Improving the Effectiveness of Incremental Mutation Testing

Luke Bajada

Supervisor: Dr Mark Micallef
Co-Supervisor: Dr Christian Colombo

Faculty of ICT
University of Malta

30th May 2014

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)
Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Improving the Effectiveness of Incremental Mutation Testing

submitted is my work, except where acknowledged and referenced.

Luke Bajada

30th May 2014
Acknowledgements

I would like to thank Dr Mark Micallef and Dr Christian Colombo for their continuous guidance throughout this work. Special thanks also goes to Björn Scorey, Edward Mallia and Joe Cordina on behalf of CCBill EU Ltd. who made carrying out the industry case study possible. Last but not least, I would like to thank my friends and family for their continuous support throughout this year.
Abstract

With software systems becoming more complex, the need to have good quality code is becoming increasingly important. Confidence in the code can only be gained if one has confidence in the test suite which checks that code. Mutation testing is a technique which can be used to find shortcomings in a test suite. It is a fault-injection testing technique, which creates faulty versions of a program, and checks if the test suite catches the faults. Although effective, it is also computationally expensive, making it unattractive to be used in software development environments.

As an improvement, incremental mutation testing was introduced. This is a technique which exploits the iterative nature of agile environments, and performs mutation testing on only the changed parts of the code, greatly reducing the execution time. Although the execution time decreases considerably, the parts of the code that depend on the changed parts are not included in incremental mutation testing. This in turn results in a problem, because faults can surface when the parts which are dependent on the changed parts are executed. Since the dependent parts are not included in incremental mutation testing, it is not possible to check whether these new faults are caught by the test suite.

This work addresses this problem by introducing a tool which improves the effectiveness of incremental mutation testing. By using static analysis on the source code and generating a call graph, the parts of the code that depend on those changed are also included as part of the incremental mutation testing process. This made incremental mutation testing more effective, by detecting shortcomings in the test suite which can appear on the tests that execute the dependent parts of the changed methods. The results obtained in the evaluation have shown us that this tool is effective on codebases which have test suites that require a long time to execute. On the other hand, it was also shown that it was not feasible to use the tool on codebases which have test suites that execute in a short amount of time.
Contents

1. Introduction 1
   1.1 Aims and Objectives ........................................ 3
   1.2 Contributions ............................................. 5
   1.3 Document Outline .......................................... 5

2. Background 7
   2.1 Agile Development .......................................... 7
      2.1.1 Test Driven Development ............................... 7
      2.1.2 Levels of Testing ..................................... 8
   2.2 Mutation Testing ........................................... 9
      2.2.1 Traditional Mutation Testing ......................... 9
      2.2.2 Mutation Testing Improvements ....................... 11
         2.2.2.1 Do Smarter ........................................ 11
         2.2.2.2 Do Faster ......................................... 11
         2.2.2.3 Do Fewer .......................................... 12
         2.2.2.4 Incremental Mutation Testing .................... 13
   2.3 Coupling .................................................. 13
   2.4 Static Analysis ........................................... 15
      2.4.1 Control Flow Analysis ................................. 16
         2.4.1.1 Call Graph ....................................... 17
         2.4.1.2 Abstract Syntax Tree ............................. 18
         2.4.1.3 Control Flow Graph ............................... 19
      2.4.2 Data Flow Analysis ................................... 20
         2.4.2.1 Data Flow Graph ................................ 20
   2.5 Conclusions ............................................... 21

3. Design and Implementation 22
   3.1 Design Decisions ........................................... 22
      3.1.1 Coupling Used ........................................ 22
      3.1.2 Type of Analysis ..................................... 23
      3.1.3 Chosen Internal Representation ..................... 24
      3.1.4 Chosen Call Graph Generation Tool .................. 25
      3.1.5 Chosen Mutation Testing Tool ......................... 26
   3.2 Mutating Architecture ..................................... 27
3.3 Main Components of Proposed Tool ........................................... 28
3.3.1 Call Graph Builder ............................................................ 28
3.3.2 Signature Converter ............................................................ 28
3.3.3 Method Caller Detector ......................................................... 29
3.3.4 Data Flow between Components .............................................. 29
3.3.5 Order of Execution between Components ................................. 30
3.4 Assumptions ............................................................................ 31
3.5 Implementation Challenges ....................................................... 32
3.5.1 Fine Tuning Soot ................................................................. 32
3.5.2 Issues with Mutatinc ............................................................. 33
3.6 Conclusions ............................................................................ 34

4. Evaluation ............................................................................... 35
4.1 Metrics ...................................................................................... 35
4.2 Evaluation Methodologies ........................................................... 36
4.3 Evaluation Technique ............................................................... 37
4.4 Evaluation Setup ...................................................................... 38
4.4.1 Codebases ........................................................................... 39
4.4.2 Machine Specification ........................................................... 39
4.4.3 Metrics Calculation ............................................................... 40
4.4.4 Experiment Data ................................................................. 40
4.4.5 Procedure ............................................................................ 41
4.5 Results ...................................................................................... 42
4.5.1 Results for Apache Commons CLI Library .............................. 42
4.5.2 Results for Industry Case Study ............................................ 43
4.6 Discussion ............................................................................... 44
4.6.1 Execution Time .................................................................... 45
4.6.2 Mutant Increase Analysis ....................................................... 47
4.7 Threats to Validity .................................................................... 50
4.7.1 Internal Validity .................................................................... 51
4.7.2 External Validity ................................................................... 51
4.8 Conclusions ............................................................................. 52

5. Related Work and Conclusions ..................................................... 54
5.1 Related Work ........................................................................... 54
5.2 Conclusions ............................................................................. 55
5.3 Future Work ............................................................................ 57
5.4 Concluding Remarks ............................................................... 58

A. Appendix .................................................................................. 59

References ................................................................................... 60
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Effects of changes in code</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Testing pyramid adapted from [40, 28, 10]</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Calculating the mutation score from [25]</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Mutation testing procedure</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Incremental mutation testing procedure</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Static analysis process</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Example code and call graphs adapted from [19]</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of an AST adapted from [35]</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>CFG for different constructs adapted from [3]</td>
<td>19</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of DFG construction from code adapted from [2]</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of call graph depth</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>New Mutatinc architecture</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Data Flow Diagram</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>UML Sequence Diagram</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Procedure of the evaluation</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Calculating the performance gain/loss</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Calculating the mutant percentage increase</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Graph of the number of mutants against the depth – Apache Commons CLI</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Graph of the number of mutants against the depth – Industry</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Features of different types of mutation testing .......................... 38
4.2 Specifications of machines – Home & Industry .......................... 39
4.3 Experiment data – Apache Commons CLI .............................. 41
4.4 Results for a day’s work – Apache Commons CLI ...................... 42
4.5 Results for a week’s work – Apache Commons CLI .................... 42
4.6 Results for a month’s work – Apache Commons CLI .................. 43
4.7 Results for a day’s work – Industry ...................................... 43
4.8 Results for a week’s work – Industry ..................................... 44
4.9 Results for a month’s work – Industry .................................... 44
4.10 Performance gain from traditional mutation testing vs performance loss from naïve incremental mutation testing – Apache Commons CLI 45
4.11 Performance gain from traditional mutation testing vs performance loss from naïve incremental mutation testing – Industry ............. 46
4.12 Mutant percentage increase according to depth – Apache Commons CLI .......................................................... 48
4.13 Mutant percentage increase according to depth – Industry .......... 49
1. Introduction

The need to produce high quality code is always present in a software development environment. High quality code is generally the result of a test suite which boasts similar quality, since a test suite which tests more lines of code and control flow paths has a greater chance of catching bugs. As a means of checking the code that is being executed by a test suite, and to determine how well it is being tested, the industry generally relies on code coverage tools. Whilst this helps in the development process, code coverage does not guarantee that any faults will be caught when changes are made [7]. This is because simply ensuring that each line is executed by the test suite at least once, ignores a substantial part of the complex control flow that can happen along various paths of execution, each of which can contain a fault. For example, consider the code:

$$\text{if } (a \geq 8) \text{ becomes } \text{if } (a == 8)$$

A test in which $a=8$ would pass both times and add to the total coverage, but if there was also a test in which $a=9$, it would fail when the code is changed. The second test would not add to the total coverage, since it covers the same statements as the first, but it is vital to check that the logic of the if-condition remains correct.

Mutation testing is a fault-injection testing technique which is used to uncover shortcomings in test suites by emulating errors that programmers usually make when coding. Such faults, or mutants, are injected into the program as simple
syntactic changes to create faulty versions of it. These faulty programs are then run against the original test suite. Should the tests in the test suite be of high quality, the tests will fail, showing that the fault is caught. If, on the other hand, the tests pass, shortcomings in the test suite are uncovered [25]. Consider the if-conditions used previously, if only one test in which \( a=8 \) is considered, and a syntactic change mutates the condition from \( a>=8 \) to \( a==8 \), the test would still pass and this change would not have been noticed. This results in mutation testing showing us that more tests need to be implemented.

