Regression Test Selection for Fast Feedback when Developing for the Android Platform

LUND UNIVERSITY
Campus Helsingborg

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:

Ali Muthanna
Ludwig Hellgren Winblad

Examiner
Christin Lindholm (LTH)

Supervisors
Emelie Engström (LTH)
Martin Frid (Sony Mobile Communications)

© Copyright Ali Muthanna, Ludwig Hellgren Winblad

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2018

Abstract

The work for this thesis was conducted at a department of Sony Mobile Communications, with the purpose of incorporating regression testing into the daily workflow of developers. Today regression testing is conducted by executing a test suite of approximately 58 000 tests to verify modifications done to code. Because of the long runtime, execution occurred during the night after modifications were done throughout the day. The problem was that developers had to wait until the next day to receive feedback on their changes. To be able to evaluate a possible solution, Empiricist was developed with the ability to recommend a smaller subset of the test suite given specific source code modifications.

Empiricist is based on a *regression test selection* approach: the history-based approach, where the basic outline is that historical data from previous test runs is used to construct a database of links between tests and source entities. Empiricist can then, given one or several source code modifications, query the database to recommend several tests to be run for verification.

The history-based approach that was used was extended to support multiple granularities when linking tests to modifications:

- *very fine-grained* where tests are linked to files
- *fine-grained* where tests are linked to directories
- *coarse-grained* where tests are linked to projects

Additionally, three approaches to recommending tests are supported: 1) *Connection-Based Recommendation (CBR)* where tests are recommended if a link exist between them and the modification; 2) *Normalized Connection-Based Recommendation (NCBR)*, with the same functionality as CBR but weights on links are normalized and subsequently subject to a cutoff criterion within the interval [0,1] in order to further reduce the subset; 3) *Recommend All Flipped (RAF)* where all the previously flipped tests are recommended.

The evaluation of the results of Empiricist show that it is possible to verify modification with relatively high certainty while only a small portion of the tests in the test suite is chosen for execution. Faster feedback is therefore achieved. For example, using CBR with coarse granularity the test suite was reduced by up to 99% while a large portion of all the tests that would have failed were recommended.

Keywords: Regression Test Selection (RTS), Android, Regression Testing

Sammanfattning

Detta examensarbete är utfört på Sony Mobile Communications med syftet att inkorporera regressionstestning i utvecklares dagliga arbetsflöde. I dagsläget utförs regressionstestning genom att exekvera en testsvit på ca 58 000 tester för att verifiera kodändringar. På grund av testsvitens långa exekveringstid så kördes den under natten efter att kodändringar gjorts under dagen. Problemet var då att utvecklare var tvungna att vänta tills nästa dag innan de kunde få någon feedback på deras kodändringar. För att utvärdera en möjlig lösning utvecklades Empiricist som dynamiskt ska kunna rekommendera att köra en mindre del av hela testsviten för specifika kodändringar.

Empiricist är byggt på en *regression test selection* teknik: den historiebaserade tekniken där historisk data från tidigare testkörningar används för att bygga upp en databas av länkar mellan källkodsentiteter och tester. Empiricist ska då kunna, givet en eller flera kodändringar, fråga databasen för att sedan rekommendera ett antal tester som borde köras för verifiering.

Den historiebaserade teknik som användes utökades sedan till att även innefatta att länkar etableras på olika granularitetsnivåer:

- *very fine-grained* där test är kopplade till filer
- *fine-grained* där test är kopplade till mappar
- *coarse-grained* där test är kopplade till projekt

Utöver det stödjs tre olika tillvägagångssätt för att rekommendera tester: 1*) Connection-Based Recommendation (CBR)* där tester rekommenderas om en länk hittas mellan testet och ändringen; 2) *Normalized Connection-Based Recommendation (NCBR)* som har samma funktionalitet som CBR men med tillägget att vikterna som är kopplade till länkarna normaliseras vilket sedan används för att ytterligare reducera testerna genom att utsätta de normaliserade vikterna för en viss tröskel inom intervallet [0, 1]; 3) *Recommend All Flipped (RAF)* där alla tester som tidigare flippat rekommenderas.

Utvärderingen av resultaten från Empiricist visar tydligt att det är möjligt att verifiera ändringar med en relativt hög säkerhet samtidigt som endast ett mindre antal tester väljs ut för att köras från testsviten. Exempelvis, när CBR användes med *coarse-grained* granularitet reducerades testsviten med uppemot 99% samtidigt som en stor andel av testerna som skulle ha misslyckats faktiskt rekommenderades.

Nyckelord: Regressionstestning, Android, Kodverifiering

Acknowledgments

Special thanks to our two supervisors Martin Frid at Sony for all the support and for always managing to bring up difficult but thoughtful points, and Emelie Engström at LTH for always believing in us and giving us inspiration and invaluable feedback on our work. Thank you to Kjell Erkstam for giving us the opportunity to conduct this work. Last but not least, thank you to Florian Eisl and all the others that helped by being there to answer our questions.

This work could not have been done without you!

Table of Contents

CHAPTER 1 INTRODUCTION **1**

1.1 BACKGROUND 1
1.2 PURPOSE 2
1.3 MOTIVATION 2
1.4 SUMMARY OF THESIS QUESTIONS 2
1.5 CONSTRAINTS 3
1.6 OUTLINE 3

CHAPTER 2 RELATED WORK **5**

CHAPTER 3 METHODOLOGY **7**

3.1 OVERVIEW 7
3.1.1 Understanding the Problem & Selecting RTS Technique 8
3.1.2 Empiricist Implementation & Evaluation 9

CHAPTER 4 SURVEY OF RTS TECHNIQUES **11**

4.1 PROBLEM DOMAIN 11
4.1.1 Android Open Source Project 11
4.1.2 Available Data 12
4.2 REGRESSION TEST SELECTION TECHNIQUES 13
4.2.1 Deep Learning 13
4.2.2 History-Based 13
4.2.3 Static Analysis 14
4.3 DISCUSSION 15
4.4 CONCLUSION 15

CHAPTER 5 SOFTWARE **19**

5.1 EXTENDED HISTORY-BASED TECHNIQUE 19
5.2 EMPIRICIST 19
5.2.1 Implementation 21
5.3 NORMALIZATION 23
5.4 FILTERING 23
5.5 VERSIONS 24
5.5.1 Empiricist 1.1 (Granularity) 24
5.5.2 Empiricist 2.0 (Separation of Projects) 24
5.5.3 Empiricist 2.1 (Normalizing after Selecting Subset) 24
5.5.4 Empiricist 2.2 (Eliminating the Impact of Number of Files Contained in Directory) 25
5.5.5 Empiricist 3.0 (Introduction of Coarse-grained Granularity) 25

CHAPTER 6 EVALUATION **27**

6.1 EVALUATION TERMINOLOGY 27
6.2 DEPENDENT VARIABLES 28
6.2.1 Reduction 28
6.2.2 Inclusiveness 28
6.2.3 Precision 29
6.2.4 Effectiveness 29
6.2.5 Average Number of Tests 29
6.3 INDEPENDENT VARIABLES 30
6.4 VERIFIER 31
6.5 THESIS QUESTIONS 32

CHAPTER 7 RESULT **33**

7.1 CONSTRAINTS IMPOSED BY TARGET CODEBASE & AVAILABLE DATA 33
7.2 RESULTS FROM VERIFIER 34
7.2.1 Inclusiveness 34
7.2.2 Precision 37

7.2.3 Reduction & Number of Tests *40*
7.3 RANDOM SELECTION 43
CHAPTER 8 DISCUSSION 45
8.1 EMPIRICIST 45
8.2 THREATS TO VALIDITY 49
8.3 UNIT TESTS 50
8.4 ETHICAL ASPECTS 50
CHAPTER 9 CONCLUSION 51
CHAPTER 10 FUTURE WORK 53
GLOSSARY 57
REFERENCES 59
APPENDIX A TABLES 61

Chapter 1
Introduction

The background necessary to understand the reasons and motivations for why this thesis was conducted is presented in this chapter.

1.1 Background

Google requires that their compatibility test suite (CTS) is run and its tests pass before companies can label and sell their devices as *Android*. The same applies to software that is to be run on Android-labelled devices. The devices that conform to Google's definition of compatibility allow apps created by different developers to work on the same platform version of Android [1-2]. Because Sony Mobile Communications (Sony Mobile) releases Android-labelled smartphones and monthly software updates, they continually execute CTS.

The compatibility test suite is a regression test suite used to ensure correctness when software has undergone modification. To date it contains approximately 700 000 tests. It takes 16 to 18 hours to complete when run concurrently on five to six Sony smartphones of the same product.

Because of CTS's size and long execution time, the test suite is only guaranteed to be executed at the verification phase before a release is intended. However, any faults detected at that time would cause considerably large losses since the release would have to be delayed. As such, it is important for each part of the software to have been sufficiently tested before this phase. To this end, to manage the test suite's long execution time and the inefficiency of executing it completely, CTS subsets are executed during the development phase to accommodate for more frequent testing.

The selected tests in a subset are considered to cover the part of the software that a department of Sony Mobile is working on. This selection is based on the understanding and experience that individuals have with CTS within a department.

The department where this thesis work was conducted has an automated testing process of executing their CTS subset (CTS*). The subset contains approximately 58 000 tests and a run takes between 11 and 14 hours on one Sony smartphone. It is executed overnight after modifications to code have been made throughout the day. However, although testing is more frequent, developers still have to wait until the next day to receive feedback on their modifications.

Empiricist is a tool that utilizes a regression test selection technique to dynamically select tests from CTS given changes. This is used to support the selection of CTS tests relevant to the developers' continuous modifications. A tool with the ability of such precise test selection allows for faster feedback.

In this report the implementation and evaluation of the extended history-based technique is presented. The performance of the implementation is also discussed here.

1.2 Purpose

The purpose of this thesis is to incorporate testing into the daily workflow by making it possible for developers to receive feedback relatively quickly after changes have been made. The goal was to develop a tool, using a regression test selection technique, that dynamically selects tests depending on the changes.

Regression test selection can be considered an appropriate approach to improve regression testing in Sony Mobile since their interest is to allow their developers to run tests that verify their specific code changes during the day. If developers could test their changes faster, faults could be detected and fixed earlier meaning that delays could potentially be avoided. The objective being that resources could be spent more efficiently to allow Sony Mobile to possibly include more features in their software and release patches faster to their users.

1.3 Motivation

For the authors of this thesis, the choice to conduct this work was motivated by the interest in the subject as well as the opportunity to apply and develop knowledge.

Sony Mobile's motivation for this thesis was to evaluate the possibility of incorporating testing into the daily workflow of developers. More frequent and faster response time for the testing of code changes would enable resources to be used more efficiently.

1.4 Summary of Thesis Questions

This thesis aims to provide answers to whether it is possible to reduce the time needed to get feedback on code changes by selecting a small subset of CTS* that still reveals most bugs. Further, the thesis aims to explore how much and what kind of data will be needed for this to work.

1.5 Constraints

The thesis is limited to developing a prototype to evaluate the extended history-based technique.

1.6 Outline

This report consists of four segments: the opening, the solution, the evaluation and the closing segment. The opening segment contains Chapters 1, 2 and 3. This section describes the underlying problem and the solutions that have been employed in other works as well as the purpose, goal and the questions to be answered under this thesis. The approaches taken to conduct the work for this thesis are also explained here.

The solution segment consists of Chapters 4 and 5. The chosen regression test selection technique for this work is explained here in relation to the problem domain and other techniques as well as the implementation of the chosen technique.

The evaluation segment includes Chapters 6, 7 and 8. Information needed for the evaluation as well as the results are presented and discussed here. The thesis questions are presented in detail in Section 6.4 since it is required that the terminology in Chapter 6 is understood beforehand.

The closing segment contains the conclusion and future work, Chapters 9 and 10 respectively. The rest of this segment consists of the glossary, references and an appendix.

Chapter 2
Related Work

This chapter provides information associated with the subject of regression testing as well as some other works dealing with similar problems as this thesis.

Regression testing is done to ensure that software modifications are correct while also ensuring that the intended functionalities of the unmodified parts are not adversely affected. In regression testing a pre-existing test suite is reused to test the modifications [3]. One approach to regression testing is *retest-all*, where all tests are rerun [3]. However, this approach can become expensive as the size of the test suite grows [4].

One way to improve regression testing is by using regression test selection (RTS) techniques. RTS looks to select a subset of tests that will test the changed parts of the software [4]. A multitude of different approaches to implementing RTS have been explored and evaluated in literature.

Árpád Beszédes et al. [5] implemented a regression test selection technique where they dynamically mapped C++ source files to test cases at runtime on a method level using instrumentation available in the GCC compiler. This technique was used in combination with prioritization to reduce the number of test cases recommended by the purely change based selection. It was developed and evaluated for the C++ part of the WebKit open source project where it recommended 50% of all the failing tests while 10% of the test suite was being selected. The drawback of this technique is that it is not easily portable to an environment with different programming languages as instrumenting code requires language specific tools. The structure of the Android operating system and the complexity and depth of its software stack, as well as the test suite, prevents focus to be placed on a single language. C++ is more prevalent in most layers of the source code while java is more prevalent in the API as well as in the test suite.

Fazeli Negar [6] used and evaluated machine learning, specifically logistic regression and random forests algorithm, as an RTS technique to generate the likelihood of any test case failing. The technique achieved a reduction of 13.8% of unnecessary runs of test cases with the goal of all failing test cases being selected. Although the results were far from their goals of reduction, Fazeli had several limitations on the available data that are not present in our environment. The source code revision system used in his case was ClearCase which, due to the way it works, prevented knowledge about which versions of which files corresponded to a specific change. Versions are stored on a file level, as opposed to on a repository level. In our case git is used, where these limitations are not present.

Ekelund and Engström [7] explored a technique based on historical data from test suite executions. A database was constructed and maintained with correlations between source code packages and tests. A correlation of this kind was said to be found if a source package changed and the outcome of a test flipped between two test suite executions. The term flipped here means that either a test failed before and now succeeds, or it succeeded before and now fails. In addition to identifying correlation, each such correlation had a weight assigned to it, which was incremented every time the correlation was found, to measure the strength of the correlation, i.e. how many times it had appeared. Although weights were never used, the thinking behind it was to be able to distinguish between true correlations and correlations simply resulting from random noise. The technique was found to generate the best results using the history of between 50 to 100 test suite executions. After around 100 test suite executions it eventually approached the equivalence of running the entire test suite, since the weights were not used. In addition to the need for training, another problem that might be posed by this approach in an environment where the size of the test suite is sufficiently large enough is whether during a given time interval only a subset of tests recurrently fails or flips. If the test suite is large enough and different tests tend to flip between runs, it would naturally take a long time to train such a database to become effective.

Chapter 3
Methodology

This chapter provides information regarding the methods chosen to conduct this thesis work.

3.1 Overview

The work that was conducted for this thesis can be generally split into two phases, as shown in Figure 3.1: 1) understanding the problem and selecting RTS technique; 2) Empiricist implementation and evaluation. The aim of the first phase was to identify the problem and to select a suitable approach that could be taken to solve or manage the problem. It was in this phase that the history-based approach was selected. This phase covered mostly the theoretical part of the thesis. In the second phase, the previously collected information was used to implement Empiricist. Experiments were conducted to determine the best design.

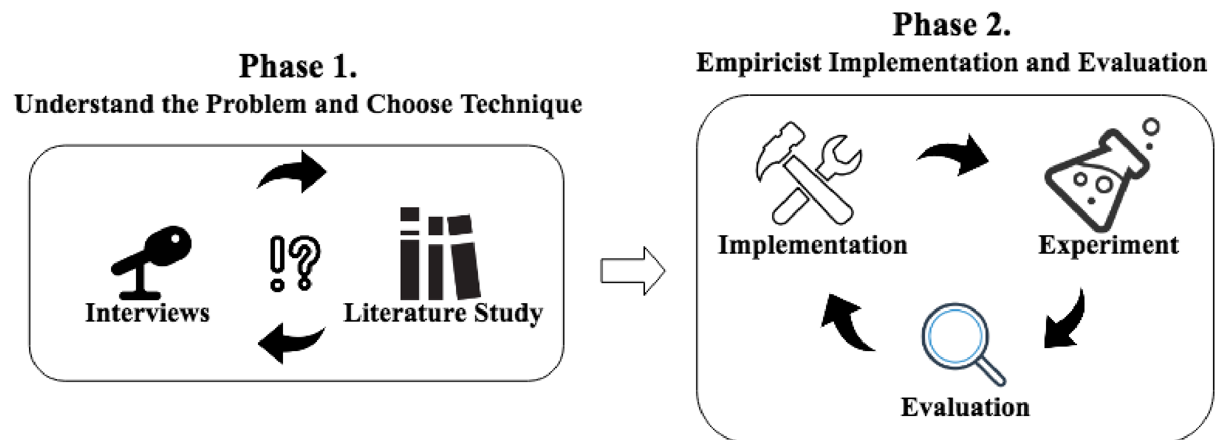

Figure 3.1: A depiction of the two work phases. The first phase was an iterative process in which literature studies and interviews were conducted to understand the problem as well as choosing a suitable technique. In the second phase the technique is implemented in the form of Empiricist and experiments were conducted to evaluate design ideas.

The overall work was iterative. A list of tasks was set up and regularly updated. At the end of every week, tasks were chosen from the list based on their importance. This approach was best suited since it could manage acute situations that could arise which gave room for flexibility of how and when to handle these situations.

