Mutation Analysis: An Industrial Experience Report

Ali Parsai

Principal Adviser: Prof. Serge Demeyer
Assistant Adviser: Quinten D. Soetens

Dissertation Submitted in January 2015 to the Department of Mathematics and Computer Science of the Faculty of Sciences, University of Antwerp, in Partial Fulfillment of the Requirements for the Degree of Master of Science.
Dedicated to my mother, who has always supported me, no matter which direction I take;
And to my grandfather, who had always encouraged me on the path to knowledge; May he rest in peace.
## Contents

List of Figures iii  
List of Tables v  
Acknowledgements vi  
Abstract vii  

1 Introduction 1  

2 Background 6  
   2.1 Testing in Continuous Integration Environment ............. 6  
   2.2 Mutation Analysis ........................................ 9  
      2.2.1 Mutation Operators .................................... 9  
      2.2.2 Equivalent Mutants .................................. 12  
      2.2.3 Mutant Sampling ...................................... 14  
   2.3 Code Coverage ............................................. 14  

3 Target System 16  
   3.1 Segmentation Component .................................... 16  
      3.1.1 Build System Structure ................................ 17  
      3.1.2 Dependency Handling .................................. 18  
   3.2 Practical Complications of Mutation Analysis ............. 18  
      3.2.1 Build System Complexities ............................ 18  
      3.2.2 Problems with Available Tools ...................... 19  
      3.2.3 Proposed Solutions ................................... 20  
      3.2.4 Required Criteria for the Tool ..................... 21
List of Figures

2.1 A typical Continuous Integration environment ....................................... 7
2.2 Mutation analysis procedure ................................................................. 10
2.3 Mutation operators implemented in tools ............................................... 12
2.4 Example for emergence of equivalence due to the context ..................... 13
2.5 Branch coverage ...................................................................................... 15
3.1 Agfa HealthCare Segmentation component build structure ..................... 17
4.1 LittleDarwin flow chart ........................................................................... 25
4.2 LittleDarwin components ......................................................................... 27
4.3 Mutant Sampler selection rate .................................................................. 28
4.4 LittleDarwin easy-to-read report (requirement 4) .................................... 31
4.5 LittleDarwin individual source file report (requirement 5) ....................... 32
4.6 Comparison between PITest, LittleDarwin and branch coverage ............. 32
4.7 Mutation coverage of complete set vs. randomly selected sample .......... 33
4.8 Example of difference in branch coverage and mutation coverage ........... 34
5.1 Mutation coverage results of classes in Segmentation component .......... 39
5.2 Mutation coverage results separating the Category 4 classes ................ 40
5.3 Mutation coverage results of classes in jOpt Simple ............................... 41
5.4 Mutation coverage results of classes in Commons Codec ....................... 41
5.5 Categorization results using different threshold values .......................... 42
5.6 Distribution of classes by number of generated mutants ........................ 43
5.7 Mutation coverage results for full set and sampled set .......................... 44
5.8 Difference in mutation coverage between full set and sampled set .......... 46
B.1 Article selection procedure ..................................................................... 66
B.2 Mothra mutation operators (adapted from [34]) .................................... 67
List of Tables

2.1 Reduced-set mutation operators (adapted from [46]) . . . . . . . 10
2.2 Object-oriented mutation operators (adapted from [44]) . . . . . 11
4.1 LittleDarwin mutation operators . . . . . . . . . . . . . . . . . 27
5.1 Summary of experiment results . . . . . . . . . . . . . . . . . 43
I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this thesis. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the thesis.

I express my warm thanks to Prof. Demeyer and Mr. Soetens for their support and guidance during the course of the project. I also appreciate the help and guidance of Dr. Murgia, Dr. Perez, and the rest of my colleagues in the Ansymo group for their helpful advice.

I would also like to thank Mr. De Witte from Agfa HealthCare for clarifying the problems and paving the way to success of this research, and all the people who provided me with the required facilities and conducive conditions for my thesis.

Finally, I would like to thank my family and my friends for their moral support during the past months.

Thank you,
Ali Parsai
Abstract

In order to assess the ability of a test suite to catch bugs, a quality metric is needed which can simulate realistic situations. Mutation analysis provides this metric with a reliable and repeatable approach. However, because of the computationally intensive nature of mutation analysis and the difficulties in applying such a technique to complex systems, it has not been widely adopted in industry. This study aims to determine the feasibility of using this technique on an industrial case, and to find out if the information gathered by this method is worth the performance costs.
One of the major goals in the software development process is quality assurance. To make sure that the software fulfills the quality standards it is supposed to have and avoid any unintentional behavior, thorough testing of the software is needed. Small software systems can easily be tested manually, however, for larger software systems that consist of complex and enormous structures manual testing is infeasible. Thus, automated testing of software became a standard. Automated tests are usually collected together in a test suite.

It is necessary to measure the quality of the test code in order to maintain a high quality test suite, which in turn, assures sufficient testing of the software. As software systems grow larger and older, it becomes more important to assess the quality of the code, if the code is suspected to be of low quality e.g. open source projects written by community or legacy code inherited from another company. There are simple metrics to estimate this quality. Statement coverage determines the percentage of executed statements by test code. In a similar fashion, Branch coverage determines the percentage of the branches of code that are executed by the test code. A branch is created in a program when a control statement (e.g. if or case statements) provides two or more paths of execution. These metrics provide an overview of the quality of the test suite in an easily attainable manner; Yet, they are inadequate in their purpose of estimating quality [71]. Therefore, building confidence on the final product based on such simple metrics increases the probability of bugs not being caught. Even a 100% branch coverage would leave a lot of room for a bug to escape [49]. Furthermore, branch coverage is also a poor measure to determine a detailed map of the weaknesses in a test suite because first, it
lacks the ability to discover which type of bugs are being caught and which are not and second, it is difficult for practical tools to trace the execution paths during the runtime of complicated software systems. This brings about practical problems for companies wishing to use these metrics to establish their strategy on how to improve the quality of their test suites. Hence, there is a need for more powerful metrics, not only to find out the quality of the test suite accurately, but also to precisely pinpoint the potentially problematic parts of the test suite.

Mutation testing offers another metric by introducing a repeatable and scientific method of measuring the quality of the test suites. This approach consists of two phases; First, generating buggy versions of the code by injecting a single bug into the code (creating a mutant) and then, executing the tests on this buggy version of the code to determine the outcome. The output of mutation testing would be a quality metric defined by the percentage of the bugs that resulted in failure of at least one test (killed mutants) by the total number of created mutants. This metric is proven to simulate the bugs realistically [4; 38]. This is due to the fact that the bugs introduced into each mutant are modeled after the common mistakes which are often made by developers. Because this metric relies on a fault model for its calculation, it often produces more accurate results than simple coverage criteria if the model used to produce the bugs is close to reality [5]. Moreover, because every mutant is instantly traceable to the part of the code it has changed, it is easy to obtain a detailed map of where the uncaught bugs are.

There are, however, downsides on using mutation analysis. One that has limited its functionality in industrial practices is its computationally intensive nature [25]. So, it is not a suitable technique to use regularly without any further optimization. Another problem with mutation testing is that the scores are dependent on the type of tool used and the model used to generate bugs. Therefore, any change in the mutation analysis tool would make the comparison between new results and the older ones difficult; and general mutation analysis tools may need a specialized model to deliver the expected accuracy for a certain special-purpose case.

There are many software industries which require higher quality standards for their products; e.g. medical software or military industries. To ensure the best quality of the ultimate product, testing the software thoroughly is a necessity [68]. Hence, it is important to have an up-to-date, complete and adequate test suite for each component of the software. This demand makes mutation analysis a viable candidate as a metric to find the weak spots by giving importance to quality instead of time and cost. Because of the popularization of the agile development methods, the common practice in industry is to use continuous integration tool chains as the backbone of their development cycle. For any technique to be useful, it is crucial to have a tool that can integrate
into such tool chains. Still, it is worth mentioning that there are no industrial
grade mutation analysis tools that works out of the box. For that reason, it is
not immediately clear if it is possible to use mutation analysis on an industrial
system. Most of the time, such systems include internal technologies which
might not be readily supported by academic prototypes. This leads us to the
first question we try to answer:

- **RQ1.** Is it feasible to apply mutation analysis on an industrial project?

Mutation analysis has been around since 1978 as a test quality metric. Even
though it was an important topic of research, only a handful of industrial cases
were subject to mutation analysis. This is attributable to several reasons:

- Many of the developers working in the industry are unfamiliar with mu-
tation analysis and its benefits or weaknesses [26].

- Mutation analysis is computationally intensive. As a result, many in-
dustries tend to opt for less complicated measures like code coverage.

- There are no mature tools available for the programming languages used
frequently in industry. Consequently, no long-term plan can be formed
based on a single tool; while the main use case of mutation analysis is to
form a long term test design strategy [34].

Nevertheless, there is a case to be made for the practicality of mutation
analysis in an industrial setting. There are many companies that have limited
resources to maintain a product they have been developing for years, and as a
result, they are not capable of a total refactoring of their product. Therefore,
analytic techniques like mutation analysis provides an opportunity to optimally
detect the areas of weakness in their test suite. Even if it takes a long time
to analyze every test suite in a product, it is still advisable to form long-
term plans based on mutation analysis in case of refactoring legacy code, or to
determine the test suite quality where code coverage alone has been proven to
be inadequate in practice. The applicability of the available tools can be tested
on the product to acquire results even if it is only possible for a small part of
the product. Even though there are many software companies which put time-
to-market in front of quality on their agenda, there are also many industries
(such as the health industry) that consider quality as a critical factor. Thus, it
is important to determine the feasibility of applying this technique in industry.

The goal that we try to reach by answering this question is to demonstrate
that it is possible to use mutation analysis even though the system under test
has a complex software structure. It is important that the resulting mutation
analysis tool is capable of adapting to different software structures usually
found within a company. This would encourage other companies to adopt the technique regardless of the complexity of their software.

To answer this question, Agfa HealthCare Segmentation component was chosen as the target project. This component is used as a part of Agfa HealthCare Impax ES and provides imaging algorithms to perform image segmentation on 3D modeled volumes. This software system is a fair representation of the challenges in the industry for mutation analysis. Its complex structure allows to examine the compatibility problems with existing tools. The continuous integration development environment that this software system is being developed in, provides an opportunity to assess the usefulness of mutation analysis in this context. Due to the complexity of the build system of this component, none of the available mutation analysis tools could be used to perform the analysis; Therefore, a new tool called LittleDarwin\(^1\) was developed for this purpose.

The other issue that we try to investigate, is to answer if it is worth it to use mutation analysis instead of simple metrics in the system. On the one hand, mutation analysis can provide detailed information necessary to detect flawed tests using an appropriate fault model, though, with extremely high computational demand; On the other hand, code coverage metrics are very easy to calculate, but they only provide a vague picture of the flaws in a test suite, bringing their usefulness in improving the quality of the test suite in question. Therefore, the choice of the metric depends both on the system under test, and the resources available to the development team. While there has been a great emphasis on developing practical and stable tools that can calculate different sorts of code coverage metrics for various platforms and systems, such effort has not been present in developing similarly practical mutation analysis tools, leaving the extent of the performance versus accuracy trade-off unknown.

Most of the industries rely on branch coverage as the quality metric of choice [26]; because it can be calculated quickly during the build process and it provides a more meaningful metric than statement coverage for the developer by giving importance to the control structures rather than just counting the covered statements. On that account, branch coverage was chosen as the simple metric of choice to be compared with mutation coverage. Hence, the question we try to answer is as follows:

- **RQ2.** What are the trade-offs involved in mutation analysis vs. branch coverage for determining test suite quality?