Although mutation testing has been shown to be more effective than code coverage in checking for shortcomings in a test suite, it is still not widely used. The main reason behind its unpopularity is because it entails a heavy computational expense to generate all the mutants and run the tests [38, 37]. In turn, this results in a long feedback loop to the developers. An improvement to address this issue was introduced, called incremental mutation testing [7]. By taking advantage of the iterative nature of system evolution in an agile development environment, this technique applies mutation testing to only the changed parts of a program between iterations. This therefore greatly reduces the length of the feedback loop, encouraging the use of such a technique.

Whilst incremental mutation testing is much faster than its traditional counterpart, it does not come without disadvantages. In [7] it is described that to get an instantiation of incremental mutation testing, the system is split between the changed and the unchanged code, and mutation testing is performed on the changed code. This instantiation is described as naïve since it does not account for the interconnections between methods and thus, the main consequence of this technique is that the methods which depend on the changed methods are not included in incremental mutation testing.

To better explain, consider a method \( A \), which is changed. Such a method could have a number of other methods which call it, known as the dependent methods. Should the change in method \( A \) return results which are unintentionally different
and not what the dependent methods expect, they in turn could also return results which are unexpected. This is where the naïve term originates, since incremental mutation testing does not mutate these dependent methods, it fails to catch possible shortcomings in the test suite. In order to retrieve these dependent methods, we need to investigate static analysis techniques.

Static analysis is the process of extracting information about a program from its source code without executing it [5]. Projects can easily have several thousand lines of code, making it difficult to find the interdependency between methods. This makes the problem non-trivial in a way that a solution needs to be found in order to analyse the code to find the dependent methods using static analysis techniques, and include them in incremental mutation testing, all while still maintaining the speed advantage.

1.1 Aims and Objectives

The aim of this work is to answer the question: “Can the effectiveness of incremental mutation testing be improved?” In the context of this work, we define incremental mutation testing as being more effective if it manages to retrieve the parts of the code that depend on the changed parts and include them in its mutation testing process.
Consider Figure 1.1, where changes in the source code can have a ripple effect which end up affecting a larger part of the code. The parts affected by the changes may behave differently than before. Therefore, by also including them in the incremental mutation testing process, we will be checking if the test suite also manages to catch any new faults which can be introduced by these changes. Doing so, we will improve the effectiveness of incremental mutation testing. In order for this aim to be achieved, the following objectives need to be addressed:

- Is the increase in time taken to retrieve the dependent parts feasible?

By finding the dependent parts of the changed code, we hypothesise that our solution will be slower than naïve incremental mutation testing, but faster than the traditional mutation testing method (i.e. mutating the whole codebase). Therefore, our solution will be feasible to be used if the extra time taken to find the dependent parts will still be noticeably faster than traditional mutation testing.

- Will including the dependent parts substantially increase the amount of mutants generated?
By including the dependent parts in the incremental mutation testing process, we expect that the number of mutants generated will be greater than that of the naïve approach. Should the number of mutants not increase, or increase only by a very small amount, the use of our solution would not be justified.

1.2 Contributions

In this work, we have investigated different static analysis techniques in order to find a solution to the problem presented. Afterwards, we managed to create a tool which analyses the call graph generated for the given Java projects. The call graph is a directed graph which represents the calling relationships between methods [19]. With the help of the call graph, the methods that call the changed methods were extracted and included in the incremental mutation testing process. Our solution was then applied to two case studies. Different depths of the call graph were investigated, with each increasing depth containing the method callers of the methods in the previous depth, until the number of methods retrieved stopped increasing, i.e. the methods retrieved had no methods that call them. This was done to see the difference in the results and to find an optimal depth to be used in general. When used on codebases that have test suites which require a large amount of time to execute, our solution was found to be considerably faster than traditional mutation testing while at the same time, it successfully improved the effectiveness of incremental mutation testing. We also found out that it was not feasible in terms of execution time for codebases which have fast executing test suites.

1.3 Document Outline

In this section, the different chapters of this work are listed along with a brief description of each.
Chapter 1. Introduction

Chapter 2: Background  An overview is given on the topics needed to understand this work. Agile development, traditional and incremental mutation testing are explained, along with further details regarding static analysis and the different types of coupling.

Chapter 3: Design and Implementation  This chapter talks about the design decisions taken and highlights how they help in accomplishing our goal. The main components and the interaction between them are discussed in detail. The assumptions taken and the challenges encountered during the implementation of the tool are also explained.

Chapter 4: Evaluation  This chapter details the chosen evaluation technique. The results obtained are shown and the improvement over incremental mutation testing is discussed in detail.

Chapter 5: Related Work and Conclusions  In this chapter, the successful implementation of this work is explained as well as the positive effects it can have on incremental mutation testing. Related work is discussed along with several improvements which are suggested as future work.
Before going through the work done and the results yielded, one must understand the underlying concepts which are essential to this work. In this chapter, we will discuss in detail agile development, mutation testing and its optimisations, coupling, and on different types of analysis, along with their internal representations.

2.1 Agile Development

Agile software development seeks to address the problems of traditional software development approaches, such as changing requirements and difficulties with visibility, by introducing activities which provide control, feedback and change [14, 49]. Dybå et al. [14] mention six different types of agile software development, all of which approach software development in iterations. At each iteration, the client tests the new features which are implemented and provides feedback to the developers. These iterations are critical to software development, since they provide visibility of the progress made and the changes done in between.

2.1.1 Test Driven Development

Test driven development is a concept used in agile software development. Tests are first written before the code, and then enough code is written for the tests to pass. This type of development tends to be tedious and slow. However, the end result
is elegant code which is well tested. Tests are generally short and are directed towards one particular feature of the code written. Overall, code written using test driven development tends to have less failures in tests [31].

2.1.2 Levels of Testing

Four levels of testing are used in an agile development environment. These are unit, integration, system and user acceptance testing [40, 28].

![Testing pyramid adapted from [40, 28, 10]](image)

Figure 2.1: Testing pyramid adapted from [40, 28, 10]

Figure 2.1 illustrates the different levels of testing, each of which test different aspects of a given codebase. We will now explain in more detail these different levels of testing.

**Unit Testing** This is the most basic form of testing. As its name implies, it is used to test small units of source code, which usually consist of a method or a set of related methods. Unit testing is described as a white-box testing method, since unit tests are written by a person who has knowledge of how the system works. Therefore, this type of testing is used to individually test sub-parts of a system [40, 28].
Integration Testing  This type of testing seeks to determine whether multiple components of a system work correctly when combined together. Therefore, integration tests determine whether conversions and data transmissions between independent modules of a system are being done correctly [40]. Two different modules of a system can pass their units tests, which shows that they work as expected independently, but fail their integration tests, which shows that there are problems in their interconnection [28].

System Testing  This is used to ensure that the system under test works correctly as a whole and all of its modules are successfully communicating and executing as expected [40, 28].

User Acceptance Testing  This is the final type of testing performed, which is done by the client himself [40]. The aim of user acceptance testing is to make sure that the system developed is what the client expected and that all the features required are available and working correctly [28].

2.2 Mutation Testing

Mutation testing is a fault-injection testing technique which is used to find shortcomings in a test suite [25]. Although it is very effective in doing so, it comes with a heavy computational expense, discouraging its use in the industry [37, 38].

2.2.1 Traditional Mutation Testing

Traditional mutation testing is the type of mutation testing that was originally conceived. The process (refer to Fig. 2.3) involves generating errors in the source code by introducing small syntactic changes to create a set of mutant programs which are faulty, each containing a different syntactic change. The original test suite then runs these mutant programs and if some tests pass, it may indicate that
Chapter 2. Background

there are shortcomings in the test suite [25]. If tests fail against a given mutant, it is considered to be killed, otherwise it is said to have survived [25] or to be un killed [7]. The mutation score is one outcome of the mutation testing process and it is used as a factor to determine the quality of the test suite. As shown in Figure 2.2 this is calculated by the percentage of the killed mutants divided by the total number of mutants [25].

\[
\text{Mutation Score} = \frac{\sum \text{killed mutants}}{\sum \text{total mutants}} \times 100
\]

Figure 2.2: Calculating the mutation score from [25]

The fault which is injected into a program’s source code by replacing or removing an operator is called a mutation operator. There are many forms of mutation operators [25]. A common example would be replacing an AND with an OR as shown here:

\[
\text{if}(a > 0 \&\& b > 0) \quad \text{becomes} \quad \text{if}(a > 0 \mid\mid b > 0)
\]

The logic of the if-statement is completely changed. Where previously both conditions had to hold, now only one has to. Faults like these are used in mutation testing to check for any shortcomings in a test suite, since changes like these should result in tests failing.

![Mutation Testing Procedure Diagram](image)

From the process shown in Figure 2.3, and from understanding what a mutant operator is, it can be easily seen that given a large number of classes and a large number of mutants, the time mutation testing takes to output a result would be quite significant.
2.2.2 Mutation Testing Improvements

In order to make mutation testing more feasible in a software development environment, several cost reduction techniques have been invented.

2.2.2.1 Do Smarter

Weak Mutation [23] is an approximation technique, which adopts a “do smarter” approach by comparing the state of the mutant and the original program exactly when the part which is mutated is executed. If the states are not found to be identical, the mutant can be said to be effective [38]. For example, consider the following code, where the right if-statement is a mutation of the left one:

```java
if (a && b) {
    c = a;
} else {
    c = b;
}
```

Using weak mutation, this mutant is said to be effective if a test reaches a different state. That is, when \( a=true \) and \( b=false \), the first case results in \( c=b \), whereas the second case results in \( c=a \).