3.1.1 Understanding the Problem & Selecting RTS Technique

A combination of interviews and literature studies was conducted to understand the problem, as shown in Figure 3.2. Interviews were held with the individuals in the department that this work was conducted at. One interviewee was the verification engineer that wanted to find a solution to the problem, another was the test expert that designed the automated test process used in the department and is responsible for the CTS test runs.

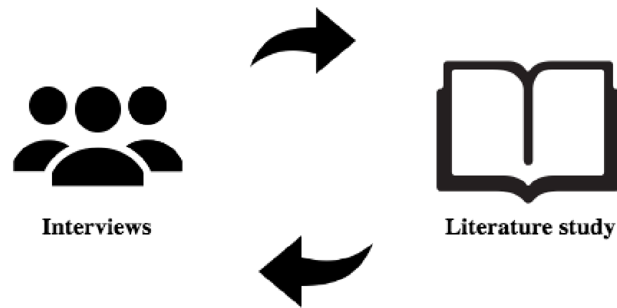

Figure 3.2: A depiction of the iterative process where interviews and literature studies were conducted to understand the problem.

At first, unstructured interviews were held to gather as much information as possible regarding CTS, the problem domain and the problem that exists concerning regression testing. Thereafter interviews became more structured when the amount of information about the problems and the problem domain increased. Some of the things mentioned in *threat to validity* in Section 8.2, such as the invalid information that is sometimes present in test results came as a direct result of the interviews.

While interviews were conducted, a literature study about regression testing and regression test techniques was also being done. An iterative process of conducting interviews and then using the knowledge obtained during the literature study to ask even more specific questions was formed.

The result from the first iteration of interviews and literature study was used in a second iteration of the same process, to dig deeper into possible solutions. The second literature study was focused on regression test selection techniques. These techniques were compared to each other and their applicability was assessed in view of the problem domain. Finally, meetings were held to discuss the practicality of the techniques and to select the most applicable one. The first prototype of Empiricist was designed and implemented based on the chosen technique.

The criteria for selecting sources when conducting the literature studies were that they needed to be trustworthy and if possible up-to-date. Scientific papers can be considered trustworthy since they are peer-reviewed, meaning that researchers in the same field have verified them. Sources 3-5, 7, 12, 16 and 19-20 in the reference list fall into this category.

Most of the papers were found in the reference lists of other scientific papers, others were found by searching for keywords, such as regression testing, through LUBsearch and Google Scholar.

Other than scientific papers, information about Android and CTS, sources 1-2 and 8-10, that are provided by Google on their official websites were used since they can be deemed trustworthy considering that they are the creators. Sources 13-15 and 17, provided by creators of tools such as Javaparser, Clang and more, were also considered trustworthy by the same logic.

The books, sources 11 and 18 can be considered trustworthy since they are written by experts in the topic covered by the books.

The thesis, source 6, can be considered trustworthy considering that it passed the standards for a master thesis in Chalmers University of Technology.

3.1.2 Empiricist Implementation & Evaluation

During the development of Empiricist, five experiment sessions, one experiment after each version of Empiricist were conducted to evaluate the different design alternatives. This approach was deemed most appropriate as logical flaws could become more apparent while the independent variables were being tweaked. The different versions of Empiricist can be found in Section 5.5.

A controlled experiment could not be achieved due to the nature of the problem domain, where the possibility of side effects resulting from other factors were unknown. Instead, semi-controlled experiments were conducted. The results of these experiments are presented in Chapter 7.

In order to evaluate Empiricist, the tool *Verifier* was created by the authors of this thesis to calculate the dependent variables. Verifier was used after each experiment session and the values were then evaluated.

The results from the evaluations of Empiricist determined the direction in which the tool was to be further developed. After each evaluation a new version of Empiricist was created. However, before creating a new version, it was important to ensure that the latest version worked as intended, since the development of the new version would be dependent on its evaluation results. The following steps were taken to verify the implementation of the latest version: 1) run unit tests during development; 2) compare the latest evaluation result with the previous version’s results.

For the first step, unit tests were created during the development of Empiricist to limit the number of bugs and logical flaws that could be introduced when creating a new version.

The second step was done to see how the latest version differed from the previous version, while also ensuring that the changes affected the results as assumed. Before the creation of a new version, assumptions were made regarding the possible impact that the version could have on the results. This was then used to determine if the changes worked as intended or not. This approach found a few logical flaws either in the previous or the latest version. The result of this can be seen in Section 5.5.

Chapter 4
Survey of RTS Techniques

Here a brief overview of the problem domain and the regression test selection techniques, read up on during the literature studies, is presented. As well as the motivation for why the history-based technique was deemed appropriate for this thesis.

4.1 Problem Domain

This section explains the framework and the constraints that the regression test selection techniques will have to work within. The problem domain consisting of the Android Open Source Project as well as the available historical data in Sony Mobile is presented below.

4.1.1 Android Open Source Project

Overall, the Android Open Source Project (AOSP) consists of, depending on the version, over 700 000 files and around 200 000 directories. The file formats range from configuration files to Java, C and C++ files. Java being more prevalent in the higher layers of the software stack while C and C++ become more prevalent in the lower levels.

To make the development of different parts of the AOSP easier it is separated into several different git repositories referred to as projects. The remote server and revision tags are, along with other configuration options, specified for each project in a manifest file located in the top-level directory [8]. A tool called *Repo* has been developed by Google which, through the manifest file, provides a unified interface for working with the collection of projects contained in the AOSP [9].

AOSP is structured as a software stack divided into five layers to provide abstractions of underlying functionality. As shown in Figure 4.1, there are five layers: Applications, Framework, Native Libraries and Runtime, HAL, and the Linux Kernel.

Figure 4.1: A depiction of the Android software stack layers

Each layer has its distinct responsibilities and provides an interface through which other layers can communicate. Despite the visual characteristics of Figure 4.1, where layers are stacked on top of each other, the communication between layers does not necessarily follow this structure. Many interfaces in the Application layer, for example, map directly to HAL interfaces [10].

4.1.2 Available Data

During the work of this thesis, the maximum amount of test results available at Sony Mobile for a single product, i.e. smartphone model, was approximately 250 while the number of test results for an Android platform did not amount to much more than a thousand. The amount of test results available from previous test runs therefore depends on how many products and/or platforms are to be included in the work of this thesis.

4.2 Regression Test Selection Techniques

Three techniques have been studied and evaluated in relation to the problem domain. An overview of these techniques is presented below.

4.2.1 Deep Learning

Deep learning is a subfield of machine learning in which models can be trained to, given some input, generate meaningful output. In this context, meaningful output is defined when comparing output generated by the model to the actual expected output. Three things are needed for this to work: input data, examples of expected output and a way to measure the difference between the output and the expected output. What distinguishes deep learning as a subfield of machine learning is a larger number of layers, i.e. data transformations that are used. Each layer consists of several weights, the values of which are what is *learned*. Every exposure to examples, i.e. input and correct outputs, will slightly modify the weights of each layer in a way that will generate more meaningful output. To this end, the optimizer and the loss function is used. The loss function calculates the distance score between the model's output, i.e. predictions, and the expected output. This distance is then used by the optimizer to update the weights of the layers in the network [11].

With historical data from previous test suite executions coupled with changes made to the source code between these executions, a deep learning model could be trained to output the probability of any test failing. The input would therefore be the specific encoded modified files and the output would be compared to the actual relevant tests resulting from those modifications. Typically, training such a model consists of around ten iterations over tens of thousands of examples [11], in which case the data available for the purposes of this thesis would be insufficient.

4.2.2 History-Based

In the technique explored by Ekelund and Engström [7] links between tests and source code entities, e.g. files, directories, or packages, can be constructed using data from historical test suite executions, i.e. the source code changes and relevant tests given those changes. *Relevant tests* are defined as tests that *flip* between test runs. This means that if test *T* failed the previous run and passes during the subsequent run or vice versa, *T* is said to be relevant to all the modifications to the source code that has been made to the software between those runs.

A link in this mapping would be established and strengthened every time a given source code entity changed and a test proves itself to have been relevant. The strengthening of the link on repeated occurrences of the same modified source code entity and the same resulting relevant test could be used to distinguish links resulting from true correlations from the

inevitable noise that will accumulate over time using this technique [7]. Another aspect to take into consideration is how to maintain the relevance of this mapping. If there is no mechanism for cleaning up noise and/or links that are no longer relevant, the map would eventually become useless. A crucial aspect to the longevity of a solution of this sort would therefore be an intelligent way to distinguish noise as well as remove outdated links. However, despite the disadvantages, the technique is very flexible. It does not depend on any specific language or file format to establish and maintain links. It could also be easily transformed to operate on different levels of the codebase, e.g. links between file and test, directory and test or a combination of both. Additionally, a database containing links of this sort has been shown to be successfully constructed using a relatively small amount of historical data [7].

The size of CTS* is also an important aspect here and more specifically how the relevant tests between test runs accumulate over time. Since the test suite contains approximately 58 000 tests, it would take a long time to construct a usable map of this sort if for each test run the tests that flip have previously not flipped. If, on the other hand, it can be shown that there is a temporal connection in relation to which tests flip, the applicability of this technique increases since a usable database should be possible to construct even with a relatively small amount of historical data. A temporal connection here means that the probability of a test flipping increases if it has recently flipped.

4.2.3 Static Analysis

Using the source code or intermediate binary code, static analysis can be performed to construct several different abstract models [12]. One of these models is a call graph which consists of a directed graph out of which dependencies between classes can be inferred.

There exist several open source tools [13-15] that can be used to construct a call graph that would serve as a basis for a mapping of dependencies between classes in the CTS* source code, the Android API, and the codebase at Sony Mobile. An important aspect of static code analysis is that the tools are language specific and the complexity of using this technique is therefore expected to grow in relation to the number of languages that needs to be supported.

Given a correctly constructed mapping, all bugs can be expected to be found since all dependencies, from source code to test, should then be known. The reduction, however, is expected to suffer as a result since it would recommend all tests that the modified file has a dependency to, regardless of any likelihood of these tests failing. A possible way to reduce the number of recommended tests further would be to introduce additional parameters that would be used to prioritize tests found through dependencies [16].

4.3 Discussion

As described in Section 4.1.1 the target codebase consists of a range of different file formats and programming languages, from C++, Java, Python and C, to config and XML files. All these files have a varying impact on the outcome of the executed CTS tests. Due to the nature of the software stack, the impact of lower level C and/or C++ code can have a wider range of effect on the test results compared to the higher-level Java code that has a more direct impact on the test outcomes. For instance, lower-level code that is directly connected to Bluetooth functionality will likely influence all tests where Bluetooth is involved. Given this, a unified approach to statically map dependencies to tests could be difficult since all the different formats and languages would have to be supported to make certain that there are no areas left untested in the source code. Due to the depth of dependencies throughout the software stack, it might not be feasible to achieve sufficient source code coverage using static analysis.

Deep Learning and the history-based technique are similar in a way that they are based on the same principle, i.e. using historical data to construct a mapping between source code and tests. The main advantages of these approaches lie in the fact that they are independent of both programming languages and file formats. This comes at the cost of needing to train and maintain both models to achieve acceptable results. However, the history-based technique is expected to require less data than deep learning to generate acceptable output given that there is a temporal connection to flipped tests.

4.4 Conclusion

Due to the complexity and the size of the target codebase, it was concluded that a static analysis technique would be too costly and not a scalable solution. The remaining two techniques studied, history-based and deep learning, share, as previously mentioned, common traits in that they are both reliant on historical data from test suite runs. The amount of data needed to properly train a deep learning neural network to, for the purpose of this thesis, achieve sufficiently meaningful output is too large.

The history-based technique was determined to best suit the problem domain. The advantages of this approach include independence of programming languages and file formats present in the codebase, and independence of connections, or lack thereof, to the Android API. Additionally, it is possible to construct a database with relatively small amounts of historical data to generate sufficiently good output, as shown by Ekelund and Engström [7].

There are a couple of preconditions that likely need to be met for the history-based technique to work. The target codebase and test suite are both large. Constructing a database that fully covers such a large codebase and test suite would take a long time. The

preconditions are therefore that there are temporal connections both in relation to which source entities have been modified and to which tests that flip for each test run. Meaning that source entities that recently have been modified are likely to be modified soon again and tests that recently flipped are likely to flip again during subsequent test runs.

A flipped test has been observed to on average, as seen in Figure 4.2, flip again about three times over 100 test runs. The first time a test flips will not be covered by the history-based technique, but the three later flips of the same test will be covered if all flipped tests are recommended. Additionally, only a fraction of the tests within CTS* ever flips. This is evident from Table 4.1 where only 343 and 432 unique tests flip over 100 and 96 runs respectively on one product. Regarding source entities, it can be seen in Figure 4.3 that modification recurrence takes place. As expected the recurrence rate increases with the coarseness of the granularity.

Table 4.1: Unique test flips over number of test runs on a different product

Test runs	Number of unique flipped tests
100	343
96	432

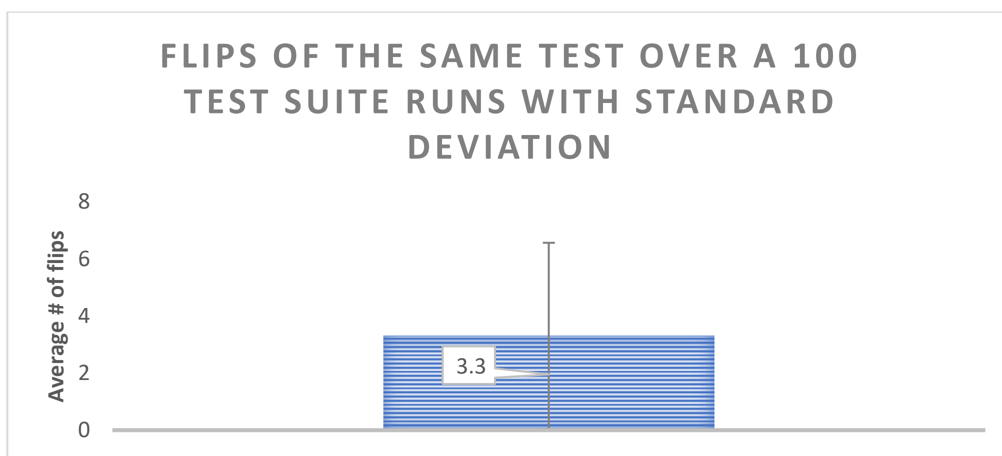

Figure 4.2: Average number of times the same test flipped over 100 runs.

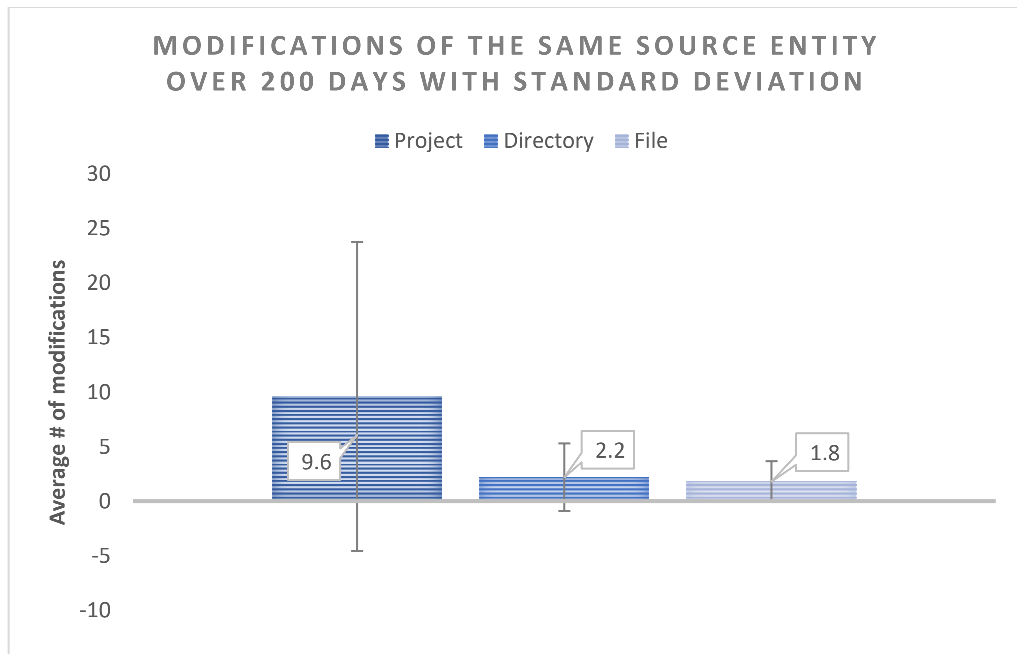

Figure 4.3: Observed average number of modifications of the same source entity over 200 days.

Despite the historical approach being deemed the most suitable given the problem domain, several potential disadvantages were considered. The effect of using historical data from different Android platforms is unknown. Although Android platforms share much code and many features, differences could potentially pose a threat to the validity of a database that has been constructed using a previous platform while its intended to be used on a later platform.

Large amount of modifications has been observed to sometimes be contained between software versions. As a consequence of this, the historical data between these versions is expected to contain a considerable amount of noise. This noise is due to the unlikeliness that all, or even most, of these modifications have true connections to the identified relevant tests. The noise will have a negative impact on precision by which tests are recommended but for the work of this thesis low precision values can be tolerated initially. Low precision affects the number of unnecessary recommended tests but since the CTS* is so large to begin with, this should not be an issue for quite some time. Given that precision is dependent on the amount of modifications that has been made between two tested software versions, the precision could be improved at a later stage by testing software versions more frequently using the full scope. The reason for this is that the number of modifications made between tested software versions is likely to be less when testing is done more frequently.

