MASTER'S THESIS | LUND UNIVERSITY 2015

Test Selection Based on Historical Test Data

Edward Dunn Ekelund

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-20

Test Selection Based on Historical Test Data

Edward Dunn Ekelund (eddie.dunn@gmail.com)

2015

Master's thesis work carried out at Axis Communications AB.

Supervisor: Ola Söder, ola.soder@axis.com

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

This thesis is the culmination of an action research project conducted at Axis Communications, which tries to enable test selection based on historical test data. In order to realize this, a prototype program – the Difference Engine – was created, which can parse and analyze historical test data drawn from a large database. The analysis produces a weighted correlation between code packages and test packages, and this correlation can then be used to select only a recommended subset of the full regression test suite in order to minimize the time spent running tests which are not related to the code that was changed.

The evaluation results of the Difference Engine shows that its algorithm for finding correlations is robust, and running only the recommended tests is more effective than running the full regression test suite by a significant factor. On average it successfully identifies nearly all relevant tests (median success rate is 100%), and it discards 14 times more irrelevant tests given the code that has been changed compared to the default selection strategy of running all tests. Furthermore, the average size of the recommended subset is only 5% of the size of the full regression test suite, meaning that the time spent running tests can be decreased by a factor of 20.

Keywords: regression testing, test selection, test prioritization, RTS

Acknowledgements

I would like to thank my supervisor, Ola Söder, for his invaluable ideas and enthusiastic feedback, my boss, Roger Pettersson, for wanting me to succeed and giving me the support to do so, and my examiner Emelie Engström for telling me what I did not wish to hear in order to make the thesis better.

I also wish to thank Axis Communications for giving me the opportunity to finally finish what I began so long ago. It is a great place to work.

Contents

Chapter 1

Introduction

Many companies perform what is called *regression testing* in order to make sure that introducing modifications to software does not affect existing system functionality. The simplest regression testing strategy is to rerun all test cases. It is an easy strategy to implement, but often it is also unnecessarily expensive, especially if changes affect only a small part of the whole system under test (*SUT*). Therefore, applying regression test selection (*RTS*) techniques in order to only select and run a subset of the full test suite is of great advantage, as long as critical bugs are not missed.

Some existing RTS techniques are very meticulous in finding exactly which function affects which test, and they must therefore perform a deep analysis which may be very time consuming (Yoo and Harman 2012). At a company with a large codebase such a fine grained correlation of functions in the code to tests in the test suite is not necessarily needed, or even possible, if the SUT is too large and complex, involving interaction between several different components and layers; furthermore, this complex interaction is quite difficult to quantify by *static analysis* alone. Instead, a faster, high-level analysis correlating whole code packages to test packages is better suited to the task. Additionally, since the analysis will be cheaper it means it will be performed more often, leading to that it can better keep up with the transformations of an active, ever-changing codebase.

My work therefore focuses on implementing software that enables a continuous analysis of the history of changed packages and their test outcomes, which can then be used to predict the tests that need to be run when a certain *code package* is changed or updated. After the software has been implemented, the high level analysis it produces has to be evaluated in order to determine whether *test selection* based on historical data is an effective and feasible means of minimizing time spent performing regression tests.

The research was conducted at Axis Communications. Axis has a big regression test suite that is run several times daily and nightly to verify that any changes to packages do not break anything. The problem is that their large, complex codebase requires a large, complex regression test suite, and the whole test suite takes approximately seven hours to run. This effectively means that a developer has to wait a whole work day before knowing whether regressions were introduced after making any kind of change to a package, severely delaying integration of new code into the platform. Having such a slow turnaround time is obviously a huge waste of time, and makes development more difficult and costly than necessary.

The SUTs at Axis contain several hundred packages encompassing several millions of lines of code. As alluded to earlier, with so many packages, an overview connecting code packages to test packages would be more useful than mapping a single function to a single test, which is the most common way of performing RTS selection today (Yoo and Harman 2012). This can be solved with a tool that continuously connects changes to code with failing tests on a *package level*. Since we do not want to run any tests that are not historically affected by the altered packages, said tool needs to select a subset of the full regression test suite containing only tests that are historically relevant. In other words, we need software that can analyze the history of regression tests and correlate changed code with failing tests. The result of such an analysis is a model that can predict that if, for example, changes are made to packages `x`, `y` and `z`, the tests `a`, `b` and `c` have a historical risk of being affected by said change. The changed packages, the potentially affected tests, and an abstraction of the risk level (referred to as *weight* in later chapters) for each test will then be presented for a tests coordinator to assess, so she can better choose a subset of tests to run. Alternatively, the regression test system could be configured to make the selection automatically, for example running all tests that are correlated to the changed packages and which have a risk level above some specified threshold. In the end, this should ultimately enable cutting down on regression test time to a fraction of the time a full test suite run takes at Axis today, enabling a shorter verification feedback loop.

In summary, as software development today moves ever faster, fast turn around time enabling continuous integration[1] becomes ever more important. Waiting a full workday for integrating a new feature is not acceptable. If intelligently selecting and running only the necessary regression tests enables more integrations per day, Axis will benefit greatly; immense cost savings are possible, as it will enable faster development of new features, and bug fixes can be pushed out to customers earlier.

This document is structured in the following manner: The Introduction chapter is followed by a chapter discussing the background of RTS strategies. Chapter 3 expands on the background by discussing related work in the area of test selection. The fourth chapter gives a somewhat brief summary of the software written in order to perform a high level analysis on historical test data. Chapter 5 reveals how the results of the analysis achieved by the software are to be evaluated, and chapter 6 discusses the results of the evaluation. A general discussion featuring a summary

[1]See http://en.wikipedia.org/wiki/Continuous_integration. Basically, CI is a way to make software development and release more effective.

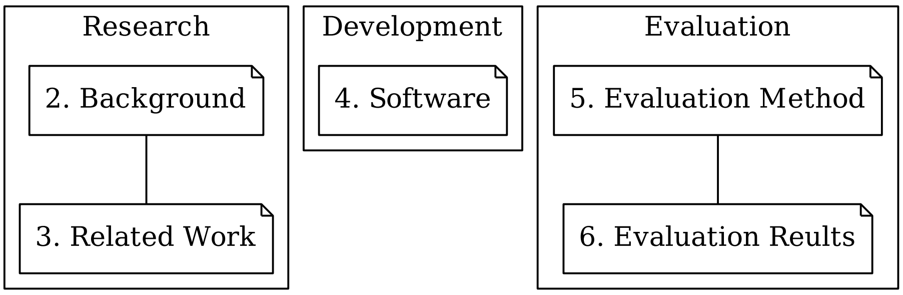

Figure 1.1: How chapters 2-6 are grouped into research, development and evaluation

and conclusion regarding the results is presented in chapter 7, as well as some future work and improvements that can be implemented to achieve better results.

Finally, chapter 8 includes a glossary which the reader can use for reference if any of the concepts discussed herein are unfamiliar. At their first occurrence in this text, all words in the glossary have been italicized to indicate that they are explained in further detail there.

The work underlying this thesis can be divided into three main components: research, development and evaluation. Figure 1.1 shows how each area relates to the chapters of this document. Most of my effort was spent on development, writing the software necessary to perform the analysis. Effort wise the evaluation part comes a close second, as it also involved writing a lot of code. In data science, a robust evaluation is necessary since results alone are useless unless we can be sure that they are also extensively tested.

Chapter 2

Background

As more and more companies leave the Waterfall development model behind and begin adopting more Agile practises, the concept of fast and continuous integration has become increasingly more important, and tests are run a lot more often than before. This means that determining how to minimize the time spent running unnecessary tests has become more important, and, consequently, it has become an increasingly more popular research topic since the early 1990's.

In a comprehensive survey of 159 papers, Yoo and Harman (2012) have analyzed trends in RTS strategies. Most strategies fit within either the *test case minimization*, *test case selection* and *test case prioritization* categories. The test case minimization strategy involves permanently minimizing the *regression test* suite in order to cut down on testing times. Test case prioritization attempts to optimally change the order in which regression tests are run so as to find faults as soon as possible. Test case selection can be seen as a middle road between the two; given a defined selection criteria only the most relevant tests in the regression test suite are selected and run, yet there is no permanent change made to the suite. As I see it, test case selection gains the benefit of less time spent on testing – as with minimization – by running only a subset of the tests, but also maintains the benefit of prioritization by not permanently excluding any tests that might turn out to be important later on.

As of late, research has shifted towards *test prioritization* (Yoo and Harman 2012). One obvious reason for this is that a correctly prioritized test suite will find faults early, but will not miss out on any later potential test faults as all tests are run eventually. This differs from test minimization or test selection which only run a subset of the whole suite, and might miss out on important faults. Another reason may be that a lot of papers published around the advent of this millennium showed that minimized test sets on average performed worse than even randomly selected sets, while requiring a significantly more expensive analysis (see e.g. Graves et al. 2001).

However, I attribute this more to the manner in which the minimized sets were achieved (using static white-box analysis of code, and no continuous re-evaluation of the analysis) than to any inherent weakness in the concept of minimizing the test sets.

As we shall see in the results chapter of this thesis, test set selection by manner of optimally selecting a smaller regression test suite is definitively a viable and effective solution for achieving a smaller regression test suite. With the context of this thesis in mind, prioritization is not needed or even recommendable – the desired outcome of my research is to minimize the *time* of the regression testing feedback loop during the day when developers are working on their code. During the night, nightly builds will ensure that the whole regression suite is run, to catch any potential regressions the minimized test selection might have missed, and to update the database with historical data that can be used in subsequent analyses for finding correlations between tests and code packages.

In their survey Yoo and Harman (2012) observe that while there is a lot of academic research on RTS techniques, application of said techniques at an industrial level remains limited. This thesis should therefore be interesting in that it is an empirical study within an industrial context. The lines of code (LOC) involved are at least *an order of magnitude*[1] larger than what has been evaluated in virtually all of the papers surveyed by Yoo and Harman, where only 14% of the papers evaluated systems of more than 100k LOC[2].

[1] An exact estimate is very difficult to make due to all the third party code in Axis products. The Linux kernel itself, even reduced and customized as it is in Axis products, consists of almost 16 million lines of code. An approximate estimate would be about 30 million LOC.

[2] The largest, being a significant outlier, contained 18 million LOC.