The trade-off between performance and preciseness is also a point to consider. To be able to decide if mutation analysis is practical in a certain context,

\(^1\)http://littledarwin.parsai.net
there must be a small scale study first to determine how often it is possible to perform the analysis; and if the cost is worth the extra information gathered using this technique.

By answering this question, we try to gather practical evidence on the benefits and disadvantages of mutation analysis in comparison to branch coverage in delivering detailed information about the quality of the test suite. This would provide a fair assessment of mutation analysis, making it easier to decide if it is desirable to use in a software project.

To find an answer, the results of our tool are compared to the available branch coverage results from the default tool currently being used in the continuous integration tool chain of the system under test.

The rest of this report is structured as follows: In Chapter 2, a brief overview of mutation analysis and code coverage is given. In Chapter 3, the rationale for developing a new tool is discussed. In Chapter 4, the design specifications of our tool is described. In Chapter 5, you can find information about the experimental setup and the analysis of the results. And finally, in Chapter 6, our final conclusions are discussed.
CHAPTER 2

Background

This chapter aims to provide an overview of the background information necessary to understand the concept of mutation analysis and the code coverage, in addition to the reasons of their importance in the current state of industrial development environments. In Section 2.1, Continuous Integration environments are briefly introduced, and the importance of the test suite quality is discussed. In Section 2.2, the mutation analysis is explained in detail, and related concepts, which are considered in development of LittleDarwin, are discussed. Section 2.3 is a brief explanation about code coverage metrics relevant to this research.

2.1 Testing in Continuous Integration Environment

One of the major goals in software engineering has always been to provide developers with tools that can automate the mundane tasks, and as such, ease the software development process. One of those mundane tasks is the testing of a system. This task is indispensable in the process of developing a software system as it helps to find and fix bugs. With the popularization of agile development during the past decade, the nature of software testing has transformed from being an auxiliary aspect of development into one of the foundations of the software development process. Incorporation of the agile concepts, however, demand the development process to change. Hence, Continuous Integration has replaced the older development processes.

Continuous Integration is defined as the practice of merging the developed
code with a central source code repository as often as possible. The concept of Continuous Integration was first proposed by Booch [9] as a way to avoid integration problems. This method has been in the center of attention in the past decade as a result of its adoption as a basis for agile development techniques. A common structure for Continuous Integration environment is shown in Figure 2.1. As shown in this figure, the developer commits the local changes into the repository, and retrieves the updated code developed by other developers. Then, according to the schedule, the modified code is automatically built by the build server using the build scripts; and the automated test suite among other tools would be run on the resulting build. The results are then available for the developer’s consideration.

There are several tool chains developed which provide an implementation of the described procedure. For example, Jenkins\(^1\) is an extensible open source continuous integration server widely used in the industry. Subversion and GIT are popular version control systems providing the functionality of a source code repository. Jira\(^2\) is a successful commercial product for the scheduling of tasks and issue reporting. These tools can be used in combination to provide a smooth automated development environment. There are also build systems which are designed to build the software, execute the test suite and generate

\(^{1}\text{http://jenkins-ci.org/}\)

\(^{2}\text{http://www.atlassian.com/software/jira}\)
2.1. TESTING IN CONTINUOUS INTEGRATION ENVIRONMENT

reports. Maven\(^3\) is an example of such build systems currently employed for Java software projects. Another famous example of such systems are those which are based on GNU Make\(^4\). In principle, it is possible to use Make to produce build scripts for various types of software projects. The existence of multiple options for any given situation allows the project managers to design their Continuous Integration environment according to the demands of their projects.

The Continuous Integration allows the software to be tested as often as possible. The preferred testing strategy is to use the unit test suite before committing to the main code repository, to discover test failures as early as possible \([51]\). As a result, the test suite has become a crucial part of the development cycle; Even to the point that it becomes the basis of the development in methods such as Test-Driven Development. As a part of Extreme Programming techniques, Beck suggests that the requirements should be turned into tests, and the development should be driven by tests \([8]\). This is known as Test-Driven Development \([7]\). In his book about Test-Driven Development, Astels \([6]\) proposes that this concept can achieve better test coverage.

The focus on testing since the introduction of agile development techniques has resulted in the increased interest in the quality of the test suite. Consequently, the test suite quality measures have become an important topic in both industry and academia; as a practical means to improve the effectiveness of their testing practices in the former, and for the methodology of agile practices and comparison of different testing strategies in the latter. Because of the focus on resources in the industry, the code coverage metrics have been their main choice as a test suite quality measure. While, the focus on accuracy of the metrics in academia resulted in the popularity of mutation analysis as a comparison criterion in many scientific studies.

It is, therefore, crucial to provide a solution to integrate mutation analysis in a Continuous Integration cycle to make it practical in an industrial environment, thus creating the opportunity to make use of its benefits. This, however, is not an easy task at all. The overall process of building and testing the software is often very complicated, and the analysis tools which are run during the build process are not easily replaceable. In addition, lack of commercial investment in development of mature mutation analysis tools has resulted in disregard for the requirements of such tools in major build management softwares. This, in turn, has made the development of the new mutation analysis tools particularly harder, and their easy integration highly unlikely. Therefore, introduction of a tool capable of easy integration is the first step in popularizing the use of mutation analysis; thus, encouraging the developers of the major build management softwares to provide built-in support for the

\(^3\)http://maven.apache.org/
\(^4\)http://www.gnu.org/software/make/
2.2 Mutation Analysis

Mutation analysis is the process of injecting bugs into software, and counting the number of intentional bugs which make at least one test fail. The idea of mutation testing was first mentioned in a class paper by Lipton [54] and later developed by DeMillo, Lipton and Sayward [17]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [12]. This procedure is executed in the following manner: First, faulty versions of the software are created by introducing a single bug into the system (Mutation). This is done by applying a known transformation on a certain part of the code (Mutation Operator). After generating the faulty versions of the software (Mutants), the test suite is executed on each one of these mutants. If there is an error or failure during the execution of the test suite, the mutant is regarded as killed. On the other hand, if all tests pass, it means that the test suite could not catch the bug and the mutant has survived. This procedure demands a green test suite — a test suite in which all the tests pass — to run correctly. An overview of this procedure can be observed in Figure 2.2.

\[
\text{Mutation Coverage} = \frac{\text{Killed Mutants}}{\text{Killed Mutants} + \text{Survived Mutants}} \quad (2.1)
\]

The final result is calculated using Equation 2.1. This metric provides a more detailed image of the quality of a test by reporting on its results instead of simple coverage. This makes sure that the kind of bugs simulated by mutation operators are covered by the test; Therefore reducing the chance of missing such bugs in the final product.

2.2.1 Mutation Operators

A mutation operator is a known transformation which can transform a part of code into a buggy version by introducing a single change. The first set of the mutation operators designed were reported in [40]. These operators which work on very basic entities were designed for the tool Mothra which was designed to mutate FORTRAN77 programming language. In 1996, Offutt et al [53] determined that a selection of few mutation operators are enough to produce the same coverage with a four-fold reduction of the number of mutants. This reduced set of operators remained more or less intact in the future research papers (Table 2.1).
2.2. MUTATION ANALYSIS

Figure 2.2: Mutation analysis procedure

Table 2.1: Reduced-set mutation operators (adapted from [46])

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>AOD</td>
<td>Arithmetic Operator Deletion</td>
</tr>
<tr>
<td>AOI</td>
<td>Arithmetic Operator Insertion</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>COR</td>
<td>Conditional Operator Replacement</td>
</tr>
<tr>
<td>COD</td>
<td>Conditional Operator Deletion</td>
</tr>
<tr>
<td>COI</td>
<td>Conditional Operator Insertion</td>
</tr>
<tr>
<td>SOR</td>
<td>Shift Operator Replacement</td>
</tr>
<tr>
<td>LOR</td>
<td>Logical Operator Replacement</td>
</tr>
<tr>
<td>LOD</td>
<td>Logical Operator Deletion</td>
</tr>
<tr>
<td>LOI</td>
<td>Logical Operator Insertion</td>
</tr>
<tr>
<td>ASR</td>
<td>Assignment Operator Replacement</td>
</tr>
</tbody>
</table>
## 2.2. MUTATION ANALYSIS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC</td>
<td>Access Modifier Change</td>
</tr>
<tr>
<td>IHD</td>
<td>Hiding Variable Deletion</td>
</tr>
<tr>
<td>IHI</td>
<td>Hiding Variable Insertion</td>
</tr>
<tr>
<td>IOD</td>
<td>Overriding Method Deletion</td>
</tr>
<tr>
<td>IOP</td>
<td>Overridden Method Calling Position Change</td>
</tr>
<tr>
<td>IOR</td>
<td>Overridden Method Rename</td>
</tr>
<tr>
<td>ISK</td>
<td>super Keyword Deletion</td>
</tr>
<tr>
<td>IPC</td>
<td>Explicit Call of a Parent’s Constructor Deletion</td>
</tr>
<tr>
<td>PNC</td>
<td>new Method Call with Child Class Type</td>
</tr>
<tr>
<td>PMD</td>
<td>Instance Variable Declaration with Parent Class Type</td>
</tr>
<tr>
<td>PPD</td>
<td>Parameter Variable Declaration with Child Class Type</td>
</tr>
<tr>
<td>PRV</td>
<td>Reference Assignment with Other Compatible Type</td>
</tr>
<tr>
<td>OMR</td>
<td>Overloading Method Contents Change</td>
</tr>
<tr>
<td>OMD</td>
<td>Overloading Method Deletion</td>
</tr>
<tr>
<td>OAO</td>
<td>Argument Order Change</td>
</tr>
<tr>
<td>OAN</td>
<td>Argument Number Change</td>
</tr>
<tr>
<td>JTD</td>
<td>this Keyword Deletion</td>
</tr>
<tr>
<td>JSC</td>
<td>static Modifier Change</td>
</tr>
<tr>
<td>JID</td>
<td>Member Variable Initialization Deletion</td>
</tr>
<tr>
<td>JDC</td>
<td>Java-supported Default Constructor Create</td>
</tr>
<tr>
<td>EOA</td>
<td>Reference Assignment and Content Assignment Replacement</td>
</tr>
<tr>
<td>EOC</td>
<td>Reference Comparison and Content Comparison Replacement</td>
</tr>
<tr>
<td>EAM</td>
<td>Accessor Method Change</td>
</tr>
<tr>
<td>EMM</td>
<td>Modifier Method Change</td>
</tr>
</tbody>
</table>

Table 2.2: Object-oriented mutation operators (adapted from [44])
2.2. MUTATION ANALYSIS

Figure 2.3: Mutation operators implemented in tools

With the popularity of the object-oriented programming paradigm, there was a need to design new mutation operators to simulate the bugs that occur only in these kind of programs. Several studies proposed new mutation operators [44] [13], and some of them were designed to prove the usefulness of object-oriented operators [42] [18]. Ahmed et al in [3] did a complete survey on this subject. As an example, object-oriented mutation operators which were developed into MuJava [45] can be seen in Table 2.2.