Another “do smarter” technique is used in [42] to run tests which are relevant to the mutated code. This was achieved by performing code coverage analysis and determining the lines that each test covered. Therefore, when a statement is mutated, only the tests which execute that particular statement are executed. Javalanche [42] is a mutation testing tool which uses this technique to reduce the overheads of running irrelevant tests. Also, Javalanche performs bytecode mutation in order to eliminate the overhead of compilation.

2.2.2.2 Do Faster

A “do faster” approach seeks ways to execute mutants at a faster speed. In [12], to avoid slowdowns due to compilations, a compiler-integrated mutation application
was developed that is able to execute the compiled code while avoiding a lot of the overhead due to compilation bottleneck. The program being mutated is compiled using this custom made compiler and during the compilation, mutants are generated as code substitutes. In order to execute a mutant, the code “snippet” which belongs to the mutant is applied before execution. This is an inexpensive approach and the mutant executes at compiled speed [25, 38].

Mutant Schema Generation (MSG) is another “do faster” technique which encodes all mutants into one source level program. The native compiler is then used to compile the program in the same environment at compiled-program speeds. Since systems based on a mutant schema do not need to provide any run-time semantics or environment configurations, they are significantly less complex and easier to build than interpretive systems, rendering them portable [38].

The execution of tests against each mutant is fully independent, therefore, similar to Javalanche [42], running the tests in a parallelised way should significantly reduce the time of test execution.

2.2.2.3 Do Fewer

Not all mutation operators are considered to be effective. In a “do fewer” approach, the number of generated mutants is reduced. A study in [38] showed that out of the 22 mutation operators in use, 6 were responsible for 40%–60% of the mutation kills. Selective mutation was created so that only mutants which are very effective in detecting flaws are applied to a system [25, 33, 38]. When selective mutation was implemented in [37], discarding two mutants which were deemed as ineffective provided a 30% increase in time saved, while discarding six resulted in 60% of the original time saved. This was at the cost of a mutation score reduction of only 2%. Javalanche [42] is a mutation testing tool which applies this technique.

Mutant sampling is another “do fewer” approach which randomly, or using some form of heuristic, only runs a subset of the mutant programs [25]. Using a heuristic was found to be more effective than choosing randomly. With a 10%
sample of mutant programs, the fault detection effectiveness decreased by only 16%, compared to the full set [38]. Mutant clustering is a subset of mutant sampling, which uses clustering algorithms in order to select mutants [25].

### 2.2.2.4 Incremental Mutation Testing

Incremental mutation testing [7] (refer to Fig. 2.4) was developed to address the issues of mutation testing. By taking advantage of the iterative nature of the agile development process, it applies mutation testing on the sections of code which have been changed between one iteration and another. This means that applying incremental mutation testing on all iterations of the code would effectively mean that mutation testing has been done on the whole system.

![Incremental mutation testing procedure](Figure 2.4)

The speed difference between this and the traditional mutation testing method is substantial mainly because of localised mutation [7]. In localised mutation, only the methods which are changed are mutated, compared to all the methods in traditional mutation testing. Mutatinc [6, 7] is a mutation testing tool which uses this technique as an optimisation.

### 2.3 Coupling

The problem of incremental mutation testing is that the methods which depend on the changed methods are not included in the mutation testing process. This
can lead to potential shortcomings in a test suite which are not caught using this technique, since a change in a method may alter the behaviour of its dependent methods. Therefore, in this work, we need investigate the several types of coupling available to characterise the ripple effects of a code change in a system.

Coupling is a qualitative measure, which shows the degree of interdependence between classes. The more classes and their components are dependent on each other, the higher the level of coupling. It is always a good design practice to keep coupling as low as possible [40]. In their book, Lethbridge and Lagamiere [29] describe several different types of coupling:

- **Content Coupling** This is the highest form of coupling. It occurs when a component “surreptitiously modifies internal data of another component” [29]. This should always be avoided.

- **Common Coupling** Occurs when several components make use of the same global variable. This is sometimes unavoidable, but it can lead to unforeseen and unexpected side effects when the global variable is changed.

- **Control Coupling** This occurs when having two methods, $fooA()$ and $fooB()$, where $fooA()$ calls $fooB()$ and passes it a parameter. This parameter, or control flag, can change the logical flow of $fooB()$. Thus, an unrelated change in $fooB()$ can give a different result to $fooA()$, with the possibility of creating an error.

- **Stamp Coupling** This type of coupling happens when a parameter of a method is one of the application classes. If one only needs the name and surname, it is better to pass these as arguments, rather than the whole Person application class. Failing to do so could result in the system becoming needlessly more complex.

- **Data Coupling** This is the result of a method containing a large amount of data arguments and increases the difficulty of testing and maintenance.
Chapter 2. Background

**Routine Call Coupling** Occurs when one method invokes another. Although this is quite common and necessary, it increases the connection between system components. In order to reduce this, encapsulating repeated method calls that are called after each other reduces this type of coupling, and the maintainer would only have to change the encapsulated method if the methods undergo changing.

**Type Use Coupling** This occurs when a class A uses a data type defined in another class. Although this can be a common occurrence, a change in that data type could result in a change to all the places that use it. Thus, the same solution as with stamp coupling applies; always use the simplest data type that is possible to work with.

**Inclusion or Import Coupling** The simple importing of a file or package results in this type of coupling. When packages or files are not used, simply removing them reduces this coupling.

**External Coupling** This type of coupling occurs when the system is dependent on elements outside its scope, such as the operating system, hardware or other libraries. To address this, reduce the places which are dependent on such elements.

As it can be seen, coupling is a necessary evil, however, one should try to keep coupling as low as possible and understand the effects of high coupling if it cannot be eliminated [40].

### 2.4 Static Analysis

Static analysis can be used in order to detect the different types of coupling. Binkley [5] describes static analysis as the process of extracting information about a program from its source code without executing it. In this case, the information
we need to know is how changes in the source code affect other parts of the same code.

Figure 2.5 shows the generic process of static analysis according to how it is explained in [5]. The three part process is split in the following: the parser, which transforms the source code into abstract syntax to make it applicable for further analysis, the internal representation, and the analysis of the given representation.

The scope of our work involves finding a suitable internal representation in order to retrieve the methods that depend on the changed methods. An internal representation is an abstraction of the program into a form which is more suitable for static analysis. Several types of internal representations exist which can be used for different types of analysis. In the following sections, control flow and data flow analysis will be discussed, along with the internal representations that allow such analysis.

2.4.1 Control Flow Analysis

Control flow analysis is used to determine the control flow relationships in a program [1]. The control flow is the order in which statements or function calls in code are executed. In [43], control flow analysis is used as a static analysis technique in order to retrieve the possible set of callers of a function or a method call. The following sections detail some internal representations which can be used for control flow analysis.
2.4.1.1 Call Graph

The call graph of a program is defined by [19] as a directed graph that represents the calling relationships between the program’s procedures or methods. Two types of call graphs exist, those which are context-sensitive and those which are context-insensitive.

procedure main() {
    return A() + B() + C();
}

procedure A() {
    return min(1, 2);
}

procedure B() {
    return min(3, 6);
}

procedure C() {
    return min(5, 3);
}

Figure 2.6: Example code and call graphs adapted from [19]

(a) Example Code

Procedures in a context-insensitive call graph can have a maximum of one node, with each node having a list of call sites and each call site being the source of zero or more edges to other nodes, which represent possible callees of that site. A call site is the location in the source code from where a function can or might be called. In a context-sensitive call graph, each procedure can have more than one node, according to the context in which it is called [19].

An example of these is shown in Figure 2.6. Figure 2.6(b) shows a context-insensitive call graph corresponding to the code in Figure 2.6(a). Since all the methods are calling \textit{min}, the arrows are pointed to the \textit{min} node. Figure 2.6(c) shows a context-sensitive call graph. The difference from Figure 2.6(b) is that for every invocation of the method \textit{min}, an instance (node) is created so that the different calling contexts of that method are distinguished [48, 5].
Chapter 2. Background

2.4.1.2 Abstract Syntax Tree

An abstract syntax tree (AST) is a form of an abstraction of the source code which is created after syntactic analysis, or parsing, is done [27, 35]. An AST contains only the functional parts of the code, meaning that irrelevant parts like white spaces and comments are ignored since they have no impact on the outcome of the program, while variable declarations, expressions and scopes are represented. There are two types of nodes in an AST:

- Scope nodes – Act as a container for other nodes, represents a block of code which can contain other scopes or expressions.
- Expression nodes – Describe an expression, which can be an assignment, declaration, function call etc. and can also point to other inner expressions.

It is also important to add that in an AST, each node corresponds to the position its related text is in the source code [35].

Figure 2.7: Example of an AST adapted from [35]

Figure 2.7 is an example of an AST, where the circular nodes are expression nodes and the rectangular nodes are scope nodes. The placement of the nodes, as
previously said, corresponds to the order the code is written in the actual source code. This makes it easier to understand and also to reproduce the code shown in the AST.

### 2.4.1.3 Control Flow Graph

A control flow graph (CFG) is a directed graph where each node represents a set of statements in a program and the edges connecting the nodes represent the control flow paths [3]. According to Arora et al. [3], most programs contain three different types of constructs, which are the sequence, selection, and iteration constructs.