Chapter 3

Related Work

As I have mentioned, a lot of the RTS research is based on employing static code analysis on a functional level to perform the test case selection (Rothermel and Harrold 1997; G. Rothermel and Harrold 1996; Graves et al. 2001). Such an analysis can be slow because it goes into such detail. If an analysis is too slow it might become so costly to perform that more cost-effective results can be achieved by not employing any RTS techniques at all. This means that if the analysis is so slow that it cannot be continuously updated, it will result in analyses that quickly become outdated for codebases where people constantly modify the source code.

Some studies, e.g. Kim & Porter (2002), have tried to improve on the problem of test selection from costly static analysis by including a historical context as well as taking into consideration the fact that the tests need to be continuously re-evaluated and re-prioritized. They achieved lower predicted costs without reducing effectiveness by running only carefully selected subsets of the test suite.

However, in their work, Kim & Porter only analyze a couple of smaller programs (less than 10k LOC) on a function-based level involving only a relatively low amount of tests, which is not always sufficient for the needs of the companies who wish to streamline the regression testing process. A function based approach like theirs might also be unnecessarily granular when performed on several thousands of lines of code, where it would be sufficient to keep the analysis at the package level. Also, much of the work behind this thesis is based on the premise that the analysis should be continuously re-evaluated to provide the best test selection. It is in other words important to note that if the analysis is too granular it will also be more costly to perform.

Wikstrand et al. (2009) worked on a paper that discusses the usage of a file cache for regression test selection. If a source code file that exists in the cache is changed, all its dependent tests will be run. One drawback of their approach is the process involved to link tests to source code files; human interaction is needed since the cache is updated based on error reports that have been closed during the day. This is only an acceptable approach if there is sufficient existing infrastructure to aggregate the data in this manner, which is not the case at Axis. The idea of a cache is explored in this thesis, though this cache will hold the correlations between code and test based on an analysis of historical test data extracted from a database, and it will be fully automated, with no human interaction necessary.

While the fault prediction effectiveness was good with the method proposed by Wikstrand et al (2009), the efficiency of the regression testing based on its recommendations was not empirically evaluated. Engström, Runeson, and Wikstrand (2010) report on an empirical evaluation of test suite efficiency with the fix-cache method compared to manual selection.

They arrive at the conclusion that a fix-cache regression test selection technique is simple yet efficient, if it can make use of information that is already collected and stored in different databases. Data in a form that can easily be aggregated in the manner needed for fix-cache selection is unfortunately not available at Axis, so their approach was not applicable in my case.

In the study *Empirical Study of Regression Test Application Frequency* (Kim, Porter, and Rothermel 2005), the authors argue that while RTS technique can produce cost savings, the effectiveness varies widely with the characteristics of the workloads, where said characteristics may be program size, test suite size, change size and change distribution. Notably, they conclude that the frequency of regression testing has a strong effect on the behavior of the RTS techniques.

The effectiveness of minimization RTS techniques increased as the number of changes increased, given that the testing interval was frequent. Infrequent testing yielded high minimized test numbers, since many changes were made to the code between regression test runs.

Their results concluded that random selection worked very well and the most minimized selection performed the worst of all their tested selection models. As this is virtually the opposite of my findings based on the work behind this thesis, one should consider that our approaches to test selection are quite different: They based selection not on test result history but on static source code analysis. With only one exception, their analysis utilized very small programs and small test suites of less than 1000 LOC. None of the programs contained more than 10k lines of code.

These differences are telling because one of the improvements they suggest is to extend the study to larger programs with more tests and a wider variety of naturally occurring faults. Another suggested improvement is to explore techniques that incorporate previous testing history. As I have mentioned, many previous papers I have studied seemed to have analyzed smaller programs, and smaller test suites, basing the test selection on static analysis of the code packages. Though my test selection techniques differ from theirs, this thesis fulfills both of the suggested improvements, as my analysis uses an abundance historical test data drawn from regression testing on a much larger codebase.

Finally, there is also some interesting research focusing on improving the test selection by adding more intelligence to the selection algorithms. In their paper, Rees et al. (2001) argue on the benefits of employing Bayesian graphical models (BGMs) for more effective test selection. Once a BGM has been constructed it can be used for optimal reduction of the probabilities of failures for a given number of tests. Alternatively, a BGM can be used to analyze how good a given subset of a test suite is likely to be, and indicate whether or not there are particular test cases which could be deleted/replaced by tests that reveal more about the quality of the software. Unfortunately the process was not fully automated and it was inconclusive whether creating a BGM is suitable and/or justifiable for all testing situations. While I believe utilizing Bayesian statistical reasoning is a very robust way of incrementally building a prediction model for how tests and code are correlated, I decided that in the early stages it would be unnecessary to implement. I was content to keep such a possible future development in mind, if the selection produced by a simpler algorithm would prove to be too inadequate.

Huang *et al.* have written about cost cognizant history based test case prioritization techniques (Y.-C. Huang, Peng, and Huang 2012). Their technique employs what they call a genetic algorithm to analyze historical data on regression test runs to produce a test suite ordered by level of importance; test cases which have poor historical records are given a chance for a higher priority. Their results show that their technique improves the effectiveness significantly in comparison with three coverage analyzing techniques. They also conclude that prioritizing test cases based on historical information can provide great effectiveness.

While adding greater intelligence to the test selection algorithm is an interesting prospect, I eventually decided against it, reasoning that it is better to keep test history analysis and test case selection as simple as possible in order to make introspection of results, maintenance, and possible future extension of the algorithm as easy as possible. I figured that if the basic algorithm would then prove to be wildly ineffective I could always reconsider and perhaps explore more exotic algorithms.

The basic algorithm did not prove to be wildly ineffective. In fact, it worked quite well from the start, as we will see in the later chapters.

Chapter 4

Software

Developing the software enabling the correlation analysis is where I put down most of the work and effort behind this thesis – by a large margin – yet this chapter will provide little more than a cursory overview, as discussing the implementation details, bugs, and the difficulty of correctly *wrangling* the vast store of data would considerably shift the scope of this thesis. In addition to the analysis software presented below, a lot of code was written in order to perform the evaluation of the analysis. This code will not be discussed here at all, since, while it helps lend credence to the RTS strategy suggested herein, it is not directly affiliated with the goal of this thesis as such: to provide a means to use historical regression test data to perform a more effective test selection in future regression test runs. For the curious, however, most of the code is published and available on Github at https://github.com/eddie-dunn/test-selection.

All code was written in the language Python[1], which is a very flexible and clear language with many readily available machine learning libraries and diagram generating utilities[2]. The code was written with the functional programming paradigm in mind, in order to produce very few side effects and make *unit testing* easier; currently the core code modules have nearly 100% unit test coverage, and should therefore be easy to modify and improve should the need arise. Some object oriented features are introduced, though, in order to gather functions into logical units in order to enable composability and ease of use.

The analysis software is split up into three main modules: The Difference Engine, providing the actual analysis of the correlations, and two auxiliaries, Seeker, the

[1]https://www.python.org/

[2]The library `sklearn`, for example, provides a flexible module for running the *K-fold cross validation* technique presented in the Evaluation chapter, and the library `matplotlib` has been used to generate all the diagrams present in this thesis.

program that extracts the historical test data, and the Simulatron, a program that generates simulated historical test data which can be used to ascertain that the analysis performed by the Difference Engine is plausible. All three programs were developed in parallel, and iteratively, where the results of improving Seeker or Simulatron could be used to improve the analysis of the Difference Engine.

In summary, we have the following software

- The Difference Engine. What is to be considered the end result of this project; the code that generates the actual correlations between tests and code packages
- Seeker. The application that communicates with the MySQL database containing the real world historical regression test data, parses the result, and passes it on to the Difference Engine.
- Simulatron. Generates a simulated regression test history that the Difference Engine can analyze

For a graphical representation see Figure 4.1.

4.1 The Difference Engine

Simply put, the Difference Engine takes a set of *Builds* as input, and gives a weighted correlation between code packages and test cases as output. Each Build is extracted from the data generated by either Seeker or Simulatron, containing packages and their revisions, and the regression tests and their result. Once a given set of Builds have been analyzed the cumulative correlations can then, in future regression test runs, be used in order to determine which tests to run given changes to one or several packages.

4.1.1 The Difference Engine Algorithm

The Difference Engine takes data generated by either the Simulatron or Seeker, parses it, and calculates the correlations between tests and code packages. Since the raw historical data in the database does not contain information of changed packages or flipped tests, this must be calculated on the fly when performing the analysis, by comparing code packages and tests with those of the previous test run.

The correlation is calculated using a simple algorithm, namely:

- From the raw data containing historical regression test runs, extract packages and test results, building an intermediary data structure containing only the test run results that are of interest for the analysis; i.e. packages and their revisions, and regression tests and their outcome.

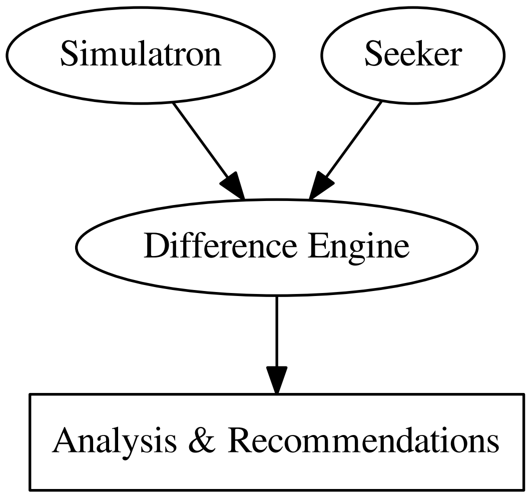

Figure 4.1: Graph describing program dependency chain

- Given this set of test run results, go through every regression test run (aka *Build*), and find each *code package* that was changed for that particular test run, by comparing the revisions with the previous test run.
- For every package that was changed, look at each test outcome for the same test run and compare it with said test's outcome in the previous test run.
- If the test result has *"flipped"*, meaning that it goes from pass to fail or from fail to pass, add one to the correlation coefficient between the changed package and the test, since changes to said package might have affected the outcome of the test.
- Iterate over the whole set of Builds, repeating the process described above, building a correlation database of code packages and the tests that are dependent on them.

Or, in pseudo code:

```
For each <build_result> in <set>:
    For each <code_package> in <build_result>:
        If <code_package> is changed:
            For each <test> in <build_result>:
                If <test> flipped:
                    correlation_coefficient += 1
```

This algorithm thus attempts to map package changes to changes in test outcomes, i.e. when tests “flip”.

4.1.2 Difference Engine example

The following example attempts to illustrate how the correlation weights indicate that a test outcome is dependent on changes to a code package. The example shows the results produced when running the Difference Engine analysis on simulated data generated by the Simulatron.