During the course of previous research internship, a literature survey on mutation analysis was performed (Appendix B) which resulted in the understanding that during the past decade, the focus of the researchers were on creating new mutation operators for special purposes such as targeting certain security problems [64] [75] or language specific mutation operators [1] [11] [65]. These mutation operators, even though important in their own context, do not relegate into the general concept of mutation analysis. An overview of the 25 tools produced during the last decade (Figure 2.3) shows that the classical operators are by far the most often implemented. One reason for this is that using more mutation operators (such as the ones mentioned in Table 2.2) produces much more mutants; which makes the procedure take longer to finish, and as a result, less practical. The reduced set of operators mentioned in Table 2.1 provides a small collection which produces results with enough detail for any practical purpose, even though the confidence in such results are less than those retrieved by using additional mutation operators.

2.2.2 Equivalent Mutants

If a mutant produces the same output as the original program for all input values, it is called an equivalent mutant. The creation of equivalent mutants is undesirable, but they are not easy to detect [47]. The problem is that the creation of such mutants depends on the context of the program itself. For example, in Figure 2.4, the introduced change in proc1 will change the
Figure 2.4: Example for emergence of equivalence due to the context
outcome, while the same change in proc2 does not. The error introduced in proc2 is, therefore, not detectable by any test. The existence of equivalent mutants would create false positives, since there is no way for a test suite to catch them. Therefore, it is advisable to manually check the results of the analysis if the mutation coverage is less than it is expected.

2.2.3 Mutant Sampling

To make mutation analysis practical, it is important to reduce the time it needs to run. One way to achieve this is to reduce the number of mutants. A simple approach to achieve this reduction is to randomly select a sample set of mutants instead of processing all of them. This idea was first proposed by Acree [2] and Budd [12] in their PhD theses. Since then, there were many studies confirming the effectiveness of this approach in improving performance [72; 50; 40].

The random selection rate can be a constant, or it can be dynamically adjusted based on the characteristics of the source file the mutants are generated for. A method resembling the latter was proposed by Sahinoglu and Spafford [59] to randomly select the mutants until the sample size becomes statistically appropriate. They concluded that their model achieves better results due to its self-adjusting nature [34].

2.3 Code Coverage

Code coverage is defined as the quantity of the code which is tested by the test suite. There are several ways to calculate code coverage. The most often used code coverage criteria are statement coverage and branch coverage. Statement coverage is the number of statements in the program that are executed by the test suite divided by the number of total statements. Similarly, branch coverage is the number of branches executed by the test suite divided by the total number of branches (Equation 2.2). Branch coverage is a stronger measure than statement coverage, because if all branches are examined, all sub-statements are examined in the process [76]. Note that branch coverage is a weaker measure than path coverage, in which the number of paths executed by the test code is measured. Because the number of paths in a program grows exponentially with the increase in the number of branches in a program, it is practically impossible to develop test code which tests all possible paths. Hence, the little interest for the use of path coverage as a practical metric for statement level.

\[
\text{Branch Coverage} = \frac{\text{Executed Branches}}{\text{All Branches}}
\]  

(2.2)
In Figure 2.5, three control structures (CS 1, CS 2, and CS 3) exist. These control structures create six branches and four paths. As we can see, even though only one path is covered by the test (CS 1 → True and CS 2 → False), two branches are considered as covered. This means that the branch coverage for this code is 33%.

Calculating code coverage is easy and fast. This has made it into an industry standard to measure test suite quality. Even though code coverage can give an estimation of the quality of test suite, it can be inadequate to guarantee the quality of the test suite [71; 32].

By comparing the reasons for widespread use of code coverage metrics, and lack of such popularity in case of mutation coverage, one can argue that practicality plays an important role in the decision to adopt them by industry. Therefore, it is important to use available ideas to improve the performance of mutation analysis tools. Using simple techniques such as a reduced set of mutation operators, and mutant sampling, would result in better performance without getting caught in the implications of adapting complicated performance enhancement techniques to complex industrial software structures.
CHAPTER 3

Target System

In this chapter, first the complexities of the architecture of Agfa HealthCare Segmentation component is discussed; Then, problems with the application of available tools are mentioned. There were several proposed solutions, of which, the most practical was to develop a new tool. So, based on the encountered problems, the requirements for the tool are defined.

3.1 Segmentation Component

There are three main components which create the core of IMPAX ES Clinical Applications. One of these three components is Segmentation component. The main use of this component is to provide imaging algorithms to segment 3D volumes. This component is average in size compared to other components of the software system, and it includes a test suite which is currently under active development. The component is entirely written in Java, for which several mutation analysis tools is available. The majority of the companies using agile methods use Java as their programming language [58].

The team that is developing the software is located in four cities around the globe, thus making coordination a critical part of the software development. This also increases the importance of the test suite, because, it is important to find out any errors introduced in any part of the code as soon as possible to reduce the need of long distance communication. The team uses the scrum model of development, by holding a weekly sprint meeting to coordinate their efforts. As the backbone of this agile development environment, they use a typ-
3.1. Segmentation Component

Figure 3.1: Agfa HealthCare Segmentation component build structure

Agfa HealthCare software components use Apache Maven as their build system. Due to the extensive use of OSGi1 headers throughout the whole project, tycho plugins are used during the build process in order to fetch dependencies, compile source files and run the test suite. The build system includes several other plugins to calculate code coverage, generate reports, obfuscate the target classes, etc.

The build system is structured as seen in Figure 3.1. The component is divided into 6 subcomponents. Main code is located in `com.agfa.hap.segmentation` while test suite code, some resources used by unit tests and a feature manifest for eclipse are located in `com.agfa.hap.segmentation.{utest, test, feature}` subcomponents respectively. The other two subcomponents provide early prototype features and their unit tests which should be excluded from the final build. Each subcomponent has a separate `pom.xml` file and can be built on its own. There is also a parent `pom.xml` file which selects and builds these

---

1Open Service Gateway initiative
subcomponents based on the called profile (production or prototype).

3.1.2 Dependency Handling

Maven uses a repository-based system to retrieve required dependencies to build the software. These dependencies are either provided as hard dependencies in the project object model (POM) file, or determined by the build plugins from other sources such as OSGI headers. This component uses OSGI to detect and use its dependencies. These can be required dependencies without which the software would not work, or optional dependencies which allow the software to provide certain functionalities. OSGI headers are used to resolve the dependencies for each package. This is done either by eclipse automatically during development, or by tycho while building with Maven.

3.2 Practical Complications of Mutation Analysis

Mutation analysis is a complicated process by itself. So, complexities inside the target software does not help the implementation of the procedure. Our first tries to use existing tools on this component have failed due to several complications arising from the complexities of the component itself. What follows is an explanation of the difficulties we have faced and the solutions proposed to overcome the problems.

3.2.1 Build System Complexities

As discussed in section 2.1, the choice of the build system is dependent on the requirements of the development environment, as well as the desired capabilities suitable to the needs of the software developers. Because of the interconnectedness of the Continuous Integration environment, replacement of the build system as a means to reduce its complexity is out of the question, as it requires a humongous effort to make it compatible with the rest of the tool chain. Therefore, the only solution is to solve the issue of complexity without making any fundamental changes to the structure of the build system.

OSGI-compliant projects are by design incompatible with most generic tools designed for Java projects. The reason lies in the way OSGI handles its dependencies. OSGI uses extra headers to provide all the information necessary for modularization of the software, including those needed to identify dependencies [67]. Therefore, those tools which are not OSGI-compliant would ignore such information altogether and run into errors. Moreover, any tool running from within the build system needs to include its own mechanism of executing the test suite, because during a single build iteration, the test suite
will be run only once by the build system, but mutation analysis requires the test suite to be executed on each mutant. This fact results in three solutions for mutation analysis of an OSGI-compliant software using a non-OSGI-compliant tool:

- **Strategy 1.** Convert the software into a non-modularized form; refactoring all OSGI-dependent code in the process.
- **Strategy 2.** Fetch all the dependencies during build process before running the analysis.
- **Strategy 3.** Apply the tool on source code and avoid interfering with the build process.

Other than the strategies mentioned above, there exists another obvious strategy to consider:

- **Strategy 4.** Using an OSGI-compliant tool.

Of course, an OSGI-compliant mutation analysis tool must work out of the box; However, a working prototype of such a tool proved to be non-existent.

### 3.2.2 Problems with Available Tools

To perform mutation analysis on Segmentation component, PITest\(^2\) was chosen as the first candidate. Given its status as a mature and successful tool used in many experiments in academia and industries alike\(^3\), it had the best chance of being successful in this task. Furthermore, PITest applies the mutations on the byte code, making it very fast in performing the analysis. This tool failed to achieve its goal due to several reasons:

1. PITest is not a OSGI-compliant tool. Therefore, it could not run the OSGI-dependent code by itself
2. PITest does not support a test suite in a separate package; Therefore, the test code and main code needed to be merged, resulting in further complications
3. PITest relies on coverage calculation to select the tests. For some unknown reason, it always returned 0 coverage for any class; resulting in selecting no tests at all
4. PITest could not find all the tests.

---

\(^2\)http://pitest.org/
\(^3\)http://pitest.org/java_mutation_testing_systems/
Other mutation testing tools like MuJava\textsuperscript{4} put even more constraints on the target project which makes them practically impossible to use in a realistic situation.

### 3.2.3 Proposed Solutions

After a brainstorming session with the developers of Segmentation component, five preliminary solutions were proposed that may solve the problem:

1. Manual refactoring of the OSGI-dependent code; consistent with Strategy 1
2. Using a prototype tool called Cheshire as a non-OSGI interface to OSGI; consistent with Strategy 2
3. Using an OSGI-compliant tool such as MuUnit [30]; consistent with Strategy 4
4. Adapting PITest to run as a Java agent; to make it available through the build process as a plugin; consistent with Strategy 4
5. Writing a new tool from scratch in which Strategy 3 is applied.

What follows explains the efforts to implement these solutions and the possible reasons they were deemed inapplicable.

#### 3.2.3.1 Manual Refactoring

Manual refactoring was the first option that was considered. However, there are limits to the refactoring process in order to keep the results relevant to the original code. One such limit is the fact that the test suite must not change. Also, parts of the code which the test suite covers must experience minimal change at best. Otherwise, the results of the analysis on the refactored code cannot be reliably representing the original code. This proved to be the main problem; Because not only the main code is OSGI-dependent, but also the test code is OSGI-dependent and needs to be refactored. This, coupled with the fact that both main and test code needed months of work to fit into a non-modularized schema, resulted in the invalidation of this strategy as a way forward by making it a risky and time consuming choice.

#### 3.2.3.2 Cheshire

Cheshire is a prototype tool that provides an interface for OSGI-compliant software to resolve and retrieve dependencies during compile time. Since it is a prototype tool, only a little more than that is known about it. After a discussion with the developer of Cheshire, it was clear that many recipes

\textsuperscript{4}http://cs.gmu.edu/~offutt/muja/
must be written for different dependencies of Segmentation component; which according to the developer’s estimation would take weeks for someone not familiar with the details of the component. Even then, the result would still be unclear, since no practical result was available of this tool ever working with another tool like PITest. Therefore, this solution was discarded as being too risky, with unclear results and no guarantee for success.

3.2.3.3 MuUnit
This solution looked like the most probable option to succeed. Yet; problems arose after tinkering with the tool. MuUnit turned out not being able to perform the tasks it promises. After a quick inspection of the code it became clear that the tool is an early prototype which is able to run its analysis only on very simple projects. Contacting the author was unsuccessful as no reply was ever received. So, this option led to a complete dead-end and was discarded as such.