Chapter 5
Software

The logic of the extended history-based technique (ExtHB) and its implementation, Empiricist, are described in this chapter.

5.1 Extended History-Based Technique

The extended historical-based technique *(ExtHB)*, implemented here in the form of Empiricist, is built based on the approach taken by Ekelund and Engström [7], explained in Chapter 4. In their approach, a weighted link between source code entity E and test T was established and incremented every time E was modified and T revealed itself to be relevant, i.e. flipped since the last run. Although weighted links were maintained, they were never used in their implementation. In Empiricist the potential of these weighted links is explored by introducing a cutoff criterion on the normalized weights using a min-max scaler. This is done in an attempt to maintain a relatively small subset of recommended tests since the tests recommended eventually become equal to running the entire test suite when all tests within the test suite are present in the database [7].

5.2 Empiricist

Empiricist was developed for evaluating the extended history-based approach. Although the training process would be the same in the real use case, the construction of test scopes, i.e. recommended tests, would be different. In the real use case developers would submit their modifications. In Empiricist these submissions are substituted for modifications made between two tested software versions. The reason for doing this was to enable a faster evaluation process.

Empiricist has two base functionalities: 1) the training process, as detailed in Figure 5.1, and 2) the process of constructing a test scope, as detailed in Figure 5.2. The details of these two processes and how one is dependent on the other are explained further in Section 5.2.1.

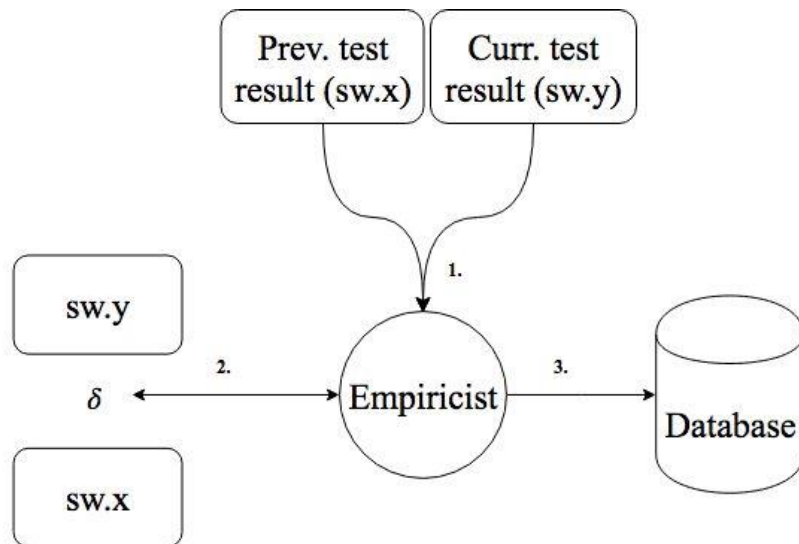

Figure 5.1: The training process of Empiricist. 1) Relevant tests are calculated by performing an outer join on the failing tests in the test results for sw.x and sw.y. 2) The modifications between the versions are extracted. 3) The links in the database are updated.

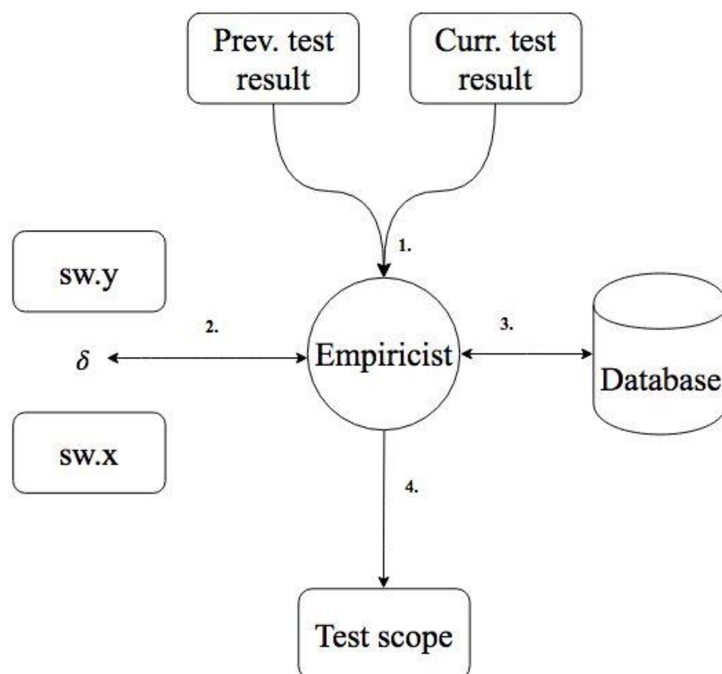

Figure 5.2: The construction of test scopes. 1) The software versions contained in the test results are extracted. 2) The modifications between the versions are extracted. 3) The database is queried for tests connected to any of the modifications. 4) A test scope is constructed.

5.2.1 Implementation

To logically substantiate a connection between source code modification and a test, it was decided to use the definition of relevant tests put forth by Ekelund and Engström [7], as the tests that had flipped between two test runs. To extract the relevant tests between the results of two test runs, an outer join was performed on the failing tests of each, illustrated in Figure 5.3.

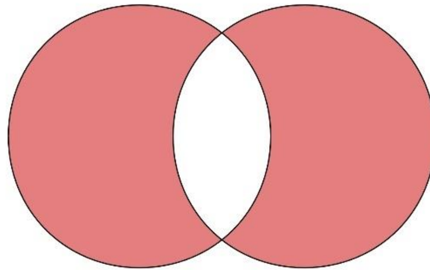

Figure 5.3: Outer join of failing tests in two test results.

By this logic, the tests that failed the previous run and are still failing in the current run are ignored. The remaining tests are the tests that either failed and now pass or the tests that passed and now fail. This is expressed by the pseudocode in Algorithm 5.1.

```
relevant_tests = ([test in prev_testresult and not in current_testresult or in
current_testresult and not in prev_testresult])
```

Algorithm 5.1: Pseudocode of an outer join of failing tests within two test results.

Test results are stored on a continuous integration (CI) server. A selected series of such test results is collected in Empiricist before the construction of the database and test scopes. This series is chosen so that the tested software versions are both in a sequential order as well as close to each other in terms of modifications. The reason for this is to attempt to minimize the situation where a modification $\boldsymbol{\delta}_1$ gives rise to test T being recommended which turns out to be relevant to the unrelated modification $\boldsymbol{\delta}_2$ that was also present in the same software difference, i.e. set of modifications, but would not by itself have given rise to the recommendation.

The test results retrieved from the CI server contain, in addition to failing tests, the ID of the specific software version they resulted from. The IDs are used to sort the test results in the correct revision order. In addition to sorting, the IDs are also later used to retrieve the modifications that have been made between tested software versions.

Empiricist supports three levels of granularity: *very fine-grained granularity* where files are connected to tests, *fine-grained* where directories are connected to tests, and *coarse-grained* where projects, as defined in Section 4.1.1, are connected to tests.

To construct a database of links between tests and source code modifications, the previously mentioned concept of relevant tests was used in combination with the source code modifications present in the difference between two software versions. Each modification is connected to each relevant test. In case the link is already present in the database its weight is incremented. This constitutes the training process of Empiricist. Special care has to be taken here so that the test scopes containing recommended tests, generated using modifications between software version *sw.x* and *sw.y* are not using a database where the links from these same modifications are already present. Such a situation would guarantee that all relevant tests were included in the test scope since they are already connected to the modifications. With this in mind, as shown in Algorithm 5.2, test scopes are always generated **before** updates are made to the database using the test results and modifications corresponding to the versions of the current and previous test results.

```
previous_version = previous_testresult.software_id
current_version = current_testresult.software_id
for modification in modifications(previous_version, current_version)
    tests = database.connected_tests(modification)
    testscope.include(tests)
write testscope
for relevant_test in outer_join(previous_testresult, current_testresult)
    for modification in modifications(previous_version, current_version)
        db.add_strengthen_connection(relevant_test, modification)
```

Algorithm 5.2: Test scopes are constructed before the database is updated.

The construction of test scopes involves several steps seen in Figure 5.2. In the first step the IDs of the software versions are extracted from the test results. Second, the modifications made between the software versions are retrieved. These modifications are then used to get the connected tests from the database. The last step, involving the construction of test scopes, supports two different approaches:

1. Connection-Based Recommendation (CBR)
2. Normalized Connection-Based Recommendation (NCBR)

The first approach does not take weights into account and simply recommends all connected tests. This includes a fallback approach for the very fine-grained granularity where all tests connected to the directory containing the modified file is recommended if no links were found using the file.

In the second approach weights are normalized and subsequently subjected to a cutoff criterion. This means that the links with lower weights will be filtered out. The amount of links lost here is dependent on the distribution of weights as well as the value of the cutoff criterion. The reason for including this approach is to be able to filter out the links resulting from noise that will inevitably be present in the database. The hypothesis here is that links of this sort, i.e. resulting from noise, will occur less frequently than links resulting from true correlations. Thus, true links will eventually distinguish themselves from the false ones.

The term noise is used for the situation where links are established erroneously. Several modifications could have been made between two tested software versions but only a subset of those gave rise to a test flip. However, all the modifications will be linked to the tests that flipped. The modifications that did not have any correlation to any of the tests flipping are said to be noise.

5.3 Normalization

Normalization is a technique for data pre-processing and scaling of data to smaller ranges. The technique used in Empiricist is min-max scaling which scales the data using the following Eq. (5.1) [17]:

$$X_{norm} = \frac{(X - X_{min})}{(X_{max} - X_{min})} \tag{5.1}$$

The reason for selecting min-max scaling, as opposed to a technique such as standardization, is that through min-max scaling a range within [0, 1] can be achieved which makes inserting a cutoff criterion simple. There are, however, drawbacks to this technique. Min-max scaling is sensitive to outliers in the data set since all data points will be pushed towards zero if there exists a value significantly larger than the rest in the data set [18].

5.4 Filtering

Extracting the modifications between two built software versions can pose several problems. If the modifications made between the versions are sufficiently large, some files and/or directories might always have been modified. This naturally leads to all previously flipped tests being recommended. As would be expected, this amount will only grow with time and eventually become equal to the entire test suite. Another problem can be that files resulting from building the software, as opposed to individually modifying one or several files in a commit, are such that they can never be present in the case of commits. Given this, it is of no use to store such connections. It was decided to identify files of the latter

kind and filter them out from the difference taken between software versions. Files and/or directories of the former kind were decided to be kept since, although they have a significant impact on the number of recommended tests, they are generally files that potentially have an impact on the outcome of the test suite runs.

5.5 Versions

Different versions of Empiricist were developed iteratively and each with an additional function and/or parameter that was expected to influence the dependent variables, defined in Section 6.2. The reasoning behind the changes was firstly that Empiricist functions correctly with reference to the description in Section 5.2.1 of how it is supposed to work; and secondly, to try and explore the effect that different approaches might have on the measured outcome. The modifications in versions 2.0 and 2.2 fall into the former category since they were in contrary to the view of how Empiricist was to work with the source entities. Version 2.1 and 3.0 fall into the latter category and were made to accommodate for normalization.

5.5.1 Empiricist 1.1 (Granularity)

This first functional version of Empiricist contained two different mapping granularities: very fine- and fine-grained. In fine-grained granularity source directories were connected to tests and in very fine-grained granularity source files were connected to tests. Additionally, in very fine-grained granularity a fallback to tests connected to the directory containing the file was used if no connected tests to the file were found.

5.5.2 Empiricist 2.0 (Separation of Projects)

Directories from different projects were previously treated as the same directory, i.e. if the path to a modification found in the project p_1 between two software versions was *a/b/c.java* and another modification found in the project p_2 was *a/b/d.java,* these were stored as connected to the same directory *a/b*. In version 2.0 this was changed and the files from the previous examples were stored connected to their project as well as their path within that project. Therefore, *a/b/c.java* becomes p_1*/a/b/c.java*. This had the expected effect that less links were found in both fine -and very fine-grained granularity. Although this effect will initially generate worse measured values on inclusiveness since less tests are recommended than before for most links, it will also reduce the likelihood of outliers in the data that previously were more prevalent when treating directories from different projects the same.

5.5.3 Empiricist 2.1 (Normalizing after Selecting Subset)

The point at which normalization was performed was changed. Instead of normalizing before a subset of links were selected depending on source modifications, it was decided to normalize **after** this subset had been selected. With this change the effect of possible outliers is reduced since a smaller subset is normalized. Additionally, different source entities can have different modification frequencies which will result in entities with higher

modification rates being more likely to have higher weighted links. Normalizing before would thus mean that source entities with comparatively smaller modification rates will be more likely to have small normalized weights. With the change to normalize after a subset has been selected the occurrence of such a situation will now be limited only to weights within the selected subset.

5.5.4 Empiricist 2.2 (Eliminating the Impact of Number of Files Contained in Directory)

If multiple files within a directory were modified in the same software difference, each of these files incremented the weight of the link between the directory and tests. This was removed since directories with a larger number of files and/or more frequent modification of them quickly gained larger weights. These weights are then not only dependent on the frequency of a found connection, i.e. a modified directory to a relevant test, but also on the number of files modified within that directory.

5.5.5 Empiricist 3.0 (Introduction of Coarse-grained Granularity)

Coarse-grained granularity was introduced to investigate the impact on the distribution of weights. Modification frequencies of the same source entities naturally become higher as links are established at coarser granularity. The working hypothesis is that a wider distribution of weights will allow links resulting from true correlations to distinguish themselves better and therefore accommodate for normalization.

Chapter 6
Evaluation

This chapter presents the dependent and independent variables used in the experiments as well as how verifier calculates the dependent variables.

6.1 Evaluation Terminology

The terms *reference test result*, *hits* and *misses* are used throughout this chapter to present the dependent variables and how they are calculated.

Reference test result refers to the two consecutive test result files that were used by Empiricist to construct a specific test scope, see Section 5.2.1 and Algorithm 5.2 for further details.

Both a hit and a miss can refer to two different things depending on what version of the dependent variable is calculated. A *hit* can refer to:

- A flipped, i.e. relevant, test between the two reference test results that is also present in the recommended test scope.
- A failing test in the later reference test result that is also present in the recommended test scope.

A *miss* can refer to:

- A flipped, i.e. relevant, test between the two reference test results that is not present in the recommended test scope.
- A failing test in the later reference test result that is not present in the recommended test scope.

6.2 Dependent Variables

The following dependent variables have been measured to evaluate the performance of Empiricist.

It is important to note that precision, inclusiveness, and reduction need to be considered as a group since each one reflects a distinct metric on the performance of Empiricist. Maximum inclusiveness and reduction can easily be achieved on its own: maximum inclusiveness through selecting the entire test suite and maximum reduction by selecting nothing. Additionally, maximum precision could be achieved by selecting only a single fault revealing test while potentially missing several other vital fault-revealing tests.

The values of reduction, inclusiveness, and precision will all be converted to percent when presented in Chapter 7.

6.2.1 Reduction

Reduction is a measurement of by how much the test suite is reduced. It is calculated according to Eq. (6.1) where S_{ts} is the size of the recommended test scope and S_{cts*} is the size of CTS*. This will give a result within the interval [0, 1] where a value of 1 would mean that no tests were recommended and 0 would mean that the entire test suite was recommended.

$$R = 1 - \frac{S_{ts}}{S_{cts*}} \tag{6.1}$$

6.2.2 Inclusiveness

Inclusiveness reflects the percentage of all the relevant or fault revealing tests that have been identified and thereby included in the recommended tests. This metric is the quotient of hits and all relevant tests or fault revealing tests. The latter is equivalent to the sum of hits and misses. Inclusiveness is thus calculated according to Eq. (6.2) as the result of division of hits H by the sum of hits and misses M.

$$I = \frac{H}{(H + M)} \tag{6.2}$$

Inclusiveness will be used to define two measurements: fault revealing inclusiveness (FRI) and relevant inclusiveness (RI). The difference between these two measurements is what a hit and miss refer to. In *FRI* a hit refers to a fault revealing test that is also present in a test scope and a miss to a fault revealing test that is not present in a scope. In *RI,* however, a hit refers to a flipped test present in a test scope and a miss refers to a flipped test not present in a scope.

6.2.3 Precision

Precision is a measurement of how many of the recommended tests in a test scope are hits. As with inclusiveness, hits can refer to either relevant tests or fault revealing tests. The former case results in fault revealing precision (FRP) and the latter in relevant precision (RP). Both are calculated according to Eq. (6.3) which will give a value within the interval [0, 1]. A value of 0 would mean that no tests in the test scope were a hit while a value of 1 would mean that all the tests in the test scope were hits, i.e. relevant or fault revealing tests.

$$\boldsymbol{P = \frac{H}{S_{cts*}}} \tag{6.3}$$

6.2.4 Effectiveness

Effectiveness is a variable defined by the environment in which RTS is to be performed. The definition can therefore differ depending on which resources are viewed as most important and where the threshold for acceptable values on inclusiveness, precision and reduction lies. Empiricist is to be used for fast feedback for changes made during the day. Important resources are therefore the time it takes to run a test scope and the percentage of the total amount of fault revealing tests that are included in the test scope. The percentage of fault revealing tests included in a test scope is reflected in *FRI*. Runtime for a test scope will be estimated with an average execution time for a test and the number of tests included in a scope.

6.2.5 Average Number of Tests

The average number of tests recommended.

6.3 Independent Variables

The independent variables presented below were all tweaked to measure the effects they had on the dependent variables.

