 to 1. With a low α value the term $(1 - \alpha)$ gets bigger by which the impact of the history variable h_k gets lower. A high α value increase the impact of the last test result. The selection probability of a test that fails multiple times in a row rises for each failure up to a value of 1. Once the test passes again, the value falls. The parameter α controls how fast the selection probability rises and falls.

The original approach is designed for time constrained environments. Tests are executed until the available testing time is exhausted. More precise, the implementation presented by Kim and Porter assumes that only n% of the tests can be executed due to limited time and selects the top n% with the highest selection probability. In the approach proposed for this thesis, the tests are sorted by the assigned selection probabilities.

5 Realization

Contents

This chapter presents the implementation of the selected RTO techniques and the associated components. First, the necessary background is given. This includes relevant parts of Lazzer, test coverage tools for Java software and how a list of changed files is obtained. Then, it is shown how the RTO techniques are realized by presenting and explaining relevant parts of the implementation. Finally, changes that make the evaluation process easier are described.

5.1 Background

This section gives the necessary background for the implementation of RTO techniques. First, relevant parts of Lazzer that are needed to realize the RTO techniques are explained. Then, the code coverage tools OpenClover and JaCoCo are compared and one of them is selected.

5.1.1 Lazzer

While the background section of chapter 4 presented the conceptual architecture of Lazzer, this section describes the classes and interfaces that are used to implement the techniques and a test coverage data store.

Optimization techniques are called *strategies* in Lazzer and are grouped in a Maven module *lazzer-strategies*. Each module is organized in a Maven module, which is a child module of *lazzer-strategies*. These modules contain a class that implements the interface `OptimizationStrategy`. This interface has a method `optimize` that takes the original test suite as parameter and expects an optimized test suite in return. To implement a new strategy, a new child module with a class that implements the

`OptimizationStrategy` has to be created in *lazzer-strategies*. Required data stores are added as parameter to the constructor that is annotated with `@Inject`.

Data stores are organized in the Maven module *lazzer-datastore*. To create a new data store, a new module has to be added to this module. Then, a new class that implements the `DataStore` interface has to be created. This interface has two methods, `preOptimizationDataCollection` which is called before the optimization and `postTestRunDataCollection` which is called after the test run. The method `postTestRunDataCollection` has the test results as a parameter.

5.1.2 Test Coverage

The general concept of test coverage was discussed in chapter 4. This section explains how the coverage tool is chosen and how it is implemented in Lazzer.

Java code coverage: OpenClover vs. JaCoCo

The code coverage tools OpenClover and JaCoCo were considered for the implementation. JaCoCo seems to be the most commonly used coverage tool. It has an active forum and gets regular updates, but does not support per-test coverage by default. It is possible to get per-test coverage with JaCoCo anyway, as described later. OpenClover supports per-test coverage and is also regularly updated. Both tools can be integrated via Maven plugins. Other coverage tools for Java software are for example JCov, Cobertura and CodeCover. But these tools are either not actively maintained anymore or do not provide per-test coverage.

The selected tools, OpenClover and JaCoCo, are compared and it is explained which tool is chosen. To decide which tool to use, mainly the usability and the effort that is needed to integrate the tool into Lazzer are considered. Performance is not a criterion, because the focus is on the evaluation of regression test optimization techniques and not test coverage tools.

JaCoCo does not support per test coverage natively. It is possible to generate per test coverage with JaCoCo anyway. To do so, every test start and end has to be detected, in order to dump the collected coverage data. This produces an overhead in performance and disk space. According to Florian Dreier, coverage reports of bigger projects take up to several gigabytes [Dre17]. He managed to reduce the size by compressing the coverage reports. Dreier also found a way to reduce the performance overhead by caching results.

So, all in all it is possible to generate per-test coverage with both tools. But compared to OpenClover, there is more work to do to achieve this with JaCoCo. Considering the available time and that the focus of this work is more on evaluation of regression test optimization techniques, OpenClover is chosen as coverage tool. Another advantage I experienced is that OpenClover is, in general, easier to use. OpenClover provides a Java API that allows to read in the generated coverage data. From my point of view this seemed to be more complicated with JaCoCo.

Implementation in Lazzer

The generation of test coverage is not included into Lazzer directly. Instead Lazzer just reads in the coverage data that was generated with OpenClover.

To make the coverage information available to the techniques, a new `DataStore` is implemented. That means, a new module with a class that implements the `DataStore` interface is created. Furthermore, data access objects (DAO) are created for file, method and statement level coverage data. The DAO is the object by which the techniques get access to the coverage data. In the `preOptimizationDataCollection` method the coverage data is read in. The source code for this is shown in listing 5.1. First, a new `CloverDatabase` object is created and the method `loadCoverageData` is called to read in the coverage data. The `cloverDbPath` variable points to the OpenClover database file `clover.db`, which is located under "target/clover". Then, the `PerTestCoverage` object is received by calling `getCoverageData().getPerTestCoverage()` on the `CloverDatabase` object. This object provides coverage data for each individual test. Additionally, the `FullProjectInfo` object is received by calling `getFullModel`. The `FullProjectInfo` is then used to iterate over all files, like it is explained in the OpenClover Documentation in section "Database Structure" under "Reading from a Clover database" [Opec]. For every file the data store reads in the tests that cover the file by calling `getTestsCovering` on the `PerTestCoverage` object. This information is stored in a `HashMap` with the path to the source file as key and the set of covering tests as value. The path to the source file is unique, thus it is suitable to be used as key. Additionally, the data store saves for every test which files it covers. This is also done with a `HashMap`. Storing the coverage information in both directions, from test to covered files and from file to covering tests, allows a faster access to these information.

```
1  cloverDatabase = new CloverDatabase(cloverDbPath);
2  cloverDatabase.loadCoverageData();
3
4  CoverageData perTestCoverage =
       cloverDatabase.getCoverageData().getPerTestCoverage();
5  FullProjectInfo projectInfo = cloverDatabase.getFullModel();
6
7  Set<TestCaseInfo> testCovering;
8  for (PackageInfo packageInfo : projectInfo.getAllPackages())
9  {
10   for (FileInfo file : packageInfo.getFiles())
11   {
12     testsCovering =
13       perTestCoverage.getTestsCovering((CoverageDataRange) file);
14     //add coverage data to file level DAO
15
16     for (MethodInfo method : file.getAllMethods())
17     {
18       testsCovering =
19         perTestCoverage
```

```
20          .getTestsCovering((CoverageDataRange) method);
21        //add coverage data to method level DAO
22
23        for (StatementInfo statement : method.getStatements())
24        {
25          testsCovering =
26            perTestCoverage
27              .getTestsCovering(
28                (CoverageDataRange) statement);
29          //add coverage data to statement level DAO
30        }
31      }
32    }
33  }
```

Source Code 5.1: OpenClover Database Read

5.2 Changed Files

There are different ways to get a list of changed files, depending on which kind of changes should be considered. Two approaches were implemented: list of modified files by the last Git commit and the list of locally modified files since the last test run.

Files Modified by Git Commit

For the first approach a new method `getChangedFiles` was added to the existing `GitDataStore`. This method returns a set of files that were modified by the last commit. Basically, a list of changed files is retrieved with the help of a git-diff command. Listing 5.2 shows how this was realized.

```
1  ObjectReader reader = gitRepository.newObjectReader()
2
3  ObjectId currentCommitId = gitRepository.resolve("HEAD^{tree}");
4  ObjectId previousCommitId = gitRepository.resolve("HEAD^^{tree}");
5
6  CanonicalTreeParser oldTreeParser = new CanonicalTreeParser();
7  oldTreeIter.reset(reader, previousCommitId);
8  CanonicalTreeParser newTreeParser = new CanonicalTreeParser();
9  newTreeIter.reset(reader, currentCommitId);
10
11 List<DiffEntry> diffEntries = git.diff()
12   .setNewTree(newTreeIter)
13   .setOldTree(oldTreeIter)
14   .call();
```

Source Code 5.2: Get Changed Files with JGit

By calling `gitRepository.resolve("HEAD^{tree}")` the underlying tree id of the current HEAD is retrieved. The call `gitRepository.resolve("HEAD^^{tree}")` returns the parent of HEAD, which is the previous commit. Then, a `CanonicalTreeParser` is created for each tree to construct a diff command, which returns a list of `DiffEntry` objects. The current path of a changed file can be retrieved by calling `getNewPath` on a `DiffEntry`. For deleted files, this returns `"/dev/null"` [Ecl]. Deleted files are simply ignored. Changed and new files are added to the list of changed files.

Files Modified Locally

The second approach computes a list of files that are modified locally. To be able to compare two versions of a file, the MD5 hash is computed. The hashes are stored in a `HashMap` with the file path as key and the hash as value. After every test run this `HashMap` is serialized into a JSON file. The JSON file can then be used in the next run to compare the freshly computed hash value with the old one. If the hash values differ, the file is added to the list of changed files. When a file is added, there is no previous hash value and these files are added too. Deleted files are ignored, because files that do not exist anymore cannot be covered by any test.

5.3 Implementation

This section presents and explains relevant parts of the implementation. Therefore, small code snippets are provided to show how the techniques are realized. Furthermore, it is discussed what was changed in Lazzer to make the evaluation process easier.