3.2.3.4 Adapting PITest
Another promising option was to adapt PITest to become an OSGI-compliant tool. Currently, PITest tries to find the classes it needs to mutate and test classes it needs to execute by itself. Therefore, it cannot be integrated into the build process. To be able to do so, it needs a Java agent to be able to run before each test and run the test on changed code on the fly. Unfortunately, due to the structure of PITest, any effort to make this possible requires major refactoring of the code. PITest started off as a distributed computing platform and later developed into a mutation testing tool. This causes extreme complications in any try to change a major part of the tool. Other than that, there are still problems to be resolved about the final design; as it is still not known how to rerun the test suite in multiple iterations during the build process without interfering with the build process itself. Thus, this option was discarded as requiring too much effort to work and most of this effort being completely out of scope and far away from the purpose of the thesis.

3.2.3.5 Developing a New Tool
The only remaining option was to develop a tool from scratch implementing Strategy 3, to avoid the hassles of OSGI and guarantee its reusability. In order to create such a tool, the criteria for the tool must have been clear. These criteria are discussed in the next section.

3.2.4 Required Criteria for the Tool
After trying to integrate the aforementioned mutation testing tools into the build system of segmentation component, it was clear that the complexity of the build system and software architecture might not be cooperative towards the procedure these tools use to run the tests. Therefore, it became obvious that one of the important criteria for the new tool was to be able to act
independently from the build system. This means that the build system must be integrated into the tool and not vice versa. This brings about certain restrictions for the way the tool can be designed. For example, it would be impossible to apply the mutations on byte code (like PITest does), and the mutations need to be applied on source code. Furthermore, it is not possible to create a single result parser to parse the results of the build procedure; since the output it generates varies between different build systems. Consequently, the results must be obtained in another way.

As a result, the following requirements can be deduced from both, the problems that are explained before, and the requirements expected of such tools:

- **Requirement 1.** The tool must work independently from the build system of target software.
- **Requirement 2.** The tool must apply the mutation on source code.
- **Requirement 3.** The tool must gather results of the tests from the build system.
- **Requirement 4.** The tool needs to generate results in the form of easy-to-read reports.
- **Requirement 5.** The tool needs to retain information which would allow the manual inspection of the results.
- **Requirement 6.** The results of the tool must be considered sound, i.e. be of practical use.

It becomes clear that the complexities in the build system and the architecture of Segmentation component would not allow just any tool to work out of the box. Considering the fact that many industries use the same techniques to design their products, it can be concluded that there is a deficiency of the mature mutation analysis tools which can be adopted to such systems easily. Therefore, it is of great importance to design a tool with this fact in mind. This led us to define the characteristics of such a tool, and attempt to implement it to prove the feasibility of performing mutation analysis on industrial projects.
LittleDarwin aims to solve the problem of easy integration of the mutation analysis tool in a continuous integration environment. To achieve this goal, several requirements were devised to inspire the design of LittleDarwin. One of the main factors considered in designing LittleDarwin was to make the procedure separate from the target system as much as possible. Therefore, reducing the complexity of the tool itself, allowing it to analyze complex systems with ease.

LittleDarwin is capable of performing mutation analysis on a variety of systems with ease, as explained in Section 4.2. It is designed in an expandable way, so that interested developers can develop their own mutation components and still use the structure of the main software to run the mutation analysis. This broadens the scope of its applicability. The main target audience for

\[1\text{http://littledarwin.parsai.net/}\]
the tool are those companies with small teams maintaining a large code base. LittleDarwin provides such teams with a means to analyze their test suite without spending a lot of time on compatibility issues. Since LittleDarwin is written in Python, it has the capability of running on various platforms. The main use case of LittleDarwin is to examine test suites which are known to have issues, but are too large to be examined manually. The primary content of such test suites must be unit tests; As acceptance tests often take too long to be part of the analysis. However, LittleDarwin is capable of calculating the mutation coverage for acceptance tests, to allow developers to assess the quality of such tests.

4.1.1 Algorithm

LittleDarwin uses a simple algorithm to implement mutation testing. The procedure is divided into two phases: Mutation Phase and Test Execution Phase (Figure 4.1) These phases act independently from each other, even though the latter is dependent on the former. This allows for manipulation of the mutants after generation.

1. Mutation Phase In this phase, the tool creates the actual mutants to be used in the next phase. As seen in Figure 4.1, the tool first searches for all source files contained in the given path and adds them to the processing queue; Then, while the queue is not empty, it selects an unprocessed source file from the queue and parses it; Then, applies all the mutation operators on the resulting abstract syntax tree; Then, saves all the resulting mutants on disk, and adds an entry for the file in the mutation database to allow access for the test execution phase.

2. Test Execution Phase As seen in Figure 4.1, in this phase, first the build system is executed without any change to ensure that it runs without any problems, and all tests pass. Then, the information about the source files and their corresponding mutants are read one by one from the database generated during the mutation phase. Each mutant is put in place of the corresponding original source file and the output of the build system is recorded. If the build system fails or times out, the mutant is categorized as killed. If the build system is successful, the mutant is categorized as survived. After all mutants of a source file are processed, a report is generated for that source file. After this, the source code is returned to its original state and the process begins for the next source file. In the end, an overall report is generated based on recorded data for all the files.
Figure 4.1: LittleDarwin flow chart
4.1. DESIGN SPECIFICATIONS

4.1.2 Components

An overview of the internal components can be seen in Figure 4.2. In order to implement the mutation phase, three major components have been implemented:

1. **JavaRead.** This component provides methods to perform input/output operations on Java files; including reading the files into a string buffer and generating mutated files from the created mutants in their appropriate location.

2. **JavaParse.** This component parses the input java file into an abstract syntax tree to make it ready for applying the mutation operators. This is necessary to make LittleDarwin *understand* the syntax of the program, so that the mutants are valid and compilable. To implement this functionality, Antlr4\(^2\) Java 8 grammar was used along with a customized version of Antlr4 runtime. Beside providing the parser, this component also provides the functionality to pretty print the final tree back to a Java file. In order to make the output file compilable, the Antlr runtime needed to be customized to generate such output.

3. **JavaMutate.** In this component, the abstract syntax tree created in the parser is used to create all possible mutants by applying all mutation operators. In total, there are 9 mutation operators implemented in this component in 6 categories. These operators are collectively known as the minimal set. The description of each operator along with an example can be found in Table 4.1.

The other two components perform operations before and after the test execution phase.

1. **Mutant Sampler.** This component creates a new mutation database by randomly selecting mutants. The selection rate is determined based on the total number of mutants per file (Figure 4.3). This component is provided as an optional feature for the user as a way to improve performance. As Figure 4.3 shows, the rate varies between 100% and 10%. A high percentage of the mutants are selected if the total number of mutants is small, and it gets lower as the size grows, and it will rest in a constant 10% rate if there are more than a 1000 mutants. As a result, the effects of this reduction in the number of mutants remains insignificant.

\(^2\)http://www.antlr.org/
4.1. DESIGN SPECIFICATIONS

Figure 4.2: LittleDarwin components

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR-B</td>
<td>Replaces a binary arithmetic operator</td>
<td>$a + b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a - b$</td>
</tr>
<tr>
<td>AOR-S</td>
<td>Replaces a shortcut arithmetic operator</td>
<td>$++ a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$-- a$</td>
</tr>
<tr>
<td>AOR-U</td>
<td>Replaces a unary arithmetic operator</td>
<td>$- a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$+ a$</td>
</tr>
<tr>
<td>LOR</td>
<td>Replaces a logical operator</td>
<td>$a &amp; b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a \mid b$</td>
</tr>
<tr>
<td>SOR</td>
<td>Replaces a shift operator</td>
<td>$a &gt;&gt; b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a &lt;&lt; b$</td>
</tr>
<tr>
<td>ROR</td>
<td>Replaces a relational operator</td>
<td>$a &gt;= b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a &lt; b$</td>
</tr>
<tr>
<td>COR</td>
<td>Replaces a binary conditional operator</td>
<td>$a &amp;&amp; b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a | b$</td>
</tr>
<tr>
<td>COD</td>
<td>Removes a unary conditional operator</td>
<td>$! a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>SAOR</td>
<td>Replaces a shortcut assignment operator</td>
<td>$a *= b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a /= b$</td>
</tr>
</tbody>
</table>

Table 4.1: LittleDarwin mutation operators
while the total processing time reduces greatly in case of large systems with complex classes [73]. It is worth mentioning that these values have been determined by experimenting on a small sample project. Therefore, they might not be the optimum values. More information on the selection rate can be found in Section 2.2.3.

2. Report Generator. During the test execution phase, the data is gathered from the output of the build runs for each mutant. Afterwards, Report Generator component generates HTML reports for each file (Figure 4.5), and an overall report in the end (Figure 4.4).

### 4.2 Target System Requirements

Generally, when designing the architecture of a software system in an industrial setting, the decisions on architecture of the software are taken based on factors other than complexity [31]. Consequently, build system structure for such software are often complicated. Available mutation testing tools are mostly proof-of-concept prototypes and their design is optimized to provide a platform to examine new ideas. Therefore, they are not suitable for practical purposes. For example, PITest needs to have all classes already compiled and all the test classes in the same place. Then, it will try to calculate statement coverage,
and run the tests that cover some part of the code, and calculate the mutation coverage based on these tests. As a consequence, if any part of the target system is incompatible with the way PITest performs its job, either the results would be incorrect or the analysis would not be run at all.

This problem arises from the fact that these tools try to create their own testing structure and then run the analysis. This problem is solved in LittleDarwin by making the tool independent from the testing structure. As a result, LittleDarwin demands much less compatibility from the target system in order to perform its analysis. Thus, it can be run on any build structure no matter how complex it is given following conditions:

1. The build process must be able to run the test suite.
2. The build process must return non-zero if any tests fail, and zero if it succeeds.
3. The build process must be sufficiently fast in order to keep the total run time practical.

There are certain functionalities added to the tool to make it more practical for complex systems. For example, it can run a clean-up series of commands in order to remove any leftovers from previous iterations after each build. It can also run a test suite which is separate from the code being tested by separating the commands to compile the mutant and the commands to run the test suite. There is also an option to change the commands for the initial build, in case there is a need to initialize the build system before running the tests. As it stands, In theory, LittleDarwin is able to mutate any software system that conforms to the conditions previously mentioned.

### 4.3 LittleDarwin in Action

While LittleDarwin looks very promising in theory, it needs to be validated against its requirements to prove useful in practice:

- **Requirement 1.** The tool must work independent from the build system of target software.
- **Requirement 2.** The tool must apply the mutation on source code.
- **Requirement 3.** The tool must gather results of the tests from the build system.
• **Requirement 4.** The tool needs to generate results in the form of easy-to-read reports.

• **Requirement 5.** The tool needs to retain information which would allow the manual inspection of the results.

• **Requirement 6.** The results of the tool must be considered sound, i.e. be of practical use.

To validate requirement 1, the tool was run on two different systems, one using Apache Ant\(^3\), and the other using Apache Maven. LittleDarwin has met this requirement by successfully gathering results from both cases.

The requirements 2 through 5 were met during the design phase; being validated each time the tool analyzes a system. These requirements are specially important to make the tool practical for any form of analysis. Therefore, the validation of requirement 6, automatically depends on the validation of these requirements. The generated results of the component developed according to requirements 4 and 5 can be seen in Figures 4.4 and 4.5.