- **Granularity of Source Code Entity**
  - The granularity level at which weighted links are established between source entities and tests

.

- **Number of Software Versions Used for Training**
  - The number of software versions that has been used, through their difference and test results, to construct a database when the measuring of dependent variables starts.

- **Cutoff Criterion**
  - The normalized weight threshold. All links with normalized weights lower than this threshold are removed.

- **Normalization Stage**
  - The stage at which normalization is done during test scope construction

.

- **Product**
  - The historical data used from test runs on a specific product, i.e. smartphone model.

6.4 Verifier

In parallel to Empiricist, Verifier was developed. Verifier calculates and presents the dependent variables, defined in Section 6.2, relevant and fault revealing inclusiveness, relevant and fault revealing precision, reduction, and average number of tests. It does this over intervals of 20. That is, the amount of software versions and consecutive test results used for training will for each interval be incremented by 20. Empiricist then builds on the trained database while generating test scopes, as described in Section 5.2.1. This is done over 20 software versions and test results. The average of the dependent variables is taken over these intervals. The reason for choosing to calculate the dependent variables over intervals of 20 is that it was deemed suitable to the amount of historical test results that was available. A larger value would have given less intervals to measure the variables over while a lower value would have been more sensitive to sudden spikes and dips.

Verifier calculates the variables using the test scopes generated by Empiricist and the reference test results obtained from the CI server. Every recommended test scope contains within its name the software versions between which modifications were used to generate it. For instance, if modifications made between the versions *sw.x* and *sw.y* were used, the test scope that is generated will be named *sw.x-sw.y.xml*. As seen in Algorithm 6.1, the content of a test scope is compared to both the actual relevant tests between the reference test results that tested software versions *sw.x* and *sw.y*, and to the fault revealing tests in the later reference test result that tested software version *sw.y*. The former comparison gives hits and misses with respect to relevant tests and the latter gives hits and misses with respect to fault revealing tests. These hits and misses are used to calculate relevant and fault revealing inclusiveness as well as relevant and fault revealing precision.

```
start_version_id = testscope.get_start()
end_version_id = testscope.get_end()
start_testresult = testresults[start_version_id]
end_testresult = testresults[end_version_id]
for relevant_test in outer_join(start_testresult, end_testresult)
    if relevant_test in testscope
        hits++
    else
        misses++
for failed_test in end_testresult.failing_tests
    if failed_test in testscope
        hits++
    else
        misses++
```

Algorithm 6.1: The extraction of hits and misses in reference to relevant and fault revealing tests respectively.

6.5 Thesis Questions

Considering the use of an RTS technique, the answers to the following question are of interest:

- **TQ.1.** By how much can the number of tests selected from CTS* be reduced while maintaining high inclusiveness?
  - **TQ1.1.** Can high inclusiveness be attained while high reduction is maintained
  - **TQ.1.2.** By how much would the time be reduced, compared to running the CTS*, taking the overhead of dynamically selecting a subset of tests into account.
- **TQ.2.** How much and what kind of data is needed to get a satisfying result, considering TQ.1.

In order to get an estimate on what the user would perceive as acceptable inclusiveness and acceptable waiting time for receiving feedback a survey, presented in Appendix Table A.8, was sent out. The survey was sent to potential future users of Empiricist, i.e. developers at Sony Mobile. Overall, eleven responses were received. The table presents the average values on the responses to each question.

The average expected minimum percentage of bugs found can be seen to be 63% while the average acceptable time to wait for feedback (excluding the time it takes to set up the environment for the tests to run) is 17 minutes. It is important to note here that the value of 63% refers to the percentage of bugs found which is not directly analogous to *FRI* since multiple tests can fail for the same bug. However, it will be used as an estimate for the threshold of *FRI* from the user's point of view. High *FRI* will therefore have to be more than the minimum of 63%.

Given these values, effectiveness, as defined in Section 6.2.4, can be specified. An approximation of the number of tests that fit within 17 minutes can be calculated using the average runtime of a test, which was found to be approximately 0.83 seconds. This was calculated by dividing the time it takes to run the CTS* with the number of tests it includes, which on average is 58 000. With the average runtime of a test, the value of 17 minutes can be directly translated into number of tests as well as reduction. The number of tests that can be recommended while keeping a runtime of 17 minutes is $\frac{17*60}{0.83} \approx 1229$. And, since the CTS* contains on average 58 000 tests, an estimate of reduction is therefore $1 - \frac{1229}{58000} \approx$ 0.98, i.e. 98%. With this information, an effective solution can be concluded to have the minimum requirement of 63% *FRI* while not recommending more than 1229 tests.

Chapter 7
Result

The results from the evaluations and the analysis of them and the calculated results of using random selection are presented here.

7.1 Constraints Imposed by Target Codebase & Available Data

Although the work conducted in this thesis encompasses the entire codebase of Sony's vendor specific version of AOSP, test results are only available from one department. This limits this work to focusing on the subset of the CTS that is executed at this department. The reason for this is that the approach taken to select a subset of the test suite for fast feedback requires access to historical data concerning tests that passed and failed within that test suite.

Additionally, due to the way the source code modifications between software versions were extracted, they might not contain all the actual modifications made. This is because the modifications were extracted between all common projects in two manifest files, see Section 4.1.1 for explanation of the manifest file and project. If a project is removed, added, or modified, this will not be reflected in the software difference. With this said, most of the modifications made are contained in common projects between two manifest files of two software versions.

7.2 Results from Verifier

The results presented here are the dependent variables calculated using Verifier, as described in Section 6.4. These variables are presented for the Recommend All Flipped (RAF), where all previously flipped tests are recommended, as well as all granularities within Connection-Based Recommendation (CBR) and five different cutoff criterions in Normalized Connection-Based Recommendation (NCBR).

7.2.1 Inclusiveness

Figure 7.1, 7.2, 7.3, and 7.4 show the progression of both fault revealing inclusiveness (FRI) and relevant inclusiveness (RI) as more test results are used to train Empiricist.

Looking at Figure 7.1 and 7.2, inclusiveness can be seen to generally increase along with coarser granularity as well as the amount of data used for training. The increase in inclusiveness with coarser granularity is expected since source entities at a coarser granularity will contain all the links of the finer granularity source entities within it. Therefore, recommendations will at least be equal to the comparatively finer granularities and, in most cases, higher in number.

Recommending all the previously flipped test produces the highest values since this approach will always be equal to the maximum possible inclusiveness for Empiricist at any point.

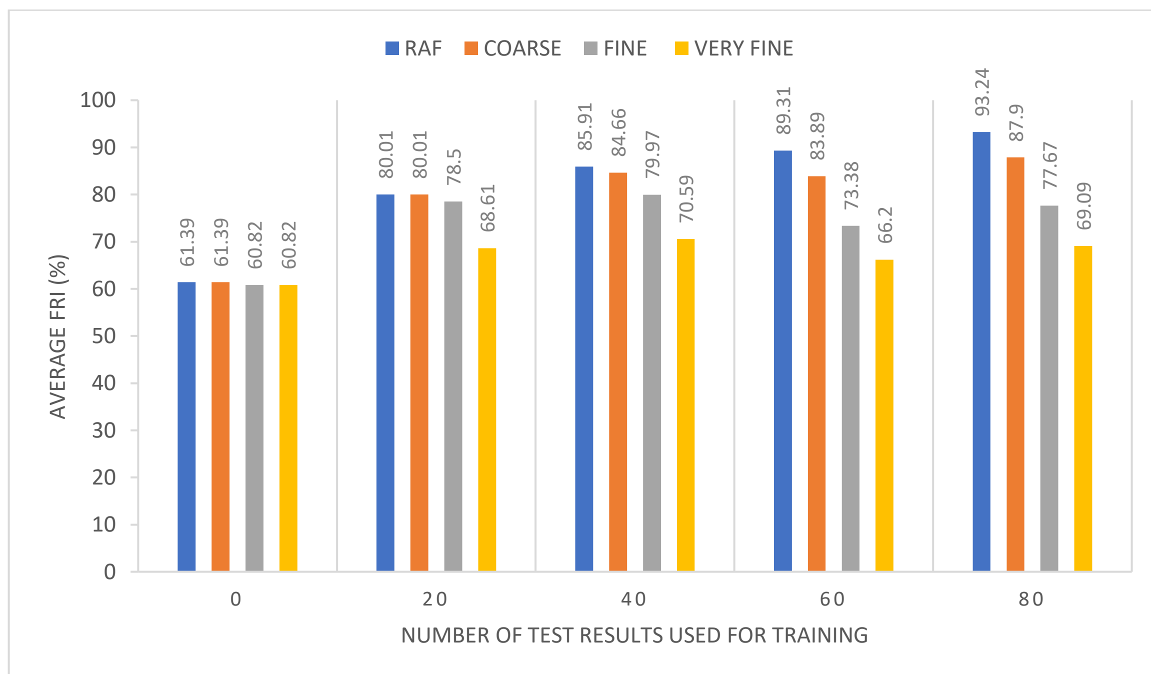

Figure 7.1: Fault revealing inclusiveness (FRI) for RAF and the granularities within CBR.

.

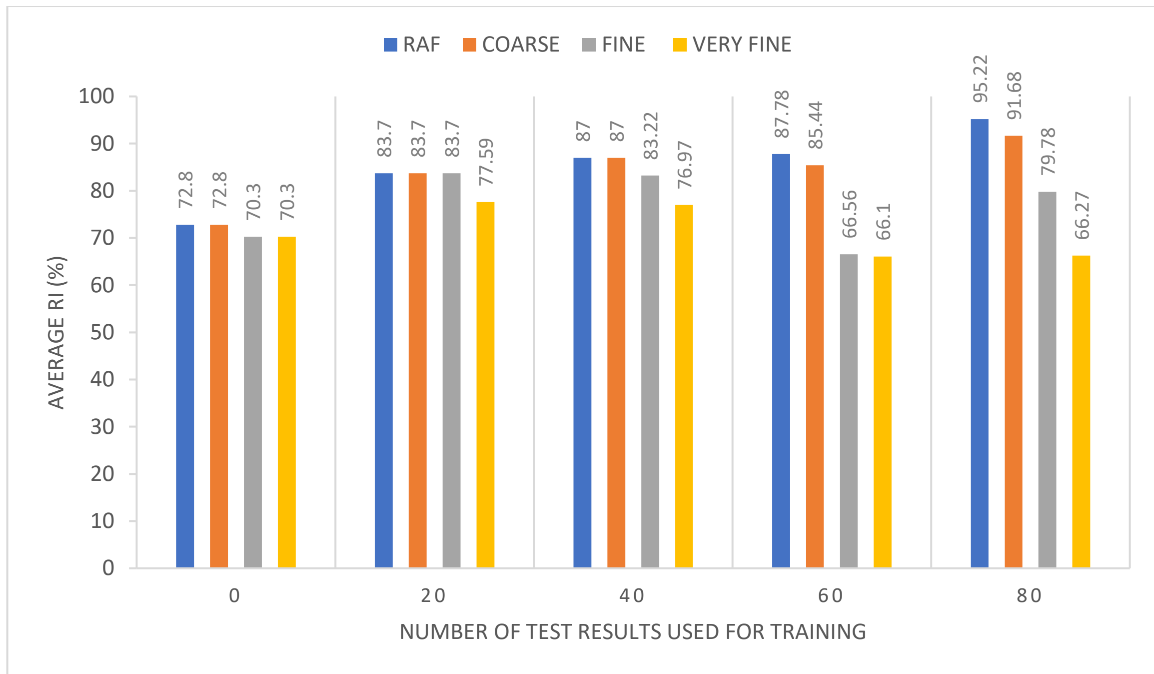

Figure 7.2: Relevant inclusiveness (RI) for RAF and the granularities within CBR.

The inclusiveness for NCBR using five different cutoff criteria is displayed in Figure 7.3 and Figure 7.4. The progression of inclusiveness for each cutoff criterion is very similar, although the inclusiveness sees an increase with a lower criterion. This is expected since a lower cutoff criterion means more recommended tests and therefore a higher likelihood of recommending a fault revealing and/or flipped test.

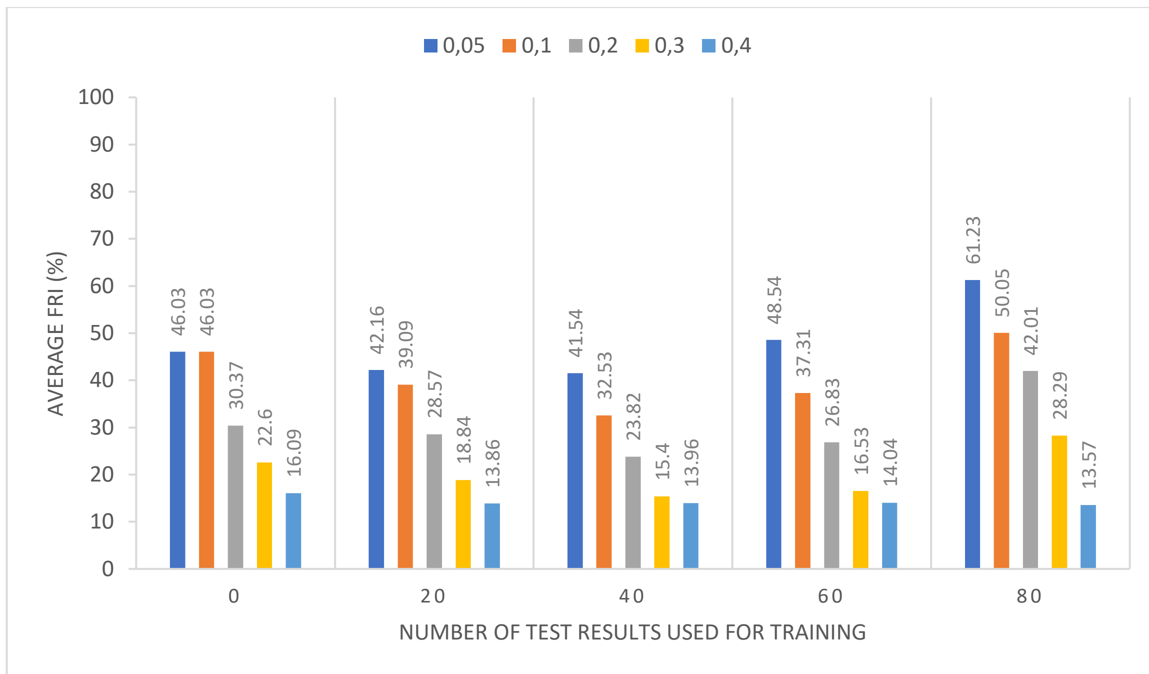

Figure 7.3: Fault revealing inclusiveness (FRI) for different cutoff criteria using NCBR on coarse-grained granularity.

.

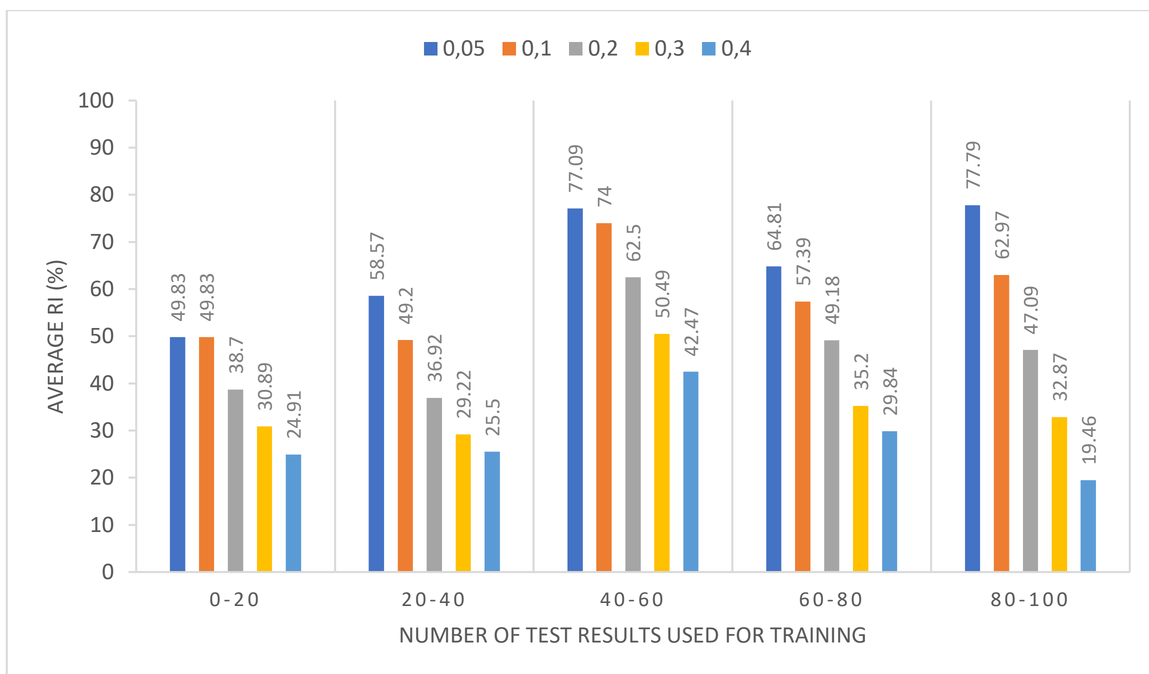

Figure 7.4: Relevant inclusiveness (RI) for different cutoff criteria using NCBR on coarse-grained granularity.

7.2.2 Precision

Figure 7.5, 7.6, 7.7, and 7.8 display the progression of fault revealing precision (FRP) and relevant precision (RP) as more training data is used, i.e. test results to train Empiricist.