5.3.1 Cover Changes Selection Technique

The Cover Changes Selection technique aims to select all tests that cover modified parts of the code. Listing 5.3 shows the relevant parts of the implementation.

```
1  Set<SourceFile> changedFiles = gitDataStore.getChangedFiles();
2  Set<String> affectedMethods = changedFiles.stream()
3    .map(testCoverageDataStore::getTestsCovering)
4    .flatMap(Set::stream)
5    .collect(Collectors.toSet());
6
7  for (TestClass testClass : testSuite)
8  {
9    List<TestMethod> selected = new ArrayList<>();
10   for (TestMethod testMethod : testClass)
11   {
12     String tcQualifiedName = MethodName
13           .computeQualifiedName(testClass, testMethod);
14     if (affectedMethods.contains(tcQualifiedName))
15     {
16       selected.add(testMethod);
```

```
17      }
18    }
19
20    if (!selected.isEmpty())
21    {
22      TestClass tc = TestDescriptionFactory
23          .newTestClass(testClass.getTestClass(), selected);
24      optimizedTestSuite.add(tc);
25    }
26  }
27  return optimizedTestSuite;
```

Source Code 5.3: Select Test Covering Changes

At first, the technique requests a set of changed files from the Git data store. This set contains the files that were changed by the last Git commit. Then, the stream operation in lines 2 to 5 collects all tests that cover a changed file. The `flatMap` method in line 5 transforms a stream of lists into a stream with all the elements of the lists. Here, a stream of sets of strings is mapped to a stream of strings. Finally, a new test suite is created, containing only test classes with at least one test method that covers a modified file. This is done by iterating over all test classes and creating a new test class containing only test methods that are affected by the changes. If a test class contains no affected methods, it is rejected. It is not possible to simply add all affected methods to one new test class. Test classes have set up methods that are needed for the successful execution of test cases. It is necessary to compute a qualified name of the test methods out of the method and class name in lines 12 and 13, because this is the way the covered methods are returned by the coverage data store. Furthermore, the `TestMethod` object only stores the name of the method, which is not unique.

5.3.2 Additional Coverage Prioritization Technique

The Additional Coverage Prioritization technique sorts tests by additionally covered code. In the first step a new empty test suite is created. Then, the technique iteratively adds the test case that covers the most elements that are not-yet covered. While the technique is implemented for three coverage granularities, the source code presented in listing 5.4 is kept general. The word element is used as placeholder for file, method or statement.

```
1  Set<Integer> alreadyCovered = new HashSet<>();
2
3  while (!testSuite.isEmpty())
4  {
5    int biggestAdd = Integer.MIN_VALUE;
6    TestMethod biggestAddT = null;
7
8    for (TestMethod testMethod : testClass) {
9      Set<Integer> elementsCoveredBy =
```

```
10        testCoverageDataStore.getElementsCoveredBy(testClass,
              testMethod);
11      elementsCoveredBy.removeAll(alreadyCovered);
12
13      if (elementsCoveredBy.size() > biggestAdd) {
14        biggestAdd = elementsCoveredBy.size();
15        biggestAddT = testMethod;
16      }
17    }
18
19    TestClass optimizedTc = optimizeTestClass(tc, alreadyCovered);
20    alreadyCovered.addAll(
21        testCoverageDataStore.getElementsCoveredBy(tc));
22    testSuite.remove(tc);
23    optimizedTestSuite.add(optimizedTc);
24
25    if (alreadyCovered
26        .equals(testCoverageDataStore.getCoveredElements()))
27    {
28      alreadyCovered.clear();
29    }
30  }
31
32  return optimizedTestSuite;
```

Source Code 5.4: Additional Coverage Prioritization

The while loop is executed until the original test suite is empty. This is the case, when all tests have been added to the optimized test suite. The for loop in lines 8 to 17 searches for the test class covering the most elements that are not yet covered by the optimized test suite. The method `getElementsCoveredBy` returns a set of integers. These integers represent a file, method or statement, depending on which coverage granularity is used. This allows a common implementation for all three coverage levels. The test class with the most additional covered elements is also optimized. That means, the test methods within the test class are sorted according to the additional coverage criterion, with respect to the set of already covered elements. The optimize method for test classes creates a copy of `alreadyCovered`, so that the passed object is not changed. After the test class was optimized, all newly covered elements are added to the set of already covered elements. The test class is removed from the original test suite to mark it as sorted. Finally, the optimized test class is added to the optimized test suite. At the end of the while loop it is checked if all elements are already covered. If so, the set of already covered elements is cleared. The method `getCoveredElements` returns all elements that are covered by any test. After all test classes have been added to the optimized test suite, the optimization is finished and the optimized test suite is returned.

Fail History Prioritization Technique

Fail History Prioritization sorts tests by their historical test results. This technique is a slightly modified version of the prioritization and selection technique proposed by Jung-Min Kim and Adam Porter [KP02]. Listing 5.5 shows its implementation.

```
1  //Load previous selection probabilities
2  probHistory =
       SelectionProbabilitiesHistory.load(projectDescriptor);
3
4  for (TestClass testClass : originalTestSuite) {
5    for (TestMethod testMethod : testClass) {
6
7      Double lastProbability =
8        probHistory.getLastProbability(testClass, testMethod);
9
10     Optional<TestResult> lastTestResult =
11       testHistoryDataStore
12         .retrieveLastTestResultFor(testClass, testMethod);
13
14     if (lastProbability == null || !lastTestResult.isPresent()) {
15       //first test run for this TestMethod with this strategy or
             test didn't run last time
16       double newProbability = 1;
17       probabilites.put(
18         MethodName
19           .computeQualifiedName(testClass, testMethod), prob);
20     }
21     else {
22       TestResult lastResult = lastTestResult.get();
23       int h = lastResult.wasSuccessful() ? 0 : 1;
24       double newProbability
25           = ALPHA * h + (1 - ALPHA) * lastProbability;
26       probabilites.put(
27         MethodName
28           .computeQualifiedName(testClass, testMethod),
               newProbability);
29     }
30   }
31 }
```

Source Code 5.5: Fail History Prioritization

The first step is to load the selection probabilities from the last run. These values are saved in a JSON file which is stored and loaded with the `ObjectMapper` from the Jackson JSON library. Then, the test results from the previous run are received from the test history data store. These value are used to compute the new selection probability. The alpha value is a final static variable. If there is no previous selection probability or test result from the last run, the test has not been executed in the last run. Thus, the

```
1 [ {
2 "classname" :
    "de.rwth.swc.examples.triangletester.tests.ExampleClass",
3 "testMethods" : [ {
4 "name" : "exampleMethod",
5 "classname" :
    "de.rwth.swc.examples.triangletester.tests.ExampleClass"
6 }]
7 }]
```

Source Code 5.6: Test Suite JSON

test gets the highest priority. When the technique is run for the first time, every test case gets the highest priority. This way, the original ordering is preserved in the first run. After every test case got a selection probability assigned, the test classes are sorted by the average selection probability of their test methods. The test methods within the test classes are sorted by their selection probabilities.

5.3.3 Changes for the Evaluation

To automatically evaluate the techniques, some changes were made to Lazzer. After the optimization is finished, Lazzer writes the optimized suite in JSON format to `optimized_suite.json` and the original suite to `original_suite.json`. An example for a test suite with only one class and one method can be found in Source Code 5.6. This task is realized by adding a new stage to the pipeline.

Lazzer originally included only test classes that contain at least one method that is annotated with `@Test`. To allow testing older versions of projects that do not use JUnit annotations, Lazzer was modified to also include test classes that follow the Surefire naming rules for methods [Apa]. That means, a test class is included if it contains at least one method that starts or ends with 'test'. This does not fulfill the complete Surefire naming rules, but it was enough to test the selected projects.

To measure the performance dependent on the number of changed files, also required changes to Lazzer. It must be possible to specify how many of the recent commits are considered for the list of changed files. To realize this, the method `getChangedFiles` in the Git data store reads in a files that contains a number. This number represents how many commits back in history should be taken into account. The file gets created during the evaluation. After each run, the results are read in and the number is increased for the next run.

6 Evaluation

Contents

This chapter presents the evaluation of the selection technique and the additional coverage prioritization technique. The questions this evaluation aims to answer with respect to the selected open-source projects are:

- Is the implemented selection technique safe? That means, does the optimized test suite contain all failing tests?
- How many tests can be excluded by the selection technique? How much time is saved by that?
- How fast does the size of the test suite grow with an increasing number of changed files?
- Can the additional coverage prioritization technique increase the APFD value in comparison to the original ordering? What effect does the coverage granularity have?

First, the evaluation methodology is explained. Then, the results are presented and discussed. Finally, the chapter explains the weaknesses of the evaluation process and its threats to validity.

6.1 Methodology

This section explains how the techniques are evaluated. The goal for the selection technique is to find out how many tests get selected and if the technique is able to cover all failing tests, i.e. if the technique is *safe*. For the additional coverage prioritization

technique the APFD value of the optimized test suite is compared with the APFD value of the original test suite. An explanation of the APFD value can be found in chapter 3. In contrast to evaluations that tested RTO techniques on projects with artificial faults, this thesis aims to evaluate the techniques on open-source projects with real faults. Fortunately, there is the project Defects4J that collects bugs from open-source Java projects, which were detected and fixed in the past. The following explains how Defects4J is built up and which projects are chosen for the evaluation. Finally, this section presents the evaluation process and explains how it is automated.

6.1.1 Defects4J

Defects4J is a database of historical faults that were identified in open source Java programs. Additionally, it provides a command line tool that allows to checkout the source code of a program. This can be a version containing the bug or a fixed version of the project. It is designed such that new bugs can be added to the database with little effort once a program was added to Defects4J. According to the associated paper by Just et al. [JJE14], the goal of Defects4J is to allow reproducible studies in the field of software testing research.

The faults provided by Defects4J are bugs that were identified and fixed in the past. Only those bugs were added that are explicitly related to source code. There must be a commit that is labeled as bug fix, i.e., the commit references a bug in the bug tracking system or the bug tracking system references the commit. Another requirement is that there is a test case that detects the bug, i.e., it passes in the fixed version and fails in the buggy version. The test may not have existed before, but may be introduced with the bug-fix. Furthermore, the bug fixing commit must not contain changes that are unrelated to the bug fix, e.g., features. If all these requirements are fulfilled, the bug can be added to the database.

