

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mutation Testing Cost Reduction Techniques for Java Applications

David Roberto Cravo da Mata



Mestrado em Engenharia de Software

Supervisor: Prof. Ana Paiva, PhD

Supervisor: Prof. João Bispo, PhD

July 20, 2023

Mutation Testing Cost Reduction Techniques for Java Applications

David Roberto Cravo da Mata

Mestrado em Engenharia de Software

July 20, 2023

Abstract

Software systems have increased in size and complexity over the years and are now an integral part of our lives, being present everywhere. This rapid expansion has brought some drawbacks, as nearly all people have encountered software that works differently than intended. This is especially noticed in mobile applications due to the latest increase in mobile devices like smartphones and tablets, which has dramatically changed how people interact with each other. This creates a huge need for software testing to ensure the quality of the application and the detection of defects before the application reaches the end user.

Exhaustive software testing is well known to be impossible to achieve, as testing all the possible combinations of inputs is not feasible due to the time and resources needed. Normally, a test suit is created in the testing phase, combining several unit tests. In this context, mutation testing was conceived by DeMillo et al. [9] as a means to evaluate the effectiveness of a test suite. By introducing small artificial mutations into the code, mutation testing evaluates if the test suit detects or not those mutations. Each mutation represents a slight modification of the original code, simulating a potential fault. A mutant is said to be killed if it leads to the failure of at least one test. Typically, there is a high computational cost associated with mutation testing that is related to the high number of mutants that are generated that result in a high number of test executions.

For a long number of years, new techniques to reduce the cost of mutation testing have been proposed by researchers. However, mutation testing is still not widely used in the software industry. The main value this research adds is the proposal of two new methodologies that reduce the cost of mutation testing.

The first proposal intends to improve Mutant Schemata by allowing the mutation operator to be agnostic to whatever is being applied in the traditional or schemata format. This approach enables the application of mutation operators that have not been implemented in schemata format. Additionally, different implementations for Java projects that use Maven and Gradle as build system are proposed and analysed. The performance degradation of the mutant schemata is also analysed against the original program.

The second proposal tackles the fact that mutation testing does not consider this evaluation of the software projects. Software, throughout its life, undergoes continuous updates and improvements. Usually, different versions are produced when introducing new features, security updates and optimisations. Each change represents a new version of the application that must be tested. Our proposal uses Git versioning to identify what files to mutate, reducing the number of mutants generated up to 87%.

Keywords: Mutation Testing, Mutant Schemata, Android Testing

Resumo

Os sistemas de software têm aumentado de tamanho e complexidade ao longo dos anos e agora são parte integral de nossas vidas, estando presentes em todos os lugares. Esta rápida expansão trouxe algumas desvantagens, pois a maioria das pessoas encontra ou já encontrou software que funciona de maneira diferente ao pretendido. Isto é notado especialmente em aplicações móveis devido ao aumento mais recente de dispositivos móveis, como smartphones e tablets, que mudou drasticamente a forma como as pessoas interagem umas com as outras. Isto cria uma enorme necessidade de testar para garantir a qualidade do aplicativo e a detecção de defeitos antes que o mesmo chegue ao utilizador final.

É conhecido que o teste exaustivo de software é impossível de ser alcançado, pois testar todas as combinações possíveis de entradas não é viável devido ao tempo e recursos necessários. Normalmente, um test suit é criado na fase de testes, contendo vários testes unitários. Nesse contexto, mutation testing foi concebido por DeMillo et al. [9] como um meio de avaliar a eficácia de um conjunto de testes. Ao introduzir pequenas mutações artificiais no código, mutation testing avalia se o test suit detecta ou não essas mutações. Cada mutação representa uma pequena modificação do código original, simulando uma potencial falha. Diz-se que um mutante foi morto se levar à falha de pelo menos um teste. Normalmente, há um alto custo computacional associado a mutation testing que está relacionado com o alto número de mutantes gerados que resultam num elevado número de execuções de teste.

Por um longo período de anos, novas técnicas para reduzir o custo de mutation testing foram propostas pelos investigadores. No entanto, mutation testing ainda não é amplamente utilizado na indústria de software. O principal valor que esta pesquisa introduz é a proposta de duas novas metodologias que reduzem o custo de mutation testing.

A primeira proposta pretende melhorar Mutant Schemata permitindo que o operador de mutação seja agnóstico ao formato que está sendo aplicado. Esta abordagem permite a aplicação de operadores de mutação em Mutant Schemata que não tinha sido possível até ao momento. Adicionalmente, diferentes implementações para projetos que usam Maven e Gradle como build systems são propostas e analisadas. A degradação do desempenho dos Mutant Schemata é também analisada em relação ao programa original.

A segunda proposta aborda o fato de mutation testing não considerar a evolução dos projetos de software. O software, ao longo de sua vida, passa por atualizações e melhorias contínuas. Normalmente, diferentes versões são produzidas ao introduzir novas funcionalidade, atualizações de segurança e otimizações. Cada alteração representa uma nova versão do aplicativo que deve ser testada. A nossa proposta utiliza o versionamento do Git para identificar quais os ficheiros que devem ser mutados. Desta forma foi possível reduzir o número de mutantes gerados até 87%.

Keywords: Mutation Testing, Mutant Schemata, Android Testing

Acknowledgements

I want to express my deepest gratitude to my dissertation advisers, Ph.D. Ana Paiva and Ph.D. João Bispo, for their continuous expertise, guidance and motivation throughout all the stages of this research work. Their support and knowledge has encouraged me throughout this process and has immensely contributed to my education and knowledge.

I also want to acknowledge my appreciation for the Faculty of Engineering of the University of Porto, as well as all my professors who accompanied me. Thank you for your warm welcome to Porto, providing unique challenges, learning opportunities, and encouragement to students.

Finally, I also want to acknowledge my family and friends, especially my girlfriend Raquel Faria, who has been a constant source of support, care and encouragement throughout this journey.

David Mata

*“In the middle of every difficulty
lies opportunity.”*

Albert Einstein

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem Statement and Research Goals	2
1.3	Outline	3
2	State of the Art	5
2.1	Mutation Testing	5
2.2	Mutant Generation	6
2.3	Approaches to Cost Reduction	7
2.3.1	Mutant Reduction Techniques	8
2.3.2	Equivalent Mutant detection techniques	12
2.3.3	Execution Cost Reduction Techniques	13
3	Mutation Testing Tools	17
3.1	Java Mutation Testing Tools	17
3.1.1	MuJava	17
3.1.2	JavaLanche	18
3.1.3	Judy	18
3.1.4	Jumble	18
3.1.5	Major	18
3.1.6	PIT	19
3.2	Android Mutation Testing Tools	19
3.2.1	MuDroid	19
3.2.2	μDroid	19
3.2.3	MDroid+	20
3.2.4	Edroid	20
3.2.5	DroidMutator	20
3.2.6	BacterioWeb V2	20
3.3	Summary	21
4	Mutation Testing Methodology	23
4.1	Traditional Methodology	23
4.2	Proposed Methodology	24
4.2.1	Mutant Schemata	25
4.2.2	Application of Mutations Only to the Changed Part of the Code	25

5	Framework Proposal	27
5.1	Introduction	27
5.1.1	LARA Framework	27
5.1.2	Mutator API	29
5.2	Tool Design	29
5.2.1	Data Model	32
5.3	Git Integration	33
5.4	Mutant Generation	34
5.4.1	Classpath Configuration	37
5.4.2	Mutation Process	38
5.5	Mutation Operators	42
5.6	Test Execution	43
5.7	Application Interaction	46
6	Empirical Evaluation	47
6.1	Impact of Mutation Operators	48
6.1.1	Impact of Mutation Operators On Java Projects	48
6.1.2	Impact of Mutation Operators On Android Projects	51
6.1.3	Summary	52
6.2	Impact of Mutant Schemata	53
6.2.1	Impact of Mutant Schemata on a Java Application	54
6.2.2	Impact of Mutant Schemata on an Android Application	55
6.2.3	Summary	57
6.3	Impact of Git Versioning Reduction	57
7	Conclusions and Future Work	59
7.1	Results	60
7.2	Further Work	60
A	Lara Environments Code	63
A.1	Main Lara Environment Code	63
A.2	Traditional Mutation Lara Environment Code	66
A.3	Java Mutant Schemata Lara Environment Code	69
A.4	Android Mutant Schemata Lara Environment Code	72
	References	79

List of Figures

2.1	Basic idea of mutation testing	6
2.2	Example of a mutation on the AST	7
2.3	Number of mutants per operator	9
2.4	Exponential growth of the number of kth-order mutants [36]	11
2.5	Polo et al. mutation process approach [36]	11
4.1	Conventional Mutation Process described by Pradeep Singh et. all[3]	24
5.1	LARA Framework Architecture [34]	28
5.2	Mutation Algorithm from LARA Mutator API	30
5.3	Tool Developed Architecture	31
5.4	Data Model of the developed tool	32
5.5	Parallel execution of the different LARA Environments	35
5.6	File Structure of Result Mutant Schemata Project	36
5.7	File Structure of Result Traditional Mutation Project	36
5.8	Tool API Description	46
6.1	Generation Time For Each Java Mutation Operator	49
6.2	Total Size For Each Java Mutation Operator	49
6.3	Average Time for each mutant execution using Traditional Mutation and Mutant Schemata	50
6.4	Execution Times for Java Traditional Mutation and Mutant Schemata	51
6.5	Generation Time For each Android Mutation Operator	52
6.6	Total Size For Each Android Mutation Operator	53
6.7	Execution Times for Android Traditional Mutation and Mutant Schemata	54
6.8	Total Execution and Elapsed Time for the Different Mutant Schemata Projects	55
6.9	Total Test Execution Times and Elapsed Times for the Different Mutant Schemata Projects in Android	56

List of Tables

2.1	An example of a mutation	6
2.2	Mothra Mutant Operators.	10
2.3	Mutation operators analysed by Bluemke et al. [6]	11
2.4	An example of Equivalent Mutation	13
5.1	List of Operators - General Specific [41]	44
5.2	List of Operators - Java Specific [41]	44
5.3	List of Operators - Android Specific [41]	45
6.1	Generation Results for Java Operators	48
6.2	Test Execution Results	50
6.3	Generation Results for Android Specific Operators	52
6.4	Test Execution Results Android	53
6.5	Schemata Performance Degradation in Java	55
6.6	Schemata Performance Degradation in Android	55
6.7	File Metrics From Different Versions of Aegis	57
6.8	Generation Metrics of Mutant Schemata with Git Improvement	58
6.9	Generation Metrics of Mutant Schemata Without Git Improvement	58