![Figure 2.8: CFG for different constructs adapted from [3]](image)

Figure 2.8 (a) shows the sequence construct of a CFG, where it is merely the flow from one node to the next. Figure 2.8 (b) demonstrates the selection construct, where an if condition can alter the flow to different nodes according to the criteria. Lastly, Figure 2.8 (c) is a depiction of the iteration construct using the while loop, where the control flow depends on the loop condition.
2.4.2 Data Flow Analysis

Data flow analysis is used to discover information about the flow of data in a program, such as the definitions and uses of data defined. The definitions, or defs, and uses of data form def-use chains, where a definition D is where a variable is assigned a value, and a use U is when D is referenced or used [21].

One of the main applications of data flow analysis is to understand the behaviour of a program, mainly for debugging and testing purposes [44]. Data flow analysis can be done on almost any level of scope in a program, with Srikant et al. [44] mentioning three different types:

- Local data flow analysis – Analysis across statements within a basic block of code.
- Global (intra-procedural) data flow analysis – Analysis across basic blocks but within functions or procedures.
- Inter-procedural data flow analysis – Analysis across functions and procedures.

In the next section, we will discuss data flow graphs, which are a possible internal representation for data flow analysis that utilise def-use pairs in order to show the flow of data in a program.

2.4.2.1 Data Flow Graph

A data flow graph (DFG) is a directed graph whose edges connect each def D to every use U of D [2]. Figure 2.9 shows an example of a DFG and the code used to create it.
1. int increase(int x, int y)
2. while (x<y){
3.   x++;
4. }
5. return x;
6. }

(a) Example Code

(b) DFG for given code

Figure 2.9: Example of DFG construction from code adapted from [2]

From this example, it can be seen that not all lines of code are represented in the DFG. This is because the only lines of code that are relevant, are those that contain defs and uses. Each node in the graph contains one or more defs and uses, called the def set and use set respectively [2]. For example, line 2 contains the uses of x and y which are taken from the def set in line 1. Line 3 kills the def of x in line 1, but since it is possible not to enter lines 2–4, both data flow arrows from node 1 to 5, and 3 to 5 need to be made.

2.5 Conclusions

In this chapter, we discussed agile development, mutation testing, different types of analysis, and coupling in detail. These are the techniques and measures needed in order to fully understand the context of this work. In the next chapter, we will use what was mentioned here in order to design a tool which can help us reach our aims and objectives.
3. Design and Implementation

In this chapter, we will discuss the design decisions which needed to be taken to implement our tool. These involve finding the types of coupling to be used, choosing a suitable internal representation, finding a tool which generates the chosen internal representation and then merging everything with an existing mutation testing tool in order to extend its functionality. After these decisions are discussed, the main components of this tool will be explained. Finally, we will explain the assumptions that had to be taken and challenges encountered during the implementation process, along with how they were overcome.

3.1 Design Decisions

In this section, we will explain the design decisions taken in order to construct the tool.

3.1.1 Coupling Used

Our work deals with characterising the ripple effects of a code change in a system. Incremental mutation testing is described as naïve because only the changed methods are mutated, and the methods which are dependent on the changed methods are ignored. A change in a method can have an effect throughout all the types of coupling, but due to time constrains, we addressed the types of coupling which
directly deal with the interdependencies between methods. These types of coupling are identified from Section 2.3:

**Control Coupling**  A syntactic change in a method can lead to the return of a different result to the methods which are calling it, therefore the inclusion of these methods can detect if there are shortcomings in the test suite.

**Routine Call Coupling**  This type of coupling is created by having a method simply calling another one. This is the basis of this work and therefore, this coupling is being used to find the connection between methods.

**Import Coupling**  The inclusion of packages in a class means that the class is being connected to other classes. This can include a method calling another method from a different class, which falls in the scope of this work.

### 3.1.2 Type of Analysis

Analysis can be either static or dynamic. Dynamic analysis occurs during runtime and takes program input into account. Because of this, it does not include all of the execution paths, and therefore, can ignore substantial parts of the code. In our case, certain methods might not be called if dynamic analysis is used. This is because the input might result in a control path which does not call certain methods (eg, a method inside an if-condition which is not entered). Therefore, if dynamic analysis is used, all the possible program inputs need to be used in order to guarantee that all the execution paths are entered, which is not always feasible. On the other hand, static analysis does not require the execution of the program and takes all the execution paths into consideration [5]. Thus, using static analysis ensures that all the method callers of a given method are retrieved.

As mentioned in Section 2.4, two types of static analysis exist, which are control flow and data flow analysis. In order to take advantage of the coupling types previously mentioned, control flow analysis is the suitable choice. This is because
it uses internal representations to analyse the relationships between functions or statements in a program, as can be seen from call graphs, control flow graphs and abstract syntax trees. Data flow analysis deals with the definitions and uses of variables in a program, something which is not best suited for finding the interdependencies between methods.

### 3.1.3 Chosen Internal Representation

There are several different internal representations used in control flow analysis. Some of these are generated by the parser, most notably the abstract syntax tree, the control flow graph, and the call graph [5]. The most suitable internal representation which was used for this project is a context-insensitive call graph. As described in Section 2.4.1.1, a call graph provides the calling relationships between methods [19]. This is ideal in our case since we need the find the callers of methods. The decision to go with a context-insensitive instead of a context-sensitive call graph was because we only need the method callers, irrespective of the context that the methods are called in.

An abstract syntax tree is a tree representation of an abstraction of the source code created after parsing [35]. Although it is possible to retrieve a method’s callers from an AST, this requires the generation of a symbol table, which is a data structure that stores the information about the types and names used in a program [39]. Therefore, an AST, although it can be used, is not suitable for the scope of this project because it generates unnecessary information which is not used for this project and can have an added overhead on the system.

A control flow graph is a directed graph, in which nodes represent a set of statements in a program and edges represent the flow of data [3]. Whilst a CFG is optimal for an intra-procedural view of the control flow within a single method or procedure, it does not provide an inter-procedural view [22], something which is necessary for this work and which is provided by a call graph.
3.1.4 Chosen Call Graph Generation Tool

Several tools exist which generate a call graph for the given program, such as Soot [34], java-callgraph [18], Graphviz [45], JProfiler [15] and the DMS Software Reengineering Toolkit [13]. Of all these, Soot was selected as a suitable tool for call graph generation. This is because java-callgraph and Graphviz provide only visualisations of call graphs and do not allow for their call graphs to be used in code in order to extract the method callers that we need. JProfiler and the DMS Software Reengineering Toolkit were not considered since they are not free to use.

Soot [47, 34] is a Java bytecode optimisation framework which provides several internal representations for Java bytecode, most importantly, call graph generation. Soot also has a flow analysis framework, making it useful to map out the dependencies of a project as well as how the data travels between different parts. This flow analysis framework allows us to generate the context-insensitive call graph of the given programs and also use it in order to create the tool which extracts the method callers by traversing different depths of the call graph.

![Figure 3.1: Example of call graph depth](image-url)
Figure 3.1 illustrates what is being referred to when the depth of the call graph is mentioned. The grey node in the middle represents the method which contains the syntactic change. It is always considered to be at depth 0. The nodes directed towards the grey node refer to the methods which directly call it, and are in depth 1. The nodes at depth 2 are the methods that call the methods at depth 1. This can keep on going until a depth is reached where there are no nodes.

3.1.5 Chosen Mutation Testing Tool

MuJava [32], Javalanche [42] and Mutatinc [6] are all mutation testing tools for Java. Since this work is aimed at improving the effectiveness of incremental mutation testing, only Mutatinc could be used, since it is the only tool capable of such a technique. The following is an overview of the different mutation testing algorithms provided by Mutatinc:

- Traditional Mutation Testing – Mutates all of the codebase and runs the tests until a test is found which kills the mutant.

- Incremental Mutation Testing – Mutates and tests only the changed methods between iterations of a codebase. This type of incremental mutation testing is called naïve [7] since it does not mutate and test the methods which depend on the changed methods.

The changed methods are inputted in our tool and in return, our tool provides Mutatinc with the methods that call the changed methods, which are then also included in incremental mutation testing.

The Apache Commons BCEL [11] is the bytecode engineering library Mutatinc uses in order to perform mutations on the operators. It is also used to detect the methods that have changed between iterations at a bytecode level. The methods that are provided to us by Mutatinc are in the format specified by this library. Therefore, we need to convert the methods to be compatible with Soot and after
the results are retrieved, they need to be converted back to be compatible with Mutatinc.

### 3.2 Mutatinc Architecture

Figure 3.2 shows the architecture of Mutatinc after our tool was integrated with it. It has the same architecture it previously had with the component marked with a dotted line, labelled ‘Retrieve Method Callers’, signifying our addition to Mutatinc, along with the place where it was inserted.

![New Mutatinc architecture](image)

**Figure 3.2: New Mutatinc architecture**

Mutatinc first starts by listening to syntactic changes done in the source code.
After it notices any changes, it detects the methods which are part of those changes. Then, our tool is used in order to retrieve the callers of these methods. The tests are then found and the methods are mutated. The tests are run against the mutated methods and the results are shown on the front end part of Mutatinc to provide a user friendly interface to the user.