In order to validate the results of LittleDarwin (requirement 6), a small experiment was designed to compare the results with those of PITest, as a widely-used mutation analysis tool, and to branch coverage, as a widely-used quality metric. Even though the mutation operators of PITest are different from LittleDarwin’s, the expectation was that the results should be similar. For the sake of assessing the effect of mutant sampling on the final results, a comparison was made between complete set of mutants and a sampled set generated by MutationSampler component. To perform this experiment, AddThis Codec\(^4\) was chosen as the target project. This project has 46 classes in total, which allows for manual inspection due to its small size.

PITest generated a total number of 1038 mutants of which 586 were killed by the test suite; resulting in a mutation score of 56%. LittleDarwin created 476 mutants with 295 of them killed; which lead to a mutation score of 62% for the whole project. The sampled project consisted of 310 mutants, which is near 65% of the original number of mutants, resulting in 35% reduction of total mutants. Of these, 190 mutants were killed; making the final result for the sampled database 61.3%, which is very close to the original score. In fact, the average difference between full database and sampled database was 1%, which is insignificant. Figure 4.7 visualizes the mutation coverage for each class of AddThis Codec, as computed by full set of mutants vs. the sampled set of mutants. The red parts show the coverage calculated by full set, and the blue parts represent the coverage computed by sampled set; The overlapping purple part shows what they have in common. As it seems, most parts of the

\(^3\)http://ant.apache.org/

\(^4\)https://github.com/addthis/codec
Figure 4.4: LittleDarwin easy-to-read report (requirement 4)
4.3. LITTLE DARWIN IN ACTION

Figure 4.5: LittleDarwin individual source file report (requirement 5)

Figure 4.6: Comparison between PITest, LittleDarwin and branch coverage
The results of the experiments could be seen in Figure 4.6. As seen in the mentioned figure, the mutation coverages calculated by PITest and LittleDarwin are close to each other for some classes; while classes with most fluctuations are discovered to be those which had less than 10 mutants generated in total after the manual inspection of the results.

These results roughly follow the pattern created by branch coverage; because it is expected that those classes which have higher branch coverage are tested more thoroughly. This, however, does not mean that the results of branch coverage match those of mutation coverage. For example, applying LittleDarwin’s mutation operators to the code in Figure 4.8 results in 6 mutants (two RORs which would change == to ! =, and four AOR-Bs which would change * to /, and + to – and vice versa). If the test suite tests main_method for input values of 1 and 2 (green part), the branch coverage would be 66%, because else is untested (red part). However, only 2 of the mutants would be killed, because, two RORs would return unexpected results, and four AOR-Bs are untested. Therefore, the mutation score for this class would be 33%. This great difference in scores is justifiable, as mutation coverage is providing more detailed information about the code; in this case, the fact that the untested
LittleDarwin was designed based on the requirements discussed in Chapter 3. We then discussed the architectural details of the tool, and the theoretical range of systems it can analyze. Afterwards, we have shown how aforementioned requirements are met by our prototype tool, and we performed a small experiment to confirm the validity of the results of the tool. We have also shown how we can improve performance without significant effects on the final results.

```java
class example {
    int method1() {
        return 1;
    }
    int method2() {
        return 2;
    }
    int main_method(int input) {
        if (input == 1)
            return method1();
        else if (input == 2)
            return method2();
        else
            return method1() * 4 + method2() * 5 - 3;
    }
}
```

Figure 4.8: Example of difference in branch coverage and mutation coverage branch contains more possibilities of a mistake.
In previous chapters, we discussed the motivation of performing mutation analysis on an industrial system and the problems we encountered when we tried to use the available tools; we, then, defined the requirements of a tool which can analyze an industrial system, and developed LittleDarwin based on those requirements.

After validating the results of LittleDarwin using a small experiment in previous chapter, we used it to analyze Segmentation component as an example of an industrial project. In this chapter, the details of this experiment are discussed. First, we describe the experimental setup; Then, we discuss our analysis of the final results.

5.1 Experimental Setup

As explained in Chapter 3, the build system for Agfa HealthCare Segmentation component is complex; and when it comes to running it in several iterations, this complexity becomes critically important. As a result, the tool included several workarounds as mentioned in Chapter 4 to be able to overcome the problems originating from this fact. In order to carry out the Test Execution phase, two separate sets of commands (compilation and test execution) were passed to the tool; therefore, making it feasible to use LittleDarwin on this component.

The build system complexity resulted in a lot of overhead, which increased the time it takes to build and test the component significantly. Therefore,
to make the process practical, few time consuming and irrelevant tests were excluded from the test suite. This would have minimal impact on the results, because these tests were either performance tests, which would not add any additional coverage, or acceptance tests, which are considered out of scope. Removal of these tests reduced the total time for compilation and testing of the component from few minutes to less than a minute in each iteration.

Branch coverage is the standard quality metric being used in many industrial softwares nowadays. Segmentation component is not an exception. Due to this popularity in the industry sector, it is not sensible to assess the usefulness of mutation coverage without comparing it to branch coverage. That being said, branch coverage can only be used as a simple guide; and it is not expected to be as precise as mutation coverage in estimating the quality of a test suite [71]. Thus, we expect mutation coverage to reveal much more details about the software compared to branch coverage.

During the build process, JaCoCo\(^1\) acts as a maven plugin which calculates branch coverage for Segmentation component. The data for the branch coverage can be acquired by executing the build system — thus executing the test suite — once.

Mutation coverage was calculated by executing LittleDarwin with two sets of commands. The first set was to compile the mutated source code and install the final result into the local maven repository; And the second set was to execute the separate test suite on aforementioned artifact. All experiments were executed using the same computer, operating system, and python interpreter to make the performance comparison valid. The final results were extracted from the reports generated by LittleDarwin, after manually inspecting the results for interesting cases.

In order to keep the performance results consistent, all of the experiments have been run on the same machine. This machine used two Intel Xeon 2.80 GHz processors with 16 GB (4x4 GB) of DDR3-1333 memory\(^2\). Since LittleDarwin only uses a single thread to perform its analysis, there are no gains from the multi-core architecture of the hardware it runs on.

The quality of the test suite of Segmentation component was unknown prior to this experiment. This might lead us to wrong conclusions if the comparison is done using only a single case. It is important to distinguish the aspects of the results which are generalizable, and those which are specific to this case. Therefore, the same comparison between the results of branch coverage and those of mutation analysis was performed on two open source projects (jOpt

\footnotesize{\(^1\)http://www.eclemma.org/jacoco/}\\
\footnotesize{\(^2\)Full specifications available here: http://h20564.www2.hp.com/hpsc/doc/public/display?docId=emr_na-c01709707}
Simple\(^3\) 5.0, and Apache Commons Codec \(^4\) 1.7), which came with high quality test suites. The same tools (JaCoCo, and LittleDarwin) were used to gather the results of each metric to minimize the effects of the tools on the acquired results. Studying the differences between the test suites of these projects and the test suite of Segmentation component provides enough evidence to tell these aspects apart.

### 5.1.1 Comparison Criteria

In order to determine the differences between the results of mutation analysis and those of branch coverage, we need to define the criteria by which the information extracted from these results are judged. The following categories are used to describe the results of this experiment:

- **Category 1.** Classes with similar branch and mutation coverage.
- **Category 2.** Classes with low branch coverage and high mutation coverage.
- **Category 3.** Classes with high branch coverage and low mutation coverage.
- **Category 4.** Classes which do not have a branch coverage but have a mutation coverage.
- **Category 5.** Classes which have a branch coverage but do not have a mutation coverage.

Considering \(m\) as mutation coverage percentage, and \(b\) as the branch coverage percentage for a class, the classes with non-zero scores with the difference of \(t\)% or without either coverage are considered to be in Category 1 \(((|m - b| \leq t \land m, b > 0) \lor m = b = 0)\). Any classes with non-zero scores for both metrics that are not in Category 1, are in either Category 2 \((m - b > t \land m, b > 0)\) or Category 3 \((b - m > t \land m, b > 0)\). Those that are not part of the first three categories, are part of either Category 4 \((m > 0 \land b = 0)\) or Category 5 \((b > 0 \land m = 0)\). The value of \(t\) determines the threshold we set in the difference of the results of two metrics for a single class to consider them as similar. This threshold does not have any effect on the number of classes in Categories 4 and 5, and only changes the values in Categories 1, 2, and 3.

Each of these categories provide an insight into the differences between these two metrics. In the classes belonging to Category 1, the number of killed mutants is proportional to the code coverage of the class. Classes in Category

---

\(^3\)http://pholser.github.io/jopt-simple/

\(^4\)http://commons.apache.org/proper/commons-codec/
2 contain few covered code blocks with high probability of occurrence of a bug, and more uncovered code blocks in which the probability of occurrence of a bug is low. Classes in Category 3 contain few uncovered code blocks with high probability of occurrence of a bug, and lots of covered code blocks in which the probability of occurrence of a bug is low. Category 4 contains classes for which branch coverage could not be calculated, but mutation coverage is available. Category 5 contains classes that are partly covered by at least a test, but are not truly being tested, or those in which the mutants lie in the code blocks which are uncovered.

By counting the number of the classes in each category we can determine the usefulness of each metric for our purpose. For classes in Category 1, the information can be acquired using either metric, meaning that, at least without any deeper analysis, the difference between two metrics regarding these classes is minimal. For classes in Categories 2 and 3, both metrics should be used in conjunction to acquire the insights described above; meaning that neither metric on its own provides any extra information. Information about classes in Categories 4 and 5 are, however, only attainable by one of the metrics; Therefore, showing a distinct advantage for each metric in gathering information.

5.2 Analysis of the Results

The purpose of designing this experiment was to answer the two questions introduced previously. The first question we try to answer is the feasibility question.

- **RQ1.** Is it feasible to apply mutation analysis on an industrial project?

The design and implementation of LittleDarwin has evolved around answering this question. Therefore, by successfully using LittleDarwin on Segmentation component we already have demonstrated that it is feasible to apply mutation analysis on this industrial project. Due to the complexity of Segmentation component, and the ability of LittleDarwin to overcome such complexity, we are confident that other industrial projects could also be analyzed using LittleDarwin. However, to prove this fact, more industrial experiments are needed.

What follows, are an analysis of the final results generated by LittleDarwin, and a comparison with the results of branch coverage analysis performed by JaCoCo as the default quality metric for Segmentation component. In this analysis, our main focus would be to answer the second research question:
5.2. ANALYSIS OF THE RESULTS

Figure 5.1: Mutation coverage results of classes in Segmentation component

- **RQ2.** What are the trade-offs involved in mutation analysis vs. branch coverage for determining test suite quality?

In total, 12825 mutants were generated for 211 source files of segmentation component. 4955 of these mutants were killed by the test suite, resulting in 38.6% overall mutation coverage. It is worth mentioning that 107 corrupted mutants were generated which resulted in compile errors. These mutants were wrongly generated due to a bug in the tool, and were excluded from the final results. In Figure 5.1, the horizontal axis are the classes sorted by their mutation coverage, and the vertical axis is the coverage score by percentage. The blue bars are the representation of branch coverage and the red line is the calculated mutation coverage. Figure 5.2 shows the same data, this time sorted by branch coverage. The red areas are a representation of mutation coverage, and the blue area is the representation of branch coverage.

As seen in Figure 5.1, there are a lot of classes which lack a branch coverage metric, but mutation coverage is calculated for them (Category 4). In total, 90 classes are classified as Category 4. After manual inspection of several classes, it was clear that mutation coverage is in fact providing the correct analysis. For example, in case of *RegionGrowerNeighbours*, branch coverage was not available; However, LittleDarwin reported that all 22 created mutants were killed because of the failure of *VolumeGrowerTest* and *RegionGrowerNeighboursTest*. Since these tests explicitly test the algorithms used in this class, it is unlikely that the real branch coverage is 0% and this means that the calculation method used by JaCoCo cannot capture this fact. These additional information about the parts that were not considered to be covered before is crucial in preparing a plan to improve the quality of the test suite. This information allows major
The results of this procedure can be observed in Figure 5.5. The value of \( t = 11 \) was chosen as the threshold based on this procedure.