If the project uses Git, Defects4J provides a copy of the original repository, enriched with additional commits by Defects4J. There is a commit labeled as buggy version and a commit labeled as fixed version. The buggy version contains the test cases that can reveal the bug, but not the bug fix. That means, the fault revealing tests fail in this version. The fixed version contains these tests and the bug fix. There should be no failing test in this version.

Selection of Programs

Defects4J covers six projects as shown in table 6.1. Three of them are used for the evaluation. The selected programs are Apache Commons Lang, Apache Commons Math and Joda-Time. Only Maven projects were considered, because Lazzer provides a Maven plugin and Lazzer's command line tool did not fully work and could not be fixed in reasonable time. JFreeChart and Closure Compiler are also Maven projects but the bugs provided by Defects4J occurred in versions before march 2010. Back then, these projects used a different format for specifying the Maven project definition instead of a `pom.xml`. Because of that JFreeChart and Closure Compiler are not used.

Identifier	Project Name	Number of Bugs
Chart	JFreeChart	26
Closure	Closure Compiler	133
Lang	Apache Commons Lang	65
Math	Apache Commons Math	106
Mockito	Mockito	38
Time	Joda-Time	27

Table 6.1: Overview of Projects in Defects4J [Def]

Some versions of the selected projects cannot be used for the evaluation. Version 5 of Apache Commons Lang was excluded, because it did not compile successfully, due to a dependency that could not be resolved. Version 42 to 65 are also excluded because the source levels of version 42 to 59 are too low for Open Clover and the versions 60 to 65 have no `pom.xml`. In Version 27 of Joda-Time and Apache Commons Math 100 to 106, the source level is also too low for OpenClover. Altogether, 165 project versions are selected for the evaluation, these are: versions 1 to 4 and 6 to 41 of Apache Commons Lang, versions 1 to 99 of Apache Commons Math and versions 1 to 26 of Joda-Time.

6.1.2 Concept and Realization

The following describes the steps that are needed to run Lazzer on a Maven project from Defects4J. Additionally, it is shortly described what problems came up and how the process is automated.

In general, the things that need to be done are: checkout the project, add Lazzer and OpenClover to the build automation tool (Maven's `pom.xml`), generate coverage data and compute the runtime, run Lazzer to obtain the optimized test suite, analyze the results. These steps are divided into three phases: initialization, optimization and analysis. In the first phase, **initialization**, everything that is needed to run Lazzer and later analyze the results is done. The first task in this phase is to get the fixed version of the project. The coverage data must be generated in the fixed version. Otherwise there would be no coverage data available for the tests that expose the fault, because these tests fail in the buggy version. Then, the Lazzer plugin and the code coverage tool must be added to the project. After this, coverage data and the average runtime of the tests is generated. At the end of this phase the buggy version of the project is checked out.

In the second phase, **optimization**, the optimization technique is executed. Apart from that, there is not much to do in this phase, because Lazzer does all the work. Lazzer optimizes the test suite and writes out the results as described in chapter 4.

The third phase, **analysis**, uses the results from the previous phases to analyze the optimization. It is computed how much time the optimized and original test suite need to detect the fault. Additionally, it is checked whether the optimized suite contains all tests that reveal the fault. Enumeration 6.1 summarizes these steps.

1. Initialization
   - checkout project
   - add Lazzer and code coverage plugin to project
   - generate coverage information
   - generate average runtime
2. Lazzer
   - optimize test suite
3. Analysis
   - analyze results of optimization
   - check if optimized suite contains all fault revealing tests
   - write optimization report

Figure 6.1: Evaluation Steps

Realization of the Evaluation Process

First, some difficulties are described. Some project versions provided by Defects4J need to be compiled and tested with the Java Development Kit (JDK) Version 7. The Lazzer framework and the evaluation application use features of Java 8 and can only be compiled and run with JDK8. This leads to the problem that the JDK has to be switched during the evaluation process. Another problem is that Maven tries to recompile the project when Lazzer is executed. This happens because Lazzer is attached to the default `test` goal, which performs a compilation by default. The problem here is that Lazzer is executed with JDK8, but, for example, one version of Apache Commons Lang cannot be compiled with JDK8. Thus, the compilation phase needs to be skipped. This is done by setting the property `Maven.main.skip` and `Maven.test.skip` to true. Setting the property `Maven.test.skip` to true also has the effect that the default test execution is skipped.

To realize and automate the evaluation process, a simple Java program was created that performs all specified tasks. In the first step, the fixed version of a program using the Defects4J framework is checked out. Then, the program adds plugins of Lazzer and OpenClover to the Maven `pom.xml`. This is done with the help of the DOM interface of the Java API for XML Processing.

Afterwards, coverage data is generated. To generate coverage information, OpenClover needs to be setup with the goal `clover:setup`. This instruments the source code and initializes the coverage database file. Then, the instrumented sources need to be compiled. The goal `test-compile` compiles everything that is needed to run tests, i.e., all source files, including the tests. Finally, the results are collected with the goal `clover:clover`. This writes the collected coverage data into the `clover.db` database file and creates

an HTML report. The Maven goals are executed with the Apache Maven Invoker plugin. First, all necessary parameters are set with a `DefaultInvocationRequest` object. These parameters are, for example, the location of the POM file and the Maven goal. Additionally, the Java version is set by setting the Java home directory with the method `setJavaHome` and by setting the properties `maven.compiler.source` and `maven.compiler.target` to the desired source level.

Then, the buggy version is checked out. Afterwards, a cleanup of the project is performed and the average runtime for each test method is computed. This is done by running the goal `surefire:test` five times and calculating the average runtime of each test afterwards. Surefire generates an XML file for each test class with information about the runtime and, if a test failed, the reason for the failure. These files can be found under `target/surefire-reports/`. After each run these files are read in and the runtime for each test method is stored. The cleanup before this step is done to speed up the runtime of the tests. Instrumented tests have a higher runtime than non-instrumented tests. But as we are more interested in time difference between optimized and original suite, the cleanup is not absolutely mandatory. This was the last step of the initialization phase.

The next step is to run Lazzer. Lazzer needs to be executed with JDK8 but the projects need to be compiled with JDK7. Even if no source files were changed since the last compilation, Maven still detected changes and tried to recompile the project. In some projects this leads to compile errors. To prevent the recompilation, the properties `maven.main.skip` and `maven.test.skip` are set to true. This skips the compilation of all source files and the test execution with Surefire. Finally, Lazzer is run by executing the Maven goal `test`. After the optimization is finished, Lazzer writes the optimized suite in JSON format to `optimized_suite.json` and the original suite to `original_suite.json`. This is done with the help of the Jackson JSON API.

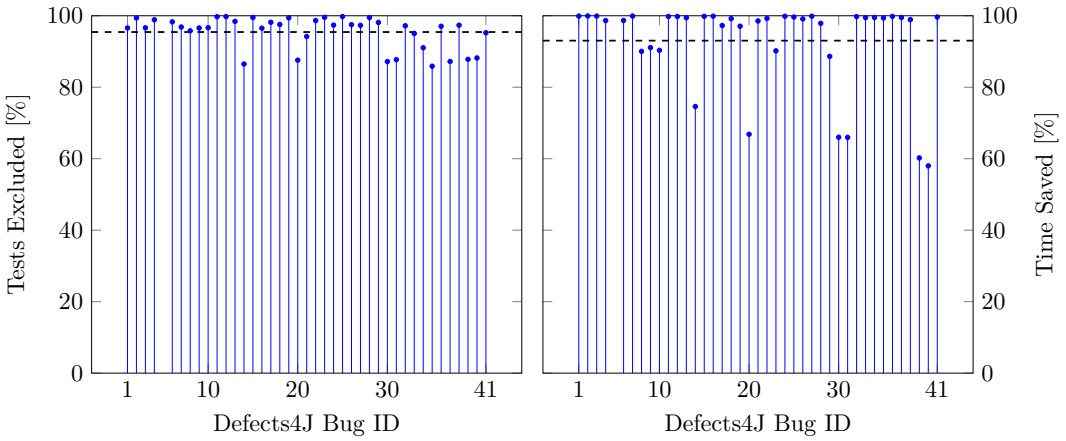
It follows the analysis phase. At first, the program reads in the JSON files of the optimized and the original test suite. Then, the execution times of the original and optimized suite are computed. Additionally, the number of tests until a failing test occurs is stored. An important step is to check whether the optimized test suite contains all tests that reveal the bug. These fault revealing tests are specified by Defects4J and are obtained by running `defects4j export -p tests.trigger` in the projects directory.

Finally, the results are written to a CSV file, which is used as data source for the subsequently presented graphics. Enumeration 6.2 summarizes the evaluation process and, if applicable, gives the executed Maven or shell commands.

All evaluations are performed on a computer with Arch Linux (Kernel 4.18.5-arch1-1-ARCH), OpenJDK 7 version 7.u171_2.6.13-1, OpenJDK 8 version 8.u181-1, Apache Maven version 3.5.2 and Defects4J version 1.2.0. The machine is equipped with an Intel Core i7 2600K @ 3.4GHz, 16GB of RAM and a Crucial MX500 SSD with 500GB.

1. Checkout the fixed version of a project from Defects4J
   - `defects4j checkout -p <project_id> -v <bug_id>b -w <work_dir>`
2. Add the Lazzer and OpenClover plugin to the `pom.xml`
3. Generate test coverage information
   - `mvn clover:setup test-compile surefire:test clover:clover`
   - needs to be done with JDK7
4. Copy coverage information to temporary folder
5. Switch to buggy version with failing test(s)
   - checkout tag `D4J_<project_id>_<bug_id>_BUGGY_VERSION`
6. Compute average runtime of tests
   - `mvn clean test-compile surefire:test`
   - needs to be done with JDK7
7. Run Lazzer
   - `mvn -Dmaven.main.skip=true -Dmaven.test.skip test`
   - needs to be done with JDK8
8. Collect results
9. Check if optimized suite contains all tests that reveal the bug (specified by Defects4J)
10. Write optimization report

Figure 6.2: Evaluation Process



(a) Tests Excluded (b) Time Saved

Figure 6.3: Apache Commons Lang - Cover Changes Selection. The chart on the left side shows how many tests were excluded. The chart on the right side shows how much time was saved by running only the selected tests. On average 95.44% of the tests were excluded and 93.03% of the time was saved.