Input:

```
$ ~/difference_engine regression_test_data.db
```

Output:

```
Analyzing data...

Results:

code/package3.py
  package3_test: 19
  package2_test: 2
code/package94.py
  package94_test: 8
  package3_test:  3
  package17_test: 1
```

Here we see that `package3_test` has a correlation weight of 19 to `package3`. This means that a change to `package3` that constitutes a new bug or a bug fix correlates very strongly to a change (flip) in the test outcome for this test. Since we are using simulated data we can tell from the name that this test is in fact dependent on `package3` and that this strong correlation is quite correct. If this analysis had been performed on real data, it would then be prudent to run `package3_test` every time a `package3` is changed.

The correlation weight between `package94` and the test for `package3` is interesting, since it means that three times when this test flipped `package94` was changed as well. The correlation is totally random however, and is just due to the *noise* that the Simulatron adds when generating simulated regression test data. In reality `package94` just happened to be changed three times at the same time as a bug was either fixed or added to `package3`.

Additionally, there are a *lot* of packages with correlation weights of 1, but most have been removed from the heavily pruned example output above. A weight of 1 is too low to be able to distinguish true correlations from random noise. That is, if two packages are changed and only one test flips, both packages will have their correlation weight increased by one, yet only one of the changes caused the flip. It isn't until we start to see a trend – that is, weights of 2 or more – that we can be more sure of the correlations between the code packages and test packages. It is easy to intuit that the likelhood that a test and package are truly correlated increases exponentially with the correlation weight, since each increment to the weight makes it ever more likely that the correlation is not just due to random noise.

4.2 Auxiliaries – Seeker and Simulatron

4.2.1 Seeker

The purpose of Seeker is to communicate with the database containing the historical regression test data, extract the relevant records (e.g. based on a specific Axis product and its package configuration), and organize them into a format that is parsable by the Difference Engine.

The test history at Axis resides in a vast MySQL database called *DBDiffer*. In this database all sorts of regression test data is kept, and among this data one can extract which packages were changed before the regression test suite was run, and which tests failed and succeeded. DBDiffer contains everything we need in order to find the correlation between code packages and tests, but the data in the MySQL database exists in several tables which are loosely linked to each other by MySQL `ids`, which serve as unique identifiers. Therefore, one problem with working with the database is that while all the necessary data is available, it has to be parsed, analyzed and aggregated into a format that the Difference Engine understands. Builds only exist conceptually; in order to find which modules where changed in a build, and which test cases flipped, packages and tests first have to be linked to the build in question.

When parsing the database, Seeker needs to perform two actions:

1. Extract all code packages and their revisions and map them to their unique MySQL `Build` identifier.
2. Extract the statuses and names from all testcases and map them to their unique MySQL `Build` identifier.

After this, Seeker joins all code packages and testcases based on their unique identifiers to create the *Build* data structures that the Difference Engine uses in its analysis. All subsequent Builds are then added to an ordered list to form a *BuildSet*, which is the data metastructure passed on to the Difference Engine for analysis.

As mentioned, the database contains no explicit information on which packages where changed in a given build. As a consequence of this, the changed packages need to be somewhat awkwardly calculated: In an intermediary step, the Difference Engine iterates through the BuildSet generated by Seeker, and for each Build looks at the packages and their revisions in the previous build in order to generate a diff between them and the packages and revisions in the current build. The flipped tests are calculated in a similar manner – by comparing the previous build with the current – and the changed packages and flipped tests are then put in a new list that is used for the actual correlation analysis.

4.2.2 Simulatron

Since the database containing the historical test results is is gargantuan – around 100 gigabytes – and the code base contains several millions of lines of code, it is impossible to intuitively determine whether the correlations found by the Difference Engine are correct; how do we know whether the correlations are due to real dependencies between tests and code packages, or if they are due to random noise, or if they may be due to errors in the correlation algorithm?

Therefore, in order to be able to assess whether the Difference Engine produces reasonable results, an important auxiliary program – the Simulatron – was developed. The Simulatron is used to create a fake regression test history where packages and tests are named so that it is immediately apparent whether they should be correlated or not. In other words, using data from the Simulatron makes it obvious if the Difference Engine finds erroneous correlations, something which is not the case when analyzing and correlating real historical test data. In their survey Yoo and Harman (2012) argue that a realistic simulation may provide an efficient manner in which to assess RTS selection strategies, which lends credence to the idea of using simulated data to evaluate the Difference Engine.

The Simulatron is used to generate

- a variable number of code packages
- tests for each package
- associated test flips when changing a package
- noise, i.e. flipped tests with no corresponding package change, or changed packages with no corresponding test flip
- a fake regression test history containing the packages, tests and noise specified above.

Noise in this context is Simulatron adding random changes to packages, or random flips to tests, in order to simulate the noise present in the real historical data extracted from the database. Noise in the real data is not entirely random however, and may be due to any number of reasons. For example, with real historical data, 2 packages are, in average, changed per build. This means that any potential test flips that occur will correlate both code packages to the tests, even if only changes in one package is responsible for the flip. In practice, this means that one of the code packages will be falsely correlated to the test, and we have “noise”. One other possible source of noise is when something outside of the control of the regression test suite – for example a network error – causes tests to flip.

Simulatron uses package and test names such that you can immediately tell which test should fail due to a bug in a package. For example, with real data, packages are named similarly to `os/boot`, `os/linux` and `apps/storage_manager`, and tests `test_image`, `test_recording`, and `test_syslog`. If the Difference Engine shows that `test_image` is correlated to `os/boot`, there is no way to immediately assess the validity of this correlation. In contrast, the Simulatron packages are named `package1`, `package2`, `package3` etc., and tests are named `package1_test`, `package2_test`. Later on, in the analysis stage, this very simple but explicit mapping can be used to determine whether the predicted correlations are correct or not.

A great advantage of having the Simulatron is that if the correlation algorithm used by the Difference Engine proves to be inaccurate or inadequate, a modified or entirely different algorithm can be quickly tested and evaluated easily with simulated data from the Simulatron. Another advantage is that fake test history can easily be generated by the Simulatron to test different variations in input data, to see in what manner such variations affect the performance of the selection algorithm. We will see some examples of this in the Evaluation chapters.

4.3 Iterative development process

Neither of the three main modules were developed in a vacuum. Rather, they were implemented in parallel, where improved results from one of the programs would be used to modify the code of the other two in turn, improving their results. Developing the software in such a manner leads to iterative, incremental and robust improvements of the code, though it is necessary to realize when "enough is enough", and decide it is time to cease development and improvement efforts. Deciding when to stop sounds easier than it may seem. For example, while the data produced by the Simulatron is designed to be similar to the real data extracted by Seeker, there are still some differences; Simulatron generates much more dense data than Seeker, meaning that changes (changed code, flipped tests) are guaranteed to happen for every Build, though this is not the case with real historical regression test data, where large portions of the data have either no changed packages or flipped tests and therefore cannot be used for analysis. The denseness of the simulated data is a conscious choice since – as we shall see in the Results chapter – it can reveal long term trends not immediately apparent when analyzing real historical data, though this also means that there is a risk of missing some vital characteristic that will only be apparent when real data is used.

In the end, it is important to realize that, while I spent a considerable amount of time on it, none of the code is perfect and can therefore still be improved. In its current state, the data that is generated, extracted or analyzed will give results that can be used to draw significant conclusions, which is enough for the scope of this thesis. It is not, however, production ready, and will need further refinement before it can be used in production at Axis for example.

Chapter 5

Evaluation Methodology

This chapter presents how the results of the high level analysis produced by the Difference Engine will be evaluated. The F-measure is introduced as a tool with which the accuracy and effectiveness of the test selection can be compared between different RTS strategies. The three RTS strategies that I will compare are then presented, and finally it is explained how the Difference Engine will be evaluated based on whether its algorithm can be trusted to find correlations, how effective its suggested tests are at finding test flips, and whether any time is saved by running only the suggested tests.

5.1 F-measure

The F-measure is the tool I have used to evaluate the accuracy of the test selection.

The goal of the test selection, apart from saving time, is to be sure that the tests that *are* run are relevant tests. A relevant test is a test that will reveal a flip in the changed code. The trivial method of finding all possible flips – achieving the highest *recall* – is to run all tests, but then accuracy is low. Accuracy can be high if the only tests we suggest are fault revealing tests, that is, we say that the selection has high *precision*. However, even if all selected tests include flips, we might miss some flips since we cannot guarantee 100% recall. Both precision and recall are therefore important for test selection, and the *F-measure* (Witten and Frank 2005, Wikipedia (2014)) is used to calculate a singular measurement statistic based on the ratio of precision and recall. In other words, the F-measure is a way to compress two dimensions of selection quality – recall and precision – into one dimension, in order to make direct comparisons between different selection models easier.

- *Precision* is defined as the fraction of selected tests that are relevant to a particular build. The precision score ranges between `0.0` and `1.0`. A precision of 1.0 means that 100% of the selected tests are included in the set of relevant tests.

$$precision = \frac{|\{relevant\ tests\} \cap \{selected\ tests\}|}{|\{selected\ tests\}|} \tag{5.1}$$

- *Recall* is defined as the fraction of relevant tests that are included in the presented selected tests. The recall score ranges between `0.0` and `1.0`. A recall of 1.0 means that 100% of the relevant tests are included in the set of selected tests.

$$recall = \frac{|\{relevant\ tests\} \cap \{selected\ tests\}|}{|\{relevant\ tests\}|} \tag{5.2}$$

- *relevant* is defined in this context as a test that has flipped.
- *selected* is defined in this context as a test suggested by the Difference Engine.
- The *F-measure*, or balanced F-score, is the harmonic mean[1] of precision and recall and is defined as

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.3}$$

- Just like with precision and recall, the F-measure lies in the range `[0.0, 1.0]`. `1.0` is a perfect score, meaning that both precision and recall are perfect as well.

Remember these definitions (or bookmark this page), because intimately understanding the definitions of recall, precision and F-measure are necessary in order to understand the discussion in the succeeding sections.

5.2 Test selection strategies: *All*, *wide* & *narrow*

The evaluated test selection strategies are *all*, *wide*, and *narrow*. In spite of being a successful selection strategy in other papers (See the Background and Related Work chapters), randomized test selection is not a chosen strategy for my evaluation; the reasoning behind this is explained in the Evaluation Results chapter.

[1]Compared to the arithmetic mean, the harmonic mean is an average value that is less affected by a small amount of extreme outliers.

All No test selection is performed; run all tests.