Abbreviations

API	Application Programming Interface
APK	Android Application Pack
AST	Abstract Syntax Tree
CPH	Competent Programmer Hypothesis
DSL	Domain Specific Language
FOM	First Order Mutant
HOM	Higher Order Mutants
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
JVM	Java Virtual Machine
TCE	Trivial Compiler Equivalence
TUMS	Testing Using Mutant Schemata
UUID	Universally Unique Identifier

Chapter 1

Introduction

1.1 Context

The software industry is constantly growing and evolving, and software is now an integral part of our lives. Software is everywhere, from public services (healthcare, education and justice) to computer-controlled systems (transports and IoT). These systems have increased in size and complexity over the years, making us increasingly dependent on them. This rapid expansion of information technology systems has brought many benefits. It is easier than ever to travel, you can call a driver with a click of a button and you can make a payment by taking a photo of a QR Code with your phone. However, this rapid expansion has come with some drawbacks. Nearly all people have already encountered software that did not work as intended causing inconvenience, loss of time and money and even, in extreme cases, health injuries and death. This is especially noticed in mobile applications due to the latest increase in the number of mobile devices like smartphones and tablets, which has dramatically changed the way people interact with each other. As a matter of fact, it is estimated that there are more connected mobile devices than people [33]. This creates a huge need for software testing to ensure the quality of the application to the end-users and to detect the majority of the defects before release.

Exhaustive software testing is well known to be impossible to achieve, as testing all the possible combinations of inputs is not feasible due to the time and resources needed. Also, the increasingly faster time to market does not help the detection of defects before the application reaches the production environment. To mitigate the presence of defects, testing is introduced in the early stages of software development, sometimes even before the actual development of the product (test-driven development). Normally, a test suite is created in the testing phase, combining several unit tests. A unit test compares the result of a small code block against the expected one, failing if different. But how do we know if a test suite is effective? Several metrics can briefly explain how well-designed a test suite is. Decision coverage (also known as branch coverage) requires testing each program's decision outcome. Statement coverage requires that tests cover every code

statement. These metrics, amongst others, only consider the percentage of code that is executed and sometimes that does not show the true quality of the test suit.

Mutation testing is also a technique that can be used to evaluate the test suite. The underlying principle of mutation testing is that faults present in software are due to small and simple syntactic errors. Mutation testing creates several mutants, each one being a copy of the original code with a small syntactic change that aims to generate an artificial defect. If the test suite detects the change introduced by the mutation, we can have a higher confidence level in its quality.

1.2 Problem Statement and Research Goals

Although the benefits of mutation testing have already been shown, it is still not widely adopted in the software industry. The biggest problem associated with mutation testing is the high computational cost. Depending on the size of the project, we can have a high amount of changes that can be applied and, subsequently, a high quantity of tests that need to be run to detect those changes. This is especially noticed in Android apps, where the tests are executed on devices with less computational power. Also, the current mutation testing applications lack documentation and ways to integrate into the software development process.

This dissertation aims to find relevant ways to reduce the cost of mutation testing and make it more applicable in the software industry. Its objective is to study proposed techniques and see the impact that each one has. With the intention of addressing mutation testing cost-reducing techniques, the following questions that will be responded to in the following chapters were formulated.

- **RQ1** - Which techniques can reduce the cost of mutation testing?
- **RQ2** - Which mutation tools exist, and how do they approach cost reduction in mutation testing?
- **RQ3** - Can different approaches (e.g. traditional, schemata) use the same implementation of mutation operators?
- **RQ4** - Which impact do mutation operators have on the cost of mutation testing?
- **RQ5** - What is the impact of the number of mutations in the performance of the schemata?
- **RQ6** - In the context of an evolving software project, do we maintain the effectiveness of fault detection when we apply mutation testing only to the parts of the code that have been recently changed?

1.3 Outline

Beyond this introductory chapter, this document is structured into additionally 6 chapters. Chapter [2](#), the literature review on mutation testing and the already implemented cost reduction techniques, where relevant contributions on the area of mutation testing are analysed. Chapter [3](#) analyses and compares mutation testing tools for Java and Android applications. In chapter [4](#), we present our methodology that improves the schemata implementation by turning the operator implementation agnostic to whatever is being applied in schemata format or traditional. We also present a second methodology that only applies mutation to changed files on Git. In chapter [5](#) is presented the developed mutation tool, with emphasis on the decisions taken that defined the architecture of the tool. In chapter [6](#) we compared the results obtain with the developed tool, analysing the resulted gains. Finally, chapter [7](#) presents an overview of the conclusions and future work.

Chapter 2

State of the Art

This chapter presents a comprehensive overview of the current status of mutation testing. The goal is to find and present relevant information, including recent search findings and methodologies on cost reduction techniques.

2.1 Mutation Testing

Mutation testing as originally conceived by DeMillo et al. in 1978 [9] is based on two fundamentals: the Competent Programmer Hypothesis (CPH) and the Coupling Effect. The CPH implies that most software faults present in programs delivered by competent programmers are due to small and simple syntactic errors. The CPH also alludes that these programs tend to be very similar to the expected and are close to being correct. The Coupling Effect states that complex faults are coupled with simple faults and test data that finds those simple faults can also find those complex ones. Mutation testing is an alternative approach to test the quality of the test suit. Mutation testing is defined by Jia and Harman [17] as a fault-based testing technique that provides a testing criterion known as mutation adequacy (also known as mutation coverage or mutation score). In the most basic form, a mutant is a program with a small syntactic change in the code that aims to generate an artificial defect. For example, a possible change to the program p in the mutant p' could be a mutation from the arithmetic expression $(a < b)$ to the arithmetic expression $(a > b)$ as seen in table 2.1. This syntactic change follows a certain rule that is called mutant operator. Typically, mutant operators are designed to introduce defects in edge cases, creating faults in a particular part of the code. In this case, when the mutant only differs from the original program by one simple modification, it is called First Order Mutant (FOM). Mutants, however, don't have to be restricted to having only one mutation. That is the case of Higher Order Mutants (HOM). HOM differ from FOM by having more than one mutation, typically creating more complex defects.

The goal of mutation testing is to infer the quality of the test suit based on if it detects deviations in behaviour between mutants and the origin program 2.1. However, not all mutants are

Table 2.1: An example of a mutation

Program p	Program p'
...	...
if (a<b)	if (a>b)
return 1;	return 1;
...	...

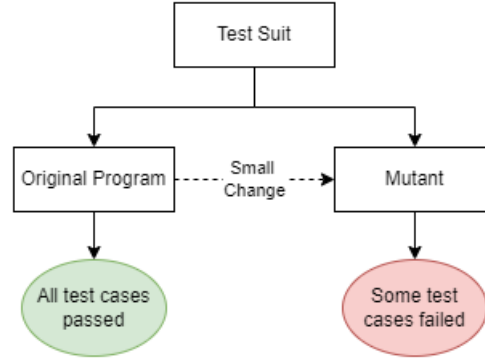


Figure 2.1: Basic idea of mutation testing

relevant, as there are mutants that are killed by all tests (trivial mutants), and mutants that suffered changes that do not modify the meaning of the original program (equivalent mutants). Having in mind all of the aforementioned, the testing criterion of mutation testing, mutation score (MS) can be defined as the division of the number of killed mutants (K) and the subtraction of the equivalent mutants (E) to the total number of generated mutants (M) as seen on equation 2.1.

$$MS = \frac{K}{M - E}, \text{ with } 0 \leq MS \leq 1 \quad (2.1)$$

2.2 Mutant Generation

Two main processes can be used for the creation of mutants, the first is through the Abstract Syntax Tree (AST) and the second by the byte-code. The AST is a tree-like representation of the code, where every entry of code creates a node on that tree. By applying mutation operators it is possible to introduce changes in the AST (figure 2.2). That way, when converting from the AST to the source code it is possible to achieve a semantic change. The other process of creating mutant through the byte-code is similar to the one of using the AST, with the difference being that the mutation operators are applied in the Byte-code. This brings the advantage of not being needed the compilation of the mutated code. However, this approach brings two negative aspects [4]. The first is that if a new version of the compiler is created, the byte-code structure can change, requiring more maintenance for supporting those changes. The second is that working on the compiled code is harder to design mutants than when using the AST.

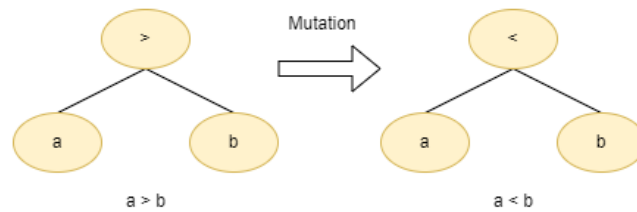


Figure 2.2: Example of a mutation on the AST

The mutation process is composed of four steps, (1) execution of the original program, (2) generation of mutants, (3) execution of each mutant and finally, (4) analysis of the mutants [35]. Typically, there is a high cost associated with mutation testing that is related to the last two steps, the execution and analysis. This high cost is related to the high number of mutants that are generated, especially if we considered that there is a huge number of mutations that can be applied in a huge number of places of the program. Additionally, in many cases, it can be difficult to determine if a mutant is laborious to kill or if it is an equivalent mutant that can not be killed.

2.3 Approaches to Cost Reduction

This section address RQ1 by presenting relevant Cost Reduction approaches to mutation testing. For a long number of years, new techniques to reduce the cost of mutation testing have been proposed by researchers. In the traditional classification proposed by Offutt and Untch [29], those cost reduction techniques could be classified as “do fewer”, “do smarter”, and “do faster”.

"The 'do fewer' approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The 'do smarter' approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution. The 'do faster' approaches focus on ways of generating and running each mutant program as quickly as possible." [29]

This classification was created a long time ago and it is no longer representative of the recent techniques for cost reduction. Some of them fit in more than one category and some don't fit in any at all. This was a problem noticed by Pizzoleto et. al. [35], who decided to create a classification composed of six different types of techniques, namely:

- Reducing the number of mutants: the main goal is to reduce the quantity of mutants that will be executed without decreasing effectiveness.
- Automatically detecting equivalent mutants: the main goal is to detect automatically equivalent mutants and to remove them.
- Execution faster: the main goal is to reduce execution time.