Precision generally decreases with the amount of training data used since more tests will have flipped and therefore more tests will also be recommended by Empiricist. Both RAF and all the granularities within CBR mirror each other in precision for the first three intervals. After this, they roughly separate into two factions where very fine- and fine-grained granularity almost mirror each other while RAF and coarse-grained granularity also mirror each other.

While the trend shows a decrease in precision, the occasional increase is likely due to the number of tests that flipped and/or failed during an interval. If more tests flip and/or fail, the fraction of relevant and/or fault revealing tests will increase.

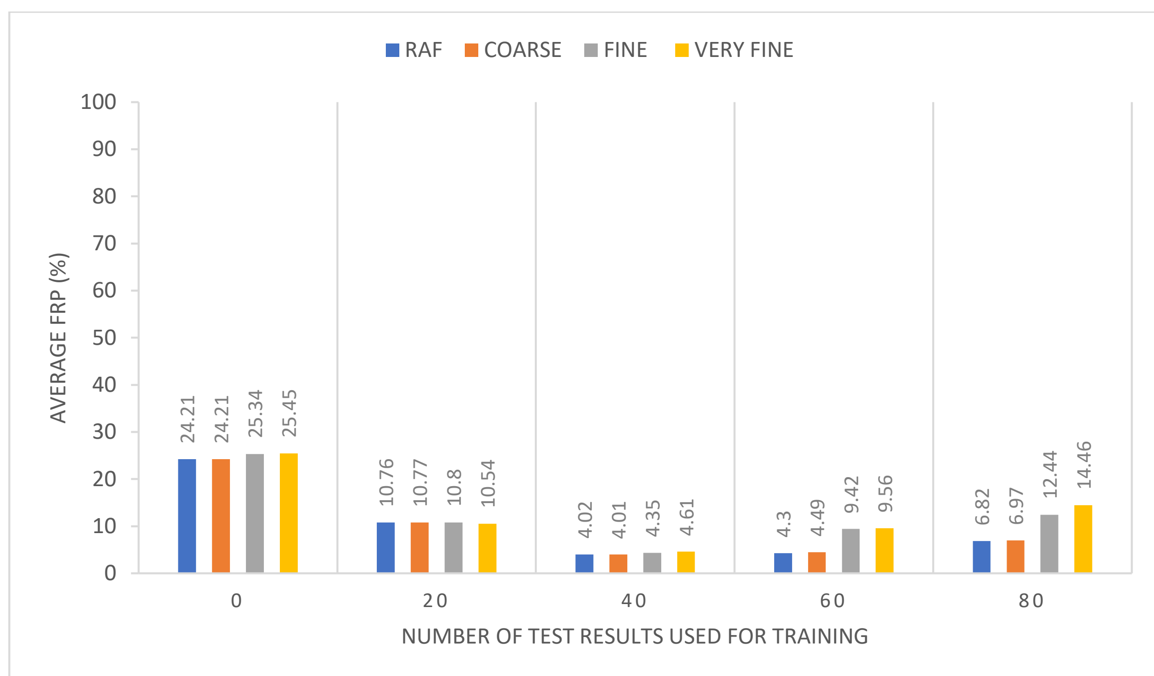

Figure 7.5: Fault revealing precision (FRP) for RAF and the granularities within CBR.

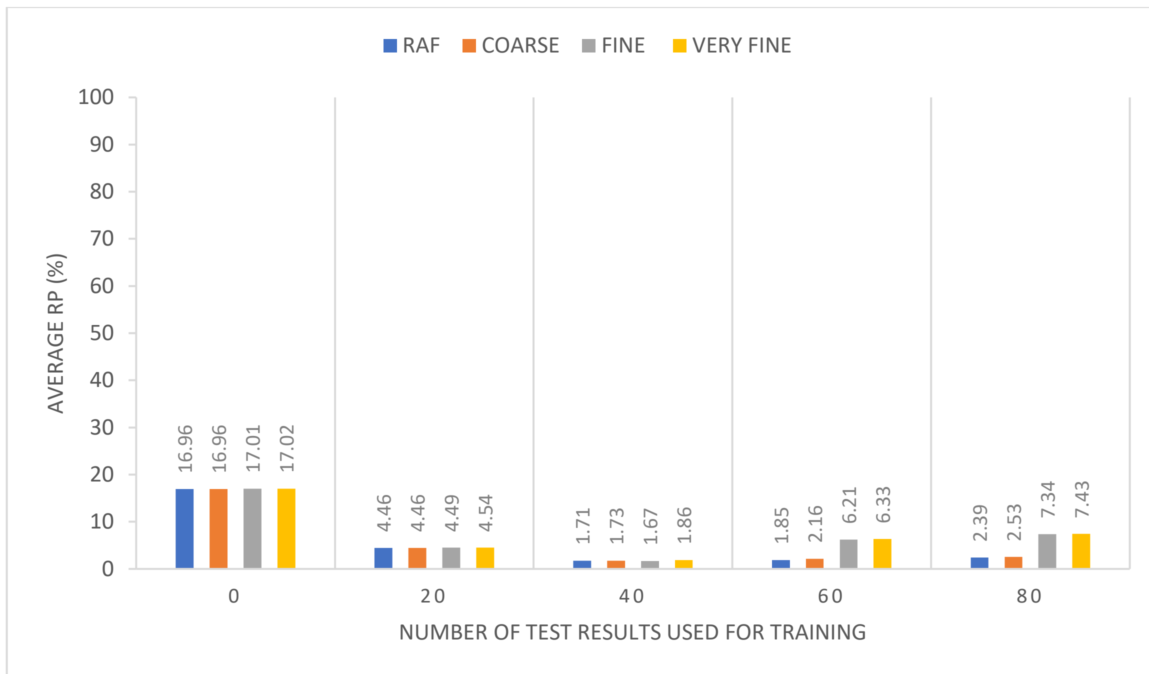

Figure 7.6: Relevant precision (RP) for RAF and the granularities within CBR.

Using the NCBR on coarse-grained granularity, the relevant precision, as seen in Figure 7.7, generally decreases as the amount of historical data increases, reaching the lowest measured value when a cutoff criterion of 0.05 is used. Since 0.05 is the lowest criterion, it is likely to recommend more tests than the higher criterions. The precision increases notably with higher cutoff criteria; the difference between the lowest and highest *RP* in the final interval is approximately 6% while the difference between the lowest and highest *FRP* is around 27% in the final interval, as seen in Figure 7.8.

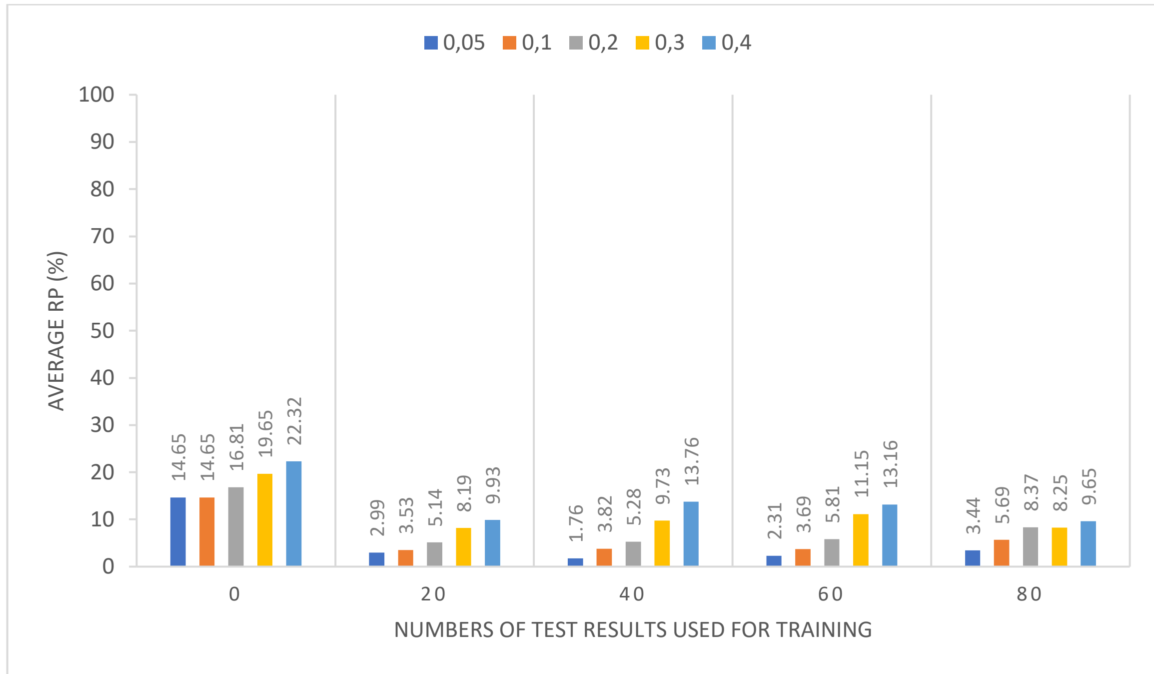

Figure 7.7: Relevant precision (RP) for different cutoff criterions using NCBR on coarse-grained granularity.

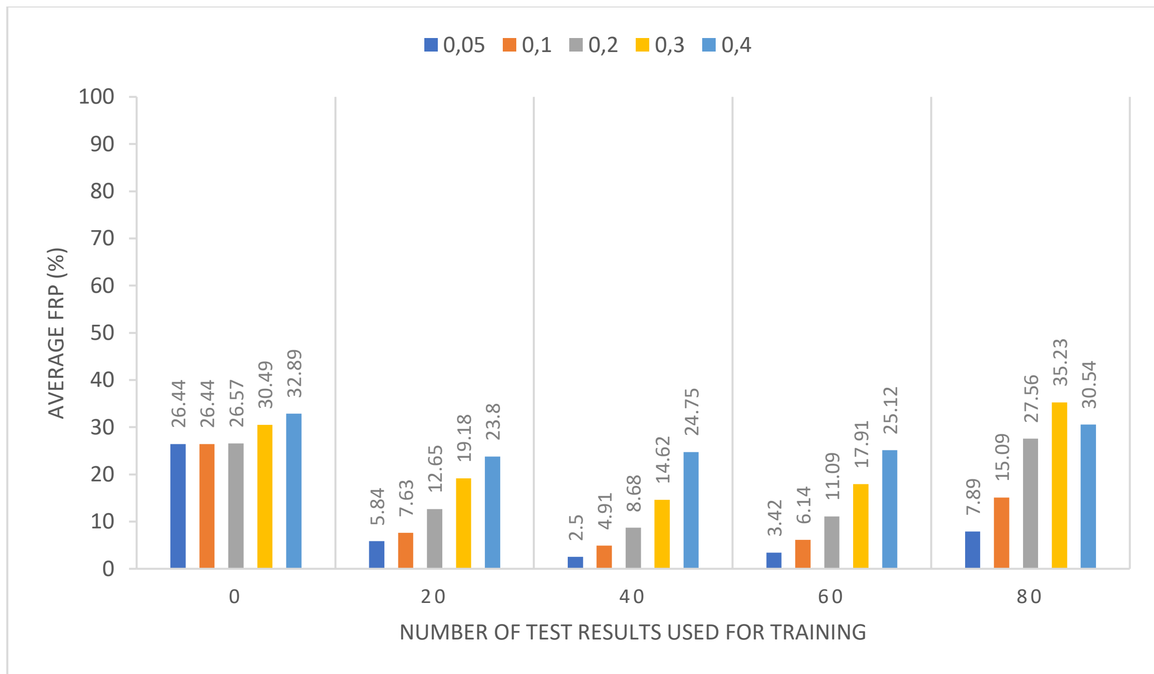

Figure 7.8: Fault revealing precision (FRP) for different cutoff criterions using NCBR on coarse-grained granularity.

7.2.3 Reduction & Number of Tests

Figure 7.9 and Figure 7.10 show the progression of reduction as more test results are used to train Empiricist.

The reduction decreases along with the amount of training data, although reduction never falls below 99.4%. The reason for this being that the test suite is large to begin with and the tests that flip tend to be tests that recently flipped, i.e. the increase in unique tests is low. This is evident from Figure 7.11 for RAF where the difference in average number of recommended tests is less for each consecutive interval. Given this decrease of the difference in average number of tests and that RAF recommends all the previously flipped tests, it is evident that the number of unique tests that flip for each interval decreases.

It can also be seen when comparing Figure 7.9, 7.1, and 7.2 that reduction is the complete opposite to inclusiveness. With coarser granularity, higher inclusiveness but lower reduction can be expected since coarser granularity translates into more tests being recommended.

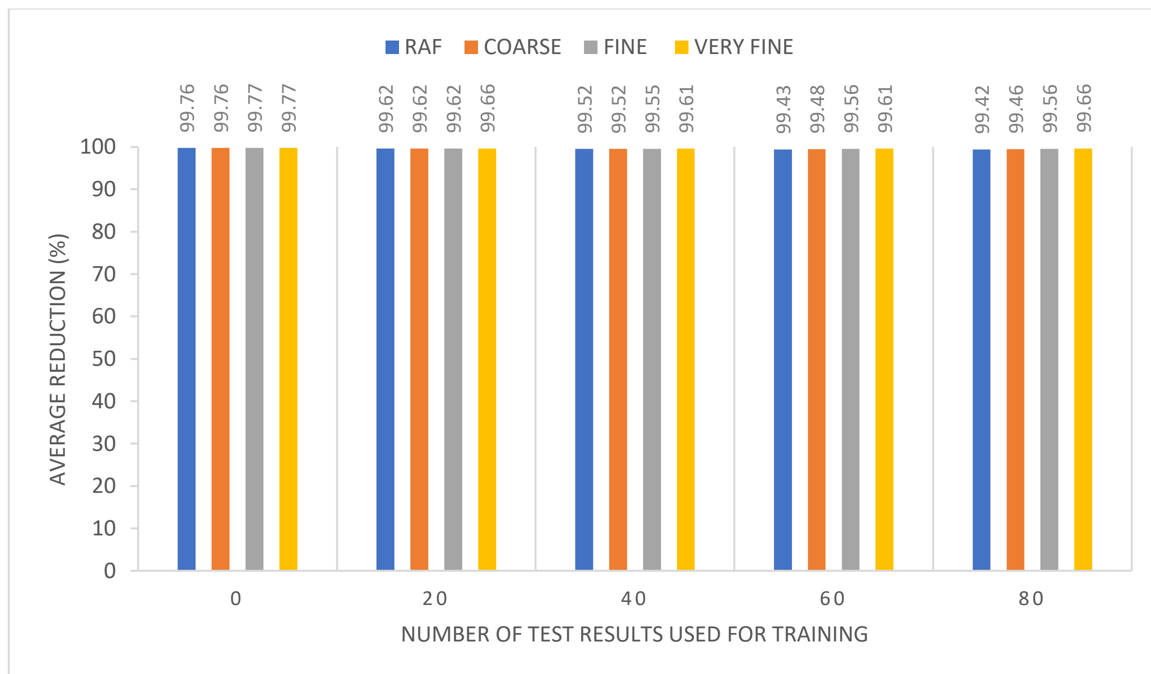

Figure 7.9: Reduction for RAF and the granularities within CBR.

Since the number of links that are excluded increases with the cutoff criterion, the reduction is also expected to increase. This is evident from Figure 7.10 where the highest cutoff criterion achieves the highest reduction.

Excluding the lowest criterion, the reduction can be seen in Figure 7.10 to be stable around 99.8 to 99.9 percent.

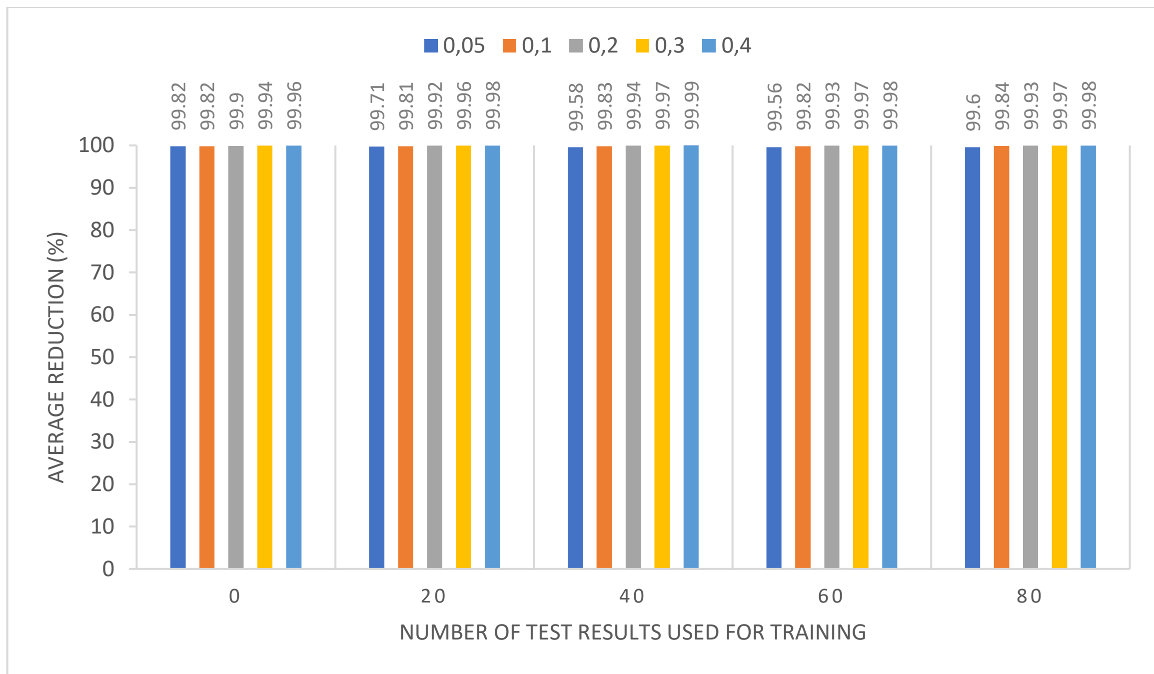

Figure 7.10: Reduction for different cutoff criterions using NCBR on coarse-grained granularity.