Wide selection Select all tests that have ever flipped, regardless of which packages have been changed. Can be useful if the narrow selection method returns very few suggested tests, or none.

Narrow selection Select tests based on which packages have been changed for a particular build. Some packages will have no correlated tests, which is when it might be useful to fallback to the wide test selection strategy.

5.3 Evaluation procedure overview

The evaluation attempts to answer the following questions:

1. **Correctness**: Are the test correlations found correct?
2. **More builds, better F-measure**: Does the F-measure get better when more builds are available for analysis?
3. **Flip-finding**: Will running the tests suggested by the Difference Engine find all the flipped tests?
4. **Time**: What is the time savings compared to running the full regression test suite?

Each point is discussed in further detail below.

5.4 Correctness

Are the correlations correct?

A "correct" correlation in this context is a correlation between a code package and a test that we know must be true. For example, if we add a bug to `package1`, we might know that `package1_test` must fail. This is basically what the RTS strategies mentioned in the Related Work chapter achieve through static analysis of code and tests.

Unfortunately, at Axis it is virtually impossible to know whether the correlations found are correct when running the analysis on real data as there is no available mapping between code packages and tests. And, as mentioned earlier, the codebase at Axis is so large and complex that achieving such a mapping through static analysis is impractical, if not impossible. Therefore the evaluation of the correctness of the correlations will be performed on simulated data produced by Simulatron, which produces results where the correctness of the correlations are known. With data generated by Simulatron we know whether the correlations found are valid since it is immediately apparent from the names of the code packages and test names.

The evaluation of the test selection algorithm's ability to find correct correlations will be performed by determining whether the most highly weighted correlations are true correlations (i.e. that the correlations are not due to package changes and test flips that coincide incidentally). To evaluate this we will look at the result of the finished analysis on simulated data, going through each package, and checking whether their most highly correlated test is also the one we know is associated with said package. A 1 will represent this being true, a 0 will represent this being false. The resulting ones and zeroes will be added to a list from which the mean value can be calculated.

Simulatron parameters:

- 75% package noise and 30% test noise, for a variable number of builds.
  - This means that in 75% of the builds a random package will be changed, but its corresponding test wont flip, and in 30% of all builds a test will flip but its corresponding package will not be changed.
- 99% package noise and 99% test noise, for a variable number of builds.
- 500, 1000 and 10 000 builds.
- A pool of 100 packages to change.
- A pool of 100 tests to flip.

75% package noise and 30% test noise were chosen because a cursory inspection and analysis of the historical test data show that these numbers provide fake data that resembles the real data, approximately. Fake data with 99% package noise and 99% test noise is more noisy than the real data, but is included to see how the Difference Engine performs with data that is noisier than the real data.

It should be noted that the Simulatron generates much more dense data than the true historical data extracted with Seeker. That is to say, many of the builds that Seeker extracts are *blank builds*, meaning that either no packages have been changed or no tests have flipped since the preceding build. Such builds cannot be used for an analysis, and are therefore discarded. With data generated by the Simulatron, however, all builds are guaranteed to have at least one package changed, and one test flipped.

5.5 More Builds, better F-measure

Does the F-measure get better when more builds are analyzed?

Early prototyping work during the iterative development of the code showed that one weakness of the Difference Engine is that it does not work well if the BuildSet it analyzes contains too few Builds, which is not very surprising. The minimal number of Builds necessary to provide a reasonable analysis is highly dependent on the input data, and which packages you are interested in finding test correlations for. For example, if you look for correlations for one specific package, you may be very lucky and the package will have been changed often in many recent builds, and therefore have many relevant and/or high correlations with tests, or very unlucky and the package will never have changed in the BuildSet you analyzed, and will therefore have no correlations.

If the Difference Engine does not work well with too few Builds, then what about the opposite? With an increasing number of Builds, we can hypothesize that the F-measure must increase and become better as well, as with enough Builds to analyze the recall must be near a 100%.

This evaluation will therefore compare the F-measures for different sizes of build sets. The range of builds will be 10, 100, 500, 1k, 2.5k and 10k builds. The Simulatron will be instructed to randomly change a code package and its corresponding test, creating a *Build*. First, ten Builds will be created in this manner, joined together as a *BuildSet*, and then the BuildSet will be analyzed by the Difference Engine. With the data from the analysis the median F-measure for the whole BuildSet can be calculated. Then I will increase the number of generated Builds for each corresponding BuildSet; a hundred, a thousand, and so on, Builds will be generated and joined into a BuildSet, and the average F-measure calculated for each.

The Simulatron parameters will be the following:

- 75% package noise, meaning that 75% of the builds will contain changed packages with no corresponding test flips.
- 30% test noise, meaning that 30% of the builds will contain flipped tests with no corresponding package changes.
- A pool of 100 code packages to change.
- A pool of 100 test packages to flip.
- 10, 100, 500, 1000, 2500 and 10 000 builds.

From this data a graph can be produced that shows a better F-measure as the number of builds increase. 10 builds is really not enough to get a good F-measure, 100 is a better, and hopefully at 1000 the F-measure will be quite high. At 10 000 it should be clear that the correct tests are suggested for the code package that is changed, as recall must be nearly 1.0, i.e. close to 100%.

5.6 Flip-finding

Will the Difference Engine find the actual *test flips?*

As you may recall, the technique of finding flips is by passing a sanitized data structure of sequential builds to the Difference Engine, whereupon its algorithm finds package changes and test flips to build a correlation database. These correlations are then used to suggest which test should be run when given a set of changed packages. Even if the Difference Engine performs well with simulated data in the two prior evaluation steps, the question is if these suggestions can be used to accurately predict the flips that actually occur on real data?

To evaluate this, I will first use the regression test database to compile the correlation database. Then, in a second pass, I will

1. go through each build in the regression test database,
2. see which packages were changed,
3. query the correlation database for suggested tests to run, given the changed packages,
4. check which tests actually flipped,
5. and finally use the F-score to measure how well the suggested tests predicted the actual outcome.

In other words, I will be using the historical builds – where the test outcomes are already known – to test the prediction model. But the prediction model itself has been created by analyzing the historical builds, which presents a problem since I am basically training and validating the Difference Engine by using the same data. If the training data and the validation data is the same, the Difference Engine will of course perform quite well – it is similar to taking an exam where the only studying material is the exact same exam you will be taking, with the exact same questions! You will obviously become quite adept at passing that particular exam. If you are then subjected to a new exam with different but related questions, you will most likely *not* pass however, since you have only memorized each particular solution to each particular problem, but have no concept of applying a generalized solution to a generalized question.

In data science this problem is referred to as overfitting your model to the training data. Your model is overtrained to solve particular problems, and therefore will not be as effective in the general case. The solution to this problem is to split the data into a training set and a validation set, see Figure 5.1. The prediction model can then be then tested on the validation set after being trained on the training set, without risking overfitting. Unfortunately, while splitting the data into two sets like this may solve the overfitting problem, it presents the risk that some vital information exists only in the validation set. Since this vital information is not present in the training set, there is no way for the prediction model to make a correct prediction concerning that particular data.

To make sure that this does not happen, we can split the data several times, with a different training set and validation set each time. A variant of this method is called K-fold cross validation (Witten and Frank 2005), where you split the data `K` times, into `K` different parts. In my evaluation I have `K=10`, so the data will be split into 10 different parts, where 9 parts are used for training and 1 part for validation. Each of the ten parts is used as validation once, while the remaining comprise the training set. See Figure 5.2 for an explanation of the 3 K-fold variant.

For each of the 10 validation sets the F-measure will be calculated, and then I will take the average of all these F-measures to arrive at a final F-measure for the whole set of analyzed data. The F-measure as well as the individual average recall and precision values will then be shown in appropriate diagrams.

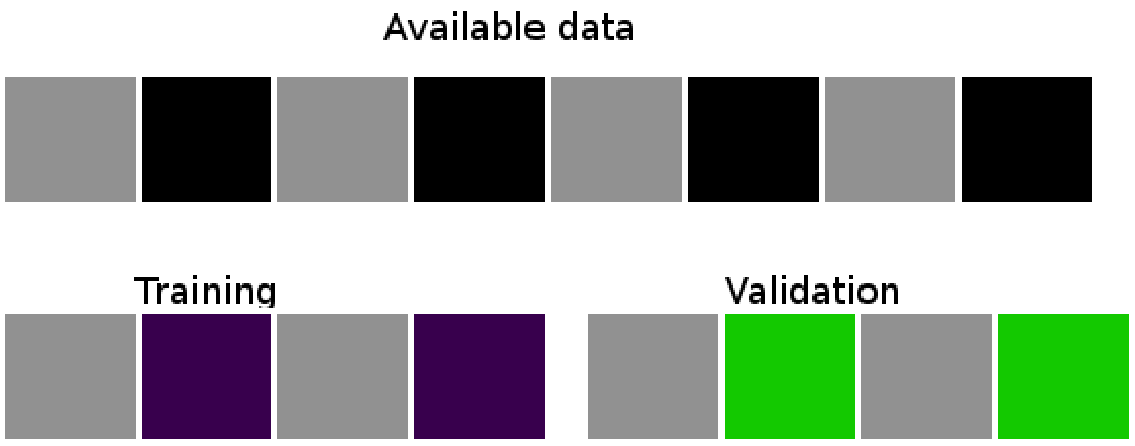

Figure 5.1: To avoid overfitting the data can be split into a training set and a validation set. There is a risk that the validation set may contain important information missing in the training set, however.

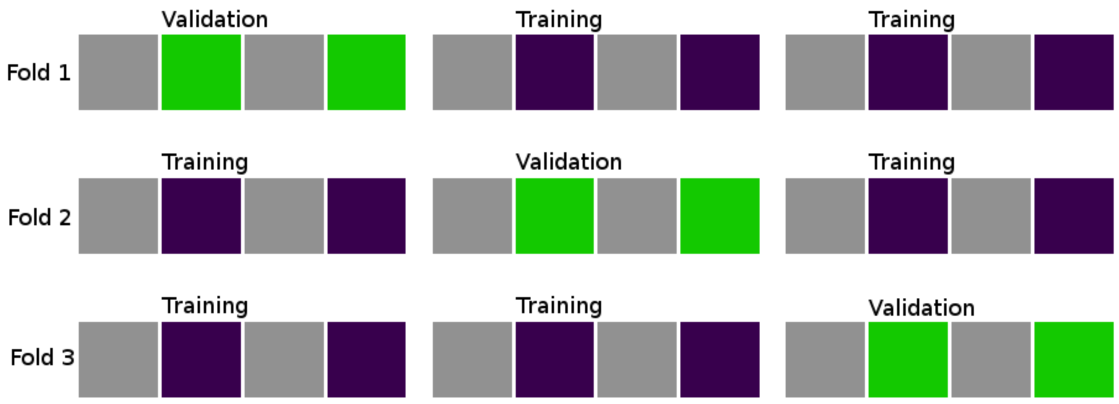

Figure 5.2: A 3 K-fold cross validation example. For each fold two sets are selected as training set and one set is selected as the validation set. All sets take turn acting as the validation set. 10 K-fold is the same, except that it splits the data into 10 folds instead of 3, meaning that for each of the 10 folds 1/10th of the whole data set is used for validation, instead of 1/3rd.