- Reducing the number of test cases or the number of executions: the main goal is to detect smaller test sets or identify groups of similar mutants to decrease test runs while keeping effectiveness.
- Avoiding the creation of certain mutants: the main goal is to avoid the creation of certain mutants by applying certain mutation operators that only create non-trivial mutants.
- Automatically generating test cases: the main goal is to automatically generate test cases, to kill the majority of the mutants.

2.3.1 Mutant Reduction Techniques

As there is a big computational cost inherent to executing the test set against all mutants, one popular research problem is how to reduce the number of mutants. For a given set of mutants M , and a set of test data T , the mutation score is given by $MS_T(M)$. The problem of reducing the number of mutants can be defined as finding a subset of mutants M' from M that respects equation 2.2.

$$MS_T(M) \simeq MS_T(M') \quad (2.2)$$

This section will introduce four techniques that reduce the number of mutants.

2.3.1.1 Mutant Sampling

Mutant Sampling reduces the number of mutants by randomly choosing a subset of mutants from the entire set. This technique was introduced by Acree [1] and Budd [7]. The process for this reduction is simple, firstly all mutants (M) are generated, then x per cent of those (M) mutants are selected. There are many studies with the primary focus on the way that random mutants are selected in different programming languages.

Anna Derezińska et al. evaluated six different sampling criteria in object-oriented mutation [11], in particular fully random ($x\%$ of mutants are randomly selected), class random ($x\%$ of mutants from each class are selected, keeping mutants equally distributed for each class), file random ($\%$ of mutants from each file are selected, keeping mutants equally distributed for each file), method random ($\%$ of mutants from each method are selected, keeping mutants equally distributed for each method), mutation operator random ($\%$ of mutants from each mutant operator are selected, keeping mutants equally distributed for each mutant operator) and namespace random ($\%$ of mutants from each namespace are selected, keeping mutants equally distributed for each namespace). The conclusions reached were that in case of object-oriented the recommended sampling method was class random sampling, although it brings a small decrease in mutation score accuracy. Also, the authors noticed, that in practice, the overall time of mutation testing is strongly influenced by the number of tests to be performed and not only the number of mutants.

2.3.1.2 Mutant Clustering

Mutant Clustering uses clustering algorithms to group similar mutants into clusters and then chooses a subset of each cluster. This technique was first proposed by Shamaila Hussain master thesis [38]. The process starts with the creation of all mutants, followed by the application of a clustering algorithm to group mutants into clusters. The idea is that mutants in the same cluster are similar and are killed by the same test cases. Then, a subset of each cluster is used and the rest of the mutants are discarded.

In Shamaila Hussain thesis, two clustering algorithms were used, the k-means clustering algorithm and the agglomerative clustering algorithm. These two were applied to five different programs, with different lengths and non-identical numbers of mutants and test cases. The results revealed that even though there was a reduction in the number of mutants, the strength of the set of mutants was not reduced [38].

2.3.1.3 Selective Mutation

Selective Mutation reduces the number of mutants by reducing the number of mutation operators applied. The principle is that if mutants generated by a mutation operator A are killed by the same tests that the mutants generated by a mutation operator B, then only one of the mutation operators is needed (either A or B). The goal is to reduce the number of mutation operators, consequently reducing the number of mutants, without reducing test efficiency. Selective mutation was first introduced by Mathur as "constrained mutation" [25] and was later extended by Offutt [28], which called it selective mutation.

In the work presented by Offutt, he analyses the Mothra mutant operators represented in table 2.2. These 22 mutation operators were defined to test FORTRAN-77 programs. Each mutation operator was defined as a three-letter acronym. From the analysis of the mutants generated by applying the Mothra mutation operators, Offutt verified that the number of mutants generated per operator was not evenly distributed. Experimental results showed that removing all the mutants operators except five created the same coverage as with all mutants. The only five sufficient mutants were ABS, AOR, LCR, ROR and UOI. This results in a reduction of cost of fifty times with large programs and 4 times with small programs, while achieving a mutation score of 99.5% [17][38].

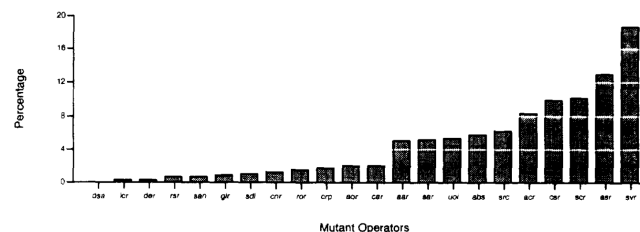


Figure 2.3: Number of mutants per operator

Different mutation operators were proposed by different authors for different programming languages. Another example is the work of Banzi et al. that uses a multi-objective approach to

Table 2.2: Mothra Mutant Operators.

Mutant Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

determine the essential operators [5]. The main difference was that Banzi et al. approach treated the selection of mutation operators as a multi-objective optimization problem, where there is no unique and single solution. The solution would depend on the different objectives, which could be the reduction of the number of mutants, improving the mutation score, amongst others (as different sets of operators exist to satisfy multiple objectives). The approach was tested with real C programs and compared with other reduction techniques namely random selection, obtaining better results.

Bluemke et al. also tried reducing the number of mutants generated by the mutation operators in the Java programming language [6]. The analysed mutation operators can be seen in table 2.3. From the reduction of the mutation operators, only the operator AOIS was unsatisfactory, as the others didn't affect the mutation score. This was due to the same test case being able to kill several different mutants, with the omission of certain mutants not affecting results.

Table 2.3: Mutation operators analysed by Bluemke et al. [6]

Mutant Operator	Description
AOIS	Arithmetic Operator Insertion, Short-cut
ROR	Relational Operator Replacement
LOI	Logical Operator Insertion
AORB	Arithmetic Operator Replacement, Binary
COI	Conditional Operator Insertion
AOIU	Arithmetic Operator Insertion, Unary

2.3.1.4 High Order Mutation

High Order Mutation combines the typical n different mutants with only one mutation, also known as first-order mutants (FOM), into one higher-order mutant (HOM). The idea was to create mutants stronger and more difficult to kill but also reduce the number of mutants, as a test case that kills a higher order mutant typically also kills the correspondent first order mutants. This approach for creating mutants was first introduced by Jia et al. [16].

Mutants can be classified according to the number of mutations. There can exist a large number of different higher-order mutants, however, not all are relevant. Higher order mutants can be classified as interesting and uninteresting [15]. The uninteresting higher-order mutants don't assist in fault-based testing (i.e. if the program becomes more faulty than we expected). The interesting mutants are the ones that are potentially able to assist the programmer in fault-based testing.

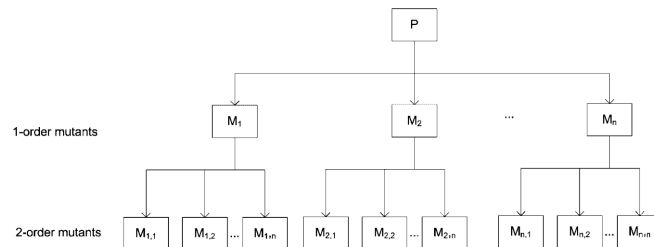
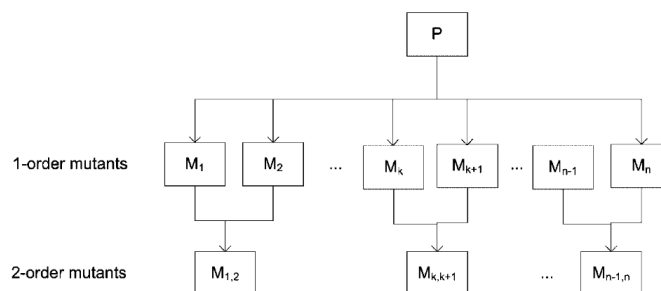
Figure 2.4: Exponential growth of the number of k th-order mutants [36]

Figure 2.5: Polo et al. mutation process approach [36]

One of the problems that also arise with higher order mutation testing is that the number of generated mutants can be higher than when using only first order mutants. This is due to the exponential growth of mutants with each generation, as shown in figure 2.4. In the approach used by Polo et al., three strategies have been proposed to tackle this problem (Last To First, Different Operators, and Random Mix)[36]. Also, the maximum number of first order mutants combined was two, creating only second order mutants as seen in figure 2.5.

There are four main types of strategies to generate second-order mutants, First To Last, Random Mix, Different-Operators, and Each-Choice [12]. These strategies choose mutants in pairs from a list of all first-order mutants (sorted alphabetically). Mutants that apply the mutation in the same line of code can only be combined into a higher-order mutant if the mutation is not in the same sub-expression. For example, considering the expression $A < B \ \&\& \ B > C$, two mutants can be combined only if one applied the mutation to the sub-expression $A < B$ and the other two to the sub-expression $B > C$. The strategies are given in the description below.

- First To Last: prioritizes the selection of mutants in the extremities of the list by groping the first and last unused mutants from a list into a higher order mutant.
- Random Mix: the higher-order mutant is selected randomly from the list of mutants.
- Different-Operators: selects the first unused mutant and then selects the next mutant that applies a different operator.
- Each-Choice: It selects the first unused mutant followed by the next unused mutant from the list.

Prado Lima et al. concluded that almost 49% of the works with higher order mutants apply search-based algorithms and that the main study programming language is Java [12].

2.3.2 Equivalent Mutant detection techniques

An equivalent mutant is a mutant p' , which is semantically different from the original program p , however, it has the same behaviour as p . Table 2.4 shows a mutant that changed the original operator $<$ in the for cycle to the operator $!=$. If the i value is not updated inside the cycle, the program p will behave like the mutant p' . As the mutation score is based on nonequivalent mutants, reaching a mutation score of 100% is impossible if there is no correct detection of equivalent mutants. This may lead the tester to believe that the test suit is inadequate when it actually is [17]. Commonly, the detection of equivalent mutants must be carried out by testers. By removing these “useless” mutants, we reduce the effort required to perform mutation as there are fewer mutants to run and fewer mutants that testers need to classify as equivalent and improve the accuracy of the mutation score [31]. This has generated interest in the community to find a way to reduce and remove equivalent mutants automatically.