6.2 Results

This section presents and discusses the results obtained by the evaluation. The questions that were presented at the beginning of this chapter are answered one after the other.

6.2.1 Cover Changes Selection

Is the implemented selection technique safe? To find out if the technique is safe, it is checked if the optimized test suites contain all failing tests. These tests are called fault exposing by Defects4J and can be obtained through the command line tool. This question can be answered shortly. Yes, all optimized test suites contain all tests that expose the fault according to Defects4J. Thus, the technique is safe with respect to the tested versions.

How many tests can be excluded by the selection technique and how much time is saved by that?

To answer this question, the size and runtime of the optimized test suite and the original test suite are compared. The results are presented in figures 6.3, 6.4 and 6.6. The bug ids from Defects4J are plotted on the X-Axis and the percentage of the number of tests excluded by the selection is plotted one thy Y-Axis. The dashed line is the average percentage.

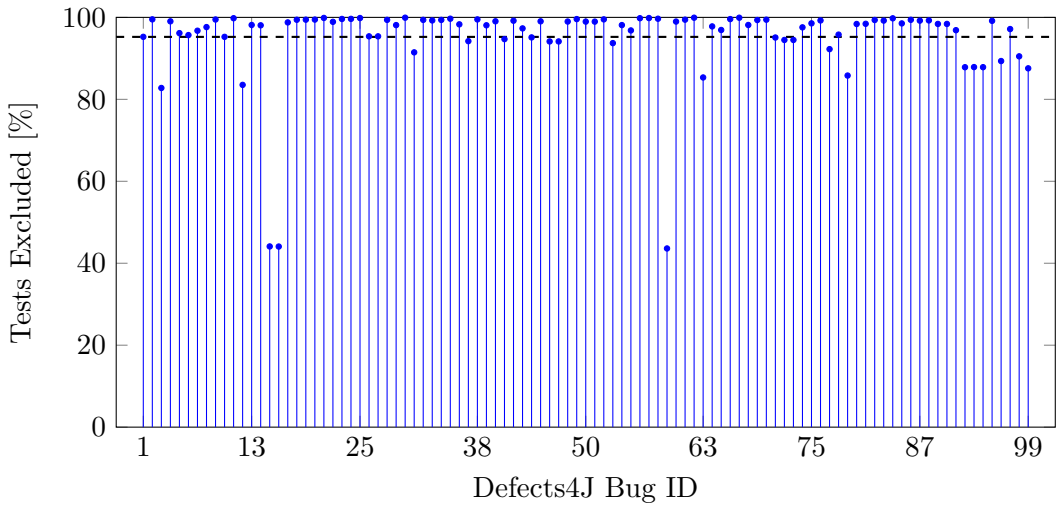


Figure 6.4: Apache Commons Math - Cover Changes Selection - Percentage of Excluded Tests. On average 89.88% of the time was saved in comparison to running all tests.

Apache Commons Lang In Apache Commons Lang (figure 6.3) on average 95.44% of the tests were excluded, which results in a time saving of 93.03%. Minimum 85.88% of the tests were excluded (bug 35). In the best case 99.78% tests could be excluded (bug 25). The minimum time was saved in bug 40 with 57.9%. The maximum time saving was achieved in bug 2 with 99.9%. A high percentage of tests excluded does not always come along with a high percentage of time saved. The biggest discrepancy between relative tests excluded and relative time saved is in bug 40, where 88.2% of the tests were excluded and 57.9% of the time was saved. There are five more outliers in relative time saved with percentages between 60.2% and 66.8%. Bug 5 was not tested, because it was not compilable.

Apache Commons Math The average percentage of tests excluded in Apache Commons Math (figure 6.4 and 6.5) is 95.24%, the average percentage of time saved is 89.88%. There are three outliers in the percentages of excluded tests and saved time, bug 15, 16, 59, with about 44% tests excluded and between 9% and 23% time saved. In the other cases at least 83% of the tests were excluded. But there are nine other bugs with a time saving between 42% and 66%. In 57 cases more than 99% of the time could be saved and in 48 cases more than 99% of tests were excluded. The biggest difference between time saved and tests excluded is in bug 3 with 83% tests excluded and a time saving of 42%.

Joda-Time In Joda-Time (figure 6.6), fewer tests were excluded by the selection than in Apache Commons Lang and Math. The average percentage of tests excluded is 73.43% and the average percentage of time saved is 81.58%. There are six outliers in tests excluded, bug 8, 9, 17, 19, 23 and 25, with a percentage between 33.86% and 35.41%.

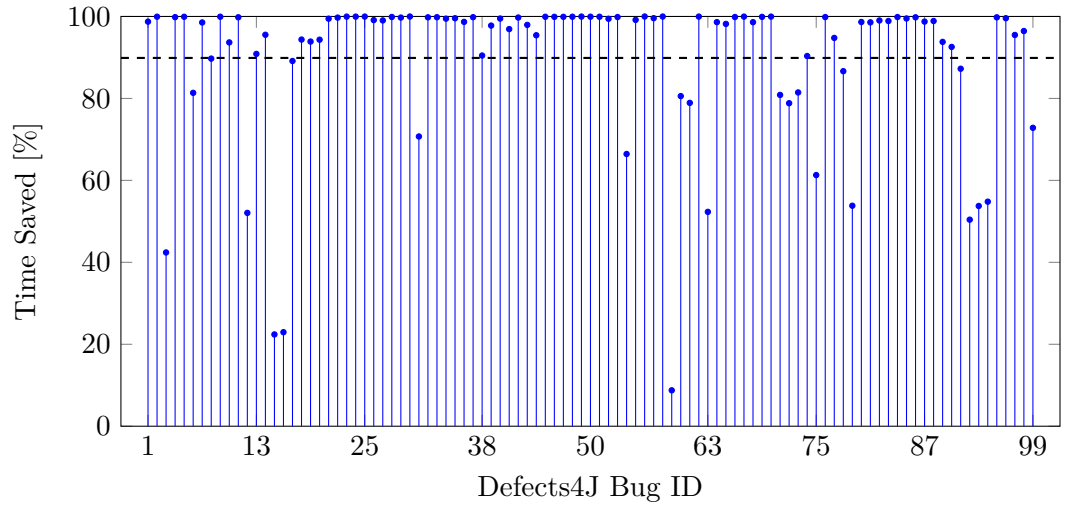
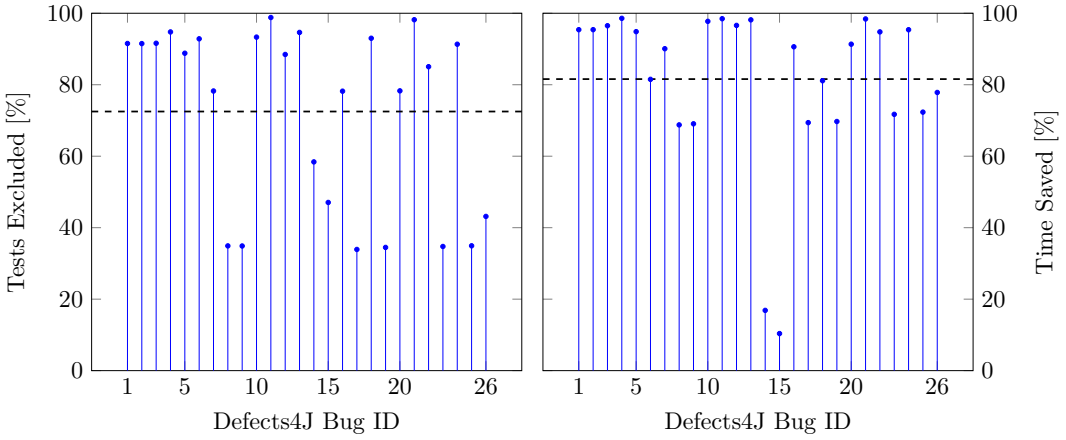


Figure 6.5: Apache Commons Math - Cover Changes Selection - Percentage of Saved Time. On average 95.24% of the tests were excluded.



(a) Tests Excluded

(b) Time Saved

Figure 6.6: Joda-Time - Cover Changes Selection. The chart on the left side shows how many tests were excluded. The chart on the right side shows how much time was saved by running only the selected tests. On average 72.5% of the tests were excluded and 81.58% of the time was saved.

Furthermore, there are two cases, bug 15 and 26, where the percentage of tests excluded is between 44.11% and 58.79%. In the other cases, at least 78.63% tests were excluded, with a maximum of 99.9% in bug 11. There are two outliers in the percentage of time saved, bug 14 and 15, with 17.0% and 10.5%. In the other cases at least 70.0% of the time was saved.

How fast does the size of the test suite grow with an increasing number of changed files?

The selection considers only changes of the most recent commit. This part of the evaluation was realized by investigating how fast the size of the optimized test suite grows with an increasing number of considered commits. The selection technique is executed 100 times. It starts with considering only the changes of the most recent commit, then the changes of the last two commits, up to the changes of the last 100 commits. The number of changed files includes every file, not only source files. The bugs 1 of Apache Commons Lang and 1 and 3 of Apache Commons Math were selected as examples for this evaluation.

Figures 6.7, 6.8 and 6.9 show the relative size and runtime of the optimized test suites in comparison to the number of Git commits back in history that were considered for the list of changed files. The X-Axis plots the number of commits that were considered for the set of changed files. The left side of the Y-Axis plots the relative size of the test suite and the right side plots the absolute number of changed files.