A total number of 90 classes out of 212 were classified as Category 4, while only 4 classes were classified as Category 5. There were 8 classes that were classified as Category 2, and 8 that were Category 3. The classes in Category 2 are less prone to the occurrence of bugs than those in Category 3, because
5.2. ANALYSIS OF THE RESULTS

Figure 5.3: Mutation coverage results of classes in jOpt Simple

Figure 5.4: Mutation coverage results of classes in Commons Codec
5.2. ANALYSIS OF THE RESULTS

Figure 5.5: Categorization results using different threshold values

the tests are covering the more mutable areas of the code. Such a low number of classes in these two categories suggests that the classes with high mutability areas are either tested thoroughly, or not being tested at all. The rest of the classes (102) were classified as Category 1, for which the mutation coverage and branch coverage calculated roughly the same values.

The high number of classes in category 4 suggests that the results of this experiment strongly support the fact that mutation analysis can provide extra information which branch coverage does not provide. However, the test suite for Segmentation component includes lots of black-box unit tests which are made according to the requirements of the software. These tests only consider the final results of the algorithms they test. Due to these unique features, we opted to perform the same experiment on two open source projects which contain high quality test suites and are widely in use.

Figures 5.3 and 5.4 show the results of these experiments on the two open source cases. In case of jOpt Simple, only 1 class out of 34 was classified as Category 5, while there were no Category 4 classes. This single class was an exception definition class which did not produce any mutants. In case of Commons Codec, no classes out of 38 possible classes were categorized as Category 4 or 5, and 10 classes were categorized as Category 2 or 3. Therefore, it can be concluded that neither metric provides any extra information on its own, and small gains can be achieved by employing both metrics.

In Table 5.1, a summary of the results of the experiments is provided. As is evident, the results of the experiment on Segmentation component is contrary to those of the open source projects. This is mainly due to the black-box approach in developing the test suite of the Segmentation component.
5.2. ANALYSIS OF THE RESULTS

Figure 5.6: Distribution of classes by number of generated mutants

<table>
<thead>
<tr>
<th>Experiment</th>
<th>t</th>
<th>Cat.1</th>
<th>Cat.2</th>
<th>Cat.3</th>
<th>Cat.4</th>
<th>Cat.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation (full set)</td>
<td>11</td>
<td>102</td>
<td>8</td>
<td>8</td>
<td>90</td>
<td>4</td>
</tr>
<tr>
<td>Segmentation (sampled set)</td>
<td>11</td>
<td>102</td>
<td>10</td>
<td>9</td>
<td>87</td>
<td>4</td>
</tr>
<tr>
<td>jOpt Simple</td>
<td>7</td>
<td>30</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Commons Codec</td>
<td>7</td>
<td>27</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of experiment results

Therefore, it can be concluded that in case of large and complex projects in which the tests are developed based on requirements rather than directly based on the code, mutation analysis can be a powerful method of measuring the quality of the test suite; Whereas, in small projects in which the test suite is designed with the internals of the system in mind, it might not provide additional benefits without deep inspection of the source code.

Figure 5.6 shows the distribution of classes by the number of mutants. In the top chart, the number of classes in each range is shown (classes with less than 5 mutants, classes with 5 to 10 mutants, etc.). In the bottom chart, the box plot shows the distribution of classes by the number of mutants generated for them.

Even though Figure 5.1 apparently shows great discrepancies between branch coverage and mutation coverage over all classes; This does not hold true for the larger classes. This happens because a lot of classes in Segmentation com-
ponent are only made up of few statements and therefore, the fluctuations in mutation coverage are high due to the small number of generated mutants. This fact can be seen in Figure 5.6, as the median of the set is 18 mutants for a class, and the third quantile is near 60. Also, the effect shown in Figure 4.8 plays a major role in creating these discrepancies.

Another observation that both metrics agree on, is the fact that a lot of classes in this component have no test coverage at all. This might be another area of interest due to the fact that any changes to these parts of the code will go unnoticed until later phases of the development; essentially skipping the economical benefits of catching the bug as early as possible.

The discussed results demonstrate the capability of mutation analysis to provide new insights about the quality of the test suite, and point out the areas of potential problems. However, it takes a long time to gather the results. In this particular case, the duration of the analysis was 6 days and 19 hours. To integrate mutation analysis into short-term strategy, such a slow performance is a major obstacle to overcome. Therefore, mutant sampling is an important contribution to the practicality of the analysis; and for that reason, it is of utmost importance that it verifiably provides similar results in shorter period of time.
5.2. ANALYSIS OF THE RESULTS

The mutant sampler component of LittleDarwin produced a randomly selected set of mutants which contained 4,448 mutants in total. This is 8,377 mutants less than the original 12,825 mutants in the full set, resulting in a 65.3% reduction in number of mutants. Figure 5.6 shows the original distribution of the classes and the resulting distribution side by side. It is obvious that the number of classes with 10 to 50 mutants have increased while the classes with more than a 100 mutants have decreased dramatically; This means that these classes were the major contributors to the reduction. Also, the number of outliers have reduced greatly. The median of the full set was 18, and the median of the sampled set is 14. However, the great change was in the average which was reduced from 60.8 to 21.1. This means that the sampled set is more balanced, as the average and the median are much closer to each other.

Out of 4,448 mutants in sampled set, 1,721 were killed by the test suite, resulting in a 38.7% overall mutation coverage. The difference between the overall coverage calculated from the sampled set and the one calculated from the full set is only 0.1%. As shown in Table 5.1, the difference between the number of classes in each category remains roughly the same when using the sampled data instead of the original data. Figure 5.7 shows the difference between the results from full set to the results from sampled set sorted by the mutation coverage from full set. The red areas show how much the sample set coverage was more than the original, and the blue areas show how much it was less. This figure shows that the mutation coverage calculated from the two sets remained roughly the same.

To discover the significance of this difference, we should consider how it is distributed and what the average is. Figure 5.8 shows the distribution of the differences in mutation coverage between full set and sampled set for each class (bottom) and the scattered plot of the differences and number of mutants for each class (top). Note that the X axis of the top plot is in logarithmic scale, and the full set was considered as the source for number of mutants for each class. One simple observation from this figure is that the third quantile of the differences lie in just above 2%, while 90% of the differences are less than 5.6%. The average difference is close to 1.8% which can be considered insignificant. It is also apparent that the number of mutants for a class has no effect on the difference value, since the scattered plot follows the same pattern for those classes with 10 to 100 mutants and those with 100 to 1,000 mutants. It is worth mentioning that the reason the difference is 0% for those classes with less than 10 mutants is that mutant sampler selects all mutants in such classes.

The analysis using sampled set took 2 days and 10 hours to finish, which is about one third of the time it needed to complete the analysis using the full set. This great enhancement in performance along with the insignificant change in the final result proves that mutation analysis provides a lot of useful information at manageable performance costs; making it a viable option to
5.3 Threats to Validity

Figure 5.8: Difference in mutation coverage between full set and sampled set consider in the testing procedure of an industrial product.

5.3 Threats to Validity

Threats to internal validity concern confounding factors that can influence the obtained results. In this study, this includes the fact that the final mutation coverage can be affected by factors such as equivalent mutants and limited set of mutation operators. Because of the large number of generated mutants, it is inconceivable to check for equivalent mutants in the final generated results due to the manual nature of looking for and removing such mutants. Nevertheless, since equivalent mutants would add to false positives, they would be discovered when the developers are trying to create new tests or improve the available tests by referring to the information acquired from mutation analysis.

Threats to construct validity focus on how accurately the observations describe the phenomena of interest. In this study, the area of interest is to find out the quality of the test suite in general and to find which tests are performing poorly. To assess this problem, we compare the tests based on two quality metrics, and determine the correlation of these metrics. Based on this, we can determine the classes in which the metrics do not reach the same results, and
find out the problematic tests.

Threats to reliability validity correspond to the degree to which the result dependent on the used tools. Compile errors and errors in tests can affect the final results since LittleDarwin checks only if the build process has failed or not, and it does not go further to determine the reason for the failure. This was addressed by inspecting the output of the build system and excluding the problematic mutants. Another threat to reliability validity is the fact that the data gathered by JaCoCo might not be accurate, and therefore the conclusions based on the comparison between branch coverage and mutation coverage might not be accurate enough. However, since JaCoCo is the tool that is being used in the structure of Segmentation component to acquire this information in the first place, the conclusions are still relevant by challenging the previously held beliefs about the system.

Threats to external validity correspond to the generalizability of our experimental results. Since this study was performed only on a software running on a single platform with a specific target language, the results might not be representative of the concept. However, it provides an outlook on the possibility of applying mutation testing in an industrial environment. We also tried to address this problem by performing the same experiment on other open source projects. By comparing these results with those of our original experiment, we determined the situations in which our conclusions can be generalized.

Based on the experiments discussed in this chapter, we can claim that mutation analysis provides valuable information about the test suite, including lots of practical details which can be useful in forming a long term development strategy. With the massive reduction in the number of mutants created, we can also argue that it is possible to enhance the performance to make the process more practical.
Conclusions

With the adoption of agile techniques in today’s industrial environments, assessing the quality of the software has become an important step in development cycle. Often in a continuous integration environment, passing the test suite on a local machine is the first step before publishing any changes to the main source code repository. Thus, having a high quality test suite is a major step in achieving higher quality software, and reducing the software maintenance costs.

Mutation analysis is a useful method to measure the quality of a test suite. Yet, it has not been incorporated in industrial projects due to its high performance costs. The transition of mutation analysis from an idea in academic circles into a practical method of analysis used in industries requires an assessment of its feasibility, and a comparison with the methods currently being used to examine the quality of a test suite. In this study, we investigated the feasibility of using mutation analysis in an industrial setting (RQ1), and find out if mutation analysis offers enough advantages to be practically useful (RQ2):

- **RQ1.** Is it feasible to apply mutation analysis on an industrial project?
- **RQ2.** What are the trade-offs involved in mutation analysis vs. branch coverage for determining test suite quality?

Our first approach was to try to use available tools to perform the analysis. This, however, proved to be impossible due to the complexities of the build system of the target software. Based on the reasons of this failure, five solutions were proposed, of which the most logical one was to design a new tool.
Therefore, requirements for such a tool were determined with the application on complex systems as a major goal.

LittleDarwin was designed to meet these requirements. To prove this, we performed a small experiment proving the soundness of the tool’s results. We also used mutant sampling as a performance enhancement technique, and demonstrated that its effect on final results is insignificant.

To answer RQ1, we used our tool to perform analysis on the target system successfully, and used the results to answer RQ2. The results of mutation analysis on Segmentation component shows that a great portion of test capabilities of the test suite is undetected by simple metrics. It also confirms the fact that mutation analysis is more precise in determining the weak spots of the test suite. By using mutant sampling, we could reduce the time needed for the mutation analysis of the target system to a third of its original time, making it a practical option to use in industry. Based on these results, we can argue that using mutation analysis can reduce the workload in refactoring and developing the test suite by pinpointing where the weaknesses are, and where there is enough coverage already.