Papadakis et al. proposed Trivial Compiler Equivalence (TCE) technique that uses compiler technology to detect equivalent mutants [30]. The idea is that TCE can detect when two different

Table 2.4: An example of Equivalent Mutation

Program p	Program p'
<pre> ... for (int i = 0; i<10; i++){ ... (value of i is not changed) ... } ... </pre>	<pre> ... for (int i = 0; i != 10; i++){ ... (value of i is not changed) ... } ... </pre>

versions of the same program when they have similar source codes. TCE compiles each mutant, compares its machine code and detects those similarities. When this happens, it is safe to declare both as functionally equivalent, and there is no point in differentiating them with test data. This way, it is possible to reduce equivalent mutants and duplicate mutants. The empirical study showed that TCE can detect from 7% to 30% of the equivalent mutants.

2.3.3 Execution Cost Reduction Techniques

Another way to reduce the cost of mutation testing is by optimizing the mutants execution process. In this section, three different approaches will be presented and explained.

2.3.3.1 Strong, Weak, and Firm Mutation

Strong, Weak and Firm Mutation are three categories that we can classify the way that analyses if a mutant is killed during the execution process. Strong mutation is the traditional way of mutation testing, which considers that a given mutant p' from an original program p is said to be killed only if p' has a different output from the original program [17]. Weak mutation was proposed by Howden [13] to optimise mutation testing. The main difference of weak mutation compared with strong mutation is that a mutant is said to be killed right after the execution of the mutation point if the execution is different from the original program. In contrast, in strong mutation, a mutant is only said to be killed after the execution of the entire program [17]. Weak mutation, however, tends to be less effective than strong mutation. To overcome the disadvantages of weak mutation Woodward and Halewood [42] proposed firm mutation. Firm mutation combines weak and strong mutation, analysing mutants after the mutation point and after execution.

2.3.3.2 Run-time Optimization Techniques

There are several main techniques for run-time optimization, among which stand out the interpreter-based technique, the compiler-based technique and the byte-code translation technique. The interpreter-based technique was mainly used in the first generation of mutation testing tools [17].

In this technique, the result of the mutant is interpreted from its code directly, being this technique influenced by the cost of the interpretation. The compiler-based technique is the traditional and regular way to achieve program mutation [17]. In this technique, each mutant is first compiled and then executed by the test cases. The byte-code translation technique differs from the others, as mutants are generated by the compiled code instead of the source code, resulting in mutants that can be executed directly without any additional compilation.

2.3.3.3 Mutant Schemata

Metamutans or Mutant Schemata create a parameterized program that groups all mutants into only one mutant called Metamutant [35]. When running this Metamutant, a parameter needs to be provided to tell it which version to run. This technique of creating a Metamutant was called the MSG method [39].

Untch et al. first introduced a prototype system called TUMS (an acronym for Testing Using Mutant Schemata). The goal was to create automated Metamutants and compare their performance against the traditional way of creating mutants. Their empirical study concluded that the MSG-generated mutant was much faster than traditionally creating all mutants, with speed increases as high as an order of magnitude [39].

René Just implementation of mutant schemata uses an external driver class to gain access to the mutant identifier *M_NO* [18]. The class is only resolved at runtime and triggers each mutant. René Just was able to implement commonly used mutation operators to mutate Java applications. An example of the resulting method can be seen on listing 2.1.

```

1 public int eval (int x ) {
2     int a = 3 , b = 1 , y ;
3
4     y = ( M_NO==1) ? a + x :
5         ( M_NO==2) ? a / x :
6         ( M_NO==3) ? a \% x :
7                 a * x ; // original
8     y += b;
9     return y;
10 }
```

Listing 2.1: Example method of René implementation [18]

Mutant Schemata with Extra Code or MUSIC was also proposed by Pedro Mateo et al. as a way to reduce the cost of mutation testing [24]. In their approach, they successfully reduce the cost of mutation testing by reducing the number of required executions to detect infinite loops created by the mutations. Including extra code that takes the form of a loop counter that tracks the number of iterations, the execution environment can determine if a mutant is stuck in an infinite loop. Even though the execution time of the original program increased significantly, their empirical study demonstrated a reduction in the number of executions by 77%. An example of the resulting code can be seen in listing 2.2.

```

1  class ClassA{
2      public intinc(int a, int b){
3          for(b<10){
4              a++;
5              if(exec(m1)){
6                  b = b-2;
7              }else if(exec(m2)){
8                  b = b*2;
9              }else if(exec(m3)){
10                 b = b/2;
11             }else {
12                 mutantsGenerated(m1,m2,m3);
13                 b = b+2;//original statement
14             }
15             increaseLoop();
16         }
17         return a;
18     }
19 }

```

Listing 2.2: Mutant Schemata implementation of Pedro Mateo et. al [24]

Francisco Azevedo [4] also implemented the mutant schemata resorting to system properties. Each mutation would be surrounded by an *if* statement in his implementation. An example of the resulting code can be seen in listing 2.3.

```

1  void move(int x, int y){
2      if(java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_0")){
3          setXPos(getYPos() - x);
4      }
5      if(java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_1")){
6          setXPos(getYPos() * x);
7      }
8      if(java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_2")){
9          setXPos(getYPos() / x);
10     }
11     if(java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_3")){
12         setXPos(getYPos() / x);
13     }
14     if(java.lang.System.getProperty("MUID").equals(null)){
15         setXPos(getYPos() + x);
16     }
17 }

```

Listing 2.3: Mutant Schemata implementation of Francisco Azevedo [4]

Diego Naveiras [27] implemented mutant schemata using Wrappers at a byte code level. The idea is to replace the mutated class with a Wrapper class (also named controller class) with the same methods as the original class. Then, a new class that extends the controller class is created

for each mutation. The wrapper class then controls what mutated class to load based on a property saved on a file. This implementation allows an equal number of generated mutants as the traditional mutation process and allows the combination of other complex techniques like Higher Order Mutation. However, using wrappers increases the processing required with inheritance, and it does not preserve the original object-oriented characteristics of the original. Diego Naveiras also proposed a mutant schemata implementation using *MutantDriver* and meta procedures. In the listing 2.4, it is possible to verify an implementation of the meta procedure *PLUS* on the *MutantDriver*. When executing the test suite, the *PLUS* meta procedure reads the value of *currentMutant* from a file. Then, depending on the value, the original or mutated versions can be returned.

```

1 public static int PLUS(int a, int b, int... indexes){
2     loadCurrentMutant();
3     int location = Arrays.binarySearch(indexes, currentMutant);
4
5     if (currentMutant == 0 || location < 0) return a + b;
6     if (currentMutant == 0) return a - b;
7     if (currentMutant == 1) return a * b;
8     if (currentMutant == 2) return a / b;
9     if (currentMutant == 3) return a % b;
10    return a + b;
11 }
```

Listing 2.4: Mutant Schemata implementation of Diego Naveiras using *MutantDriver* [27]

To conclude, the mutant schemata approach has some limitations. Depending on how the control flow is implemented, it can be challenging to discern which specific section of code is being executed at any given time. Also, mutation operators that change methods signature or access specifiers and classes access specifiers can be challenging to implement in the mutant schemata form. Another drawback is the impact on code readability, as the number of mutants can diminish the clarity of the code. Additionally, supposing that the implementation of the Metamutant is not correctly, new bugs or issues can be introduced into the code base, resulting in a program that might have different behaviour than the original.

Chapter 3

Mutation Testing Tools

As showed by Domenico Amalfitano et al., different tools implement different features [2]. This chapter focuses on address RQ2, by giving an overview of the different cost-reduction techniques that are used by mutation tools. Firstly, all the Java-specific mutation tools will be presented, and then they will be compared against the Android-specific ones. All the significant differences and similarities will be described. Only relevant mutation testing tools that do have some type of documentation available online are considered.

3.1 Java Mutation Testing Tools

3.1.1 MuJava

MuJava [22] is one of the earliest Java mutation tools and was released in 2005. At the time of writing, it is not under active development, and it was last updated in August 2016. It was also released under an open-source Apache License 2.0¹. The last version of MuJava was compatible with Java 7 as its highest-supported Java version. Similarly, it supported JUnit 4 as the latest version of the unit testing framework. Its original goal was to reduce the execution cost of mutation testing in object-oriented programs. To do so, it implemented two strategies: mutations on a byte code level and mutations using mutant schemata and reflection. MuJava implemented two types of mutation operators, behavioural mutations and structural mutations. The behavioural mutations are done using compile-time reflection and the mutant schemata. The structural mutants are done using byte code translation. Even though the tool allows multiple Java classes to be loaded, each test is executed manually one by one.

¹<https://github.com/jeffoffutt/muJava>

3.1.2 JavaLanche

Javalanche [37] was released in 2009 with the main goal of overcoming two main problems, efficiency and equivalent mutants. It is publicly available ² and has not been updated since 2012. The last supported Java version was the 6, and it has support for JUnit 4 unit test framework. As efficiency was prioritised, many cost-reduction techniques were applied. Only a small set of mutation operators that were considered relevant was implemented (selective mutation). Mutant schemata was also used on a byte code level, avoiding the need to recompile the mutated code. Additionally, the mutation generation is executed in parallel, and the coverage data is used for the test execution. Not all tests are executed against all mutants, only the ones that cover the mutated line.

3.1.3 Judy

Judy [23] was originally conceived to outperform MuJava. The latest version stable version was the 2.1.0 from January of 2014 ³. The last supported Java version was the 8, and it has support for JUnit 4 unit test framework. It only offers a command line interface. Judy automatically generates mutations on a byte code level. Additionally, it also takes advantage of parallel execution. Judy is based on the FAMTA Light, a novel approach at the time. FAMTA Light takes advantage of the pointcut and advice mechanism that identifies specific points in the program flow. In the empirical study, it was shown that it had significant gains compared to MuJava [23].

3.1.4 Jumble

Jumble is a mutation tool that was originally conceived by a commercial company in New Zealand, Reel Two ⁴. It was last updated in May of 2015, with version 1.3.0. It has support for integration with Eclipse IDE and also has a command line interface. The last supported Java version was the Java 8, and it has support for JUnit 4 unit test framework. Jumble only allows a single class selection, and it only applies one mutation at a time on a byte code level. Multiple test cases can be selected, and it prioritises test execution. It first executes all tests on the original code and stores the time for each one. Then, it executes tests from the lowest time to the highest time. It does not have any equivalent mutation detection to reduce the impact of equivalent mutants.