5.7 Time

Is time saved by running the selected tests?

This can be a hard question to answer, as 'time saved' can be applied in a considerably general sense. For example, if factoring in the time spent on working on this thesis, when does the cost of developing a test selection program pay off in terms of saved time?

Or, in the case that a lot of time is saved at the cost of missing some crucial fault-revealing test, how much time will it take to find the error and adjust the code?

For the sake of simplicity I will evaluate the time saved by comparing the average time it takes to run the selected tests to the time it takes to run the full regression test suite. As stated earlier in this document, the full regression test suite takes seven hours. Simply stated, for the sake of this evaluation, if `analysis_time + subset_test_run_time < full_test_run_time`, time is saved[2].

If time is saved it is of course also interesting to see numbers of approximate time savings running only the selected tests. This data will be presented in the results section.

[2]Optionally, see Leung and White (1991) for an especially rigorous discussion of test cost models.

Chapter 6

Evaluation Results

In this chapter the results of the evaluation are presented and discussed, based on the criteria of how trustworthy the correlations found by the Difference Engine are, its ability to suggest the tests that will find flips, and whether any time can be saved by running only the tests suggested by the Difference Engine.

The first two evaluation tests, Correctness and More Builds, better F-measure are performed on simulated data, and the following two, Flip-finding and Time are performed on real historical test data.

6.1 Correctness

Are the correlations correct?

This section attempts to answer the question whether we can trust that the majority of correlations found by the Difference Engine are true correlations, or if they are more likely to be due to noise.

The results in Table 6.1 show that the correlation algorithm is very robust. Even when there is both package and test noise in 99% of the builds, the correct correlation is still weighted the highest in almost all cases. The more builds that are analyzed, the better the algorithm will perform in mapping the correct test to their corresponding package.

6.2 More Builds, better F-measure

Does the F-measure improve when more builds are analyzed?

Nbr builds	Pkg noise	Test noise	Mean	Median
500	75	30	0.96	1.0
500	99	99	0.88	1.0
1000	75	30	0.99	1.0
1000	99	99	0.99	1.0
10k	75	30	1.0	1.0
10k	99	99	1.0	1.0

Table 6.1: The mean and median of correct correlations weighted the highest for different values of noise and number of builds.

First, 10 builds is generally not enough to produce any meaningful test selection analysis. Most often there is not enough data to perform a narrow test selection. Even though 10 builds were suggested to be part of the evaluation, they are not shown in the graph in Figure 6.1, since the results where too random and fluctuating to be of much use.

Second, due to the denser data some trends are significantly accelerated when analyzing the simulated data compared to when analyzing real historical data. This should be considered given the results presented below, where accumulated noise will quickly deteriorate the F-measure value as the number of builds increase. It will not happen as quickly when running the analysis on real data.

In the diagram in Figure 6.1 the F-measure appears to reach a maximum at around 100 builds, before deteriorating. The Difference Engine did not perform as predicted in the Evaluation section (i.e. that more builds will lead to better F-measure values), which in hindsight should be quite obvious; given enough builds, the accumulated noise quickly overwhelms the true correlation data.

Discarding the case when there are not enough builds to perform an analysis and provide a recommended test selection, the narrow selection continuously outperforms running all tests, until the amount of noise is too overwhelming. At that stage, at 10k analyzed builds, it performs as bad as running all tests, since it will basically recommend running all tests.

This can be observed in Figure 6.1 where at 10k builds the F-measure is almost the same as the average F-measure for performing no test selection, which is 0.031 regardless of number of builds. I believe this owes a lot to the fact that at the default settings, the Difference Engine performs no filtering at all, and even correlations of weight 1 are included in the selection of recommended tests. A 1-weight correlation is just as likely to be random noise as a true correlation[1]. Thus, as the number of builds increase, more noise is added to the set of recommended tests, until finally, there is virtually no difference between the set of selected tests and the set of all tests.

The solution to this problem is to introduce an intelligent way of culling the noise.

[1]See the Simulatron section in the Software chapter.

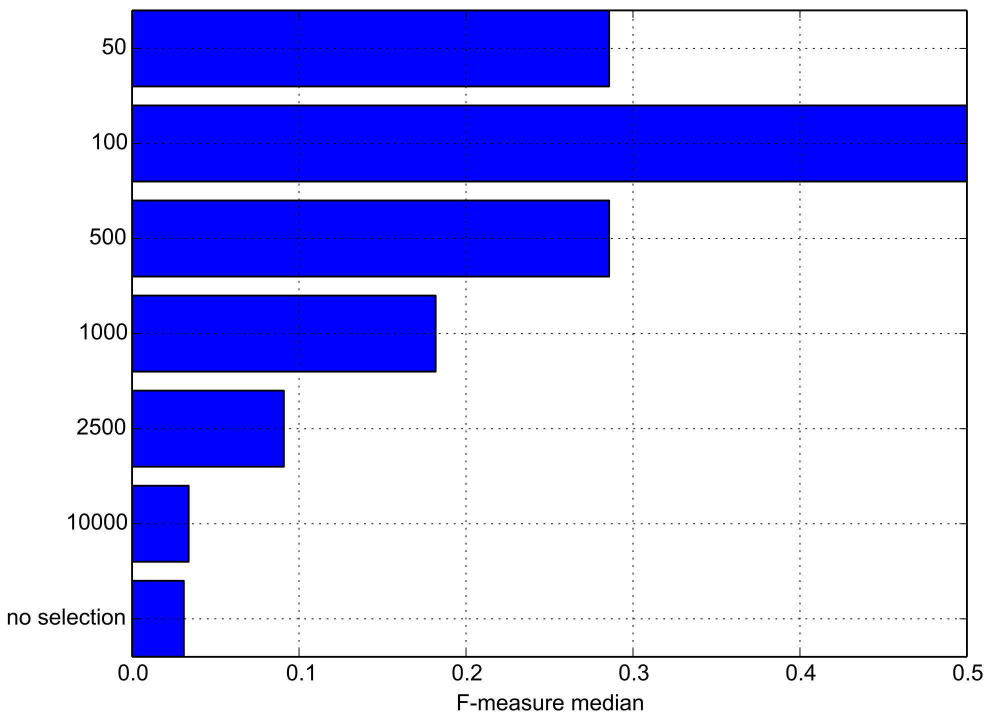

Figure 6.1: Diagram showing F-measure for 50, 100, 500, 1k, 2.5k and 10k builds, using the narrow test selection strategy. The F-measure for 10 builds is not included since only 10 builds generally do not provide enough data for the Difference Engine to construct a reliable pkg/test correlation table. The median F-measure for no test selection is also shown, for reference.

Currently the Difference Engine supports a primitive filter for discarding all test recommendations beneath a specified threshold weight. Setting this threshold at 1 (highly recommended, for the reasons stated above), will keep the narrow test selection superior to running all tests longer, but it will eventually be overwhelmed as well. Consequently, I propose that a much needed future improvement to the Difference Engine would be introducing a form of memory to the analysis: As more builds are added to the analyzed pool, the weights should be decreased for tests that have not flipped for some amount of time. For example, if a test has not flipped together with the change of a package after, say, 1000 builds, its correlation weight should be decreased by 1.

Finally, observing Figure 6.1, counterintuitively it seems that after a 100 builds, fewer analyzed builds give better results. The reason is that the median value for recall maxes out at 1.0 already at 500 builds, and cannot go any higher after that, while the precision continues dropping as more noise is added to the results of the analyzed builds. All in all, the results of this part of the evaluation provide compelling reasons for introducing a strategy for handling the accumulation of test noise.

6.3 Flip-finding

Will the Difference Engine find actual test flips?

The figures in this section display the recall, precision and F-measure of the different test selection strategies, answering the question of how effective the Difference Engine is when used on actual historical test data.

In summary, yes, the Difference Engine finds the actual test flips, and it does so much more effectively than running all tests.

6.3.1 Randomized test selection

In many of the related studies random test selection was shown to be a very effective RTS strategy. Because of this, I was planning on adding diagrams for selecting random tests, but I eventually decided against it since the analytical results I calculated were so abysmal that I felt there was little point in considering it.

A randomized choice based on the average number of tests that flip per build (2), would yield an average recall value of of `avg_number_flipped/size_of_test_pool` = `2/1265`, which is a far cry from the results obtained by any of the other selection methods, all, wide or narrow.

Another method of selecting random tests would be selecting as many tests as the number of packages that were changed for the build, but that would only improve the average recall ratio slightly, as no more than 10 packages are generally changed per build, and it would still never reach above `0.8` which is the lowest mean recall for the other selection methods.

Finally, in order to try to achieve better recall from a random RTS strategy, one could randomly choose as many tests as the wide selection method, which is approximately a third of the size of the test pool. However, this would in average never yield better results than those obtained by either the wide or narrow selection model; assuming that the test flips are evenly distributed across the test pool (I honestly doubt they are, though), randomly choosing 1/3rd of the test will give an average recall of 0.33 which is way lower than any of the other RTS strategies. Regarding precision, the median number of tests suggested by the narrow RTS strategy is 25. Since the average number of test flips per build is 2, it is impossible to achieve the same precision as the narrow strategy; the average narrow precision is 2/25 while the average precision using this version of random selection would be 2/400. Since both the recall and precision for this random RTS strategy is lower, the F-measure will be lower as well.

A randomized RTS strategy also has the disadvantage that two consecutive test runs are likely to look entirely different. For various reasons engineers at Axis often want to compare the results of the newest regression test run with older versions. When the selection of tests is totally random such a comparison will not be possible since the selection of tests is likely to differ too much between different versions.

The only benefit of random selection that I can think of is that it is cheap computation-wise, as all it needs to do is randomly draw some tests from the test pool. Therefore evaluating a randomized selection model could still be interesting if it weren't for the fact that the analysis performed by the Difference Engine is also very fast; it finishes in mere seconds. As it stands, I decided performing an evaluation on the results of a random selection was not worth the effort.