Reduction and the number of test results recommended have an inverse relationship, see Section 6.2.1 for the definition. This can be seen when comparing Figures 7.11 and 7.12 with Figures 7.9 and 7.10. The approaches and cutoff criteria that reaches the highest reduction values can be seen to have the lowest average number of recommended tests.

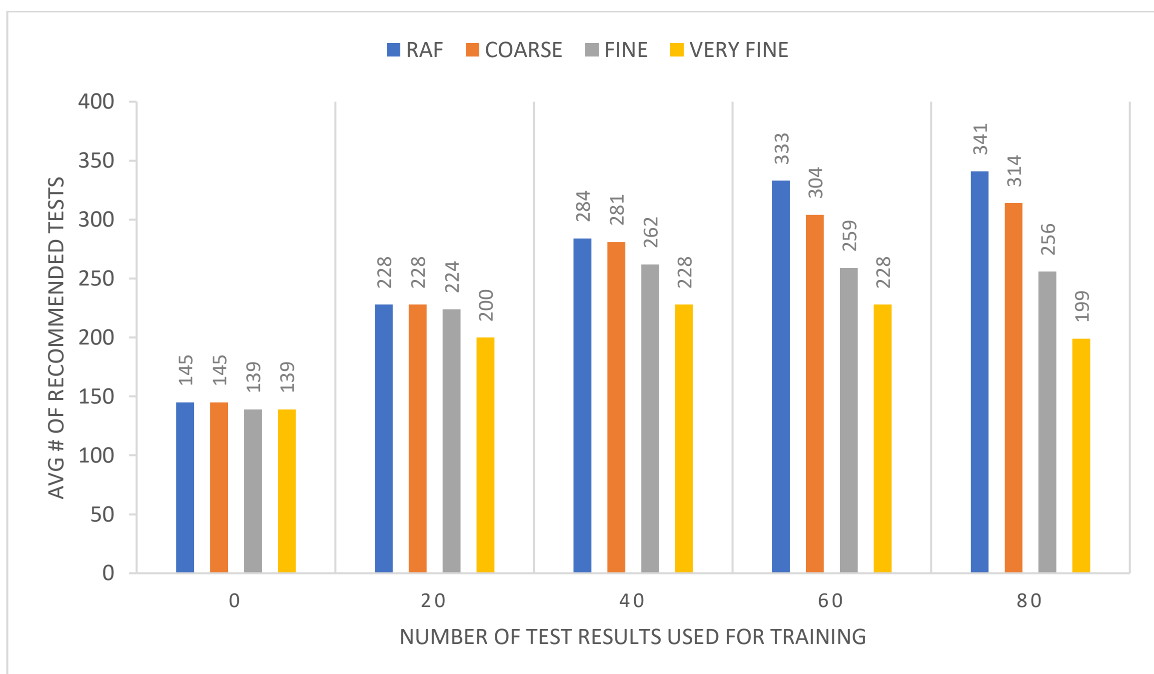

Figure 7.11: Average number of recommended tests for each interval for RAF and the different granularities within CBR.

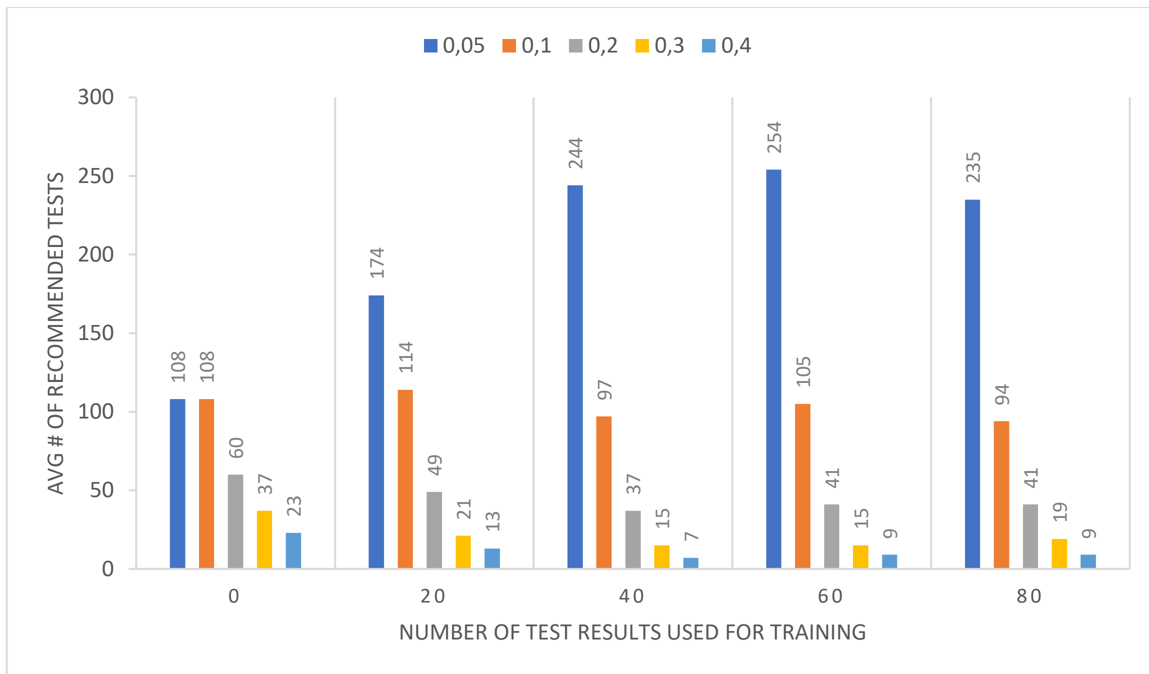

Figure 7.12: Average number of recommended tests for each interval for different cutoff criterions using NCBR on coarse-grained granularity.

NCBR has the lowest number of recommended tests. In the last interval a cutoff criterion of 0.4 only recommends 9 tests while the very fine-grained granularity in CBR recommends 199. Furthermore, the number of recommended tests for each cutoff criterion is stable over all the intervals, as opposed to the different granularities within CBR. The exception to this being a criterion of 0.05 that becomes stable in the third interval.

7.3 Random Selection

To be able to compare the results of the extended history-based approach to an alternative RTS technique, it was decided to estimate the performance of random selection. Random selection is a computationally cheap approach to selecting a subset of tests to run. The main advantage of the approach is the simplicity of it. It takes next to no resources to select ***N*** number of random tests from a pool of ***M*** tests.

By comparing consecutive test results from historical test runs on 219 software versions the *average number of flipped tests per run* was observed to be 11 while the *average number of fault revealing tests per run* is 22. Given these values, the probability of getting a certain inclusiveness can be found by calculating the classical probability, according to Eq. (7.1), where $\boldsymbol{g}$ is the number of favourable outcomes, $\boldsymbol{m}$ is the number of possible outcomes, and $\boldsymbol{S_{cts*}}$ is the size of the test suite. The number of favourable outcomes is equivalent to the number of possible ways to choose $\boldsymbol{k}$ tests from a pool of $\boldsymbol{n}$ flipped or fault revealing tests when selecting 25% of CTS* ($\boldsymbol{S_{25}}$), which is roughly 14 500 tests out of 58 000. For example, getting 1/11 flipped tests when selecting 14 500 tests can be found by the following calculation: $\frac{\binom{11}{1}\binom{14500-11}{14499}}{\binom{58000}{14500}}$.

$$\boldsymbol{P = \frac{g}{m} = \frac{\binom{n}{k}\binom{S_{25}-n}{S_{25}-k}}{\binom{S_{cts*}}{S_{25}}}} \tag{7.1}$$

The results of these calculations are presented in Figure 7.13 and 7.14. The precision using this approach will, on average, not exceed $\frac{22}{14500} = 0.0015$, i.e. 0.15%, for fault revealing tests and $\frac{11}{14500} = 0.00076$, i.e. 0.076% for flipped tests. As for reduction, it will always be 75% since 25% of the test suite is selected.

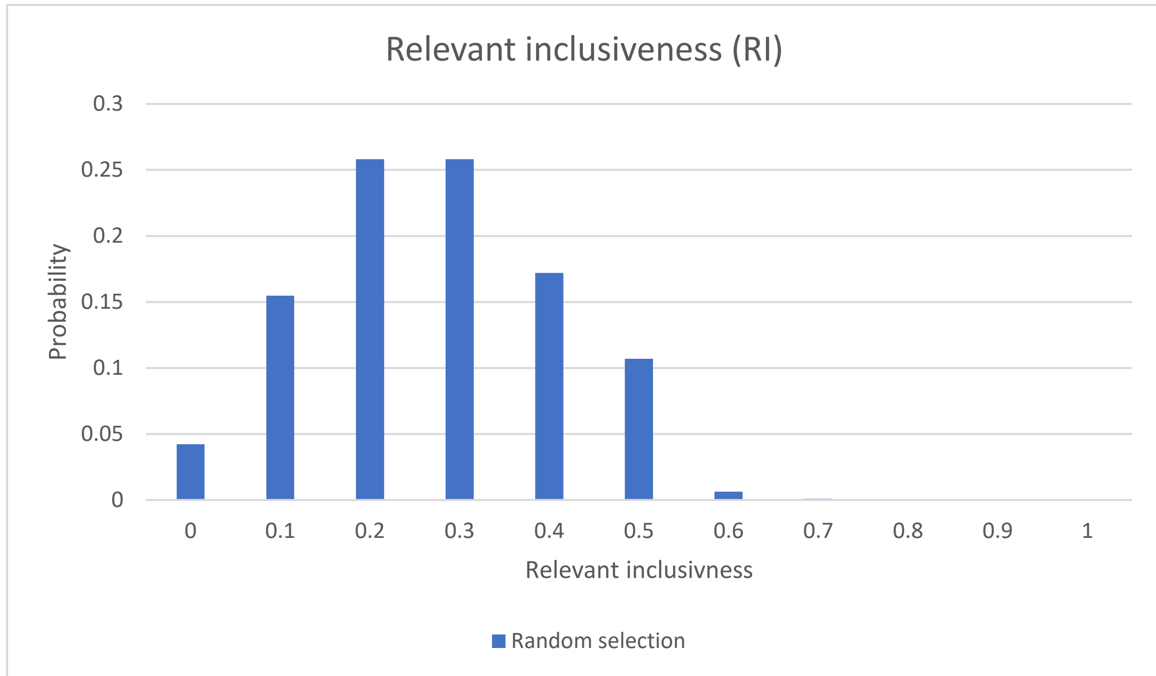

Figure 7.13: The likelihood of getting any relevant inclusiveness when randomly selecting 25% of the test suite.

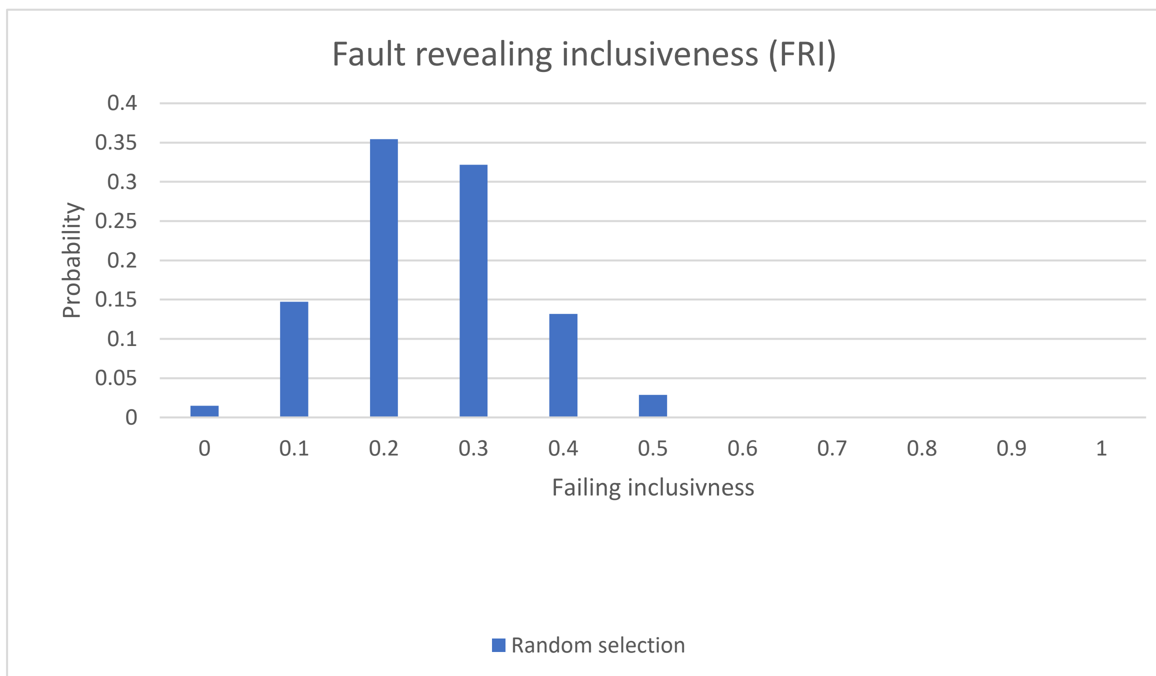

Figure 7.14: The likelihood of getting any fault revealing inclusiveness when randomly selecting 25% of the test suite.

Chapter 8
Discussion

In this chapter the results and their implications are discussed. Threats to validity are also covered and ethical aspects discussed.

8.1 Empiricist

An approach not initially considered was the recommend all flipped (RAF) approach. This approach was introduced as a side effect of a certain file always being present in the modifications between software versions. The result of this being that all the previously flipped tests were always recommended. This file was filtered out to achieve the connection-based recommendation approach (CBR) but it was decided to investigate RAF further. A weakness of CBR is that it will not recommend any tests if the source entity that has been modified lacks any links. A fallback approach was introduced at the very fine-grained granularity, i.e. file granularity, but if the directory that contains the file also lacks links there will still be no recommendations. RAF does not have this weakness. All the previously flipped tests will be recommended, regardless of where the modifications took place. The reason why this generates such good values while still recommending a relatively small subset of CTS* (e.g. 343 tests after test runs on 100 software versions) is because tests that flip and/or fail tend to be the tests that recently flipped and/or failed. This is vital information since it shows how small the fraction of flipped and fault revealing tests is given the size of the test suite, providing even more credence to the inefficiency of always running the entire test suite. Additionally, although RAF is approaching the threshold for the acceptable number of tests to recommend, i.e. 1229, when using test runs on 160 software versions on three different products to train Empiricist it is still only on average recommending 878 tests, as can be seen in Appendix in Table A.1. What is interesting here is that it relatively quickly, after 40 test runs, reaches an average of 833 recommended tests. This means that roughly 45 additional tests flipped over the subsequent 120 test runs which substantiates the temporal connections between which tests flip and fail. It is important to note that although RAF performs well, the number of tests that it recommends will never cease to grow, given that the same database is continuously built upon. Some mechanism for maintaining the relevance of the tests in the database is clearly needed.

The CBR approach performs as expected on the different granularity levels, i.e. project-, directory- and file-level. The granularity is essentially a trade-off between precision and inclusiveness. Coarser granularity provides higher inclusiveness since a project will contain all the links that every directory within it establishes. On the other hand, finer granularity provides higher precision since the links established are more precise, i.e. they contain less links related to other source entities such as directories and files. Where the threshold for inclusiveness and precision lie depends largely on the size of the test suite. Low precision

can be tolerated when the test suite is very large since the reduction will then be large regardless, which results in a lot of time saved.

In the normalized connection-based approach (NCBR) weights on links were normalized to distinguish between links resulting from true correlations and links resulting from random noise. The hypothesis being that true links will eventually distinguish themselves from false links since the same combination of source entity *E* and test *T* will be more frequent if there is an actual correlation between the two, meaning that test *T* actually tests functionality in entity *E*. The results in Figure 7.7 and Figure 7.8 show that the precision increases as the cutoff criterion is increased. Although the values achieved on both inclusiveness variables, seen in Figure 7.3 and Figure 7.4, are not acceptable from a user's point of view, the increase in precision shows that there is substance behind the hypothesis. It could be that the number of flipped tests contained in the database is not large enough yet to warrant normalization. As mentioned in Section 7.2.3, the number of recommended tests is stable over the intervals when using NCBR. This is an interesting behavior but not necessarily something that is desirable in its current state. It could be due to the time it takes for links to distinguish themselves in terms of their weights. A newly established link will need time to distinguish itself from other links since the other links have had a head start. Normalization therefore results in Empiricist never recommending the most recently established links, i.e. Empiricist becomes sluggish to newly established links. Additionally, NCBR will remove any links with comparatively lower weights. The effect of this can be that if initially a subset of links accumulates high weights and the same links always have their weights strengthened, other links will be lagging behind and therefore never catch up to the initial subset that constantly have their weights strengthened. Both of these aspects are undesirable and will likely have to somehow be compensated for.