3.1.5 Major

Major [19] is a mutation framework that implements a large number of optimisations. This resulted in Major being able to apply mutation testing to programs with more than 200k lines and a result of 150k mutants. At the time of writing, the latest release was version 2.0.0 on January 2023 ⁵. The last supported Java version was the Java 8, and it has support for JUnit 4 unit test framework.

²<https://github.com/david-schuler/javalanche>

³<http://mutationtesting.org/judy/documentation/>

⁴<https://jumble.sourceforge.net/index.html>

⁵<https://mutation-testing.org/index.html>

Moreover, Major comes with a domain-specific language (DSL), which can be used to define and extend mutant operators. Major adopts strategies of Strong and Weak mutation. Similar to other tools, it also implements test suit prioritisation. Major is integrated with the Java compiler and implements the mutations on the AST, alongside with the compilation of the original code.

3.1.6 PIT

PITest or PIT [8] is one of the few tools that continues to be developed, with the last version at the time of writing being the 1.14.2, released on June 2023⁶. It requires Java 8 or above, and either JUnit or TestNG can be used. Pit aims to be easily integrated with Integrated Development Environment (IDE), possessing extension support for Eclipse and IntelliJ, and can be integrated with Ant, Maven and Gradle build tools. Similarly to other tools, it also employs multiple cost-cutting measures, such as byte code manipulation and test execution prioritisation. PIT only applies mutations to methods that are executed, as it generates mutants based on code coverage. The mutation operators that are applied are also only the ones that do not produce a large number of mutations. Additionally, the test execution is prioritised based on three factors, line coverage, text execution speed and test naming convention. Only tests that exercise the mutated line of code are used order by increasing execution time.

3.2 Android Mutation Testing Tools

3.2.1 MuDroid

Mudroid [10] is one of the earlier Android-specific mutation tools. It was originally released in 2015. The last update at the time of writing was on May of 2016⁷. It has a command line interface. It applies mutation testing at the byte code level, and it only requires the APK to do so. It has the capability of executing tests that use either JUnit, Robotium or Espresso. Mudroid does not implement any cost reduction technique like mutant schemata, parallel execution or equivalent mutant detection. Mudroid, however, has the capability to execute the test suit against each mutant and compare the results again to the original code so that the mutation score is calculated.

3.2.2 μ Droid

μ Droid [14] is a mutation testing framework that was originally conceived in 2017. It mainly targets the lack of tools that access the energy performance of applications, as it only applies energy-related mutations. The mutations are applied to the AST, and both first-order and higher-order mutation testing are available. The tool has integration with Eclipse IDE and supports only Java 8. It is composed of two components, an Eclipse Plugin that implements the mutation operators and generates the mutated code and a Runner/Profiler component that executes the test suite and profiles the power consumption of the device.

⁶<https://github.com/hcoles/pitest>

⁷<https://github.com/Yuan-W/muDroid>

3.2.3 MDroid+

MDroid+ [26] is a mutation testing framework for Android Applications that is open source⁸. It was originally conceived in 2018, with the release of version 1.0.0 with no recent development at the time of writing for over four years. It only applies Android-specific operators, and all the mutations are implemented on the AST or the resources files (like XML files). It has a command line interface that allows the user to define which mutations to apply. It does not implement any cost-reduction technique related to mutant schemata and equivalent mutant generation. Each operator results in a copy of the original program, with only one difference the mutated Java file or resource file.

3.2.4 Edroid

Edroid [21] is a graphical user-friendly Android mutation tool that was originally presented in 2018. It only allows mutation on XML files, even though the original goal was to be extended to Android Java operators. It also does not have any cost reduction technique applied, as it does not detect equivalent mutants. All the mutations are applied using a regex over the XML files, and each operator results in a copy of the original program. Although it allows the generation of first-order and high-order mutants, it does not provide any mutant schemata capabilities. Additionally, it does not provide an automatic way to execute the test suit and calculate the mutation score.

3.2.5 DroidMutator

DroidMutator [20] is an Android Specific mutation tool that was originally presented in 2020. The tool is public, and the last development at the time of writing was in November of 2020⁹. DroidMutator is composed of three separate components. The first is the mutator component that uses a JavaParser to parse each source file into an AST. The mutations are applied only in feasible mutation locations. The second component is a Builder that has the responsibility to compile the mutant. The last component is the Launcher, which is responsible for executing the tests. DroidMutator does not implement any cost-reduction technique besides trying to reduce stillborn mutants.

3.2.6 BacterioWeb V2

BacterioWeb V2 [27] is one of the latest Android mutation tools that was proposed on June 2021. It is originally based on BacterioWeb v.1 which was released in 2017 [40]. Its main focus is to optimise mutation testing in Android applications. It was developed with distributed computing in mind, as it supports multiple executions in different machines. It also implements selective and sampling mutation, parallel generation of mutants and mutant schemata. Additionally, the

⁸<https://gitlab.com/SEMERU-Code-Public/Android/Mutation/MDroidPlus>

⁹<https://github.com/SQS-JLiu/DroidMutator>

test execution can be executed on multiple devices at the same time, as the tool manages parallel execution of tests.

3.3 Summary

Upon analysing the aforementioned mutation tools, it became evident that the Java specific tools did incorporate more cost reduction techniques compared to the Android-specific tools. Most Java specific tools did implement the mutation on a byte code level, while the majority of the Android-specific ones focused on implementing the mutations on the AST. Additionally, multiple Java specific tools did offer some sort of equivalent mutant detection, while no Android-specific tool provided this feature. Furthermore, it was possible to verify that there are Java mutation tools like PIT that do have daily updates with an open-source community, while the majority of Android-specific tools lack updates, integration in the software development process, documentation and support.

Chapter 4

Mutation Testing Methodology

This chapter starts by presenting the traditional methodology commonly followed when executing mutation testing. This methodology is considered the baseline for comparison. Building upon the foundation of the traditional methodology, the chapter then proceeds to the presentation of the two proposed methodologies that were developed in the scope of this research work. Both proposed methodologies present alternative strategies to improve the efficiency of mutation testing.

4.1 Traditional Methodology

Mutation testing has generated interest in the scientific community for over two decades, and the conventional mutation process as described by Pradeep Singh et. al[3] can be seen in figure 4.1.

The traditional process starts with the original program P . Through the application of the mutation operators, the mutants P' are generated. Subsequently, a test set T is executed on the original program P , to evaluate its correctness. If P is incorrect or if issues are identified, the necessary fixes are implemented, with the process starting all over again. If P is correct, the test set is then executed against all the live mutants P' . If all mutants are killed by the test set, the execution concludes. Otherwise, if any mutants survive, equivalent mutants should be analysed and the need for additional tests should be evaluated before executing the test set against the remaining live mutants.

This process can be time-consuming, especially when considering the time required for the compilation and test execution of each mutant. To better illustrate this, let us consider a program P that has a test suite that takes 5 minutes to execute, and let us assume that the program takes 1 minute to compile. Now, let us apply 500 mutations to program P . The total time in the traditional way is given by the time to execute the test suite in all mutants, $500 \times 5\text{min} = 2500\text{min}$, plus the time of the compilation of the 500 mutants, $500 \times 1\text{min}$. This results in a total time of 3000min, or 50 hours, which is highly undesirable. This, however, does not consider the generation time for each mutant, which is also noticeable, as seen in chapter 6. Some tools try to tackle this problem

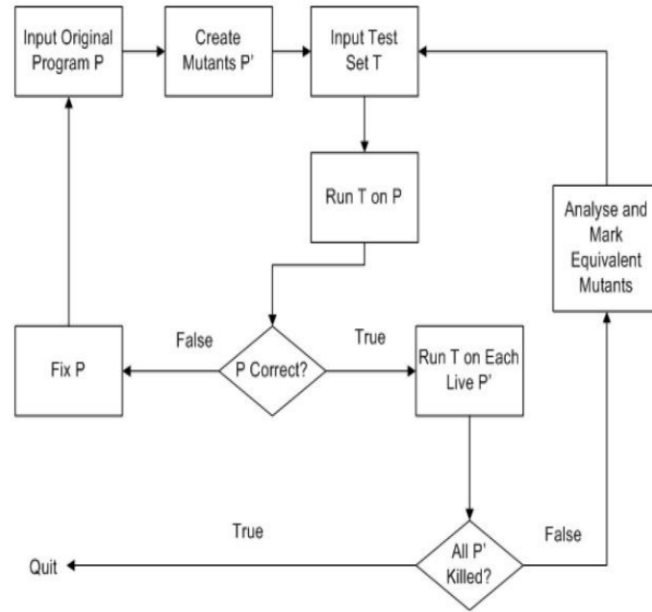


Figure 4.1: Conventional Mutation Process described by Pradeep Singh et. al[3]

by applying mutations to the compiled code, but as seen in 2.2 there are several drawback of this approach.

The cost of mutation testing is even more noticeable in Android. The Android-specific build process is more complex than the traditional Java applications, as all resources need to be packaged in an Android Application Pack (APK). Additionally, Android applications include several resources like XML files, images and layouts that need to be processed in order to build the APK. Furthermore, android-specific libraries like the ones from the Android Software Development Kit (SDK) need to be processed. All of these steps add up to the compilation time. The test execution is different in Android compared to traditional Java projects. In Java, there are unit tests that are responsible for testing small, specific parts of the code. Android has unit tests but also has Instrumented tests. These tests are specific to Android apps and require an emulator or an actual device to be executed, as they test Android-specific features.

4.2 Proposed Methodology

This research work proposes two different approaches for mutation testing. The first approach improves the application of mutant schemata to Java projects that use both Maven and Gradle as build systems in a way that the mutation operator is independent of the schemata implementation. This also includes Android applications written in java that use Gradle and Maven. The second approach targets applying mutations only to the different parts of the code instead of the traditional application to the entire project. Both proposed methodologies will be described in detail below.