6.3.2 Recall for test flips on real data

Figure 6.2 displays the recall of the different test selection strategies.

As expected the average recall value for running all tests is the highest, since running all tests guarantees that you will not inadvertently miss out on any relevant tests. The wide test selection places second, and the narrow test selection places last, also according to expectations, since they run fewer tests in turn. While the narrow selection might appear to have 20% worse recall than selecting all tests, this number is somewhat pessimistically skewed as the (arithmetic) mean is sensitive to extreme outliers. It is very likely that sometimes the narrow selection has a recall of 0, which will drag the average recall down a lot. But because the mean is sensitive to aberrations, it can be a useful indicator of outliers when used in conjunction with the median. Therefore it is important to consider both the mean and the median values when looking how effective a test selection strategy is in practice.

In spite of the lower mean recall, it is encouraging to note that the median values for all selection strategies is 1.0, even for the narrow strategy.

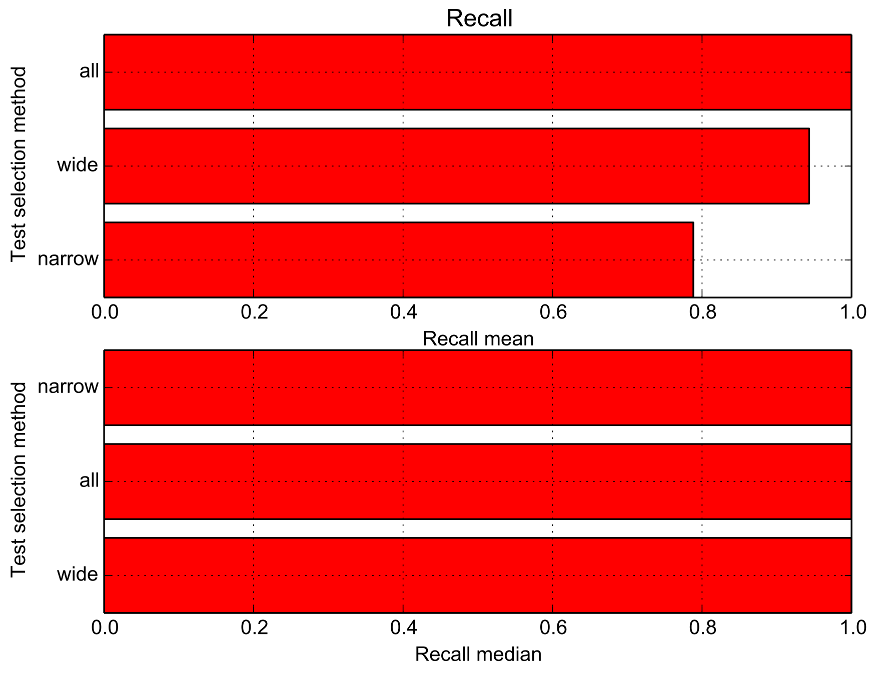

Figure 6.2: Bar chart showing the mean and median values for recall using different test selection strategies

6.3.3 Precision for test flips on real data

Figure 6.3 displays the precision of the different test selection strategies.

Now it gets interesting; the more specific the test selection strategy, the greater the precision. In other words, the most specific test selection strategy (*narrow* in the chart), yields the best results.

While the absolute values of the precision appear very low, the relative ratio between no test selection and test selection is the important factor. Greater precision means less time wasted on running irrelevant tests. Less time wasted is the final goal of performing the test selection in the context of the work behind this thesis.

Also, it is important to keep in mind that the test selection is only a *recommendation* based on historical data. Just because a (correct) correlation has been found between a code package and a test, there is no guarantee that said test will flip just because that package is changed. Given the plausible assumption that the codebase is relatively stable, i.e. that new code most of the time does not introduce faults, most of the time most tests that are run will not flip, because, as stated, the codebase is stable. If a test is recommended, but it does not flip, its precision will inevitably suffer, no matter how accurate the RTS selection technique is.

In other words, even though the precision values in Figure 6.3 are no where near 1.0, they are still perfectly acceptable. In relative terms the narrow test selection strategy significantly outperforms the all test strategy, by about a factor of 14.

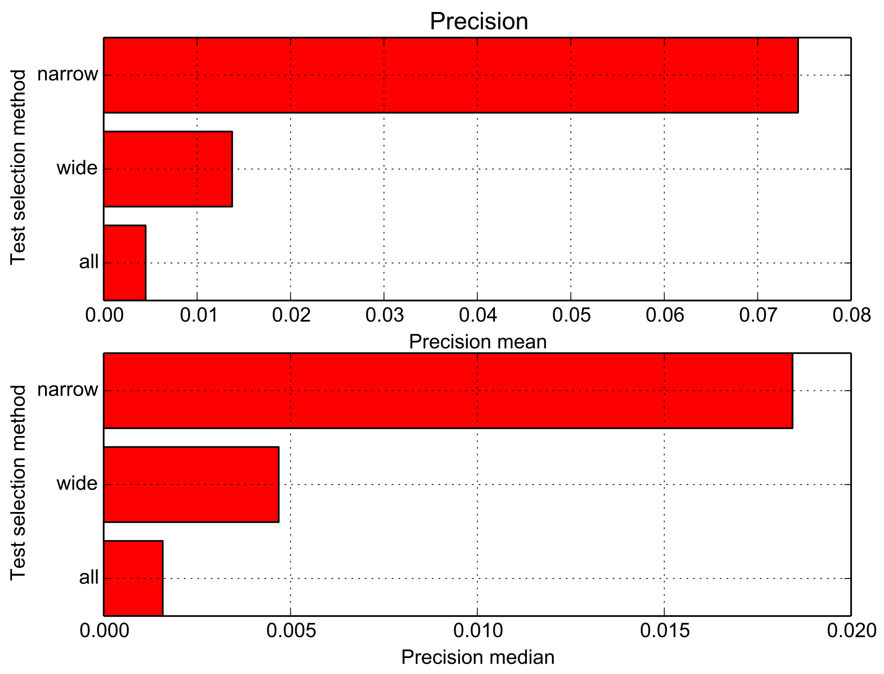

Figure 6.3: Bar chart displaying the mean and median values for precision using the different test selection strategies

6.3.4 F-measure for test flips on real data

As discussed earlier, the F-measure is the weighted harmonic mean of the recall and precision, given the recommended tests (retrieved tests) and the actual test flips (relevant tests). Figure 6.4 displays the F-measure for the different test selection strategies.

As can be seen in the charts, the F-measure ratios depend mostly on the values of the precision for the test selection strategies; this is because the recall for all selection strategies is close to 1.0 for all selection strategies, which means it will not affect the F-measure very much.

The narrow selection strategy is, again, the superior approach, while running all tests fares the worst.

6.4 Time

Is time saved by running only the selected tests?

As discussed earlier, if `analysis_time + subset_test_run_time < full_test_run_time`, time is saved.

For the sake of this analysis, all tests are considered equal in duration. This is obviously not true, and will have to be revised at a later date in order to achieve a

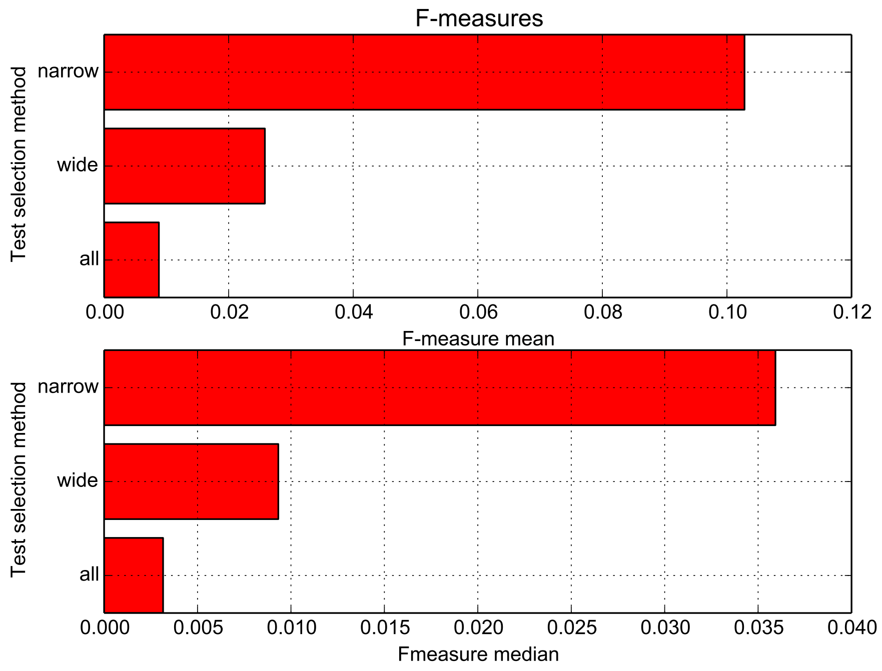

Figure 6.4: Bar chart displaying the mean and median values for the F-measure using the different test selection strategies

more correct analysis, but the work involved is considerable, and outside the scope of this thesis.

Extracting the data from the database with Seeker takes slightly less than 5 minutes. Analyzing the data with the Difference Engine takes less than 15 seconds. So we can say that `analysis_time` equals approximately 5 minutes[2].

`subset_test_run_time`, or, the duration of running the selected tests, is calculated as the number of tests suggested by a selection model, multiplied by the average duration time of one test.

All tests make up 1265 units and take about 7 hours to complete. The average duration of one test is therefore calculated to $7 \times 60/1265 \approx 0.33$ minutes.

The time savings are presented in Table 6.2 and in Figure 6.5.

As seen in figure 6.5 it is safe to say that *a lot of time* is saved by running only the tests recommended by the Difference Engine. With the narrow test selection strategy, it is possible to perform 20 integrations in the same time frame as it takes to perform one integration running all tests. At that rate, it is doubtful whether the regression test runtime is a bottleneck any longer; more likely, the bottleneck will be other manual processes that are involved in an integration of new code into the platform.

[2]This is strictly not true, however, since the database extraction and analysis can be performed incrementally, significantly reducing the duration of the analysis. We are assuming a worst case scenario here.

Task	Tests	Duration	Total	Time saved	Comment
Run all tests	1265	420	420	0	
Wide subset runtime	470	152	157	263	63% time saved
Narrow subset runtime	50	16	21	399	95% time saved

Table 6.2: Time savings calculations. The unit of time is minutes. The duration is calculated by multiplying the number of tests with the average duration of one test (0.33 minutes). The total is calculated by adding the time of analysis (5 minutes, worst case scenario) to the duration.