There are several aspects of this research which could be followed up in the future. An interesting subject to consider is to introduce the support of other programming languages to LittleDarwin, making it the first multi-language mutation analysis tool. Because LittleDarwin is designed to run outside the build system, it is capable of performing analysis on systems consisting of multiple programming languages without any additional effort in a single run.

Another aspect which can be improved is the issue of performance. The application of the mutation analysis on the same software can be implemented in a continuous way, only generating and executing the mutants according to changes in the system. This would dramatically increase the performance in a continuous integration framework, making it a practical replacement for simple metrics such as branch coverage. It is also possible to use distributed computing techniques to run the analysis in a cloud, thus reducing the time needed to complete the operation.


[38] Ren Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? Technical Report UW-CSE-14-02-02, University of Washington, 2014. (page 2), (page 64), (page 72)


L Madeyski and N Radyk. Judy - a mutation testing tool for java, 2010. (page 72)


J McGregor. Test early, test often. 2007. (page 8)


David Schuler and Andreas Zeller. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the the 7th Joint Meeting of the European


[65] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. Mutation operators for concurrent programs in MPI. In Test Workshop (LATW), 2012 13th Latin American, LATW ’12, pages 1–6, Washington, DC, USA, 2012. Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo, USP, São Carlos, Brazil, IEEE Computer Society. (page 12), (page 68)


One of the major objectives of this study was to prove the fact that mutation analysis can be useful for a company such as Agfa HealthCare. Therefore, it is only right if we provide our analysis of the final results from the subjective perspective of the company; In other words, to provide a strategy for using mutation analysis in the development of software components in Agfa HealthCare. In this chapter, we are first going to discuss what the final results mean for Agfa HealthCare; Then, propose the experiments which can be performed using current state of development of LittleDarwin; And finally propose what could be achieved if LittleDarwin was to be developed further.

A.1 Strategy Based on Current Results

The main message current results communicate is the fact that the test coverage of the analyzed tests for the Segmentation component is not adequate enough\(^1\). However, the results show that the problem is not as bad as branch coverage suggests. One reason is that the effects of functional tests are ignored by branch coverage, and captured by mutation analysis. This leads to the great

\(^1\)Segmentation component is thoroughly tested using an extensive acceptance test suite which is approved by several health standard organizations such as FDA. This study was conducted with the exclusion of these tests, and the results only apply to the analyzed part of the test suite.
difference between two results. It is also important to note that many basic algorithms lack unit testing, and are only tested by functional tests. The reason is that most of these small classes are directly inherited from the legacy software considered as stable. This strategy is potentially dangerous, increasing the risk of letting through a dormant bug into the final product. Therefore, it is advisable to develop unit tests for the important and most used components in the short term, and for the whole Segmentation component in the long term.

Development of new unit tests can be optimized by manually inspecting the survived mutants for each class. This way, it is easier to find the type of faults the tests need to look for. Looking into the detailed mutation coverage also provides a map of where the coverage is inadequate, reducing the time needed to write tests by avoiding already covered parts.

A.2 Proposed Experiments Based on Current State of LittleDarwin

One of the interesting observations about the final results is that the functional tests provide a lot of mutation coverage for diverse range of classes. Since this experiment excluded all the acceptance tests, there is not enough info on the coverage of these acceptance tests. So, it would be interesting to see how effective these tests are in catching low-level errors. There might be additional benefit in developing more functional tests instead of sticking to the development of unit tests.

One of the interesting aspects of the software which needs to be studied is the similarities of the survived mutants and the bugs that found their way into the production code before. This can help to determine the potential locations of dormant bugs by checking similar kind of survived mutants. It is also of interest to observe the effects of developing new tests for the weak spots discovered by mutation analysis in mutation coverage and branch coverage. This will provide an instant quality measure for the newly developed tests.

A.3 Future Experiments Possible by Expanding LittleDarwin

The most important improvement that needs to be developed in LittleDarwin is the introduction of a change-based system of performing mutation analysis.
It will result in major improvement in the performance of the tool, and its practicality in a continuous integration environment. This will allow the mutation coverage to replace branch coverage as the main quality metric of the test suite. It also allows the experiments which where considered impractical due to their demanding nature to be performed in such environment.

Introduction of new mutation operators provides support for more accurate modeling of the faults that occur in the program. However, addition of new mutation operators results in a lot of new mutants. With change-based mutation analysis, this problem can be solved.

It is also possible to develop support for new programming languages in LittleDarwin. This would allow testing several components in different programming languages at once, allowing the quality of integration tests to be assessed for a combination of these components. This is, of course, dependent on the availability of mutation operators which are designed to target the faults most commonly happening by integrating several components.
B.0 Abstract

Mutation testing is one of the leading methods of testing the test-suites. In this article we review the literature about mutation testing in order to provide a guide for a developer who wants to design a mutation testing framework. We explore the diverse nature of mutation operators as a main ingredient to any mutation testing framework, and also discuss the features of the tools developed in the past decade.

B.1 Introduction

Since the dawn of software engineering, one of the problems developers faced was to be able to make sure their software is free of bugs. A bug is defined as a fault in the system which results in unexpected behavior [74]. Therefore, to avoid any complications resulting from the existence of bugs in the software, it is in the interest of the developer to test their software. Automated test suites are designed to handle this task.

The demand for adequate testing requires a test suite to be of a certain level of quality; Otherwise, the ability of a test suite to catch bugs comes into question. There are a lot of legacy systems with inadequate test suites which need improvement to become of a level which can be useful for trustworthy
regression testing. This creates the need for an idea to assess the quality of a test suite in a reliable, repeatable and falsifiable way. Mutation testing [17] provides a formal method to determine the quality of a test suite by injecting intentional bugs into a system and counting the number of caught bugs. There are several studies which show that mutation testing is successful in simulating the real-life bugs a test suite might catch [4][38]. Hence, there is widespread interest in its usage as an analysis method for scientific purposes.

Even though the idea of mutation testing looks simple, in reality, it is hard to implement, adapt and run. Since changing the software in a way that fabricates a real bug is dependent on the software in question, mutation testing frameworks are usually dependent on the syntax of the language used in the software. In addition to this, technologies used in developing the software (e.g. build system, test suite runner) are also a limiting factor for implementing a general mutation testing framework for a particular software. This means that developing a universal mutation testing framework demands an enormous amount of work to be done.

In this study, a brief overview of the state of the literature in the last decade was done in order to gather information regarding the available tools and methods used in their development. The goal of this study is to help the design and/or adaptation of a mutation testing framework on a particular system by providing information regarding the existing efforts available in the literature. Different aspects of a mutation testing framework is discussed and statistical data is used to find out which approach have had more proponents.

The rest of the paper is structured as follows. In section B.2, the background of the subject is discussed. In section B.3 the research questions are discussed. In section B.4, the method to search for articles and filtering the results is discussed. In sections B.5, an analysis of the results is done. In section B.6, the statistics extracted from the literature is provided; and in section B.7, similar studies are brought into attention.

### B.2 Background

The idea of mutation testing was first mentioned in a class paper by Lipton [54] and later developed by DeMillo, Lipton and Sayward [17]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [12]. However, the computationally demanding nature of mutation testing made it a difficult subject for empirical research and the progress in the subject was very slow. Advances in hardware capabilities and the increase in demand of the market due to availability of legacy code caused a resurgence in the topic and since late 1990’s the topic has been a hot trend among software engineers.
Mutation testing is the process of injecting bugs into software and counting the number of caught bugs. This procedure is run in the following manner: First, faulty versions of the software is created by applying a single change in the system. This change is created by a mutation operator. A mutation operator is a piece of software which changes the system under test in such a way that it includes a single bug. After generating the mutants, the test suite is run on each of these mutants. If there is an error or failure during the execution of the test suite, the mutant is regarded as killed. However, if all tests pass, it means that the bug could not be caught by the test suite and the mutant has survived. If two mutants return the same output for all possible inputs, they are called equivalent mutants. The final result is the number of killed mutants excluding the number of equivalent mutants divided by the number of all generated mutants excluding the number of equivalent mutants.

The mutation operator is an important part of this procedure. The more mutation operators there is, more mutants will be generated and as a result the amount of computations needed is increased. Moreover, if the mutation operators are not sufficiently different, they increase the chance of creating equivalent mutants. Thus, there is an interest to find which subset of mutant operators would result in sufficient testing of the system. There has been multiple articles regarding this topic [53][52][36].

B.3 Research Goals

This study is designed towards those who need to acquire relevant knowledge about the current state of the art in the topic of mutation testing. This is useful for those who need to design a new mutation testing framework from scratch in order to analyze a system which does not work with the available tools, or those who are adapting an already existing framework into a usable system for their software based on their own criteria. This study also provides a simple guide for choosing a tool based on its capabilities in the form of an overview.

In order to achieve the goal mentioned above, the following research questions must be answered:

- **RQ1.** What mutation operators are essential in developing a mutation testing framework? Are these mutation operators applicable to every context, or are they specific to a special category of softwares?

- **RQ2.** Which tools implement these mutation operators? What programming languages are supported by each tool? What is the availability status of the tool?
B.4 Survey Method

To conduct this survey, several automated tools were used to gather, merge and handle the results. In order to be able to use automated tools, the reference libraries were limited to IEEE\(^1\) and ACM\(^2\). A customized version of SLRTool\(^3\) was used in order to retrieve preliminary results from the aforementioned libraries. The search was conducted using two queries — "Mutation Testing" and "Mutation Analysis" — in order to find all relevant results.

Then, Mendeley\(^4\) was used to merge all resulting files and to remove duplicate entries. After removing all the results from before year 2004, the resulting entries were manually inspected to be categorized according to the research questions. This lead to the final count of 50 results overall. Of this 50 articles, 18 were about the mutation operators and 32 were related to the developed tools. An overall summery of the survey method can be seen in Figure B.1.

\(^1\)http://ieeexplore.ieee.org
\(^2\)http://dl.acm.org
\(^3\)http://github.com/javipeg/SLRtool
\(^4\)http://www.mendeley.com
### B.5 Analysis

After narrowing down the results of the survey and categorizing them according to the subject, the resulting articles were read to seek the answers of our research questions. In the following sections, I try to summarize the information acquired through this survey.

#### B.5.1 Essential Mutation Operators

The first set of the mutation operators designed were reported in [40]. These operators which work on very basic entities were designed for the tool Mothra which was designed to mutate FORTRAN77 programming language (Figure B.2). In 1996, Offutt et al [53] determined that only a selection of few mutation operators are enough to produce the same coverage with a four-fold reduction of the number of mutants. This reduced set of operators remained more or less intact in the future research papers (Figure B.3).

With the popularity of the object-oriented programming, in order to adapt mutation testing to the new concept, the need for new mutation operators was felt. Several studies proposed new mutation operators [44] [13], and some of them were designed to prove the usefulness of object-oriented operators [42] [18]. Ahmed et al in [3] did a complete survey on this subject.
As a sample, object-oriented mutation operators which were developed into MuJava [45] can be seen in Figure B.4.

During past decade, the focus of the researchers were on creating new mutation operators for special purposes such as targeting certain security problems [64] [75] or language specific mutation operators [1] [11] [65]. These mutation operators, even though important in their own context, does not relegate into the general concept of mutation testing. Therefore, one needs to study them in case of designing a system for a specific purpose, but mostly they are ignored in general. As seen in section B.6, most popular mutation operators used in the tools available are by far the classical ones.

### B.5.2 Mutation Analysis Tools

There are several tools developed in the past decade for the purpose of mutation analysis. An overview of the specifications of these tools can be seen in Figure B.5. What follows is a small description for each of these tools.