Apache Commons Lang Bug 1 In Apache Commons Lang Bug 1 (figure 6.7), the size of the optimized suite increases from 3.5% up to 23.2% after two commits, while the number of changed files increases from 2 up to 5. Then, the size increases proportionally to the number of changed files up to 57.6% and 58 changed files until commit 91. At commit 91, there is an increase of the test suite size up to 98.8% and an increase of the number of changed files up to 194. Finally, at commit 90, the technique selected 99.0% of the tests and 216 files are changed.

The time of the optimized test suite also makes a jump at commit 2, from about 0.01% to 21.5%. Until commit 19, it increases up to 21.8%. Then, the time increases to 32.2% and further increases until commit 88 to 35.9%. At commit 90 the time reaches 99.9% and stays at this level until commit 100.

Apache Commons Math Bug 1 The graph of Apache Commons Math with bug 1 (figure 6.8) shows an increase in the relative size of the test suite from 14.8% to 60.8% at commit 10, while the number of files increases from 20 to 21. The time increases at this point from 10.6% to 91.4%. Apart from that, there are no further increases of this magnitude, neither in size and time of the test suite nor in the number of changed files. After 100 commits, the optimized test suite has a relative size of 76.4% and a relative runtime of 95.7% and the set of changed files contains 190 elements.

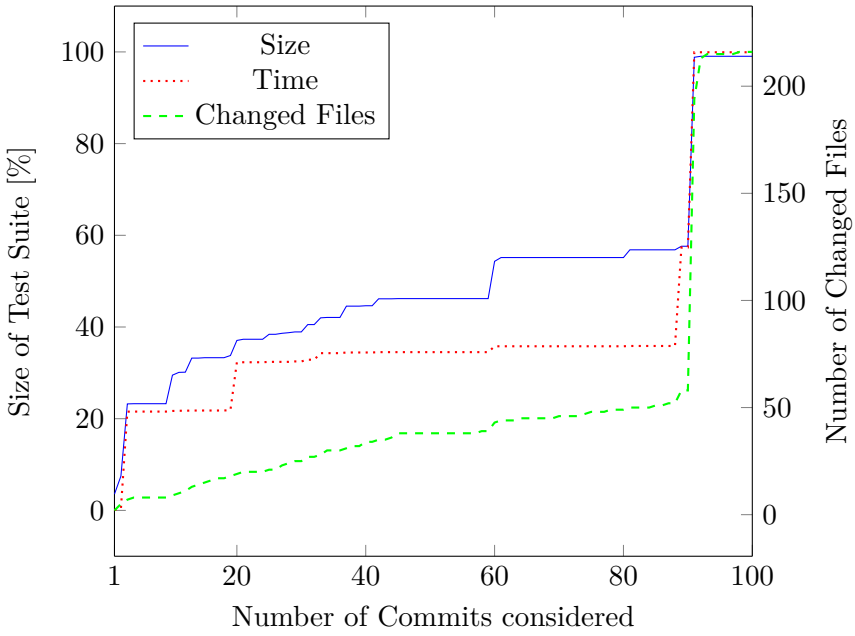


Figure 6.7: Apache Commons Lang with bug 1, size of test suite in dependence of number of commits considered to compute changed files

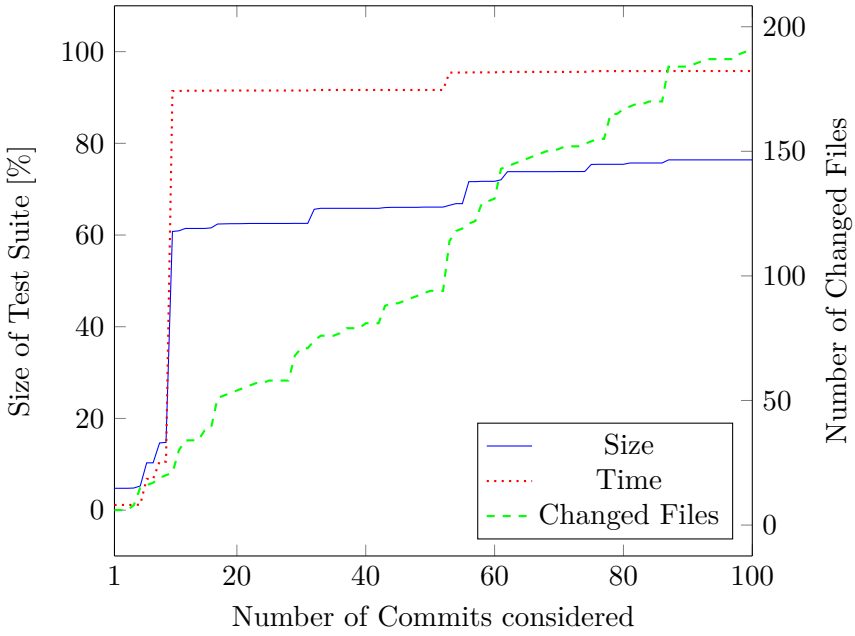


Figure 6.8: Apache Commons Math with bug 1, size of test suite in dependence of number of commits considered to compute changed files

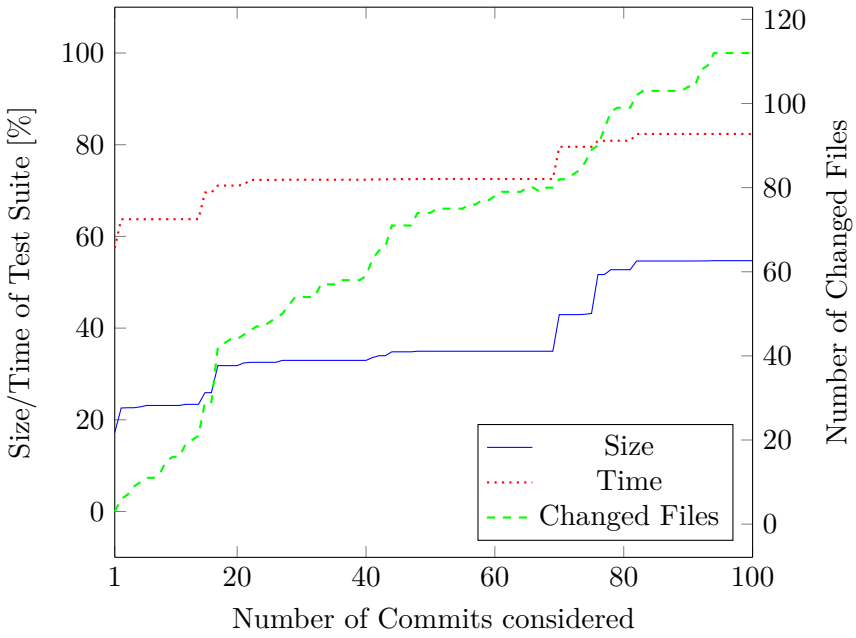


Figure 6.9: Apache Commons Math with bug 3, size of test suite in dependence of number of commits considered to compute changed files

Apache Commons Lang Bug 3 Apache Commons Lang with bug 3 (figure 6.9) starts with a test suite that has a relative size of 17.2% and a relative runtime of 57.5%. At commit 100 the test suite has a relative size of 54.7% and a relative time of 82.3%. The number of changed files starts with 3 and ends with 112. In the intervals 48 to 69, 27 to 40 and 82 to 91, the size and time do not change at all. There are no increases in size, time and number of changed files with the magnitude like in the previous two examples. From commit 17 to 69 the test suite size increases from 31.8% to 35.0% and the relative time increases from 71.1% to 72.5%, while the number of changed files increases from 42 to 80. It follows an increase in relative size to 54.6% and in relative time to 82.3% until commit 82. The number of changed files increases to 102.

6.2.2 Additional Coverage Prioritization

Can the Additional Coverage Prioritization technique increase the APFD value?

For this question, the test suites are optimized with the Additional Coverage Prioritization technique with file, method and statement level granularity. Then, the APFD metric is computed for each optimized suite and original suite. The versions provided by Defects4J contain by definition exactly one fault. Thus, the formula for the APFD value is $APFD = (1 - \frac{TF_1}{n} + \frac{1}{2 \cdot n}) \cdot 100$, where TF_1 is the position of the first failing test case and n is the total number of test cases.

The results are presented as box plots in figures 6.10, 6.11 and 6.12. The whiskers

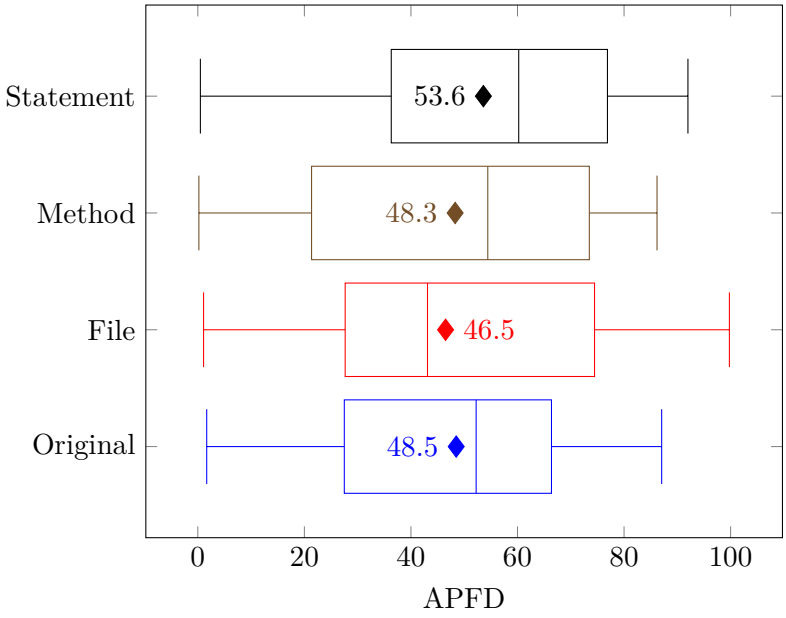


Figure 6.10: Additional Coverage Prioritization - Apache Commons Lang - APFD - Box plot

are from minimum to maximum, the boxes represent the middle 50% of all values and the line inside the boxes marks the median. The height of the boxes is the interquartile range and the diamonds mark the average APFD value.