3.3 Main Components of Proposed Tool

The tool is made up of the following components:

- Call Graph Builder – Builds the call graph for the given project.
- Signature Converter – Converts method signatures from Java bytecode to objects recognised by the call graph.
- Method Caller Detector – Provides the callers of the methods according to the specified depth.

These components will now be explained in more detail in the following sections.

3.3.1 Call Graph Builder

This component loads the project being analysed and uses Soot to generate its corresponding call graph. The call graph contains the calling relationships of all the methods in the project.

3.3.2 Signature Converter

Since Soot does not hold the same method signature format that the changed methods given to the tool by Mutatinc do, a signature converter had to be implemented so that it will translate the signature to the format that is recognized by Soot’s call graph. Consider the following BCEL method signature:
Chapter 3. Design and Implementation

```java
(Ljava/lang/String;)Lorg/apache/commons/cli/Option;
```

This is converted to the following Soot method signature:

```java
<org.apache.commons.cli.CommandLine:
  org.apache.commons.cli.Option resolveOption(java.lang.String)>
```

After the results are obtained, they are translated back to the signature format which is recognized by BCEL and Mutatinc. This component is used in the method caller detector, which is going to be explained next.

### 3.3.3 Method Caller Detector

In this component, we analyse the call graph to retrieve all the nodes which are connected to the given call site with an edge. Nodes in themselves are methods in the call graph and when we find all the nodes connected to a given method, we would essentially have found the callers of the given method. According to the depth specified, the method callers found are also used to find their own respective callers, if available, and so on. This has already been explained in Figure 3.1, where different depths are defined with respect to the changed method. Also, as it is shown in the diagram, each depth can have several methods, because a method can be called by 0 or more methods.

### 3.3.4 Data Flow between Components

Figure 3.3 is data flow diagram which illustrates how the data provided by Mutatinc is processed by our tool.
The project path is given to the call graph builder component in order to generate the call graph for the project. The changed methods and the depth are given to the method caller detector component. This uses the signature converter component to convert the BCEL methods into Soot methods, in order to retrieve the method callers of the changed methods according to the specified depth. After all the method callers are retrieved and translated into BCEL methods, they are returned to Mutatinc.

### 3.3.5 Order of Execution between Components

The data flow diagram successfully shows how the data flows between the components but fails at showing what is executed first and the order in which the components are executed.
Chapter 3. Design and Implementation

Figure 3.4: UML Sequence Diagram

Figure 3.4 is a UML Sequence Diagram showing the order of execution of the components. The call graph builder component executes first by building the call graph from the given execution. A loop is then entered where the method caller detector uses the signature converter to convert the changed methods from BCEL to Soot in order to retrieve their method callers. The method callers are retrieved, up to the specified depth, using the call graph generated, and are converted back to BCEL format using the signature converter. The set of method callers is then returned to Mutatinc.

3.4 Assumptions

During the implementation of our tool, some assumptions had to be taken:

- Source and test classes are in their respective folders, following Java conventions.
- Test classes are in the same package as the source class that is being tested.
• Test classes and test methods follow common Java naming conventions.

The first two assumptions are taken by Mutatinc, and therefore since our tool will use Mutatinc, these assumptions must also be included. The last assumption is important because classes which contain test methods must not be considered as part of the method callers since mutation testing should only mutate the source code and not the tests.

3.5 Implementation Challenges

During the implementation procedure, several challenges were encountered. These challenges had to do with merging the tool elegantly with the existing Mutatinc, and also, optimising it in a way so as to make it as fast as possible.

3.5.1 Fine Tuning Soot

Soot provides numerous customisation options in order for the user to tweak it to suit his needs. During the implementation of the tool, Soot needed to be used as efficiently as possible in order to minimise the overhead that is added to incremental mutation testing, and thus, make the tool more attractive to use. The following are some challenges encountered and how they were overcome.

Partial Call Graph By default, when Soot is asked for a call graph, the user is presented with a call graph of the application classes along with all the libraries used in the project. The application classes are the classes which belong to the project being analysed. Apart from being very expensive to generate, only a call graph for the application classes was needed for this work, since the interest is in retrieving the method callers of methods in the same project. In order to address this issue, the call graph generated was being limited to only the application classes, and if an application class used an object of a library, this object’s class is considered to be a phantom class reference. These phantom class references are
references which are known to exist, but are not loaded by Soot. This optimisation resulted in a substantial speed boost whilst preserving the results which belonged to the project being analysed.

Redundant Call Graphs  Soot automatically generates a call graph containing all the methods as entry points. Entry points in Soot are a way of knowing where to start processing. At first, each new method was being included as an entry point and a new call graph was being built each time, but after examining the inner workings of Soot, it was found that all methods of all application classes are included as entry points by default during the construction of the call graph. This led to the changing of our tool by applying the singleton pattern such that the call graph is built once during initialisation, which in turn resulted in a performance improvement.

3.5.2 Issues with Mutatinc

The tool developed in this work needed to be merged with Mutatinc in order to enhance the effectiveness of its incremental mutation testing capabilities. Mutatinc is developed with concurrency in mind in order to detect changes in more than one class file at the same time, instead of one after the other. On the contrary, a synchronized block is used in order to execute the tests. A synchronized block holds a lock and allows only one thread to enter it at a time. Our tool was inserted in this synchronized block, and the methods which are dependent on the changed methods are retrieved and also included in the mutation testing procedure.

This extension needed to include some alterations to Mutatinc’s source code. Where previously the changed methods to be included for mutation were retrieved on a class by class basis, their dependent methods could now be from any of the other classes. Apart from this, methods can belong to different depths of the call graph. A method can be at depth 1 for a particular method, and at depth 2 for another. There is also the possibility that a method which is found by traversing
the call graph, is also a changed method. In order to prevent the duplication of method mutations, a global store needed to be held so that before mutating a method, it is checked whether it has already been mutated or not.

3.6 Conclusions

It was decided that control flow analysis using a context-insensitive call graph would be a suitable choice as an internal representation to investigate the coupling mentioned. This is because it provides all the information necessary regarding the calling relationships between methods. The call graph is generated using Soot and the tool which analyses the call graph is integrated with Mutatinc. The tool is split up into several components, which when combined together, provide the method callers of the changed methods. The challenges encountered mainly involved reducing the time taken to generate the call graph and integrating the created tool with Mutatinc. These challenges were overcome and the final artefact was successfully created. In the next chapter, we will evaluate the tool created and discuss the results obtained.
4. Evaluation

In the evaluation, the developed tool allowed us to retrieve methods that are connected to each other by different types of coupling, something which always exists in code. Existing metrics and evaluation methodologies were also studied to find those which are applicable to this work. This tool was then applied to two case studies and the results obtained are discussed, along with possible threats to validity.

4.1 Metrics

The following are standard metrics used in the field of mutation testing. Along with these metrics, the call graph depth metric is mentioned which is used in this work.

- Mutation Score [33, 37, 42] – This is known by several names in other works such as “kill rate” [20] and “mutation adequacy score” [25]. As shown in Figure 2.2 in Section 2.2.1, this is calculated as the percentage of the killed mutants divided by the total number of mutants [25]. The aim is to have a mutation score which is that of 100%, achieved by having a test suite which is able to kill all the mutants that are generated [33, 25]. This metric is not directly related to our work, since the mutation score is determined by the quality of the test suite written by the developers, whereas our main
focus regards the increase in mutants by including the method callers. The mutation score is still included in the final results, since these new methods are mutated and tested by the test suite, and thus form part of the mutation score.

- Execution Time [25, 42] – This is the time taken to generate all the mutations and execute the tests for each mutant. This metric is vital for this work since an execution time lower than that of traditional mutation testing needs to be achieved for our tool to be feasible.

- Call Graph Depth – The depth signifies how deep the call graph is going to be explored, as explained in Section 3.3.3. This has a direct relation to the amount of mutants generated and the execution time, with a higher depth increasing the amount of mutants and the execution time.

4.2 Evaluation Methodologies

There exist several evaluation methodologies used in literature to determine the feasibility of a mutation testing solution.

- Comparison using another mutation testing tool – In [46], two mutation testing tools were compared based on their execution time. This methodology was not feasible in this work because the scope of this work was to improve the effectiveness of incremental mutation testing by extending Mutatinc to mutate and test the method callers of the changed methods. We will therefore be comparing the previous Mutatinc with the one modified by us.

- Comparison between different mutation algorithms – [37] compares two mutation algorithms by using a metric, such as the mutation score obtained. For the purpose of this work, this methodology is ideal because as can be seen in Section 4.3, traditional mutation testing, naïve incremental mutation testing and incremental mutation testing with different depths are used. These use
different algorithms or an extension to an already existing one and are com-
pared based on their execution time and the amount of mutants generated.

• Evaluation using multiple codebases – In order to have a good result set, [42] and [33] used several codebases in their work. This methodology is ideal, and as can be seen in Section 4.4.1, two codebases are used in order to evaluate this work. It would have been more ideal to increase the number of codebases used, but due to time constraints it was deemed unfeasible.

For this evaluation, we will compare our work with different mutation testing algorithms, namely traditional and naïve incremental mutation testing and also use different codebases in order to have a wider view of the outcome. These will be discussed in the following sections.