4.2.1 Mutant Schemata

The first proposed methodology suggests a novel approach to apply mutant schemata in a way that is independent of the mutation operator used. In this approach, the mutation operator responsibility is to alter the AST, without being aware of whether traditional or schemata-based mutations are being employed. This allows mutation operators to be applied as mutant schemata operators that have not been done until this point. The process of creating the mutant schemata involves consolidating all mutants into a single mutant. The execution is then controlled through a parameter as explained in [2.3.3.3](#). This way we aim to reduce the compilation time substantially in Java projects that use Maven or Gradle. Additionally, it is expected that the implementation of the Schemata is abstract from the mutant operator, meaning that this way, any operator could be easily implemented, as it only needs to perform the code mutation and is not responsible for creating the schemata.

It was decided that to achieve this, the mutations would have to be implemented on the AST. This way, the implementation of the mutant schemata would be easier as byte-code transformations were not necessary. The process would consist of the following steps:

1. Search the AST to store the points on where the mutations will be applied.
2. Apply each mutation to the origin code in the schemata form to the AST.
3. Convert from the AST to Java code.
4. Compile the created project.
5. Execute the tests n times for the n mutations generated.

Even though this proposed methodology does not apply the mutation on the byte code level, we would have only to compile once, independently from the number of mutants generated, as all mutants would be aggregated in a single project. Although the process consists on the same steps, there are some differences between the implementation from the mutant schemata on Java projects that use Maven or Gradle. All the differences will be detailed in section [5.4](#).

4.2.2 Application of Mutations Only to the Changed Part of the Code

The second proposed methodology uses Git versioning to identify what files to mutate. Currently, many companies use Git in software development. Git is a distributed version control system for tracking source code changes and it allows multiple developers to collaborate in the same code base by providing different branches for independent development. It also provides mechanisms for merging those branches and resolving the conflicts that arise when doing so. In general terms, Git is used for code management, version control and collaboration between developers.

Software, throughout its life, undergoes continuous updates and improvements. Usually, different versions are produced when introducing new features, security updates and optimisations.

Each change represents a new version of the application or project that must also be tested. Until this point, mutation testing does not consider this evaluation of the software projects.

Even though significant efforts have been made to reduce the cost of mutation testing, each time that there is a new release, there would exist equal mutations to parts of the code that did not change. This requires additional time to compile, test and analyse equivalent mutants. A simple example of this would be a project with 500 Java files that generated 3000 mutants. If only a slight change was made to a single file, all the mutations would have to be repeated to all files. Then, all the tests would have to be made on all the generated mutants. This is a significant problem that is correlated with mutation testing still not being widely adopted in the industry.

The proposed approach intends to remove duplicate mutants through the different versions of the applications by only applying mutations to the changed files. We consider that only applying mutation testing to the changed files would bring the benefit of reducing the number of mutants without reducing the number of detected faults on the test suit. This will also significantly reduce the cost of mutation testing, as only changed files will result in mutations instead of the entire project. The detailed implementation is explained in further detail in [section 5.3](#).

Chapter 5

Framework Proposal

This chapter starts by presenting a source-to-source compilation Framework called Lara and its APIs. This framework facilitated the creation of the mutated projects. Then, we present the developed tool architecture and data models as well as all the decisions made during its development, including the tool's integration with Git. Moving forward, we present the step-by-step process of applying mutations to both traditional Java projects and the Java Android project. Finally, we emphasize the implementation of the automatic execution of the test suite on traditional Java projects and on Java Android projects.

5.1 Introduction

The main goal of this research work is to develop a cost-reduction approach to mutation testing. The focus is not only on traditional Java projects that use Maven or Gradle as build systems but also on Java Android applications, where the mutation testing cost is much more noticeable. The objective is to compare the efficiency of the proposed solutions that tackle cost reduction in different environments. Furthermore, another challenge that is going to be addressed is the fact that most tools have limited or no integration into the software development pipeline [3.3](#). This will be overcome with the developed tool, which can integrate with GIT, a free, open-source distributed version control system.

5.1.1 LARA Framework

LARA is a framework for source-to-source compilation developed in Faculdade de Engenharia da Universidade do Porto (FEUP). It provides tools for building that apply source-code analyses and transformations described as JavaScript scripts. Currently, there are several LARA compilers,

Clava¹ for C/C++, MATISSE² for Matlab and Kadabra³ for Java (that will be used in this research work).

The architecture consists of three main Java components, the LARA Engine, the Weaving Engine and the source-to-source compiler 5.1. In the case of Kadabra, it uses Spoon as the source-to-source compiler. Spoon is an open-source library that allows developers to write their own source code analysis by generating a metamodel representative of the origin code [32]. The LARA Engine interprets and executes LARA scripts written in JavaScript. The Weaving Engine connects the LARA scripts and the source-to-source compiler, which is responsible for parsing, transforming and generating the target source code [34].

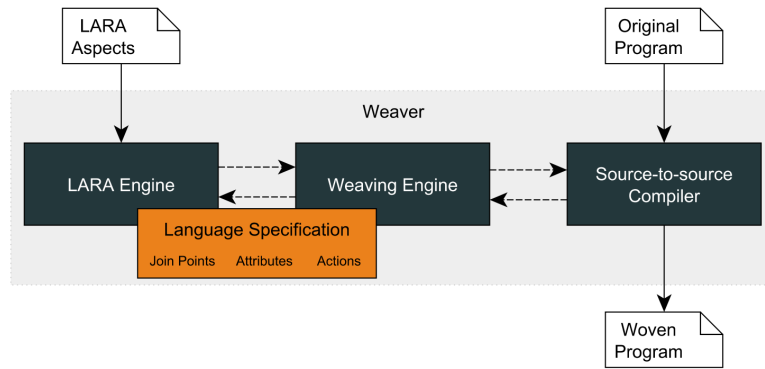


Figure 5.1: LARA Framework Architecture [34]

The LARA Framework represents each AST node as a joinpoint, which works as a common interface wrapped around the particular node. Each LARA compiler uses different AST nodes from different libraries but similar join points which share the same base interface. All types from the original code, like classes, methods, statements or expressions, are encapsulated in this representation. LARA offers a wide range of functions and APIs that allow analysing and changing the AST. These capabilities include the insertion and deletion of code, among other functionalities. The APIs used were the following:

- **lara.io:** This API was used to handle operations of input and output on files. It encloses functions like *copyFolder()*, *writeFile()*, *getSeparator()* and *getFiles()* that were used.
- **lara.Strings:** This API was used to handle operations with strings. It encloses functions like *uuid()*, *toJson()*,
- **lara.mutation.Mutator:** This API was used to handle mutation operators. The detailed implementation can be seen in 5.1.2.
- **weaver.Query:** This API was used to handle querying the AST, as it provides methods for selecting joinpoints.

¹<https://specs.fe.up.pt/tools/clava/>

²<https://specs.fe.up.pt/tools/matisse/>

³<https://specs.fe.up.pt/tools/kadabra/>

- **weaver.Weaver:** This API was used to handle parallel instances of LARA environments. It encloses functions like *runParallel()* and *writeCode()*.
- **weaver.Script:** This API was used to handle parallel instances of LARA environments, allowing one to combine the results of all instances into just one.

5.1.2 Mutator API

The Mutator API is based on the behaviour design pattern Template Method, which defines a skeleton of an algorithm for applying a mutation. This strategy allows the subclasses to override different steps without changing the actual structure of the algorithm. The algorithm for applying the mutation is composed of the following functions:

- **addJp(joinpoint):** This function is responsible for adding the given *joinpoint* to the mutation points list only if the operation can be applied for the given joinpoint. In simpler terms, if the joinpoint respect the rules defined, it will be stored in the mutation points list.
- **getMutationPoint():** This function returns the current list of joinpoints stored to apply the mutation.
- **hasMutations():** This function returns true or false depending on whether the list of mutation points is empty.
- **_mutatePrivate():** This function has the responsibility of applying the mutation. Depending on the operator, it has the ability to remove, change or add new code.
- **_restorePrivate():** This function is responsible for inverting the process made by the *_mutatePrivate()*, meaning that it will restore the code to the original code.

With the aforementioned functions described, we can define the algorithm as follows [5.2](#). The first is to pass all joinpoints from the AST through the *addJp()* function, and only the ones compatible with the mutation operator are added to the mutation points list. Secondly, it is verified the presence of any joinpoints in the list. If any is present, the call to the function that applies the mutation can be made, consequently changing the AST to conform with the defined in the *_mutatePrivate()* function. Also, the joinpoint where the mutation was applied will be removed from the mutation points list. Finally, the restore function can be called to restore the AST to its original format and move on to the next existing join point.

5.2 Tool Design

Having in mind that the LARA Framework already had a way of implementing mutations and manipulating the AST, it became evident that the tool would have to be built with the integration of LARA in mind. As the purpose of this research work is not to implement a source-to-source compilation tool that is controlled by LARA, it was clear that the developed tool would have

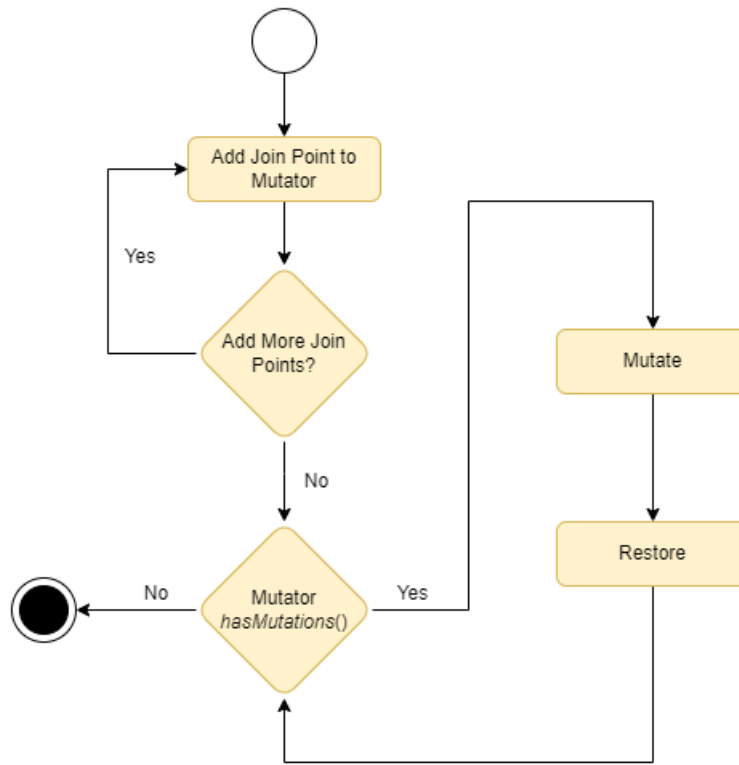


Figure 5.2: Mutation Algorithm from LARA Mutator API

integration with Kadabra. Additionally, the tool needed to have Git integration with a web-hosting service like GitHub or GitLab and a way to execute tests in the different Java projects that use Maven or Gradle.