Apache Commons Lang In Apache Commons Lang, the average APFD value of the original test suite is 48.5. The average value of the suites optimized with the technique on file and method level is below this with 46.5 and 48.3. With statement level, the average APFD is 53.6, which is an increase of about 10.5%. The median is also slightly higher on statement level with 60.23 compared to 52.24 of the original suites. The whiskers show that the values are widely spread. Minimum on statement level is 0.48 and maximum is 91.99.

Apache Commons Math In Apache Commons Math, method level brought the best results with an APFD of 65.4 on average. The original suite has an average APFD of 58.9. The file level technique could increase the average APFD value to 61.6. The values for the original suite range from 1.48 to 99.7. In the method level suite, the minimum is 7.92 and the maximum 99.89. The box of method level is smaller. It ranges from 47.38 to 85.41, which corresponds to an interquartile range of 38.03. The interquartile range of the optimized suite, however, is 49.99. This is an indication that the APFD values of the method level suite lie closer together than the values of the original suite.

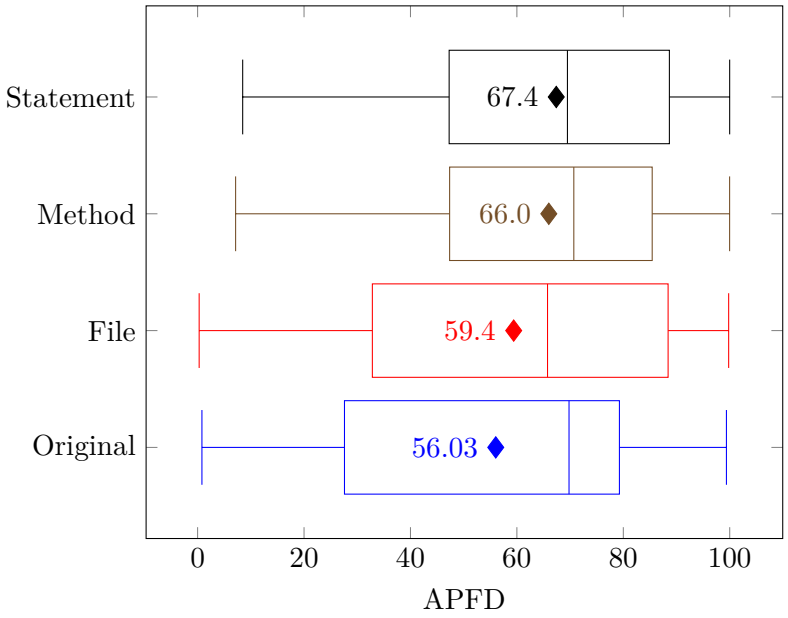


Figure 6.11: Additional Coverage Prioritization - Apache Commons Math - APFD - Box plot

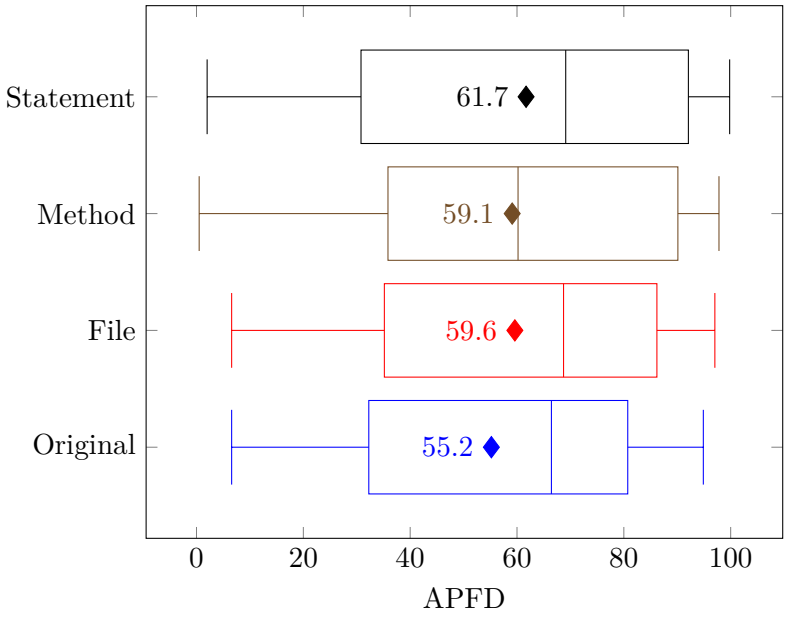


Figure 6.12: Additional Coverage Prioritization - Joda-Time - APFD - Box plot

Joda-Time In Joda-Time, again the finest granularity resulted in the highest APFD values. The statement level suite has an average APFD value of 61.7. The average value of the original suite is 55.2. This is an increase by 11.8%. The interquartile range of the statement level suite is with 61.3 higher than the interquartile range of the original suite with 48.5. Additionally, maximum and minimum are also further apart in the statement level suite. The maximum in the original suite is 94.88 and the minimum is 6.58. Against that, the maximum of the statement level suite is 99.81 and the minimum 1.99. This indicates that the values of the statement level suite are more distributed than the values of the original suite. The average of the method level suite is lower than the average of the statement level and file level suite.

6.3 Discussion

6.3.1 Select Tests Covering Changes

The *Select Tests Covering Changes* technique shows great potential to save time. But it did not perform equally well in all projects. In Apache Commons Lang and Apache Commons Math the technique selected on average about 4.6% of the test cases. In Joda-Time, however, the optimized test suite has an average size of 26.6%. Over all tested versions, on average 91.9% of the tests were excluded and 89.3% of the time was saved in comparison to running all test. These results correspond to other evaluations, e.g., by Rothermel et al. [RH97] [RH96], Gligoric [Gli15] and Dreier [Dre17].

The results also indicate that the size and runtime of the optimized test suite are highly dependent on which files were changed. For example, in bug 15, 16 and 59 of Apache Commons Math the technique selects about 56% of the tests. In these versions the class `FastMath` was modified. This is a utility class which is used by many other classes and therefore (indirectly) covered by many tests. The same thing happened in bug 8, 9, 17, 19, 23 and 25 of Joda-Time. In these cases, the technique selected about 64% of the tests on average. In all these versions the class `DateTimeZone` was modified. `FastMath` is also responsible for the huge increase in the number of tests in figure 6.8. Thus, the classes `FastMath` in Apache Commons Math and `DateTimeZone` in Joda-Time are covered by many tests.

The changes probably affect only a few lines of the 3800 lines in `FastMath` and 1300 lines in `DateTimeZone`. A finer technique that selects tests that cover changed methods or lines would probably select fewer tests in these cases. But the complexity of finer techniques is higher.

One has to keep in mind, that the time to generate coverage information and the runtime of Lazzer was not considered by the evaluation. The selection technique needed on average about 50 milliseconds to compute the optimized suite in all projects. The time to read in the coverage data was about 1 to 2 seconds. While in Commons Math and Lang the overhead is acceptable, the overhead in Joda-Time can nullify the time saving. The test suite of Joda-Time has an average runtime of about 3 seconds. Further improvements of the implementation might solve this problem.

Furthermore, the generation of the coverage data takes a long time. All tests need to be run for the initial generation. Additionally, the test runtime is higher with instrumentation compared to running tests without any instrumentation. The OpenClover documentation states that the slowdown is especially strong in CPU-intensive applications [Opea]. Apache Commons Math is given as an example there and it is stated that the test runtime is about two times higher with method level instrumentation and three times longer with statement level instrumentation. I experienced an even stronger slowdown.

6.3.2 Additional Coverage Prioritization

The evaluation of the *Additional Coverage Prioritization* revealed mixed results. Over all project versions the APFD value was increased by 16.8% on average. On the first sight, the increase of APFD seems to be lower than in other evaluations found in the literature. In these evaluations the additional method and statement coverage techniques significantly increased the APFD value [Rot+99] [DRK04] [EMR02]. In the study by Rothermel et al., for example, the average APFD value was increased by 38% with the statement level technique. But the results are limitedly comparable. Elbaum et al. state that "the relative effectiveness of prioritization techniques can vary across programs" [EMR02]. Prioritization techniques here include the additional coverage techniques. Three projects are probably too few for a meaningful comparison and the selected projects could be examples in which the technique does not perform well.

Furthermore, the experimental setup was different. In studies by Rothermel et al. [Rot+99] [RUC01] and other similar studies [DRK04] [EMR02], faulty versions were created by randomly seeding faults or adding them by hand. The faulty versions in studies by Rothermel et al. [Rot+99] [RUC01] and Elbaum et al. [EMR02] contained multiple faults and each fault caused one single test case to fail. The faulty version by Elbaum et al. contained between one and eight faults that can be detected by a test case. However, the faulty version provided by Defects4J contain only one fault, but may cause multiple tests to fail. When there are multiple faults, the additional coverage prioritization may increases the APFD more reliable. I assume, without proof, that the APFD value is more dependent on chance in test suites with only one fault than in suites with multiple faults. The risk that all fault revealing tests are positioned at the end of the test suite might be higher in versions with one fault than in versions with multiple faults.

Another issue is that prioritization was only done on test case level. However, Rothermel et al. and Elbaum et al. implemented the technique on test method level. Do et al. [DRK04] tested additional coverage prioritization technique on four open source Java programs and compared prioritization on test case level with prioritization on method level. They created 23 faulty versions by manually adding faults. The version contained in total 41 faults. Faults that were detected by more than 20% of the test cases were excluded. Their results indicate that the technique performs significantly worse when prioritizing on test class level, so they concluded that prioritization on test-class level seems not to improve the APFD value in general. The results by Do et al. are more similar to the results in this thesis than the results by Rothermel et al. [Rot+99] [RUC01]

or Elbaum et al. [EMR02]. Lazzer currently does not support prioritization on test method level, so I only considered prioritization on test class level. It would be interesting to see if prioritizing on test method level results in higher APFD values with the same evaluation setup.