4.3 Evaluation Technique

The purpose of this work is to improve the effectiveness of incremental mutation testing. In the context of this work, we define incremental mutation testing as being more effective if it addresses the issue of its current naïve approach by considering the ripple effects of a code change. This must be done while keeping the added overhead as low as possible in order for the technique to be still considerably faster than traditional mutation testing, and thus, feasible to be used. In order to address this, we will compare our solution with traditional mutation testing and naïve incremental mutation testing. The former mutates the whole codebase and runs the tests for each mutant. The latter mutates only the changes in code between iterations and runs the tests, hence the term localised mutation.

Our solution builds on incremental mutation testing by generating a call graph to include methods that call the changed methods up to the specified depth, and thus, having incremental mutation testing which is not naïve. Table 4.1 gives a brief summary of the different mutation testing types.
Chapter 4. Evaluation

<table>
<thead>
<tr>
<th>Mutation Testing Type</th>
<th>Localised Mutation</th>
<th>Call Graph Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naïve Incremental</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Incremental w/ Depth</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: Features of different types of mutation testing

We hypothesise that the execution time will be faster than traditional mutation testing but slower than naïve incremental mutation testing. This slow down signifies the cost of generating a call graph and also the cost of running the tests for the new methods which are included. The chosen evaluation technique will compare the execution time of these three approaches along with the increase in mutants according to the depth specified. This evaluation will be carried out on two different codebases in order to have a wider view of the outcome.

4.4 Evaluation Setup

In order to properly devise the evaluation, a number of factors needed to be considered:

- Codebases – The two projects that will be used in this work in order to generate the results for the evaluation. These will have their call graphs generated and analysed by our tool.

- Metrics Calculation – The metrics that will be calculated in this evaluation.

- Evaluation Machines – The machines used that will run the tool.

- Experiment Data – The different iterations of the codebases used.

- Procedure – The steps taken in order to generate the results.
4.4.1 Codebases

The codebases needed for the evaluation required to be stored in a version control system. This is because in order for incremental mutation testing to work, two versions of the same codebase are needed in order to detect the changes between them.

The Apache Commons CLI library [17] was chosen as it is open source and made it possible to retrieve revisions which could be used in incremental mutation testing. This library, with approximately 5,000 lines of code, provides the functionality for parsing command line arguments. This was also the library used in the original incremental mutation testing work done by Cachia et al. [7].

The other codebase used was taken from the industry courtesy of CCBill EU Ltd. The codebase belongs to a registration system which stores and retrieves the details of the company’s clients. With approximately 10,000 lines of code and with different revisions also available, an evaluation could also be carried out on this codebase.

4.4.2 Machine Specification

Table 4.2 shows the machines used for the case studies:

<table>
<thead>
<tr>
<th></th>
<th>Home</th>
<th>Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 (Sandy Bridge) 2.2GHz</td>
<td>Intel Core i7 (Nehalem) 2.93GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>6GB</td>
<td>8GB</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 8.1 64bit</td>
<td>Windows 7 SP1 64bit</td>
</tr>
<tr>
<td>Java</td>
<td>1.7 64bit</td>
<td>1.7 64bit</td>
</tr>
</tbody>
</table>

Table 4.2: Specifications of machines – Home & Industry
The reason why two machines were used was because the industry codebase could not be taken out of company premises, and therefore, the evaluation needed to be done on a machine which belonged to the company.

### 4.4.3 Metrics Calculation

Mutatinc calculates the execution time and mutation score automatically. The execution time is calculated from the moment when mutation testing starts until it is complete. The mutation score is calculated using the equation shown in Figure 2.2. The depth is specified before the incremental mutation testing is started.

In order to obtain these metrics in incremental mutation testing, two codebase instances are required. The first codebase being used as a starting point and the second codebase, which is that of a later date, being used to obtain the differences from the first one. The next section describes the iterations taken and the time lapse between them.

### 4.4.4 Experiment Data

For each codebase, three experiments were performed, each having a different gap in time. The changes performed by the developer between repository revisions X and Y simulate the work performed in the given time frame, as shown in Table 4.3. The numbers for each revision in the table represent the revision numbers of the codebase which was pulled. The decision to take several time gaps in each experiment was to try and identify patterns that different depths can have according to the lapse of time between revisions X and Y. The revisions obtained needed to have significant changes in the source code, since revisions which change other files which are not source code, such as documentation files, would not be useful.
Revision numbers regarding the industry case study could not be published due to confidentiality, but the same time frames are maintained, which are that of a day, a week, and a month.

### 4.4.5 Procedure

Figure 4.1 shows the basic step by step instructions used to evaluate the codebases on different depths. For each of the day, week and month experiments, three runs were performed for each depth in order to obtain an average of the execution time. The depth was increased until the amount of mutants stopped increasing.

While (i is 0) or (mutants increased from depth i-1)

For three times

For All experiment e

1. Incremental Mutation Testing was selected on the application
2. Depth i was given
3. The application was initialised on its starting revision
4. The revision was updated to a more recent codebase
5. Execution time, number of mutants and mutation score were recorded from the application

6. Increment i

Table 4.3: Experiment data – Apache Commons CLI

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Revision X</th>
<th>Revision Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A day of work</td>
<td>759392</td>
<td>779054</td>
</tr>
<tr>
<td>A week of work</td>
<td>780163</td>
<td>955156</td>
</tr>
<tr>
<td>A month of work</td>
<td>1095599</td>
<td>1570439</td>
</tr>
</tbody>
</table>

Figure 4.1: Procedure of the evaluation
Chapter 4. Evaluation

4.5 Results

This section presents the results recorded when applying our tool to the two case studies. The evaluation was performed as described in Section 4.4.5. The execution time was obtained by averaging the result of three runs. In the tables below, *Traditional* refers to traditional mutation testing and depth 0 refers to naive incremental mutation testing. Also, for each experiment, the depth was increased until the number of mutants stopped increasing.

### 4.5.1 Results for Apache Commons CLI Library

<table>
<thead>
<tr>
<th>Depth</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Unkilled</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traditional</td>
<td>253</td>
<td>110</td>
<td>43%</td>
<td>143 (57%)</td>
<td>20s</td>
</tr>
<tr>
<td>0</td>
<td>95</td>
<td>45</td>
<td>47%</td>
<td>50 (53%)</td>
<td>5s</td>
</tr>
<tr>
<td>1</td>
<td>122</td>
<td>58</td>
<td>48%</td>
<td>64 (52%)</td>
<td>18s</td>
</tr>
<tr>
<td>2</td>
<td>122</td>
<td>58</td>
<td>48%</td>
<td>64 (52%)</td>
<td>18s</td>
</tr>
</tbody>
</table>

Table 4.4: Results for a day’s work – Apache Commons CLI

<table>
<thead>
<tr>
<th>Depth</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Unkilled</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traditional</td>
<td>340</td>
<td>113</td>
<td>33%</td>
<td>227 (67%)</td>
<td>20s</td>
</tr>
<tr>
<td>0</td>
<td>183</td>
<td>57</td>
<td>31%</td>
<td>126 (69%)</td>
<td>9s</td>
</tr>
<tr>
<td>1</td>
<td>210</td>
<td>68</td>
<td>32%</td>
<td>142 (68%)</td>
<td>20s</td>
</tr>
<tr>
<td>2</td>
<td>210</td>
<td>68</td>
<td>32%</td>
<td>142 (68%)</td>
<td>20s</td>
</tr>
</tbody>
</table>

Table 4.5: Results for a week’s work – Apache Commons CLI
Chapter 4. Evaluation

<table>
<thead>
<tr>
<th>Depth</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Unkilled</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16s</td>
</tr>
<tr>
<td>Traditional</td>
<td>349</td>
<td>131</td>
<td>38%</td>
<td>218 (62%)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>158</td>
<td>73</td>
<td>46%</td>
<td>85 (54%)</td>
<td>6s</td>
</tr>
<tr>
<td>1</td>
<td>306</td>
<td>108</td>
<td>35%</td>
<td>198 (65%)</td>
<td>19s</td>
</tr>
<tr>
<td>2</td>
<td>312</td>
<td>111</td>
<td>36%</td>
<td>201 (64%)</td>
<td>20s</td>
</tr>
<tr>
<td>3</td>
<td>315</td>
<td>112</td>
<td>36%</td>
<td>203 (64%)</td>
<td>21s</td>
</tr>
<tr>
<td>4</td>
<td>315</td>
<td>112</td>
<td>36%</td>
<td>203 (64%)</td>
<td>21s</td>
</tr>
</tbody>
</table>

Table 4.6: Results for a month’s work – Apache Commons CLI

### 4.5.2 Results for Industry Case Study

<table>
<thead>
<tr>
<th>Depth</th>
<th>Mutants</th>
<th>Killed</th>
<th>Mutation Score</th>
<th>Unkilled</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63s</td>
</tr>
<tr>
<td>Traditional</td>
<td>954</td>
<td>577</td>
<td>60%</td>
<td>377 (40%)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>57</td>
<td>42</td>
<td>74%</td>
<td>15 (26%)</td>
<td>4s</td>
</tr>
<tr>
<td>1</td>
<td>182</td>
<td>140</td>
<td>77%</td>
<td>42 (23%)</td>
<td>23s</td>
</tr>
<tr>
<td>2</td>
<td>191</td>
<td>149</td>
<td>78%</td>
<td>42 (22%)</td>
<td>24s</td>
</tr>
<tr>
<td>3</td>
<td>191</td>
<td>149</td>
<td>78%</td>
<td>42 (22%)</td>
<td>24s</td>
</tr>
</tbody>
</table>

Table 4.7: Results for a day’s work – Industry
Call graph generation time for both case studies was 8.2 seconds and 14.5 seconds respectively. This is part of the execution time from depth 1 onwards. In the following section, we are going to discuss the results obtained.