As can be seen in Appendix Table A.2-A.7, inclusiveness when normalizing before a subset is selected based on links to modified source entities and normalizing after is consistently higher or equal in the latter case. The actual difference is varying: ranging from nothing to approximately 20% when looking at *FRI*. A larger difference in inclusiveness should reflect more outliers in the database being removed when normalizing after. The reason for this, as stated in Section 5.5.3, is that outliers present in the database but not present in the subset of links that are connected to the modified entities will not have an effect when normalization is done after a subset is selected.

The precision values presented in Section 7.2.2 are likely worse than what they are in reality. The likelihood of tests not failing or flipping even if they directly or indirectly test the modified code can be assumed to be high. Developers usually do their job well so the introduction of bugs in all the code that is modified is not plausible. This must be considered when viewing the precision since the objective is to select a subset of tests that test the modified code and not only the tests that fail. With this said, the ExtHB approach inadvertently does take the likelihood of a test failing into consideration since links are established only when tests flip, meaning they must at some point have failed. Tests that test a given source entity but never fail will therefore never be recommended. This trait is

of use when, as in the case of this thesis, the test suite is large, and most tests never fail over a long period of time.

An alternative approach to the one taken in this thesis could be to select a subset of tests randomly from the CTS*. This approach is well established in literature where a common approach is to select 25% of the test suite to be executed [19]. Using the data calculated in Section 7.3 and presented in Figure 7.13 and 7.14, a random approach where 25% of the test suite is selected, the values of inclusiveness can be seen to be far lower than those of ExtHB. The likelihood of even reaching an inclusiveness of 50% is very low. Instead, odds are that inclusiveness will fall somewhere within 20 to 40%. These values are very noteworthy since ExtHB achieves up to and over 90% inclusiveness while recommending only 341 of the test suite. This in contrast to the 14 500 that roughly make up 25% of the test suite. The difference in runtime between these two values is, on average, $\frac{14500*0.83}{60} - \frac{341*0.83}{60} \approx 196\ minutes$ which from the user's point of view is large. Although the random selection approach was never considered in this domain, it is nonetheless interesting to note by how much it is outperformed by ExtHB.

The number of test runs on different software versions it takes to train Empiricist to become sufficiently performant can be concluded by using the minimum acceptable FRI obtained through the survey. Looking at Figure 7.1 an average FRI of well over 63% is achieved already in the second interval when using test runs on 20 different software versions to train Empiricist. This is true for RAF as well as all granularities of CBR. It is, however, not the case for NCBR which should only be used when there are many tests in the database. The process needed to train Empiricist to be sufficiently performant is of high importance in an industrial implementation since time is of the essence. The fact that acceptable values can be achieved so fast shows the high applicability of ExtHB in an industrial setting.

Ekelund and Engström mention in their paper that the approach thrives on an unstable codebase which is not something that is desirable. This is certainly true as connections between source code and tests cannot be identified unless the source code is changed and the test flips. However, there is no need to find all the connections if they never generate anything useful, i.e. they never flip. Finding all connections between source code and tests would make the mapping equivalent to the static technique, given that true connections can be distinguished from false ones, otherwise it would make it worse due to the noise. Therefore, it is of little desire to have connections to all the tests in the test suite if nothing of importance arise from them. Though, it is also true that this view of how to construct a mapping in ExtHB will always fail to recommend a relevant test the first time it presents itself. Whether or not the test is recommended the subsequent times depends on the sub-approach taken within ExtHB, e.g. RAF, CBR or NCBR.

The results of the work conducted in this thesis further supports the efficiency of the history-based technique taken by Ekelund and Engström. Reductions were measured in the 99th percentile while in some cases achieving an inclusiveness of over 90% both in relation

to relevant and fault revealing tests. Compared to a technique like static analysis, it is also computationally cheap and easily scalable since it lacks dependencies to anything other than historical test results. In their paper Ekelund and Engström found that the precision drops with time as more connections are found and thereby more tests recommended. As can be seen in Figure 7.5, 7.6, 7.7 and 7.8, this is also mostly true for the implementation in this thesis. However, normalization has been introduced as a countermeasure to this. As previously mentioned, results show that precision markedly increases when the links are normalized based on their weights and a cutoff criterion is used. When the precision drops to undesirable values normalization could potentially counteract that by removing weaker links.

The reason for measuring both relevant inclusiveness (*RI*) and fault revealing inclusiveness (*FRI*) is that from the user's perspective relevant, i.e. flipped, tests is not as vital as finding fault revealing tests. It is of course also of use to verify that a fix is working correctly, but finding bugs is of more importance. Another reason to measure them both is to see if and how they differ over time. It can be seen from Figure 7.1 and Figure 7.2 that they do in fact differ slightly. FRI is often slightly below the values of RI. There are several potential explanations for this. A test that never pass will not be caught as a relevant test since it does not flip, thereby affecting FRI but not RI. The reason that this can happen is that certain tests in the CTS have external dependencies. For example, certain tests will inevitably fail if the smartphone is missing an SD-card. If this is consistently ignored, these tests will never pass. In the rarer case when the opposite is true, i.e. FRI is higher than RI, it would mean that a test that recently flipped fails over the following runs. The effect of this would be that the test can be recommended since it is present in the database but will be viewed as not relevant since it does not keep flipping. Rather, it keeps failing.

Much like the history-based technique implemented by Ekelund and Engström, the feedback loop from making changes to receiving quality feedback is significantly shortened. The number of tests recommended differs depending on the product used to train Empiricist. When using RAF on a certain product, 341 tests are recommended on average after test runs on 100 software versions. On another product, the number of recommended tests is 428 after test runs on 80 software versions. As is displayed in Appendix A in Table A.1, the average number of recommended tests is 872 after test runs on 100 software versions when using RAF on three different products simultaneously. In none of these cases is the average of either RI or FRI for the final interval less than 85%. In two of the cases the average of both RI and FRI are above 90%. Although when using three products simultaneously FRI drops to around 80% for a few intervals after the 140th test run.

Running the entire CTS* takes approximately 11 to 14 hours, while running a randomly selected subset equal to 25% of the CTS* would take approximately $\frac{14500*0.83}{60} \approx 200\ minutes$. Using ExtHB, the highest measured number of recommended tests was 878 which occurred when RAF was used on three products simultaneously after test runs on 180 software versions. Running this subset would on average take $\frac{878*0.83}{60} \approx 12\ minutes$.

Even taking into account the time to generate these recommendations into account, which will finish within 2 minutes, it is still below the 17 minutes that was deemed acceptable by developers.

Lastly, as Ekelund and Engstöm mention, while the history-based technique significantly shortens the feedback loop for changes made to the source code, it is not meant to and positively cannot, replace running the entire test suite regularly. The history-based technique relies on up-to-date historical data to be able to construct and maintain a relevant database of links between source code and tests. It needs to be stressed that running the entire test suite regularly, e.g. nightly, is therefore essential for the approach to work at all.

8.2 Threats to Validity

In a real world working environment, Empiricist is meant to recommend tests based on relatively small changes, e.g. the equivalence of individual commits. However, due to the time constraints it was not possible to test how Empiricist would perform in this environment. The RAF approach is not affected at all by how big the changes are when recommending tests but both CBR and NCBR are directly affected by this. The reason that CBR and NCBR are affected is that a modified entity *E1* can be present in the software difference which gives rise to test *T* being recommended. However, test *T* can turn out to be relevant to another modification *E2* which would not on its own have led to the test being recommended. If this situation occurs, the dependent variables measured are likely to be better than what they should have been. Another situations that can occur when using large amounts of modifications to generate test scopes is that there can be modifications that fix previous modifications that would have made a test fail. Since this has been fixed in the same set of modifications, there can be no way to know whether the test that would have failed but did not would have been recommended by Empiricist given only the modification that contained the bug.

As mentioned in Section 7.1, all the modifications were not obtained when retrieving the modifications made between software versions. This was due to the way taking the software difference using the manifest file works. Test flips that were caused by modifications not visible to Empiricist will erroneously be associated with the rest of the modifications present in the difference. This should conceivably only have a negative effect on the measured dependent variables since only noise is stored in the case mentioned. Precision is therefore expected to suffer.

Test results have been observed to at times report modules to have failed even if they passed. This does not affect Empiricist since the information is not used on module level, instead individual tests are checked to see if they passed or failed. However, even though false information has not been observed to be present in reports regarding whether a test failed or passed, the possibility cannot be ignored. If false information is in fact reported

on information that is used, it will affect the outcome of the dependent variables. The impact would be in relation to how prevalent the false information would be.

As with any software, although the functionality of Empiricist was verified both manually and through unit tests throughout the development process, possible bugs and/or logical flaws cannot be excluded. Reservations are therefore made regarding this.

Some tests are officially broken as reported by Google. It is also possible that there are broken tests that are not yet reported as such by Google. The flips of such test could be random. Other tests can have dependencies to states that previous tests left behind, i.e. the previous tests did not properly clean up the states that were set by them. All these situations, and possibly others, where tests seemingly flip randomly without any connection to the modifications made will have a large impact on NCBR where normalization is used. As mentioned, normalization using the min-max scaler is sensitive to outliers and tests that frequently flip will also be more likely to accumulate higher weight values which would push the relative values of other links towards 0.

8.3 Unit Tests

To continuously verify the functionality of Empiricist, unit tests were developed. Due to some functionality being dependent on an authenticated connection to the network and servers used at Sony Mobile as well as being of the sort that the output needs manual verification, full coverage could not be achieved. However, approximately 65% of the source code is covered by unit tests. The functionality and validity of the remaining classes and functions were verified manually throughout the development process

8.4 Ethical Aspects

When analyzing and discussing the results of the extended history-based technique, it is important to keep in mind how the results are presented. Transparency regarding possible disadvantages and/or aspects where the solution does not perform as well is essential. If results were manipulated or data was left out due to being unfavorable to the solution this would be deceiving both the employer and the potential future users. The employer would have needlessly wasted resources while the future users might consistently believe their code changes to be correct when in fact they are not.

Chapter 9
Conclusion

In this chapter, conclusions are made in reference to the initially posed thesis questions.

TQ.1. By how much can the number of tests selected from CTS* be reduced while maintaining high inclusiveness?

Looking at the data presented in Chapter 7, high inclusiveness is attained both in relation to relevant and fault revealing tests using both recommend all flipped (RAF) and connection-based recommendation (CBR). The reduction falls within the 99th percentile for all the approaches on one product, while it falls within the 98th percentile using three products. Although the reduction increases along with the cutoff criterion when normalizing links based on their weights, the inclusiveness is not sufficient yet from the user's point of view.

TQ1.1. Can high inclusiveness be attained while high reduction is maintained?

Since the test suite is so large and the same tests tend to flip over a given interval, it would take quite some time for reduction to fall below 90% even with RAF. Both high inclusiveness and reductions are therefore attained using RAF and CBR. As mentioned in the answer to TQ.1, while the reduction is higher in NCBR the inclusiveness is not sufficient yet. NCBR may only be applicable when there are a lot more links in the database.

TQ1.2. By how much would the time be reduced, compared to running the CTS*, taking the overhead of dynamically selecting a subset of tests into account.

The overhead for selecting and recommending a subset of tests is relatively small: it finishes within 2 minutes when the maximum number of links was present in the database. The time it takes to run CTS* is approximately 11 to 14 hours. The largest number of recommended tests was found to be 878. Using the average execution time for a test this could be estimated to take $\frac{878*0.83}{60} \approx 12\, minutes$. Using the lower estimate of 11 hours for CTS* to finish a huge amount of time is saved, the recommended subset finished in roughly 2% of the time it takes the test suite to run.

TQ.2. How much and what kind of data is needed to get a satisfying result, considering TQ.1.

The extended history-based technique needs historical data from previous full scope test suite runs. It also needs continuous access to this kind of data, meaning that the full scope test suite runs must be performed regularly, preferably nightly, to get best performance out of the approach. It can be seen from the data presented in Chapter 7 that an average of around 60% inclusiveness is achieved even within the first interval whereas when using RAF to create test scopes and 60 test runs on different software versions to train Empiricist the fault revealing inclusiveness reaches 89%.

To conclude this chapter, the extended history-based approach is an effective solution in the problem domain of this thesis. This can be concluded from the results from Empiricist coupled with the specification of the criteria, defined in Section 6.5, that an effective solution must meet.

Chapter 10
Future Work

Possible future improvements to the extended history-based approach are suggested and discussed here.

A fundamental difficulty in the extended history-based (ExtHB) technique is the maintenance of the database containing the links between source code entities and tests. Several situations occur where links should by some means be removed:

- The source entity is modified in a way that renders a connection to a test that used to be true invalid
- The test is modified in a way that removes the connection to a source entity
- Noise occurs when establishing and maintaining links

To use a database constructed with this approach an intelligent way to remove links of the aforementioned kind will eventually be necessary since the amount of links will never cease to grow. Introducing timestamps for when a weighted link was last updated, as well as last used, along with a timeout might alleviate this problem. Source entities that are removed from the codebase will then eventually be removed from the database as well. If normalization is used, links resulting from true correlations should be used more frequently than false links which would allow for removal of links resulting from noise using this method. However, a trade-off exists here between wanting to keep an up-to-date database and wanting to keep all true links. A link that is not used and/or updated for a given amount of time might still be a valid one and it would be a loss to remove it. Setting the timeout parameter to a well thought through and tested value would therefore be imperative.

Another possible approach to maintaining a database could be to identify the optimal number of test runs on previous software versions to use to train Empiricist. If this number would be found to be 100, it could then be used to remove any links that are older than the oldest test run that is to be included, i.e. the test run that was performed 100 test runs ago. The disadvantage here is, of course, that true connections are lost but if the performance is good enough with this amount of historical data then it might be a viable solution.

As mentioned in Section 5.3, the min-max scaler used in this work is sensitive to outliers. There are data pre-processing techniques that can handle such outliers better such as standardization or the robust scaler from the sci-kit learn pre-processing module. Exploring different approaches here in using weights to remove false links could potentially render better results.

The undesirable effects of normalization, discussed in Section 8.1 where newly established links have a hard time distinguishing themselves through their weights, could potentially be alleviated by modifying the way in which weights are established. Currently when a link is initially established it will be assigned the default value of 1. An alternative way to establishing links could be to assign a weight equal to the median of all weights or even equal to the highest weight present in the database. The weights of links could then be incremented when they recommend a test that turns out to be fault revealing and/or relevant test. Conversely, they could be decremented each time they recommend a test that did not turn out to be fault revealing and/or relevant.

The work of this thesis was limited to identifying and measuring the effect of different source code granularities. Different granularities can also be applied to the test suite which would possibly enable more consistently high inclusiveness. Precision will naturally suffer and the effectiveness of coarser granularities on the test suite will likely depend on the size of the test suite itself. On a large test suite, such as the CTS, coarser granularity on the mapping to test suite entities could be worthwhile exploring.

Flaky tests are tests that have a non-deterministic outcome for the same software version i.e. they can both pass and fail during different runs on the same software version [20]. These types of tests can have an impact on the performance of ExtHB since such tests are likely to accumulate high weight values on their links which in turn will affect the normalization process. Essentially, they risk becoming outliers in the data sets. In their paper Qingzhou L, Hariri F, Eloussi L and Marinov D [20] identified three common causes for flaky tests:

- Async wait
- Concurrency
- Test order dependency

Recognizing such tests automatically could be difficult since the reasons for their flakiness can be so varied. It is also not clear how to handle such tests. Flaky tests might still provide coverage and their flakiness is sometimes due to an actual bug in the tested software. Removing such a test would thus reduce the effectiveness of the test suite [20]. However, at a minimum, there should be support for blacklisting such tests manually for a given amount of time with the right expertise. As mentioned in Section 8.2, test order dependency is known to exist in CTS due to tests failing to clean up properly after they finish, leaving behind a state that affects the result of later tests.

When the database constructed using ExtHB becomes sufficiently performant, a somewhat ironical situation risk occurring. If the right tests are recommended during the day, which results in the bugs being fixed before the full scope nightly test run, the database will fail to identify and increment the weights of these links. This is a problem when weights are normalized since links that consistently reveal bugs will not have their weights incremented as often as they should have. A possible solution to this would be to use the test results

from the recommended subsets that are run throughout the day to update the database. That is, if test *T* flips on a test run of a recommended subset on a local software build, this can be used to update and/or create links given all the local modifications. The flip in this case would be in reference to the most recent official software version tested.

Using additional meta-information such as a test's estimated runtime would enable a more dynamic selection process. A developer wanting to verify changes could then specify an acceptable time to wait for feedback. Given more exact information on individual tests execution time the subset of selected tests could then further be reduced to finish within the acceptable time.

Glossary

Android Open Source Project (AOSP) The Android operating system.

Coarse-grained The granularity at which projects are linked to tests.

Codebase A term referring to all the files, directories and projects within AOSP.

Compatibility Test Suite (CTS) The test suite provided by Google for regression testing.

CTS* A subset of CTS used in the department where the thesis work was conducted. It consists of approximately 58 000 tests.

Connection-Based Recommendation (CBR) The approach where only tests connected to modifications are recommended.

Cutoff criterion The threshold by which normalized weights are removed, i.e. normalized weights that have values below the threshold are removed.

Effectiveness A measurement specific to the working environment where this thesis took place. It is defined as a combination of inclusiveness and reduction.

Empiricist The software implemented on the basis of the extended history-based technique (ExtHB).

Extended History-Based Technique (ExtHB) An extension of the technique initially explored by Ekelund and Engström where links are established on different granularities and weights are normalized and subject to a cutoff criterion.

Fallback An approach taken on very fine-grained granularity where if no links are found to the modified files, the links to the directories containing those files will be used.