Another problem, independent of APFD, is that the initialization of the coverage data store was slow. The technique itself needed on average about 300 milliseconds for the optimization in Apache Commons Lang. But the initialization of the coverage data store on statement level took on average 9.7 seconds. With method level granularity, the technique and the initialization of the coverage data store were about three times faster. The runtime of the test suite is on average 9.9 seconds. In Joda-Time this is even worse. The test suite has a runtime of about 3 seconds, but it took about 53 seconds to read in the coverage information on statement level. With method level granularity, the initialization was much faster, but it still took about 7 seconds. Apache Commons Math has the test suite with the highest runtime with at most about 100 seconds. The initialization in this project took about 7 seconds and the technique itself needed on average 550 milliseconds.

Furthermore, the generation of the coverage data with statement level granularity was very slow in Apache Commons Math. The coverage generation for the faulty versions 1 to 33 took on average about 60 minutes each. Without any instrumentation the average runtime of the tests in these versions is about 76 seconds. The documentation of OpenClover states for Apache Commons Math that the test runtime is about twice as long with instrumentation on method level and thrice as long with statement level instrumentation compared to running tests without any instrumentation. I am not sure why it was so much slower in my evaluation. In the other projects the instrumentation did not affect the test runtime that strong.

6.4 Threats to Validity

While I chose to construct an evaluation on the basis of Defects4J to create an environment closer to real world software projects, I encountered some downsides of that evaluation approach. The following weaknesses of evaluating optimization techniques on bugs from Defects4J could be found.

Not suited for regression test evaluation The bug revealing tests of the bugs provided by Defects4J are first implemented with the bug-fix. So the bugs in Defects4J are no bugs that were detected by a regression test. That is a problem, because these bugs are used to evaluate regression test optimization techniques. The evaluation simulates that a bug is introduced again, by reverting the bug-fix.

Bugs are isolated The bugs provided by Defects4J are isolated. This means, the commit that is labeled as buggy version by Defects4J, contains only changes that revert the bug fix. In consequence, only few files are changed by this commit. This could lead to unrealistically good results of the selection technique, because in practice probably more

files that are unrelated to a failing test are changed.

Given these disadvantages, it becomes clear that Defects4J is not perfectly suited for the evaluation of regression test optimization techniques. On the other hand, the usage of Defects4J brings also some major advantages. The biggest advantage is that the evaluation process is completely reproducible and can be easily automated. It is possible to rerun the evaluation with the same projects, as long as Defects4J exists and the tested bugs are not removed.

Furthermore, there are other threats to validity that are independent of Defects4J. Faults in the implementation cannot be ruled out. To keep this threat low, the implementation was tested before the techniques were evaluated. Another threat to validity is that the runtime of Lazzer, including the optimization process and generation of coverage data was excluded in the computation of saved time. This was done because the focus of this thesis is on the evaluation of the techniques itself, and not on the performance of the implementation. At some points, execution times are given to estimate the overhead.

7 Conclusion

Contents

This chapter gives a brief summary of this thesis. Furthermore, suggestions for future research and further improvements of the implementation are made.

7.1 Summary

Chapter 2 presented some existing implementations of regression test optimization techniques and their use in practice. Additionally, other techniques that can be found in the literature were presented.

Then, chapter 3 gave formal definitions for regression test optimization techniques. The framework for the evaluation of selection techniques by Rothermel et al. [RH94] and the APFD metric for prioritization techniques were explained. Furthermore, existing evaluations of optimization techniques were presented.

Chapter 4 presented the techniques that were selected for the evaluation and explains why these were chosen. These techniques are the *Cover Changes Selection* technique, the *Additional Coverage Prioritization* technique and a *History-Based Prioritization* technique. Furthermore, an overview of Lazzer's architecture was provided and the concepts of test coverage were explained.

Then, chapter 5 presented how the selected techniques were realized. The test coverage tools, JaCoCo and OpenClover, were compared and it was discussed why OpenClover was chosen. Afterwards, it was explained how OpenClover has been integrated in Lazzer and how a list of modified files is computed. The implementation of the techniques was explained and relevant parts of the source code were presented.

Chapter 6 presented the evaluation of the *Cover Changes Selection* and the *Additional Coverage Prioritization* techniques. First, the evaluation methodology was explained. The bug database Defects4J was used to get faulty versions of open-source Java programs. This approach had some downsides, but due to a lack of alternatives it was chosen

nevertheless. Defects4J entailed also advantages like a reproducible and automated evaluation.

Finally, the results were presented and discussed. The selection technique could reduce the number of test by 91.9% and the execution time by 89% compared to the original test suite. However, the additional coverage prioritization technique showed mixed results. The more precise version of the technique did not always resulted in a higher APFD value. On average, the APFD value was increased by 16.8% in comparison the APFD value of the original test suite. The time needed to generate and read in the coverage data was identified to be a possible problem in the practice.

7.2 Future Work

This section suggests topics for future research. Furthermore, possible improvements of Lazzer and the implemented techniques are proposed.

7.2.1 Overall Performance

This thesis compared the optimized test suites with the original test suites. But in practice the overall performance is important. According to Plewnia the runtime of Lazzer can be considered as negligible, especially in comparison to the runtime of the tests [Ple15]. The runtime of the selection technique was between 50 and 100 milliseconds and the runtime of the additional coverage techniques about 300ms. However, the runtime of the coverage data store with statement level granularity is a problem. It took nearly 10 seconds to read in the coverage data in Apache Commons Lang and in Joda-Time even 56 seconds. An optimization of this is absolutely necessary to get a reasonable runtime for techniques based on statement level test coverage data.

Another problem is that coverage data gets outdated. It does not suffice to compute coverage information once and use it forever. Source code is changed during development and thus, coverage information becomes obsolete. A possible solution for this is to update coverage information with every run. Pavan Kumar Chittimalli and Mary Jean Harrold [CH09] propose a "selective instrumentation" by which only tests are instrumented that were selected by the optimization technique. This way, coverage information is always up to date. In OpenClover it is possible to instrument specific files with the command line tool `CloverInstr` [Opeb]. But instrumented tests have a longer runtime. This effect was especially strong in Apache Commons Math. Thus, updating coverage data with every run might not be efficiently realizable with OpenClover. Other coverage tools might have a better performance.

Integration of RTO

A good integration of RTO into the development process is mandatory for a successful usage in practice. Lazzer can already be used as Maven plugin. Integrations in IDEs and continuous integration environments could improve Lazzer's usability.

To enable an efficient usage of RTO, a fully automated process including coverage generation, updating coverage data and executing tests is necessary. The generation and automated update of coverage data is not implemented in Lazzer yet. Thus, a future improvement of Lazzer should be a functionality to automatically generate coverage data and keep it up to date. Two examples for a complete integration of all needed steps were presented by Gligoric [Gli15] and Dreier [Dre17]. A short description of these two approaches can be found in chapter 2.

7.2.2 Metrics for Evaluating RTO Techniques

There are only few formally described metrics for the evaluation of regression test optimization strategies. There is the framework for evaluating selection techniques by Rothermel et al. [RH94] that defines some qualities of selection techniques that help to rate a selection strategy. For prioritization techniques there does not exist such a framework. An often used metric to rate the effects of a prioritization technique is APFD. However, a high APFD value is not proven to be relevant in practice at all.

Further metrics are needed to rate the effectiveness of techniques depending on the usage scenario. Some techniques may show good performance on a build server, but are less successful on developer's machines. Additionally, some techniques are based on additional information that are in general not available and cannot be generated. A comprehensive set of metrics that measure a wide range of qualities could help to decide which technique is most suitable for a specific project.

7.2.3 Practical Evaluation Approach

To find out more about the effectiveness of regression test optimization in practice, the techniques need to be evaluated in a real-world environment. But as Milos Gligoric said: "there is no dataset that would allow executing tests the same way that developers executed them in between commits" [GEM15]. Generating a history with the help of a version control system (VCS) is not useful. In general, changes that are pushed to the VCS are already tested by the developer. Especially history based techniques are therefore hard to evaluate.

Experimenting with regression test optimization techniques during the development of a software is rather undesired. To solve this, data could be collected on the developer's machines by a tool that collects data from every test run. After some time enough data is available to check if the usage of regression test optimization techniques would have saved time.

7.2.4 Bug Database for Regression Testing Evaluation

Defects4J is, to the best of my knowledge, currently the only project of its kind for Java projects. As already discussed, Defects4J is not well suited for the evaluation of regression test optimization techniques. But due to a lack of alternatives and its advantages I decided to use it nevertheless. A structured database like Defects4J, but with focus on

regression testing would be a great help in the evaluation of RTO techniques. At best, this database provides data that allows executing tests like developers do in between commits. This would not only give more realistic results for all techniques, but it would make it possible to evaluate history based techniques. The database could for example be filled with the previously proposed approach.

7.2.5 Granularity

The selection technique was implemented only on file level. A finer granularity may make the selection more precise, i.e., less non-failing tests are selected, but it also increases the complexity. Thus, a question for future research is if finer coverage granularity brings better results and if it is worth the additional effort.

Lazzer currently supports only prioritization on test class level. The additional coverage prioritization could not reliably increase the APFD value. The evaluation by Do et al. [DRK04] indicates that the additional coverage prioritization technique yields much better results with prioritization on test method level. It would be interesting to see if prioritization on test method level can increase the APFD more reliably in projects from Defects4J than prioritization on test class level.