### 4.6 Discussion

The aim of this work is to answer the question: “Can the effectiveness of incremental mutation testing be improved?” The results recorded in Section 4.5 were analysed and in the following sections, a statistical overview of the results obtained.
4.6.1 Execution Time

The first objective in this work was: “Is the increase in time taken to retrieve the dependent parts feasible?” The execution time in both case studies will now be discussed in detail in order to address this objective. The performance gain or loss in Tables 4.10 and 4.11 is calculated using the following formula:

\[
\text{Performance Gain/Loss} = \frac{\text{Execution Time Before}}{\text{Execution Time After}}
\]

Figure 4.2: Calculating the performance gain/loss

The performance gain in Figure 4.2 is defined as how many times incremental mutation testing, combined with our tool for call graph generation, is faster than the traditional mutation testing method. The performance loss is defined by how many times this new method is slower than naïve incremental mutation testing, also defined as depth 0.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Performance Gain from Traditional Mutation Testing</th>
<th>Performance Loss from Naïve Incremental Mutation Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Depth 1</td>
<td>Depth 2</td>
</tr>
<tr>
<td>Day</td>
<td>1.11x</td>
<td>1.11x</td>
</tr>
<tr>
<td>Week</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>Month</td>
<td>0.80x</td>
<td>0.76x</td>
</tr>
<tr>
<td>Mean</td>
<td>0.97x</td>
<td>0.96x</td>
</tr>
</tbody>
</table>

Table 4.10: Performance gain from traditional mutation testing vs performance loss from naïve incremental mutation testing – Apache Commons CLI

is shown and discussed in relation to the objectives which will help us answer our aim.
Chapter 4. Evaluation

Performance Gain from Traditional Mutation Testing

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Depth 1</th>
<th>Depth 2</th>
<th>Depth 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>2.74x</td>
<td>2.63x</td>
<td>2.63x</td>
</tr>
<tr>
<td>Week</td>
<td>1.62x</td>
<td>1.62x</td>
<td>1.62x</td>
</tr>
<tr>
<td>Month</td>
<td>1.17x</td>
<td>1.17x</td>
<td>1.17x</td>
</tr>
<tr>
<td>Mean</td>
<td>1.84x</td>
<td>1.81x</td>
<td>1.81x</td>
</tr>
</tbody>
</table>

Performance Loss from Naïve Incremental Mutation Testing

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Depth 1</th>
<th>Depth 2</th>
<th>Depth 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>5.75x</td>
<td>6.00x</td>
<td>6.00x</td>
</tr>
<tr>
<td>Week</td>
<td>1.56x</td>
<td>1.56x</td>
<td>1.56x</td>
</tr>
<tr>
<td>Month</td>
<td>1.54x</td>
<td>1.54x</td>
<td>1.54x</td>
</tr>
<tr>
<td>Mean</td>
<td>2.95x</td>
<td>3.03x</td>
<td>3.10x</td>
</tr>
</tbody>
</table>

Table 4.11: Performance gain from traditional mutation testing vs performance loss from naïve incremental mutation testing – Industry

As can be seen from the tables above, the performance gain in execution time varied for both of the case studies. Taking the highest depth in the Apache Commons CLI library case study, the mean performance gain was that of 0.96x. This has actually shown us that our tool was slower than traditional mutation testing. The main reason behind this was the fast executing test suite that this library has. Therefore, for such a library, it is not worth the extra overhead of call graph generation, since it would generate less mutants than traditional mutation testing in a greater amount of time.

Different results were observed in the industry case study. Taking the highest depth in Table 4.11, we saw a mean performance gain from traditional mutation testing of 1.81x. This case study has a test suite which is far larger and takes a greater time to execute, therefore, the call graph generation overhead was justified in this case.

From both tables, we can observe that the highest performance gain was obtained in the day experiment, with this decreasing when the lapse of time between
codebase revisions increased. This can also be seen in the case of the Apache Commons CLI library, whose day experiment resulted in a 1.11x performance gain, albeit almost negligible. This was the only exception in this case study where our tool was faster than traditional mutation testing. Therefore, from the results obtained, one can conclude that it would be best to use the tool on codebases with a day’s time lapse between revisions.

The performance loss in speed compared to naïve incremental mutation testing ranged from as low as 1.54x to as high as 6x. Apart from the added time taken to generate and analyse the call graph, the reason was directly related to the amount of mutants which are generated in naïve incremental mutation testing compared to the number of mutants generated when the call graph is being traversed. Taking the case where it was 6x slower, more than double the mutants were generated when comparing depth 0 and depth 2 of Table 4.7. On the other hand, taking the case where it was 1.54x slower, the increase in mutants was of only 28, because the majority of methods found with the call graph were already part of the changed method set.

From both tables, it can be seen that some results are exactly the same along several different depths. This is because either there was no increase in mutants between depths or because the increase was so minimal that the execution time was not affected.

### 4.6.2 Mutant Increase Analysis

The other objective we had to address was: “Will including the dependent parts substantially increase the amount of mutants generated?” This was important for our work because if the mutants did not increase, the extra time taken to build and analyse the call graph would have been futile. As we increased the depth in analysing the call graph, we expected that the number of mutants would also increase. The following formula was used in order to calculate the percentage increase in mutant count between different depths:
Chapter 4. Evaluation

Mutant Percentage Increase = \frac{Mutant Count After - Mutant Count Before}{Mutant Count Before} \times 100

Figure 4.3: Calculating the mutant percentage increase

Figure 4.4: Graph of the number of mutants against the depth – Apache Commons CLI

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Depth 0-1</th>
<th>Depth 1-2</th>
<th>Depth 2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>28.42%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Week</td>
<td>14.75%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Month</td>
<td>93.67%</td>
<td>1.96%</td>
<td>0.96%</td>
</tr>
<tr>
<td>Mean</td>
<td>45.61%</td>
<td>0.65%</td>
<td>0.32%</td>
</tr>
</tbody>
</table>

Table 4.12: Mutant percentage increase according to depth – Apache Commons CLI
Figure 4.5: Graph of the number of mutants against the depth – Industry

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Depth 0-1</th>
<th>Depth 1-2</th>
<th>Depth 2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>219.30%</td>
<td>4.95%</td>
<td>0%</td>
</tr>
<tr>
<td>Week</td>
<td>19.60%</td>
<td>0.70%</td>
<td>0%</td>
</tr>
<tr>
<td>Month</td>
<td>4.00%</td>
<td>0.40%</td>
<td>0%</td>
</tr>
<tr>
<td>Mean</td>
<td>80.97%</td>
<td>2.02%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.13: Mutant percentage increase according to depth – Industry

The graphs in Figures 4.4 and 4.5 represent the increase in the number of mutants with respect to the depth with which the call graph was being analysed. From these graphs, Tables 4.12 and 4.13 were obtained. These tables show that
the largest increase in mutants occurs when going from naïve incremental mutation testing (depth 0) to depth 1. In both case studies, there is a mean mutant count increase of 45.61% and 80.97% respectively. Delving further than depth 1 showed very small increases, with the highest being 4.95%. There are several reasons behind this. First of all, the depth in which a method is found is dependent on the changed method, as a method can be at depth 1 for a particular method and at depth 2 for another method. Because of this, the same method can be retrieved more than once. Therefore, it is only mutated and tested the first time that it is encountered, since otherwise, it would only add unnecessary overhead by providing the same results as when it was mutated and tested the first time. Second of all, another reason which can contribute to the small increase after depth 1, is that the methods found can also be part of the original set of changed methods.

The two studies had contrasting results in the experiment representing a month’s worth of work. The Apache Commons CLI library case study had a 93.67% increase in mutants at depth 1 while the industry case study only had a 4.00% increase. The main reason for this is because in the former, changes that were made were not connected to one another, meaning that many methods found at depth 1 were not changed during that month. Contrary to the former, the small increase in the latter’s month experiment meant that most new methods found at depth 1 were also changed during the same month, and therefore, already included in the naïve incremental mutation testing process without call graph generation.

4.7 Threats to Validity

As with all case studies, several threats to validity exist which can question the results obtained. These can be both threats to internal and threats to external validity.
4.7.1 Internal Validity

Having internal validity means that there is a direct correlation between your action and the outcome. Therefore, threats to internal validity are issues that question the conclusions drawn from the results obtained [9]. Barros et al. [4] mention several validity threats which are applicable to this work:

- **Lack of discussion on code instrumentation**  The source code used in an experiment can have certain tweaks or instrumentations which favour the results obtained. Johnson [26] recommends that the source code used in experiments is made public so that other users can experiment with it. This is possible with the Apache Commons CLI library since it is open source, but this is not possible with the industry case study due to confidentiality, which limits other researchers from using this case study to reproduce and criticise it.