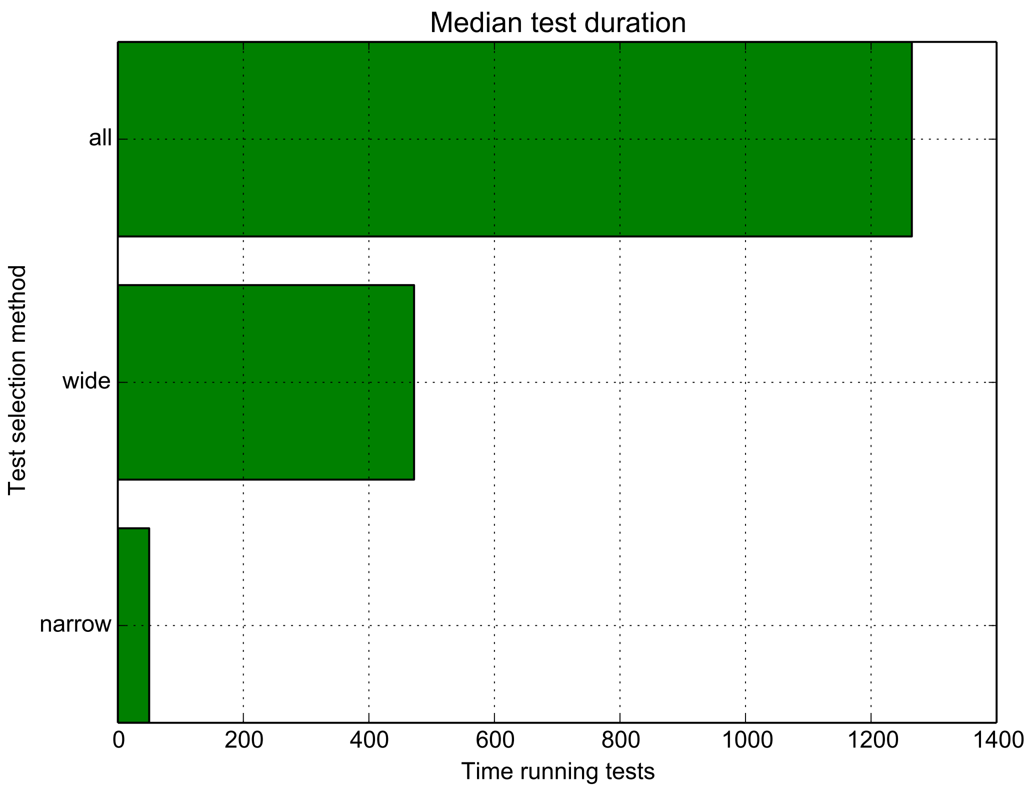

Figure 6.5: Bar chart showing the average number of tests run with different selection models. The runtime of the narrow subset selection is 5% of the duration of running all regression tests where no selection strategy is applied.

Finally, an interesting observation is that because of how the wide RTS strategy is implemented – recommending all tests that have *ever* flipped, regardless of which package has been changed – we see in table 6.2 that over 60% of all the tests in the full regression test suite have never flipped. Therefore, running the full test suite regardless of how small the change is, is generally a huge, huge waste of time. Just culling the tests that have never shown any relevant results would result in reducing the time spent on regression testing by 60%.

Chapter 7

Discussion

In this chapter we will summarize the results and draw conclusions from the summary, as well as discuss future possible improvements that can be implemented in the Difference Engine in order to make the test selection based on historical data (even) more effective.

7.1 Summary and Conclusion

In the first three chapters we saw that RTS techniques can theoretically be very beneficial if implemented properly, and that many researchers have been looking into test selection in order to enable a shorter test verification feedback loop. The other research has had an approach quite different from mine, where the most common way of performing test selection has been to analyze the code of the SUT (system under test) and then map different functions in the code to different tests. Such a low level analysis is costly – sometimes even prohibitively expensive and not possible in practice – if the SUT if too large and complex. The Difference Engine algorithm described in chapter 4 is very simple, but works fine and is fast even for large and complex systems. It is, on the other hand, highly dependent on having a regression test history to analyze and draw conclusions from. In order to provide a sufficient analysis we need a lot of historical data in order to cover all code packages and tests, and the data needs to contain enough failing tests to provide sufficient correlations. Ironically, the test selection algorithm thrives on an unstable code base that regularly introduces faults for as many different packages as possible, which is usually something you *do not want* when running regression tests. For the Difference Engine however, if the code base is too stable, test selection based on the historical regression test data might be dangerous since some packages very seldom cause test flips.

Chapter 5 dealt with how the analysis produced by the Difference Engine should be evaluated, and chapter 6 discussed the results of the evaluation. It was found that test selection based on historical data can show marked improvements in regression test execution time while still finding almost as many faults as running all tests would.

While performing the narrow test selection might on average appear to yield 20% worse recall than running all tests, it is important to remember that running all tests is a *very* expensive insurance policy, and that the median value for the narrow selection is the same as it is for running all tests. Additionally, one should keep in mind that the test recommendation given by the Difference Engine is essentially self correcting, if one makes sure that the latest build is added to the pool of analyzed builds. In other words, one might miss out once on a flipping test, but the next time it will be in the list of recommended tests, if the analysis is performed continuously.

This self correction comes at a price, however. As the recall ratio is increased and as more tests are added to the list of recommended tests, the precision will inevitably drop further. This is unavoidable since even though a code package and a test may be correctly correlated, there is never any guarantee that said test will flip just because its correlated package is changed. In fact, changing code without introducing bugs and test failures is practically the *modus operandi* of any decent developer. Yet, as long as the recall is high, and the regression test suite time is kept low, a somewhat lower mean recall value for the set of selected tests will not be a problem.

Given that with real code, most changes do not introduce test flips, I have some reservations with how many of the papers I studied relied on a statistical model that evaluate whether the selection of tests was successful based on one, or very few, analyses. Since it is safe to assume that any selection algorithm will miss out on important tests, it is better to accept this reality and instead provide a selection algorithm that can be continuously adjusted and calibrated. An organic, self adjusting selection methodology – such as the one presented in this paper – works fine, as my empirical results clearly demonstrate.

There is, of course, a weakness in that such an approach depends on easy access to historical test results. If there – for whatever reason – is a dearth of historical data to analyze I can see the benefit of test selection based on an up front static analysis of the code and tests, but then my recommendation is to as quickly as possible get access to historical data which can be used to augment the test-to-code correlations.

In other related research a lot of effort is spent on validating that the selected tests will find *all* possible faults. Our approach is different; we accept the reality that relevant tests may – and *will* – be missing from the set of selected tests. However, as stated above, we also plan for this being a self correcting problem by continuously providing the Difference Engine with fresh historical data to analyze. Since running only the recommended tests will starve the Difference Engine of historical data to analyze it is imperative that the full regression test suite is run periodically, for instance in regular nightly builds. Given this strategy, a test will only be missing from the pool of recommended tests for one day, then the full regression test run will pick it up during the night and it will be included in the set of recommended tests the next day. It is however important to remember that the (slight) risk of missing relevant tests will still be there for a full day, which has to be taken into account when verifying firmware using only the tests recommended by the Difference Engine.

As I see it, there are two apparent problems with using the F-measure to verify the efficacy of the test selection strategy proposed in this paper. The first is related to how precision is defined; in essence, precision is a value to measure whether a recommended test has flipped. A code package may be very strongly correlated to a test, but there is no guarantee that the test will flip just because that particular code package has been changed. In most cases – as mentioned above – the test will *not* flip, since most changes to the code do not cause any faults, and this means that the value of the precision will suffer greatly. As long as a low average precision is accounted for, however, and one realizes that it is the precision relative to other RTS strategies that is important, the low absolute values of precision and the F-measure are of little consequence except that they make the evaluation statistics looks worse than they really are.

The second problem is related to recall. With any form of test selection there is always the risk that a relevant test will be missing from the pool of recommended tests. This means that it is important that we regularly run the full regression test suite, for instance nightly as recommended above. In other words, we cannot solely rely on the tests selected by the Difference Engine, and we will still have test runs that take the full 7 hours to complete.

A less apparent problem with the F-measure as an evaluation tool is that for most intents, great recall and decent precision are more beneficial than decent recall and great precision, yet it is not taken into account when calculating the F-measure. In Axis case, for example, high recall should beat high precision every time, as long as the list of recommended tests is still a fraction of the size of the whole test pool. In other words, in practice it is better to run 10 tests where only 2 are relevant (recall $= 1$, precision $= 2/10$, F-measure $= 1/3$), than only run 4 tests, but where one relevant test is missed (recall $= 4/5$, precision $= 1$, F-measure $= 0.89$). The calculated F-measure scores in this paper do not reflect this reality unfortunately[1].

In the end, I ultimately think that it is valuable to start using the RTS strategy suggested in this paper since, as we saw in figure 6.5, a *20-fold increase* in integration productivity is possible. This means that the cost saving potential of implementing intelligent test selection for the daily regression tests at Axis is enormous, and the risks involved are only slight in comparison to the money that can be saved.

[1]It should be noted that the F-measure equation can be modified to take this into account, by weighting the recall and precision scores differently. It was not attempted in the evaluation of this thesis, though.

7.2 Future work

While the results of this thesis are usable already, there are of course a lot of improvements that can be added at a later date.

While the software is already performant enough, it can be enhanced. Python has an abundant selection of libraries that are specifically tuned for the kind of computations used to analyze the historical test data. I have used some of them to great effect, but there is still a lot of code that can be replaced by more competent libraries. For example, there is a library called Pandas[2] that features data structures that are much more optimized for the kind of data wrangling that is necessary for the Difference Engine to successfully analyze the raw data it has to work with. Also, currently the most time consuming process is the extraction of data from the MySQL database, and this can definitely be optimized further by someone more proficient with efficient SQL queries.

While doing the work involved with this thesis I realized that a lot can be learned by adding more metadata to the final analysis. For example, which code package causes most tests to flip? Which package causes the least amount of tests to flip? Which package is changed the most, and which is changed the least? Which tests fail or flip most often? What is the average amount of packages that change per build, and what is the average amount of tests that flip? Given more powerful and flexible data structures than the ones currently implemented in the Difference Engine, these questions can be answered quite easily. Said answers can then be used to gain a better understanding of how the regression test suite performs, provide ideas to attain better performance from the test suite, and perhaps even help explain why some previously inexplicable things occur during regression test runs. Since some tests in the regression test suite have proven to be quite unstable I think adding support for blacklisting such troublesome tests is important, which is another improvement that can be added to the Difference Engine. With proper metadata and analysis this process can even be automated.