Fine-grained The granularity at which source directories are linked to tests.

Inclusiveness The percentage of fault revealing or relevant tests that was recommended.

Link The term for a connection between a source entity and a test.

Manifest file A file containing, along with other configuration options, the remote server and revision tags for each project, i.e. repository, within AOSP.

Noise A term used to describe the situation when links are established erroneously. That is, several modifications are made between two tested software versions but only a subset of them resulted in a test flip. However, all of them will be linked to the test that flipped. The modifications that did not cause the flip but are linked to it are said to be caused by noise.

Normalization A technique used to scale data into a fixed range.

Normalized Connection-Based Recommendation (NCBR) The approach where only tests connected to modifications are recommended after they have been subject to normalization along with a cutoff criterion.

Precision The percentage of recommended tests that are fault revealing or relevant.

Product A Sony smartphone model.

Project A Repository within the AOSP.

Recommend All Flipped (RAF) The approach where all previously flipped tests are always recommended regardless of what source entities have been modified.

Reduction The percentage by which the test suite is reduced.

Reference test results The two test result files that have been used to generate a test scope.

Regression test selection (RTS) Techniques that seeks to select a subset of tests that will test the changed parts of the software.

Regression testing A testing process where tests are reused to verify that software modifications are correct while also ensuring that the intended functionalities of the unmodified parts are not adversely affected.

Relevant test A test that has flipped between two consecutive test runs.

Software difference The set of modifications made to the codebase between two software versions.

Source entity A term used to describe files, directories and projects that are contained within the codebase.

Temporal connection A term used to describe the situation where tests that flip tend to be tests that recently flipped.

Test flip A term used to convey the situation where the outcome of a test is the opposite of what it was on the previous test run, i.e. it previously passed and now fails, or it previously failed and now passes.

Test result XML file containing which tests failed during a test run**.**

Test scope XML file containing the tests recommended by Empiricist.

Training data The number of test results used to build the database in Empiricist before evaluation of the output occurs.

Very fine-grained The granularity at which source files are linked to tests.

Weight The term for a value associated with a link that reflects how many times the same source entity *E* has been modified while the same test *T* has flipped**.**

References

[1] Android Compatibility [Internet]. [updated 2017 Nov 15; cited 2018 Apr 16]. Available from: https://source.android.com/compatibility/

[2] Compatibility Program Overview [Internet]. [updated 2017 Sep 13; cited 2018 Apr 16]. Available from: https://source.android.com/compatibility/overview

[3] Rothermel G, Harrold M.J. Analyzing Regression Test Selection Techniques. IEEE Transactions On Software Engineering. 1996; (8): 529-551.

[4] Yoo S, Harman M. Regression Testing Minimization, Selection and Prioritization: A Survey. Software Testing: Verification & Reliability. 2012;22(2): 67-120.

[5] Beszedes A, Gergely T, Schrettner L, Jasz J. Lango L, Gyimothy T. Code Coverage-Based Regression Test Selection and Prioritization in WebKit. Software Maintenance (ICSM), 2012 28th IEEE International Conference on. 2012;46.

[6] Fazeli N. Machine Learning to Uncover Correlations Between Software Code Changes and Test Results. Student essay. Gothenburg: Chalmers University of Technology; 2017. Available from: http://hdl.handle.net/2077/54576

[7] Ekelund, E.D, Engström E. Efficient Regression Testing Based on Test History: An Industrial Evaluation. 2015 IEEE International Conference On Software Maintenance And Evolution (ICSME). 2015.

[8] Manifest [Internet]. Gerrit.googlesource.com. 2018 [cited 2018 May 1]. Available from: https://gerrit.googlesource.com/git-repo/+/master/docs/manifest-format.txt

[9] Overview [Internet]. 2018 [updated 2018 Mar 27; cited 2018 May 1]. Available from: https://source.android.com/setup/develop/

[10] Architecture [Internet]. 2018 [updated 2018 Mar 27; cited 2018 May 1]. Available from: https://source.android.com/setup/develop/

[11] Chollet F. Deep learning with Python. Manning publication; 2017. P.4-11.

[12] Li L, Bissyandé T, Papadakis M, Rasthofer S, Bartel A, Octeau D, et al. Static Analysis of Android Apps: A Systematic Literature Review. Information And Software Technology. 2017; 8867-95.

[13] JavaParser [Internet]. [cited 2018 May 16].
Available from: https://github.com/javaparser/javaparser

[14] Clang Parser [Internet]. Clang.llvm.org. 2018 [cited 2018 May 16].
Available from: http://clang.llvm.org/doxygen/classclang_1_1Parser.html

[15] FlowDroid Static Data Flow Tracker [Internet]. [cited 2018 May 16].
Available from: https://github.com/secure-software-engineering/FlowDroid

[16] Engström E, Runeson, P, Ljung A. Improving regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study. 2011 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2011; 367-376.

[17] Sklearn.preprocessing.MinMaxScaler documentation [Internet]. Scikit-learn.org. 2018 [cited 16 May 2018]. Available from:
http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[18] Patterson J, Gibson A. Deep learning: A Practitioner's Approach. Sebastopol: O'Reilly; 2017. P.332.

[19] Engström E, Runeson P, Skoglund M. A Systematic Review on Regression Test Selection Techniques. Information and Software Technology. 2010;52(1): 14-30.

[20] Qingzhou L, Hariri F, Eloussi L, Marinov D. An Empirical Analysis of Flaky Tests. FSE Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering 2014;643-653.

Appendix A
Tables

Table A.1: RAF using three products

Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0	43.11	33.93	75.7	64.46	99.43	338
20	5.42	3.85	90.51	76.43	98.92	641
40	4.57	5.37	91.44	89.18	98.6	833
60	0.89	1.51	96.51	91.01	98.53	861
80	0.48	1.76	98.0	92.54	98.51	872
100	0.31	1.6	96.25	92.02	98.51	874
120	0.7	2.49	99.5	94.94	98.5	876
140	0.47	1.28	99.38	89.49	98.5	877
160	0.55	0.73	99.78	83.61	98.5	878

Table A.2: Normalization before subset is selected with NCBR on very fine-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	14.53	25.79	44.9	42.72	99.83	98
0.05	20	3.5	9.15	45.79	31.97	99.79	127
0.05	40	1.59	2.64	57.64	30.9	99.72	165
0.05	60	6.84	14.48	47.87	35.76	99.75	143
0.05	80	13.62	28.91	51.56	43.71	99.83	98
0.1	0	14.53	25.79	44.9	42.72	99.83	98
0.1	20	3.5	9.15	45.79	31.97	99.79	127
0.1	40	3.54	5.02	56.55	28.06	99.79	123
0.1	60	18.49	26.0	39.69	24.4	99.92	44
0.1	80	22.03	44.39	35.08	26.21	99.96	25
0.2	0	15.37	26.77	33.78	34.7	99.88	69
0.2	20	4.3	10.49	24.15	13.76	99.94	38
0.2	40	18.04	21.08	49.16	20.87	99.94	38
0.2	60	26.08	38.4	29.64	16.55	99.97	15
0.2	80	35.32	55.8	26.81	13.05	99.98	9
0.3	0	17.78	29.79	24.26	20.87	99.94	36
0.3	20	17.91	27.55	16.4	9.17	99.98	11
0.3	40	27.38	31.83	46.59	12.4	99.98	12
0.3	60	29.62	34.81	15.39	8.02	99.99	7
0.3	80	39.64	67.67	21.14	6.9	99.99	5
0.4	0	20.23	32.86	21.78	15.17	99.96	22
0.4	20	18.61	31.8	16.4	8.21	99.98	9
0.4	40	34.29	37.78	29.98	9.52	99.99	6
0.4	60	40.38	53.25	13.72	6.91	99.99	3
0.4	80	43.19	75.54	12.57	4.19	100.0	2

Table A.3: Normalization before subset is selected with NCBR on fine-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	15.51	26.63	46.78	43.03	99.83	101
0.05	20	3.07	6.36	56.07	41.11	99.73	159
0.05	40	1.82	2.48	72.96	35.15	99.65	204
0.05	60	6.33	13.5	50.89	42.41	99.68	185
0.05	80	7.88	14.65	61.8	51.07	99.71	168
0.1	0	15.51	26.63	46.78	43.03	99.83	101
0.1	20	3.17	7.41	56.07	39.54	99.76	141
0.1	40	10.29	12.04	69.79	27.03	99.88	68
0.1	60	17.88	22.63	44.18	28.2	99.89	67
0.1	80	14.97	32.66	44.95	38.03	99.9	60
0.2	0	16.31	25.69	35.26	29.49	99.9	58
0.2	20	5.13	13.64	36.73	26.13	99.91	54
0.2	40	10.36	17.14	59.08	21.84	99.95	29
0.2	60	20.52	31.1	42.29	26.56	99.95	32
0.2	80	26.76	46.24	33.9	27.86	99.96	24
0.3	0	19.71	30.44	27.83	21.56	99.94	35
0.3	20	8.68	22.86	28.45	18.05	99.96	21
0.3	40	13.47	18.31	50.45	14.28	99.97	14
0.3	60	22.74	32.9	25.47	11.0	99.98	12
0.3	80	30.27	55.7	25.8	15.05	99.98	11
0.4	0	26.51	36.91	22.34	15.36	99.96	22
0.4	20	14.29	24.25	19.92	10.33	99.98	12
0.4	40	20.88	25.27	40.57	11.94	99.99	8
0.4	60	26.61	41.94	20.84	9.79	99.99	6
0.4	80	30.35	55.09	15.29	6.42	99.99	5

Table A.4: Normalization before subset is selected with NCBR on coarse-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	14.65	26.44	49.83	46.03	99.82	108
0.05	20	2.99	5.84	58.57	42.16	99.71	174
0.05	40	1.76	2.5	77.09	41.54	99.58	244
0.05	60	2.31	3.42	64.81	48.54	99.56	254
0.05	80	3.85	10.19	73.42	58.46	99.64	209
0.1	0	14.65	26.44	49.83	46.03	99.82	108
0.1	20	3.53	7.63	49.2	39.09	99.81	114
0.1	40	4.45	5.57	74.0	31.66	99.85	86
0.1	60	5.73	7.54	51.72	35.12	99.85	86
0.1	80	5.46	17.3	56.72	48.1	99.86	80
0.2	0	16.81	26.57	38.7	30.37	99.9	60
0.2	20	5.28	13.89	35.25	27.25	99.93	44
0.2	40	6.65	11.08	60.0	22.95	99.94	34
0.2	60	10.85	17.99	41.84	23.01	99.94	32
0.2	80	13.5	37.39	44.59	37.29	99.94	33
0.3	0	19.65	30.49	30.89	22.6	99.94	37
0.3	20	8.33	18.87	29.22	17.79	99.96	20
0.3	40	8.6	16.22	46.51	13.9	99.98	13
0.3	60	19.59	32.09	26.2	12.54	99.98	10
0.3	80	20.3	43.87	32.87	22.53	99.97	14
0.4	0	22.32	32.89	24.91	16.09	99.96	23
0.4	20	10.19	22.54	23.84	11.75	99.98	11
0.4	40	11.51	22.5	34.97	12.26	99.99	6
0.4	60	23.44	41.26	24.17	11.62	99.99	6
0.4	80	22.61	43.94	19.46	10.45	99.99	7

Table A.5: Normalization after subset is selected with NCBR on very fine-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	14.53	25.79	44.9	42.72	99.83	98
0.05	20	3.43	9.42	44.12	31.18	99.79	124
0.05	40	1.59	2.64	57.64	30.9	99.72	165
0.05	60	6.9	14.6	48.2	36.35	99.75	145
0.05	80	8.65	24.92	50.94	42.59	99.83	96
0.1	0	14.53	25.79	44.9	42.72	99.83	98
0.1	20	3.43	9.42	44.12	31.18	99.79	124
0.1	40	2.34	3.4	57.64	29.63	99.77	132
0.1	60	7.01	15.59	46.54	34.33	99.8	117
0.1	80	9.86	26.5	46.32	40.87	99.88	72
0.2	0	14.8	26.29	38.62	37.35	99.87	80
0.2	20	4.53	11.36	28.46	22.24	99.89	65
0.2	40	3.92	6.46	50.8	24.29	99.91	51
0.2	60	8.58	19.33	41.62	27.47	99.91	52
0.2	80	12.33	39.14	37.27	34.66	99.93	39
0.3	0	18.06	28.78	27.19	23.89	99.93	43
0.3	20	6.58	16.37	23.89	17.45	99.95	30
0.3	40	6.47	10.6	42.37	14.07	99.97	16
0.3	60	10.86	24.79	36.48	19.61	99.96	21
0.3	80	12.48	42.54	29.67	21.29	99.97	17
0.4	0	20.43	32.43	23.28	15.92	99.96	24
0.4	20	8.1	16.62	20.15	11.11	99.97	19
0.4	40	11.44	17.63	35.65	11.56	99.98	9
0.4	60	13.4	31.5	19.56	15.3	99.98	14
0.4	80	13.53	40.05	26.08	13.63	99.98	12

Table A.6: Normalization after subset is selected with NCBR on fine-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	15.51	26.63	46.78	43.03	99.83	101
0.05	20	3.07	6.36	56.07	41.11	99.73	159
0.05	40	1.82	2.48	72.96	35.15	99.65	204
0.05	60	6.33	13.5	50.89	42.41	99.68	185
0.05	80	7.88	14.65	61.8	51.07	99.71	168
0.1	0	15.51	26.63	46.78	43.03	99.83	101
0.1	20	3.17	6.59	56.07	40.45	99.75	147
0.1	40	3.63	4.31	71.04	29.03	99.83	99
0.1	60	7.22	15.27	45.97	32.62	99.84	92
0.1	80	9.32	18.69	51.7	44.25	99.84	91
0.2	0	16.64	27.23	37.61	34.14	99.88	68
0.2	20	4.88	11.02	38.39	30.65	99.89	67
0.2	40	4.3	8.08	60.68	23.22	99.93	39
0.2	60	9.42	21.62	42.51	28.03	99.93	40
0.2	80	11.01	31.24	37.9	36.69	99.94	35
0.3	0	20.71	31.44	29.08	21.71	99.94	35
0.3	20	8.65	20.31	30.12	19.29	99.96	23
0.3	40	7.2	13.32	50.79	15.06	99.97	17
0.3	60	11.69	22.67	35.2	15.51	99.97	18
0.3	80	12.17	36.07	29.8	25.78	99.97	18
0.4	0	22.13	31.64	25.96	16.49	99.96	25
0.4	20	9.72	22.26	23.25	14.06	99.97	15
0.4	40	9.45	17.39	46.03	13.41	99.98	11
0.4	60	13.93	32.59	29.34	12.81	99.98	10
0.4	80	13.53	33.57	26.29	14.18	99.98	10

Table A.7: Normalization after subset is selected with NCBR on coarse-grained granularity using one product.

Cutoff Criterion	Training Data	RP (%)	FRP (%)	RI (%)	FRI (%)	Reduction (%)	Avg # Tests
0.05	0	14.65	26.44	49.83	46.03	99.82	108
0.05	20	2.99	5.84	58.57	42.16	99.71	174
0.05	40	1.76	2.5	77.09	41.54	99.58	244
0.05	60	2.31	3.42	64.81	48.54	99.56	254
0.05	80	3.44	7.89	77.79	61.23	99.6	235
0.1	0	14.65	26.44	49.83	46.03	99.82	108
0.1	20	3.53	7.63	49.2	39.09	99.81	114
0.1	40	3.82	4.91	74.0	32.53	99.83	97
0.1	60	3.69	6.14	57.39	37.31	99.82	105
0.1	80	5.69	15.09	62.97	50.05	99.84	94
0.2	0	16.81	26.57	38.7	30.37	99.9	60
0.2	20	5.14	12.65	36.92	28.57	99.92	49
0.2	40	5.28	8.68	62.5	23.82	99.94	37
0.2	60	5.81	11.09	49.18	26.83	99.93	41
0.2	80	8.37	27.56	47.09	42.01	99.93	41
0.3	0	19.65	30.49	30.89	22.6	99.94	37
0.3	20	8.19	19.18	29.22	18.84	99.96	21
0.3	40	9.73	14.62	50.49	15.4	99.97	15
0.3	60	11.15	17.91	35.2	16.53	99.97	15
0.3	80	8.25	35.23	32.87	28.29	99.97	19
0.4	0	22.32	32.89	24.91	16.09	99.96	23
0.4	20	9.93	23.8	25.5	13.86	99.98	13
0.4	40	13.76	24.75	42.47	13.96	99.99	7
0.4	60	13.16	25.12	29.84	14.04	99.98	9
0.4	80	9.65	30.54	19.46	13.57	99.98	9

Table A.8: The average of the answers from the survey sent to potential users. The minimum and maximum values obtained for each answer is also presented.

Question	Average	Min	Max
Q1. On average, how much time does it take to flash and create a local build given code changes?	77 (min)	1 (min)	150 (min)
Q2. Excluding the time it takes to flash and create a local build, what is an acceptable amount of time to wait for feedback?[1]	17 (min)	1 (min)	60 (min)
Q3. How many of the total amount of bugs that potentially could result from those changes would you, at a minimum, expect the tool to find? (0-100%)	63%	10%	100%
Q4. If you were to run a subset of CTS that would give feedback the same day; what percentage of times would you accept that the subset would miss one or more bugs? (0-100%)	36%	1%	75%

[1] Two answers were filtered out in the second question due to how large they were in comparison to the rest. One was 3000 minutes while the other was 240 minutes.