**MuJava**  MuJava is a publicly available mutation system for Java that supports both traditional statement-level mutants and newer inter-class mutants. Relevant articles: [56] [45] [46] [66] [57]

**SQLMutation**  SQLMutation is a tool to automatically generate mutants of SQL database queries [69].

**Certitude**  Certitude is a commercial software tool performing mutation analysis on C programs with the focus on the microelectronics industry [29].

**ExMAn**  ExMAn is an automated, general and flexible mutation analysis framework which allows for the comparison of different quality assurance tech-
<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC</td>
<td>Access modifier change</td>
</tr>
<tr>
<td>IHD</td>
<td>Hiding variable deletion</td>
</tr>
<tr>
<td>IHI</td>
<td>Hiding variable insertion</td>
</tr>
<tr>
<td>IOD</td>
<td>Overriding method deletion</td>
</tr>
<tr>
<td>IPO</td>
<td>Overridden method calling position change</td>
</tr>
<tr>
<td>IOR</td>
<td>Overridden method rename</td>
</tr>
<tr>
<td>ISK</td>
<td>super keyword deletion</td>
</tr>
<tr>
<td>IPC</td>
<td>Explicit call of a parent’s constructor deletion</td>
</tr>
<tr>
<td>PNC</td>
<td>new method call with child class type</td>
</tr>
<tr>
<td>PMD</td>
<td>Instance variable declaration with parent class type</td>
</tr>
<tr>
<td>PPD</td>
<td>Parameter variable declaration with child class type</td>
</tr>
<tr>
<td>PRV</td>
<td>Reference assignment with other compatible type</td>
</tr>
<tr>
<td>OMR</td>
<td>Overloading method contents change</td>
</tr>
<tr>
<td>OMD</td>
<td>Overloading method deletion</td>
</tr>
<tr>
<td>OAO</td>
<td>Argument order change</td>
</tr>
<tr>
<td>OAN</td>
<td>Argument number change</td>
</tr>
<tr>
<td>JTD</td>
<td>this keyword deletion</td>
</tr>
<tr>
<td>JSC</td>
<td>static modifier change</td>
</tr>
<tr>
<td>JID</td>
<td>Member variable initialization deletion</td>
</tr>
<tr>
<td>JDC</td>
<td>Java-supported default constructor create</td>
</tr>
<tr>
<td>EOA</td>
<td>Reference assignment and content assignment replacement</td>
</tr>
<tr>
<td>EOC</td>
<td>Reference comparison and content comparison replacement</td>
</tr>
<tr>
<td>EAM</td>
<td>Accessor method change</td>
</tr>
<tr>
<td>EMM</td>
<td>Modifier method change</td>
</tr>
</tbody>
</table>

Figure B.4: Object-oriented mutation operators (adapted from [44])
niques such as testing, model checking, and static analysis on C and Java code [10].

**MUGAMMA** MUGAMMA implements a software system so that when it executes in the field, it will determine whether users’ executions would have killed mutants (without actually executing the mutants) [39].

**MuClipse** MuClipse is an automated mutation testing plug-in for Eclipse [66].

**CSAW** CSAW is a lightweight C language mutation tool [22].

**Jumble** Jumble is a byte code level mutation testing tool for Java which inter-operates with JUnit [33].

**MUFORMAT** MUFORMAT is a prototype tool which is targeted at format string bugs in C [64].

**CREAM** CREAM is a mutant generator for C# [19].

**MUSIC** MUSIC is MUtation-based SQL Injection vulnerabilities Checking (testing) tool that automatically generates mutants for the applications written in Java Server Pages (JSP) and performs mutation analysis [63].

**MILU** Milu is an efficient and flexible C mutation testing tool designed for both first order and higher order mutation testing [35].

**Javalanche** Javalanche is an open source framework for mutation testing Java programs with a special focus on automation, efficiency, and effectiveness which assesses the impact of individual mutations to effectively weed out equivalent mutants [62].

**GAmera** GAmera is an automatic mutant generator for WS-BPEL compositions. It is composed by three different elements: an analyzer, a mutant generator and a system that executes and evaluates the mutants. GAmera is based in genetic algorithms and attempts to minimize the number of generated mutants, independently of the number and type of mutation operators, without losing relevant information. It can also detect potentially equivalent mutants allowing to improve the quality of the test suite [21] [20].
<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Language</th>
<th>Operators</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>MuJava</td>
<td>2004</td>
<td>Java</td>
<td>Classical, Object-Oriented</td>
<td>Free</td>
</tr>
<tr>
<td>SQLMutation</td>
<td>2006</td>
<td>SQL</td>
<td>SQL Specific</td>
<td>Free</td>
</tr>
<tr>
<td>Certitude</td>
<td>2006</td>
<td>C, C++</td>
<td>Classical</td>
<td>Commercial</td>
</tr>
<tr>
<td>ExMAN</td>
<td>2006</td>
<td>C, Java</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>MUGAMMA</td>
<td>2006</td>
<td>Java</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>MuClipse</td>
<td>2007</td>
<td>Java</td>
<td>Classical, Object-Oriented</td>
<td>Free</td>
</tr>
<tr>
<td>CSAW</td>
<td>2007</td>
<td>C</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>Jumble</td>
<td>2007</td>
<td>Java</td>
<td>Classical, Object-Oriented</td>
<td>Free</td>
</tr>
<tr>
<td>MUFORMAT</td>
<td>2008</td>
<td>C</td>
<td>FSB</td>
<td>Not Available</td>
</tr>
<tr>
<td>CREAM</td>
<td>2008</td>
<td>C#</td>
<td>Classical</td>
<td>Not Available</td>
</tr>
<tr>
<td>MUSIC</td>
<td>2008</td>
<td>SQL</td>
<td>SQL Specific</td>
<td>Not Available</td>
</tr>
<tr>
<td>MILU</td>
<td>2008</td>
<td>C</td>
<td>Classical, Higher-Order</td>
<td>Free</td>
</tr>
<tr>
<td>Javalanche</td>
<td>2009</td>
<td>Java</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>GAmera</td>
<td>2009</td>
<td>WS-BPEL</td>
<td>Classical, WS-BPEL Specific</td>
<td>Free</td>
</tr>
<tr>
<td>AjMutator</td>
<td>2009</td>
<td>AspectJ</td>
<td>AspectJ Specific</td>
<td>Free</td>
</tr>
<tr>
<td>Proteum/AJ</td>
<td>2010</td>
<td>AspectJ</td>
<td>AspectJ Specific</td>
<td>Not Available</td>
</tr>
<tr>
<td>Judy</td>
<td>2010</td>
<td>Java</td>
<td>Classical, Object-Oriented</td>
<td>Free</td>
</tr>
<tr>
<td>X-Mut</td>
<td>2010</td>
<td>XSLT</td>
<td>Classical</td>
<td>Not Available</td>
</tr>
<tr>
<td>MutMut</td>
<td>2010</td>
<td>Java</td>
<td>Classical</td>
<td>Not Available</td>
</tr>
<tr>
<td>SMutant</td>
<td>2011</td>
<td>SmallTalk</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>MAJOR</td>
<td>2011</td>
<td>Java</td>
<td>Classical</td>
<td>Free</td>
</tr>
<tr>
<td>ConSMutate</td>
<td>2012</td>
<td>SQL</td>
<td>SQL Specific</td>
<td>Not Available</td>
</tr>
<tr>
<td>Bacterio</td>
<td>2012</td>
<td>Java</td>
<td>Classical</td>
<td>Not Available</td>
</tr>
<tr>
<td>SMT-C</td>
<td>2012</td>
<td>C, C++</td>
<td>Classical, Semantical</td>
<td>Not Available</td>
</tr>
<tr>
<td>MuCheck</td>
<td>2014</td>
<td>Haskell</td>
<td>Haskell Specific</td>
<td>Free</td>
</tr>
</tbody>
</table>

Figure B.5: Mutation testing tools
**AjMutator**  AjMutator is a tool for the mutation analysis of PCDs. AjMutator implements several mutation operators that introduce faults in the PCDs to generate a set of mutants. AjMutator classifies the mutants according to the set of join points they match compared to the set of join points matched by the initial PCD [16].

**Proteum/AJ**  Proteum/AJ realizes a set of requirements for mutation-based testing tools and overcomes some limitations identified in previous tools for aspect-oriented programs [24] [23].

**Judy**  Judy presents an innovative approach to mutation testing that takes advantage of a novel aspect-oriented programming mechanism, called ‘pointcut and advice’, to avoid multiple compilation of mutants and to speed up mutation testing [48].

**X-Mut**  X-Mut is a tool which provides an adaptation of mutation analysis for XSLT language [43].

**MutMut**  MutMut is a tool for Mutation Testing of Multithreaded Code [28].

**SMutant**  SMutant is the first mutation testing tool for Smalltalk programs [27].

**MAJOR**  MAJOR is a fault seeding and mutation analysis tool that is integrated into the Java Standard Edition compiler as a non-invasive enhancement for use in any Java-based development environment [37] [38].

**ConSMutate**  ConSMutate is a mutation testing framework which guides concolic testing using mutation analysis for test case generation for database applications [61] [60].

**Bacterio**  Bacterio is a Java mutation testing tool that automates the tasks to perform mutation analyses and that implements a set of mutation techniques that reduce the costs of mutation and the execution mutant time drastically [70].

**SMT-C**  SMT-C is a semantic mutation testing tool for C [14].

**MuCheck**  MuCheck is a mutation testing tool for Haskell programs [41].

---

5Pointcut Descriptors
B.6 Statistics

A quick look into the available data reveals that the subject has kept its importance throughout the past decade steadily. Figure B.6 shows the number of articles and number of tools developed in each year of the past decade. As it can be seen, the trend has remained more or less steady for the past decade with the average of 3.2 articles per year, and 2.5 tools per year. An interesting fact that can be observed in the Figure B.6 is that during 2013, no tools have been developed, and only one tool was developed in 2014. However, 3 articles were written during this period about the previously available tools which shows that the tools still can be expanded by incorporating new ideas.

Out of the 25 tools developed in the past decade, 10 were developed for Java, and 7 for the C family (C, C++, C#). Interestingly, SQL holds the third place with 3 tools developed for it which shows the interest due to the rise in the demand of database applications. Figure B.7 shows all the languages the tools have been developed for and their distribution.

One interesting phenomena observed from this data is the fact that most of the tools use classical mutation operators and only a few of them use object-oriented mutation operators. This, combined with the fact that the majority of the tools target Object Oriented languages show that there is a lack of interest in developing object-oriented mutation operators. This may be due to the fact that the empirical research proving the usefulness of such operators is still incomplete; However the existing empirical research supports this claim [42]. Figure B.8 shows the usage of the mutation operators in tools.
Figure B.7: Target languages of the tools

Figure B.8: Mutation operators used in the tools
B.7 Related Work

There are other surveys tackling the subject of mutation testing. One of the surveys which does a comprehensive analysis of the subject is Jia et al 2011 [34]. In this survey, the process of mutation testing is explained, and its problems and solutions in the literature are discussed. Also, empirical analysis of a lot of the tools discussed in this report is provided. Offutt et al in [55] discuss the history of mutation testing and the state of the art now, and provides insight into the future of the field. A good reference for analysis of the mutation testing tools for Java is Delahaye et al 2013 [15]. In the mentioned article, mutation testing tools for Java are compared based on efficiency, compatibility with current technologies and multiple other factors.