At the outset, I had plans for utilizing a more advanced analysis algorithm since I somewhat incorrectly assumed that the initial algorithm would not produce as good results as it did. I planned to realize part of this more advanced algorithm by employing *Bayesian inference*, which is a method of statistical inference that can be used to update the probability for a hypothesis as more evidence is acquired. While it did not prove necessary for the work in this thesis, I still think this is a suitable and natural further improvement for evolving the algorithm. The correlation weights given by the initial analysis of the current implementation of the Difference Engine can seamlessly be used as the *prior probabilities* for which code packages and tests are correlated. It is unsurprising if this topic seems a bit hard to grasp; introducing Bayesian inference to RTS techniques is a subject worthy of a full thesis in its own right, and is difficult to explain within one paragraph. Suffice it to say, Bayesian inference is a great tool that has been used effectively to improve machine learning and statistical methodologies over a broad field of sciences, and I believe it can be

[2] http://pandas.pydata.org/

used to improve the work underlying this thesis as well.

Finally, I believe it is vital to find a way to automate the selection of tests and seamlessly integrate it into the developer workflow and code verification process to ensure employing advanced RTS strategies is as easy as possible for its intended users.

Chapter 8

Glossary

As I am perhaps presumptuously assuming that a majority of the terminology in this paper is already known to all potential readers, this glossary has been added to ensure that we share a common concept of the vocabulary used herein.

Bayesian inference A method of statistical inference in which Bayes' rule is used to update the probability of a hypothesis as evidence is acquired. See http://en.wikipedia.org/wiki/Bayesian_inference

Build In this thesis a "Build" is a data structure that contains the data of packages and their revisions, and the regression tests and their statuses (pass/fail).

BuildSet A set of Builds, ordered by time

Code package One or more source code files that are compiled together into a program.

DBDiffer The MySQL database containing the historical regression test data

Difference Engine The software that analyzes the raw historical data extracted from the MySQL database. Named after what can arguably be considered the first computer ever constructed.

Flip The word used in this paper to concisely express that the outcome of a test has gone form `pass` to `fail`, or from `fail` to `pass`. It is important to not only look at failed tests since a package change might make a historically failing test suddenly pass.

Functional test Tests how several units of code work together.

LOC Lines Of Code.

Mean Unless otherwise specified, whenever the word "mean" is used in this thesis, it refers to the arithmetic mean.

Narrow test selection The regression test selection strategy proposed by this thesis. Uses the output from the Difference Engine to define which tests should be run based on the found correlations between code packages and tests.

Noise Also referred to as *random noise*, *package noise* and *test noise*. Package noise in the context of this thesis is for example when multiple packages are changed at once, but only one of the package changes cause a test to flip. All of the changed packages will be correlated with the flippsed test, yet only one of the correlations are valid – the others are package noise. Likewise there can be test noise in the case where several tests flip, but some flips are not directly due to code package changes (for example, environment issues that cause tests to fail). These other test flips will be correlated with the changed packages even though they are due to other reasons; such false correlations will be noise as well, test noise.

Precision The ratio of selected tests that are relevant to a particular build. To calculate it you take the union of the set of relevant tests and the set of selected tests, count the size of the union, and divide it by the size of the set of selected tests.

Prior The probability distribution that expresses one's beliefs about a quantity before more tangible evidence is taken into account. See http://en.wikipedia.org/wiki/Prior_probability

Recall The ratio of relevant tests that are selected for a particular build. To calculate it you take the union of the set of relevant tests and the set of selected tests, count the size of the union, and divide it by the size of the set of relevant tests.

Regression test A test that is run in order to make sure that changes to code do not introduce a *regression*, i.e. break things that have been working previously. Regression tests can be either functional or unit tests. At Axis, all regression tests are functional tests.

Relevant test A test that will flip for a particular Build.

RTS Regression Test Selection

Selected test A test that the Difference Engine has predicted will flip for a particular Build, given the changed package(s).

Static analysis Static analysis in the context of this thesis is when correlations between code units and tests are found by analyzing the source code of both, without actually running the code. The work underlying this paper is dynamic analysis – i.e. we find correlations by running the code and inspecting the results.

SUT System under test. The compiled code upon which regression testing is performed.

Test minimization An effort to permanently prune the regression test suite in order to only run the relevant test given a particular context.

Test prioritization An effort to reorder the regression test suite in order to run the most relevant tests first.

Test selection An effort to select and run only the tests from the regression test suite that are relevant given a particular testing context. This is the RTS strategy implemented in this thesis, and relevant tests are decided based on which code packages have been changed.

Test package One or more tests that test a Code package.

Unit tests Tests a single unit of code, e.g. one function.

Wide test selection A compromize between running all tests and the narrow RTS strategy. Uses the output from the Difference Engine to suggest *all* the tests that have flipped historically. It is a decent RTS strategy since a majority (>60%) of the tests in the regression test suite have never flipped.

Wrangling Also *data wrangling.* The process of converting or mapping data from a raw form into another format that is more convenient for consumption of automated tools. Data visualization, data aggregation, the training of statistical models are examples of data wrangling.

References

Engström, E., P. Runeson, and G. Wikstrand. 2010. "An Empirical Evaluation of Regression Testing Based on Fix-Cache Recommendations." In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 75–78. doi:10.1109/ICST.2010.40.

Graves, Todd L., Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. "An Empirical Study of Regression Test Selection Techniques." *ACM Trans. Softw. Eng. Methodol.* 10 (2). New York, NY, USA: ACM: 184–208. doi:10.1145/367008.367020.

Huang, Yu-Chi, Kuan-Li Peng, and Chin-Yu Huang. 2012. "A History-Based Cost-Cognizant Test Case Prioritization Technique in Regression Testing." *Journal of Systems and Software* 85 (3): 626–37. doi:http://dx.doi.org/10.1016/j.jss.2011.09.063.

Kim, Jung-Min, and A Porter. 2002. "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments." In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 119–29.

Kim, Jung-Min, Adam Porter, and Gregg Rothermel. 2005. "An Empirical Study of Regression Test Application Frequency." *Software Testing, Verification and Reliability* 15 (4). John Wiley & Sons, Ltd.: 257–79. doi:10.1002/stvr.326.

Leung, H.K.N., and L. White. 1991. "A Cost Model to Compare Regression Test Strategies." In *Software Maintenance, 1991., Proceedings. Conference on*, 201–8. doi:10.1109/ICSM.1991.160330.

Rees, K., F. P. A. Coolen, M. Goldstein, and D. A. Wooff. 2001. "Managing the Uncertainties of Software Testing: A Bayesian Approach." *Quality and Reliability Engineering International* 17 (3). John Wiley & Sons, Ltd.: 191–203. http://onlinelibrary.wiley.com/doi/10.1002/qre.411/abstract.

Rothermel, G., and M.J. Harrold. 1996. "Analyzing Regression Test Selection Techniques." *Software Engineering, IEEE Transactions on* 22 (8): 529–51.

doi:10.1109/32.536955.

Rothermel, Gregg, and Mary Jean Harrold. 1997. "A Safe, Efficient Regression Test Selection Technique." *ACM Trans. Softw. Eng. Methodol.* 6 (2). New York, NY, USA: ACM: 173–210. doi:10.1145/248233.248262.

Wikipedia. 2014. "Precision and Recall." http://en.wikipedia.org/wiki/Precision_and_recall.

Wikstrand, G., R. Feldt, J.K. Gorantla, Wang Zhe, and C. White. 2009. "Dynamic Regression Test Selection Based on a File Cache an Industrial Evaluation." In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, 299–302. doi:10.1109/ICST.2009.42.

Witten, Ian H., and Eibe Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yoo, S., and M. Harman. 2012. "Regression Testing Minimization, Selection and Prioritization: A Survey." *Software Testing, Verification and Reliability* 22 (2). John Wiley & Sons, Ltd: 67–120. doi:10.1002/stvr.430.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

EXAMENSARBETE Test Selection Based on Historical Test Data
STUDENT Edward Dunn Ekelund
HANDLEDARE Ola Söder (Axis Communications)
EXAMINATOR Emelie Engström (LTH)

Intelligent regression testing

POPULÄRVETENSKAPLIG SAMMANFATTNING **Edward Dunn Ekelund**

Intelligently selecting the most relevant tests by analyzing historical test data can cut down on the average time spent on regression testing by 95%, speeding up code integration by a factor of 20.

Automated regression testing is the practice of running tests that ensure nothing has been broken after changes have been made to the code, i.e. that no regressions have been introduced. In the 80s and 90s one of the most popular software project managing models was the *Waterfall* model. Waterfall is a sequential design process, meaning that all planning is done first, then the software is developed, and lastly it is verified by, for example, passing a regression test suite. In Waterfall, even if a test suite takes several hours to run it is not costly in relative terms since it is only run at the end of a project cycle which may have lasted several months or more.

Nowadays however, Waterfall has largely fallen out of favor, and so-called *Agile* processes have taken its place, where planning, development and verification are done iteratively and continuously. In other words, integration cycles are much shorter in Agile development and code changes are continuously verified several times per day. This means that a regression test suite that takes hours to complete has become massively inconvenient and costly.

My thesis work at Axis involved analyzing a vast database of historical regression test runs. Going through each test run in turn, I mapped the code that was changed to the tests that failed during the same run, which in the end resulted in a set of correlations between regression tests and code packages. Then, in future regression test runs, whenever code in a package is changed and has to be verified, we can use this correlation set to select only the regression tests that have historically proven to be affected by the changed package. All tests which are historically unaffected by changes in that particular part of the code are thus excluded from the regression test verification, and time is saved.

Evaluation of my results have shown that when running only the intelligently selected tests we find nearly as many regressions as when running the full regression test suite. Yet, since the test set is much smaller, time spent on regression test verification is only 5% of the time compared to running the full test suite. In Axis' case it means going from an 8 hour regression test suite to one that on average takes less than 20 minutes to complete.

The consequence of cutting down on time wasted on running unnecessary tests is that developers no longer need to wait a full work day to verify code changes. The rate of integration can be sped up by a factor of 20. With more rapid code integration, it takes less time to fix bugs and introduce new features, which leads to more highly valued software, and, ultimately, increased revenue for the company.