- **Lack of clear data collection tools and procedures**  Continuing from the previous one, this threat addresses the lack of detail in the steps executed to collect the desired results, which may show a bias in selecting favourable instances. Whilst all the details of the procedure are provided in this work’s evaluation, the revision numbers for the industry case study could not be made available due to confidentiality reasons.

These are the internal validity threats applicable to this work. While the utmost was done in order to show a completely transparent process, the cause for these threats is ultimately unavoidable.

4.7.2 External Validity

External validity refers to the applicability of this work outside the scope of the case studies used for evaluation [9]. As with all evaluations, this evaluation suffers from the external validity threat in which the results obtained cannot be generalised.
Due to time constraints, it was not possible to perform more case studies, and even though the results obtained have a relationship between each other, we cannot say for certain that this will hold for all the case studies that can be performed [16].

### 4.8 Conclusions

The aim of this work was to answer the question: “Can the effectiveness of incremental mutation testing be improved?” In order to answer this question, two objectives determined whether the effectiveness of incremental mutation testing was improved.

The first objective was: “Is the increase in time taken to retrieve the dependent parts feasible?” From the two case studies, we have seen that it is feasible to use our tool on a codebase which has a test suite that consumes a lot of time to execute, such as the industry case study. On the contrary, the overheads of the call graph generation on a codebase that has a fast executing test suite, such as the Apache Commons CLI library, resulted in an execution time similar to that of traditional mutation testing, rendering our tool unfeasible. Thus, we can hypothesise that the longer the tests take to execute, the greater the performance gain our tool will have.

The second objective was: “Will including the dependent parts substantially increase the amount of mutants generated?” This objective was important to justify the call graph generation by having an increase in mutants from naïve incremental mutation testing. From both of the case studies, we have seen that the number of amounts has substantially increased when going from depth 0 to depth 1. Further increasing the depth resulted in little to no increase in most cases, since the methods which were retrieved could have already been found and mutated in previous depths, or were part of the changed methods set.

Regarding the mutation score, there is no particular pattern as to whether it increases or decreases according to the approach chosen. The mutation score is
highly dependent on how well-tested the parts chosen for mutation are, something which is completely in the hands of the developers.

In [8], a study was carried out which showed that in order to fix a bug, 50–90% of the time is spent on comprehending the code, since the maintainer is rarely the author. Also, in [40] it is said that the cost of fixing a bug during the testing stage is at least three times more expensive than fixing it during the development stage. On top of that, fixing a bug after release is also at least fifteen times more expensive. Hence, it is more cost effective and less time consuming to fix bugs during the development stage.

In order to fix the bugs during the development stage, it is best to perform mutation testing multiple times until enough tests are written, or fixed, that can kill a satisfactory amount of the mutants generated. The additional methods included in incremental mutation testing with the call graph generated can help find shortcomings in a test suite which were previously not found using naïve incremental mutation testing. Thus, the discovery and fixing of these shortcomings would essentially aid in decreasing the cost of development, by having a thorough enough test suite which catches bugs that may be introduced by the developer.
5. Related Work and Conclusions

In this chapter, we will discuss some related work in the field of testing and then provide a summary of the work done in this area, along with some suggested future improvements. Finally, some concluding remarks regarding the expectations of this work are expressed.

5.1 Related Work

Several related work dedicated to making mutation testing more feasible already exists. These are the cost reduction techniques called “do smarter”, “do faster” and “do fewer” which are discussed in Section 2.2.2. These approaches aim to decrease the number of mutants generated or execute the mutants at a faster speed.

However, no work could be found on using static analysis techniques to improve the effectiveness of incremental mutation testing. Similar use of static analysis techniques was found in the area of regression testing [22, 41], which is a technique performed after an improvement or a fix is applied to a program. Its main purpose is to determine whether the change done to a program has negatively affected, or regressed, other parts of the same program. A subset of the test cases are usually rerun which cover the changes made. It is important to perform regression testing on a system because correcting errors found in the original source code tends to be much more error prone in itself [36]. The difference from mutation testing is that
in regression testing, bugs are found in the source code, while mutation testing is used to find shortcomings in the test suite.

In [41], internal representations which represent the control and data dependency are used in order to show the relation between classes. These are used to find sections of code that are affected when changes are done between two versions of a codebase. Their particular tests are then found from the test suite and run to check for regressions. The similarity to our work is that we are finding methods that call the changed methods, mutating them, and then executing the tests.

A regression test selection system called Retest is introduced in [22], which selects a subset of tests in a test suite in order to reduce the computational expense of regression testing. It uses a Java Interclass Graph (JIG), which is an extension of a CFG that accommodates Java language features, along with static analysis techniques, to find test cases from the original test suite that might expose faults from the new changes done to a program. This also bears some similarity to our work, since we are using a call graph to retrieve the callers of a changed method and mutate them. The tests are then executed to check for any shortcomings introduced in the test suite by these mutated methods, since the method callers may behave differently when the method they call is changed. Therefore, we can say that whereas Retest finds test cases that can expose faults in the source code, we are finding methods which, when mutated, can expose shortcomings in the test suite.

5.2 Conclusions

In this work, progress was made in order to make incremental mutation testing more effective. This was done by creating a tool which extends incremental mutation testing by also including the method callers of the changed methods in mutation analysis.

Through research, it was discovered that the dependencies of the changed meth-
ods could be extracted using static analysis on source code. Several internal rep-
resentations exist which give information regarding the source code of a program. 
Ultimately, a context-insensitive call graph was used because it provided the right 
amount of information that was needed, that is, the inter-procedural flow between 
functions of a system.

Soot [34, 47] was the bytecode engineering framework of choice which was used 
in order to generate the call graph and Mutatinc [6] was the mutation testing tool 
used, since it is the only one which has the functionality of incremental mutation 
testing. The changed methods which were being detected by Mutatinc had to have 
the methods that call them extracted using the call graph which is generated by 
Soot.

This led to the design of a tool which consolidated all the required functionality. 
The tool created was organised in several modules. The call graph builder module 
uses Soot to build the call graph by giving it the application classes of the project 
used. The signature converter module converts the bytecode signature to make it 
compatible with Soot and converts it back to make it compatible with Mutatinc. 
Finally, the method caller detector module traverses the call graph to find the 
method callers of the changed methods up to the specified depth.

Challenges were encountered during the development phase of the tool, mainly 
including the time taken to generate a call graph and the integration of the tool 
with Mutatinc. These were overcome and the time taken to generate a call graph 
was greatly reduced and a seamless integration with Mutatinc was implemented.

An evaluation was then carried out on the tool, based on the increase in mutants 
according to the depth traversed in the call graph and the execution time. The 
number of mutants considerably increased, especially on depth 1 of the call graph. 
The performance gain in execution time was determined by how much time the test 
suite of the codebase took to execute. From the Apache Commons CLI library case 
study, we observed that it was not worth using our tool for call graph generation, 
since the test suite did not take a long time to execute. On the other hand, from the
industry case study, we observed a considerable performance gain from traditional mutation testing, signifying that as test suites have a longer execution time, the performance gain increases.

With the addition of the tool, incremental mutation testing remained efficient when used on codebases which have time consuming test suites, and at the same time had its problem of not including the dependent methods addressed. Therefore, it is still feasible to be used in these cases, and at the same time helps developers to detect more bugs during the development stage.

5.3 Future Work

There are several avenues of exploration to further improve the effectiveness of incremental mutation testing. The following are some improvements which could be made to our tool or Mutatinc in order to make this new approach more efficient:

**Faster call graph generation** In this work, the utmost was done in order to generate the call graph as fast as possible. Further updates to Soot can result in a more efficient way for the call graph to be generated, and thus further reduce the execution time. Mechanisms such as call graph caching could also be introduced. This would eliminate rebuilding the call graph when changes are introduced to the source code, by adjusting the call graph according to the changes done.

**Reachability analysis** Certain areas of the source code may never be accessible during program execution. Therefore, such an analysis can be conducted so that methods which can never be called are excluded from the mutation testing process [5].

**Identification of other types of coupling** Using the call graph, or other internal representations, the interconnection between methods using other types of
coupling can be found. For example, using common coupling to find out the effect a change in a global variable can have on the methods that use it.

**Expanding call graph to test methods**  Currently, Mutatinc retrieves the tests by searching for test methods in the source code. Extending the call graph to retrieve the tests and modifying Mutatinc to use the tests obtained from the call graph can be a more efficient way of retrieving the tests.

**Integration with an IDE**  The encapsulation of the tool within an IDE would make it more accessible to developers and easier to use, similar to how current code coverage tools are integrated.

### 5.4 Concluding Remarks

The successful implementation of this tool and the subsequent evaluation showed that it was possible to improve the effectiveness of incremental mutation testing when considering codebases that contain time consuming test suites. With these results, we hope that such a tool and technique will encourage the use of mutation testing throughout the software development industry.
A. Appendix

In the CD attached, the following resources can be found:

- The Apache Commons CLI library revisions used in the case study.
- The source code of the tool implemented.
- The source code of the modified Mutatinc with our tool integrated.
- A text file explaining how to use the tool.
- A batch file which executes the runnable JAR file of Mutatinc with our tool integrated.
- A soft copy of the FYP report.
References


References


References