Bibliography

[BM07] R. C. Bryce and A. M. Memon. "Test Suite Prioritization by Interaction Coverage". In: *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*. DOSTA '07. Dubrovnik, Croatia: ACM, 2007, pp. 1–7. ISBN: 978-1-59593-726-1. DOI: 10.1145/1294921.1294922. URL: http://doi.acm.org/10.1145/1294921.1294922 (cited on page 6).

[CH09] P. K. Chittimalli and M. J. Harrold. "Recomputing coverage information to assist regression testing". In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 452–469. ISSN: 00985589. DOI: 10.1109/TSE.2009.4 (cited on page 46).

[Def] Defects4J. *Defects4J GitHub Page*. URL: https://github.com/rjust/defects4j (visited on 09/04/2018) (cited on page 29).

[Dre17] F. Dreier. "Obtaining Coverage per Test Case". Master Thesis. 2017, p. 40 (cited on pages 5, 10, 16, 18, 41, 47).

[DRK04] H. Do, G. Rothermel, and A. Kinneer. "Empirical Studies of Test Case Prioritization in a JUnit Testing Environment". In: *Proceedings of the 15th International Symposium on Software Reliability Engineering*. ISSRE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 113–124. ISBN: 0-7695-2215-7. DOI: 10.1109/ISSRE.2004.18. URL: http://dx.doi.org/10.1109/ISSRE.2004.18 (cited on pages 11, 42, 48).

[EE15] E. D. Ekelund and E. Engström. "Efficient regression testing based on test history: An industrial evaluation". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 449–457. DOI: 10.1109/ICSM.2015.7332496 (cited on pages 5, 10).

[EMR00] S. Elbaum, A. G. Malishevsky, and G. Rothermel. "Prioritizing Test Cases for Regression Testing". In: *SIGSOFT Softw. Eng. Notes* 25.5 (Aug. 2000), pp. 102–112. ISSN: 0163-5948. DOI: 10.1145/347636.348910. URL: http://doi.acm.org/10.1145/347636.348910 (cited on page 11).

[EMR02] S. Elbaum, A. G. Malishevsky, and G. Rothermel. "Test Case Prioritization: A Family of Empirical Studies". In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 159–182. ISSN: 0098-5589. DOI: 10.1109/32.988497. URL: http://dx.doi.org/10.1109/32.988497 (cited on pages 2, 9, 11, 42, 43).

[GEM15] M. Gligoric, L. Eloussi, and D. Marinov. "Ekstazi: Lightweight Test Selection". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 713–716. URL: http://dl.acm.org/citation.cfm?id=2819009.2819146 (cited on pages 5, 10, 47).

[Gli] M. Gligoric. *Website of Ekstazi*. URL: http://www.ekstazi.org/ (visited on 09/04/2018) (cited on page 5).

[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: ACM, 2014, pp. 361–372. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643019. URL: http://doi.acm.org/10.1145/2642937.2643019 (cited on page 1).

[Gli15] M. Z. Gligoric. "Regression Test Selection: Theory and Practice". Dissertation. 2015, p. 112. URL: http://hdl.handle.net/2142/88038 (cited on pages 5, 10, 16, 41, 47).

[Gra+01] T. L. Graves et al. "An Empirical Study of Regression Test Selection Techniques". In: *ACM Trans. Softw. Eng. Methodol.* 10.2 (Apr. 2001), pp. 184–208. ISSN: 1049-331X. DOI: 10.1145/367008.367020. URL: http://doi.acm.org/10.1145/367008.367020 (cited on page 7).

[Har+01] M. J. Harrold et al. "Regression Test Selection for Java Software". In: *SIGPLAN Not.* 36.11 (Oct. 2001), pp. 312–326. ISSN: 0362-1340. DOI: 10.1145/504311.504305. URL: http://doi.acm.org/10.1145/504311.504305 (cited on page 7).

[Her+15] K. Herzig et al. "The Art of Testing Less Without Sacrificing Quality". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 483–493. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818815 (cited on pages 5, 7, 10).

[HS88] M. J. Harrold and M. L. Souffa. "An incremental approach to unit testing during maintenance". In: *Proceedings. Conference on Software Maintenance, 1988.* 1988, pp. 362–367. DOI: 10.1109/ICSM.1988.10188 (cited on pages 1, 7).

[Inf] Infinitest. *Website of Infinitest*. URL: http://infinitest.github.io/ (visited on 09/04/2018) (cited on page 5).

[JJE14] R. Just, D. Jalali, and M. D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. 2014. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055 (cited on page 28).

[KP02] J.-M. 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 '02* (2002), p. 119. ISSN: 02705257. DOI: 10.1145/581339.581357. URL: http://portal.acm.org/citation.cfm?doid=581339.581357 (cited on pages 6, 16, 24).

[Lu+16] Y. Lu et al. "How does regression test prioritization perform in real-world software evolution?" In: *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. 2016. ISBN: 9781450339001. DOI: 10.1145/2884781.2884874 (cited on page 2).

[Opea] OpenClover. *OpenClover 4.2.0 Documentation - Clover Performance Tuning*. URL: http://openclover.org/doc/manual/4.2.0/ant--runtime-performance-tuning.html (visited on 09/04/2018) (cited on page 42).

[Opeb] OpenClover. *OpenClover 4.2.0 Documentation - CloverInstr*. URL: http://openclover.org/doc/manual/4.2.0/commandline--cloverinstr.html (visited on 09/04/2018) (cited on page 46).

[Opec] OpenClover. *OpenClover 4.2.0 Documentation - Database Structure*. URL: http://openclover.org/doc/manual/4.2.0/developer-guide--database-structure.html (visited on 09/04/2018) (cited on page 19).

[Oped] OpenClover. *Website of OpenClover*. URL: http://openclover.org (visited on 09/04/2018) (cited on page 5).

[Ple15] C. Plewnia. "A Framework for Regression Test Prioritization and Selection". Master Thesis. 2015 (cited on pages 13, 14, 46).

[PRB08] H. Park, H. Ryu, and J. Baik. "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing". In: *Proceedings - The 2nd IEEE International Conference on Secure System Integration and Reliability Improvement, SSIRI 2008* (2008), pp. 39–46. DOI: 10.1109/SSIRI.2008.52 (cited on page 16).

[RH94] G. Rothermel and M. J. Harrold. "A Framework for Evaluating Regression Test Selection Techniques" Research Paper". In: *ICSE '94 Proceedings of the 16th international conference on Software engineering* (1994), pp. 201–210 (cited on pages 7, 45, 47).

[RH96] G. Rothermel and M. J. Harrold. "Analyzing Regression Test Selection Techniques". In: *IEEE Trans. Softw. Eng.* 22.8 (Aug. 1996), pp. 529–551. ISSN: 0098-5589. DOI: 10.1109/32.536955. URL: http://dx.doi.org/10.1109/32.536955 (cited on pages 7, 41).

[RH97] G. Rothermel and M. J. Harrold. "A Safe, Efficient Regression Test Selection Technique". In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (Apr. 1997), pp. 173–210. ISSN: 1049-331X. DOI: 10.1145/248233.248262. URL: http://doi.acm.org/10.1145/248233.248262 (cited on pages 7, 41).

[Rot+99] G. Rothermel et al. "Test Case Prioritization: An Empirical Study". In: *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 179–. ISBN: 0-7695-0016-1. URL: http://dl.acm.org/citation.cfm?id=519621.853398 (cited on pages 6, 9, 11, 16, 42).

[RUC01] G. Rothermel, R. J. Untch, and C. Chu. "Prioritizing Test Cases For Regression Testing". In: *IEEE Trans. Softw. Eng.* 27.10 (Oct. 2001), pp. 929–948. ISSN: 0098-5589. DOI: 10.1109/32.962562. URL: https://doi.org/10.1109/32.962562 (cited on pages 2, 6, 11, 42).

[Sin+12] Y. Singh et al. "Systematic literature review on regression test prioritization techniques". In: *Informatica (Slovenia)* 36.4 (2012), pp. 379–408. ISSN: 03505596 (cited on page 1).

[SOZ11] S. K. Said, R. R. Othman, and K. Z. Zamli. "Prioritizing interaction test suite for t-way testing". In: *2011 Malaysian Conference in Software Engineering*. 2011, pp. 292–297. DOI: 10.1109/MySEC.2011.6140686 (cited on page 6).

[ST02] A. Srivastava and J. Thiagarajan. "Effectively Prioritizing Tests in Development Environment". In: *SIGSOFT Softw. Eng. Notes* 27.4 (July 2002), pp. 97–106. ISSN: 0163-5948. DOI: 10.1145/566171.566187. URL: http://doi.acm.org/10.1145/566171.566187 (cited on page 6).

[TTL89] A. Taha, S. M. Thebaut, and S. Liu. "An approach to software fault localization and revalidation based on incremental data flow analysis". In: *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*. 1989, pp. 527–534. DOI: 10.1109/CMPSAC.1989.65142 (cited on pages 1, 7).

[Won+97] W. E. Wong et al. "A Study of Effective Regression Testing in Practice". In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering*. ISSRE '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 264–. ISBN: 0-8186-8120-9. URL: http://dl.acm.org/citation.cfm?id=851010.856115 (cited on page 16).

[YH07] S Yoo and M Harman. "Regression Testing Minimisation, Selection and Prioritisation : A Survey". In: *Test. Verif. Reliab* 00 (2007), pp. 1–7. DOI: 10.1002/000. URL: www.interscience.wiley.com (cited on pages 1, 7).

[Apa] Apache Software Foundation. *Maven Surefire Plugin - Inclusions and Exclusions of Tests*. URL: https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html (visited on 09/04/2018) (cited on page 25).

[Ecl] Eclipse Foundation. *JGit API Documentation*. URL: http://download.eclipse.org/jgit/site/4.0.1.201506240215-r/apidocs/index.html (visited on 09/04/2018) (cited on page 21).