It was decided to create a backend application that could be easily deployed. To achieve this, Java Spring Boot was used. Java Spring Boot is an open-source Java-based framework that allows the creation of simple backend and front-end applications. It offers a dependency injection feature that lets objects define their own dependencies. This allows the creation of modular applications. This backend would have to use a database to store the information of the projects and from the test executions. We choose PostgreSQL, an open-source object-relational database, due to its simplicity of working with and deploying.

In figure 5.3, it is possible to verify the architecture of the build tool. It was created a backend that uses HTTP requests that can be called by clients. As the purpose of this research work was to focus on the cost reduction techniques, it was not developed a front-end that would integrate with the tool. Instead, all the tests made to the tool were made by Postman, an API platform for building and using APIs that allow to simulate a client making HTTP requests to the backend.

The tool was developed using a layered architecture composed of four elements, the presentation layer, the business or service layer, the persistence layer and the database layer. Each layer responsibility is as follows:

- **Presentation Layer:** This layer is constituted by all controller classes. They have the

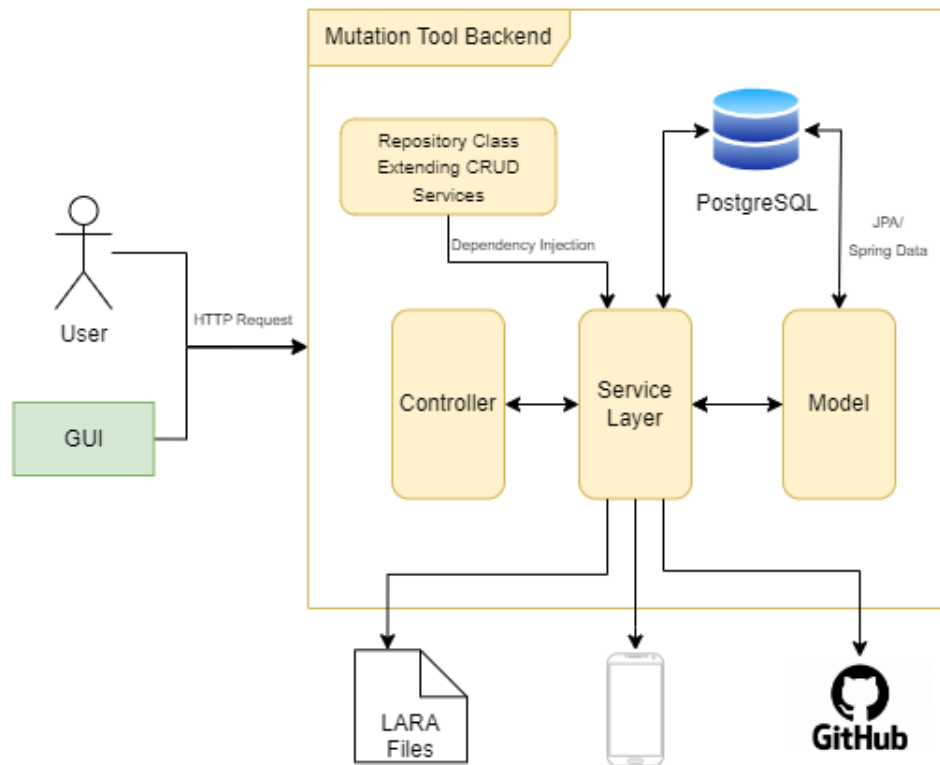


Figure 5.3: Tool Developed Architecture

responsibility of handling HTTP requests and the translation of the JSON parameters to objects.

- **Business Layer or Service Layer:** This layer is where all business logic is present. In our case, all the logic of generating the mutations, compiling the projects and executing the tests is present. Additionally, the integration with Kadabra and Git was made in this layer, as well as the communication with the Android emulators.
- **Persistence Layer:** This layer is constituted by all the models and repository classes. This layer contains all the storage logic and the capability of translating business objects from and to the database layer.
- **Database Layer:** This layer is the actual database in which all the create, read, update and delete (CRUD) operations are done.

This layered architecture brings isolation of responsibilities by dividing the application into different layers, each with a well-defined responsibility. Also, promotes modularity between components, makes easier the process of developing and modifying each layer.

5.2.1 Data Model

With the definition of the architecture of the application, the design of the data model took place. First, all the main entities that were part of the system were identified. Then the attributes of each one were defined. Finally, we established all the relationships and defined the constraints. The data model can be seen in figure 5.4

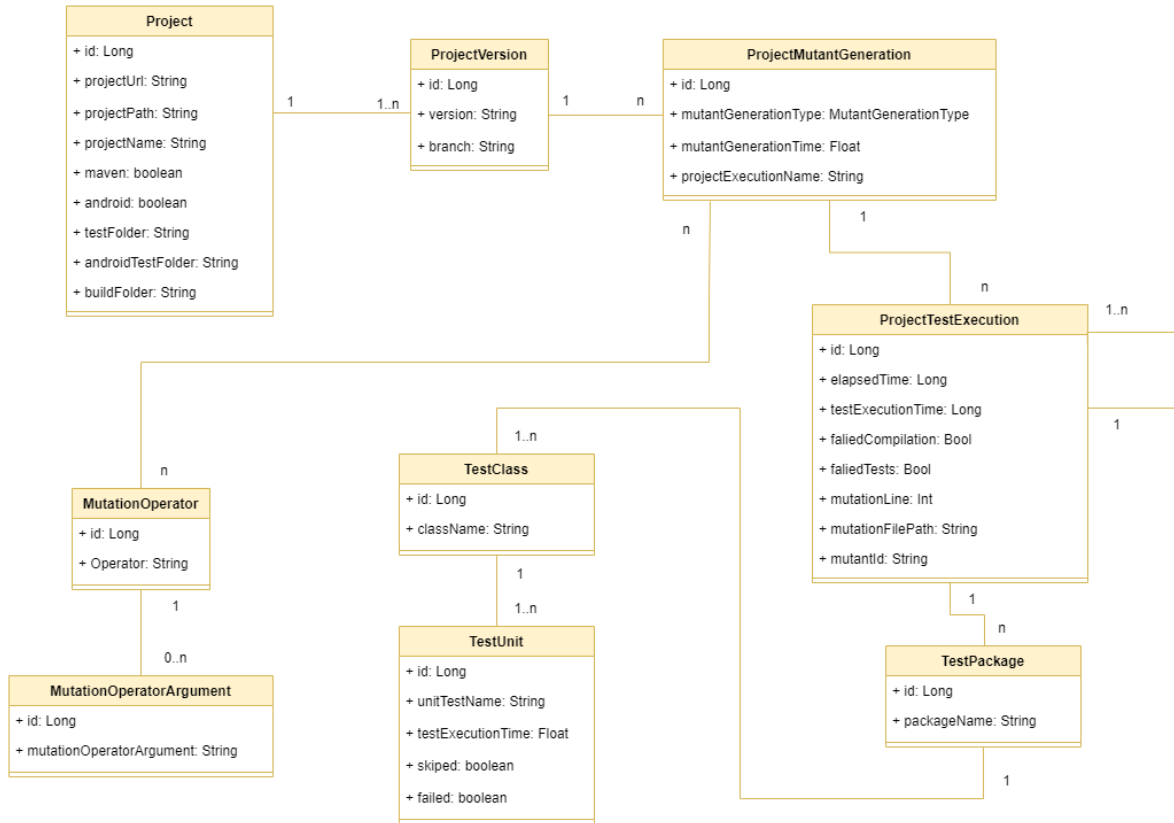


Figure 5.4: Data Model of the developed tool

Starting with the *Project* entity that can represent a Java project that uses Maven or Gradle build system, that can be a traditional java application or an Android application. It is composed of nine attributes, an identification (*id*), a web-hosting URL (*projectUrl*), a local path on the disk (*projectPath*), a project name (*projectName*), an identification of the type of build tool and type of project (traditional or Android), two test folders (*testFolder* that contains the relative path for the unit tests and *androidTestFolder* that contains the path for the Android instrumented tests) and a relative path of the build folder (*buildFolder*). A project can also have different versions (*ProjectVersions*). The *ProjectVersions* entity represents the different versions of an application that are present in the Git version system. It has three attributes, an identification (*id*), a version that represents each release of a project (*version*) and the branch of that version to be used (*branch*). The entity *ProjectVersions* can have multiple *ProjectMutantGeneration* entities. This entity represents the generation of a mutated version of a certain *ProjectVersion*. A *ProjectMutantGeneration* has four attributes, an identification (*id*), a mutation generation type (*mutationGenerationType*) that

represents the different strategies that are used to create the mutated project (e.g. traditional mutation or mutant schemata), a mutation generation time (that represents the total time of creating the copy of the project and applying the mutations) and a name that represents the mutant generation (*projectExecutionName*). A *ProjectMutantGeneration* entity is related to the *MutationOperator* (in a many-to-many relationship) and the *ProjectTestExecution* (in a one-to-many relationship).

The *MutationOperator* entity represents the mutation operators (e.g. Arithmetic Operator, Relational Operator or Assignment Operator). Its only attributes are an identification (*id*) and a name (*operator*). A *MutationOperator* is also related to the *MutationOperatorArgument* entity by a one-to-many relationship. *MutationOperatorArgument* also contains two attributes, an identification (*id*) and an argument (*mutationOperatorArgument*). This way, we could represent operators that receive arguments in order to perform the mutation (e.g. the Arithmetic operator needs two arguments, the one that is the original operator "+" and the replacing operator "-", representing the substitution of a sum of two elements for a subtraction).

The *ProjectTestExecution* entity represents a test execution from a *ProjectMutantGeneration*. It has eight attributes, an identification (*id*), the test execution time (*testExecutionTime*) and the total elapsed time (*elapsedTime*), two fields that represents if the result project compiles (*failedCompilation*) and if it has failed tests (*failedTests*), the original code line that was mutated (*mutationLine*), the project file that was mutated (*mutationFilePath*) and the mutation identifier (*mutantId*). The elapsed time represents all the time needed for test execution. It is the sum of the test execution time plus the time to build, compile and deploy the project if needed. A *ProjectTestExecution* is related to itself by a recursive one-to-many relationship. This decision was made as when mutations are applied, several test executions need to be made. This way, we can aggregate the test execution time and total elapsed time of each individual execution on a parent entity. A more detailed description will be given in section 5.6.

Finally, three more entities, the *TestPackage*, *TestClass* and *TestUnit*, combined, represent each unit test. The *TestPackage* contains only two attributes, an identification (*id*) and the package name of the test (*packageName*). It is related to the *TestClass* by a one-to-many relationship. The entity *TestClass* also has only two attributes, an identification (*id*) and the class name of the test (*className*). It is related to the *TestUnit* by a one-to-many relationship. The entity *TestUnit* contained five attributes, an identification (*id*), the unit test name (*unitTestName*), the particular test execution time (*testExecutionTime*) and two more attributes that represent if a test was skipped (*skipped*) or if it failed (*failed*).

5.3 Git Integration

The integration of Git into the application would be a complex task if started from scratch. This integration was a critical part of the methodology proposed in 4.2.2, and requires the collection of information like the different versions of an application and the changes made to each individual

file of each version. To simplify this integration, the decision to use Jgit was made. Jgit is an open-source library developed by the Eclipse Foundation that provides several functions that simplify the integration with Git. This integration simplified the creation of the following functions:

The code can be seen in our public repository ⁴

- **cloneRepo():** This function allows to clone a repository. It is used when a new project is added to the application.
- **getVersionHistory():** This function collects all the versions from a repository. It is called when there is a need to create or update the versions of a Project in our database.
- **updateCurrentVersion():** This function updates the local copy of the repository to a given version of the application.
- **getChangedFiles():** This function returns all the different files between two different versions. It is used when the user chooses to use Git to apply mutations to only changed files.

5.4 Mutant Generation

The generation of the mutations plays a crucial role in the mutation testing process. It requires the execution of each mutation operator at every relevant point within the AST. Depending on the project complexity, this can be time-consuming, especially in projects with large code bases. Recognising this, it was decided to optimise the generation process. We would only generate the AST for each Java file instead of generating the complete AST for the entire project. This way, we could parallelise the mutation execution process by executing multiple threads, each one with an independent Lara environment, that would apply the mutation to the AST of the corresponding file (figure 5.5). To achieve this, it was necessary to divide responsibilities. The application is responsible for receiving the user-provided information, processing it and forwarding it to the main LARA environment. Then depending on the information received, the main LARA environment instantiates a new LARA environment, in a new thread, for each Java file. Each instantiated environment then exclusively focuses on mutating its corresponding file.

The application receives its information from an HTTP request, which is then processed and passed into the main Lara environment. This information that goes into the main LARA environment is passed as arguments using the JSON format. In this information, it is included what type of mutation and what mutation operators are going to be applied, what project is to apply the mutations and what is its path in the file system, and what paths of the project should not be applied mutations (mutations should not be applied to test java files). In the main Lara environment, there is a collection of all the Java files that are going through the process of mutation with all of this information. A list is generated, with each position containing a Java file and the corresponding arguments necessary for instantiating the new LARA environment. This list is then passed into

⁴https://github.com/specs-feup/mutation-testing-v2/blob/main/Mutation_Testing_Backend/src/main/java/org/feup/Mutation_Testing_Backend_Final/Helper/Githelper.java

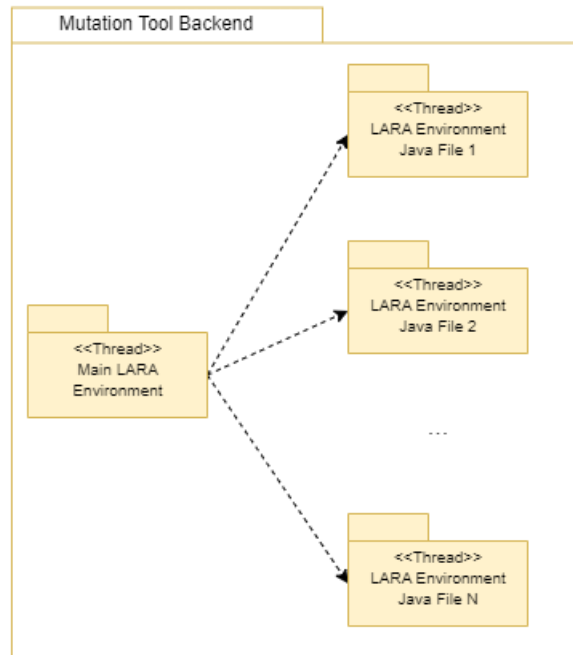


Figure 5.5: Parallel execution of the different LARA Environments

the function `runParallel()` from the Weaver API, which generates a thread pool, with each thread being a separate Lara environment that can access only the AST for its corresponding Java file. The code can be seen in appendix [A.1](#).

The application is also responsible for generating a unique identifier for the mutation execution. This unique identifier is always the combination of the project name and a Universally Unique Identifier (UUID). For example, for a project named *java-jwt*, the resulting unique identifier will be similar to *java-jwt817c563b_366a_4e80_b5ec_dd7ea62104a7*. This unique identifier is used to name the folder on the file system for the result of the mutation execution. This value is passed into the main LARA environment, and depending on whether it is being applied to mutant schemata or traditional mutation, it may or may not be responsible for generating code.

When using mutant schemata, the main LARA environment creates a copy of the entire project into a folder with the generated identifier. Then, if any mutation is applied, each executing LARA environment overrides its corresponding file (that will contain all the mutations using the schemata format). Considering the example before, at the end of the execution, the folder *java-jwt817c563b_366a_4e80_b5ec_dd7ea62104a7* has only the resulting project files, and each file has the corresponding mutations applied. The resulting structure can be seen in figure [5.6](#).

When using traditional mutation, the main LARA environment does not make any copy of the original project. Instead, that responsibility is moved into each launched environment. When a mutation point is detected, an entire copy of the project is made into a new folder inside the mutation execution folder. That new folder is always the combination of the mutant operation name plus the filename where the mutation was applied and a new UUID. Considering once again the above example and considering that the mutation operator binary was applied to the *Main.java* file result-

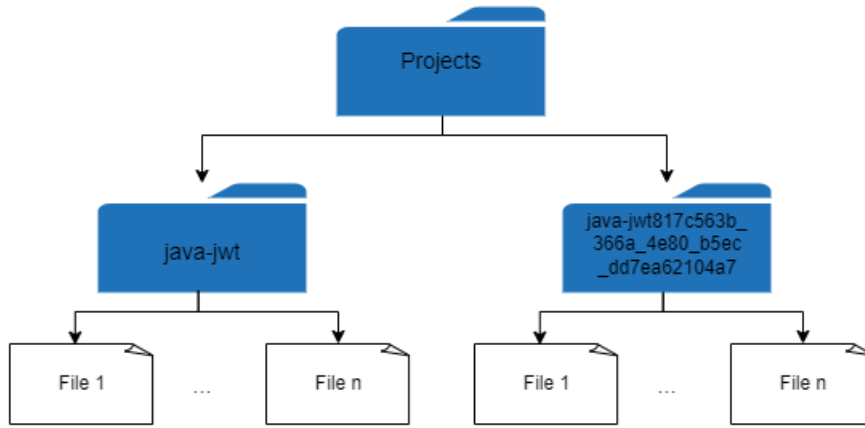


Figure 5.6: File Structure of Result Mutant Schemata Project

ing in only one mutation. Inside the *java-jwt817c563b_366a_4e80_b5ec_dd7ea62104a7* folder would be created a new folder *binaryMutator_Main_855c8349_4c65_4021_843a_5a4aa68cc8bb* where only one file would be different from the original project. That file would be, in this case, the *Main.java*. The resulting structure can be seen in figure 5.7.

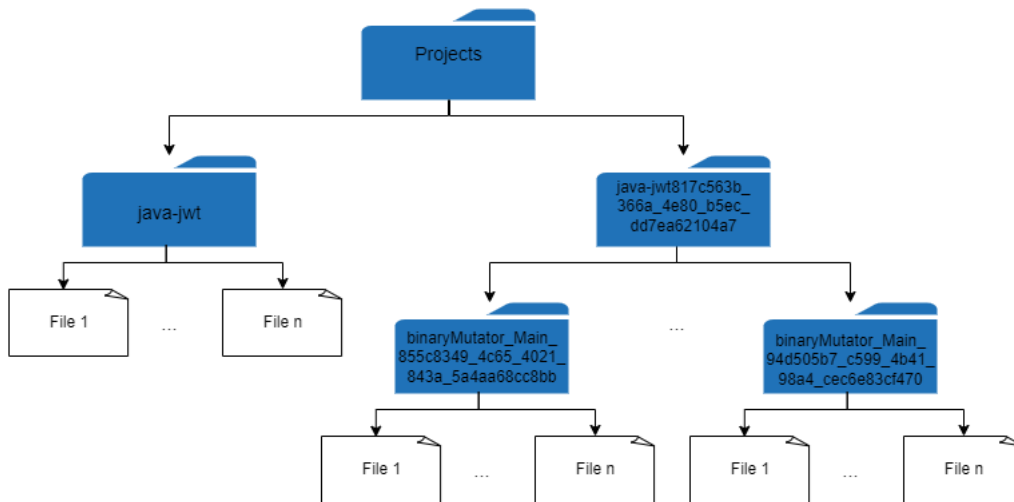


Figure 5.7: File Structure of Result Traditional Mutation Project

At the end of the execution, each thread has to return the mutations applied to the main LARA environment. This information is composed by the mutation unique identifier, the line where the mutation was applied, the mutation operator information and the complete file path. The main LARA environment then groups all the information and sends it back to the developed application, which stores it in the database. The information is also stored on a file called *MutationInfo.json*, on the root of the folder with the resulting mutated code. An example of this code can be seen in listing 5.1.

```

1  [
2    {
3      "mutantId": "/binaryMutator_Main_855c8349_4c65_4021_843a_5a4aa68cc8bb",
4      "mutation": {
5        "mutationOperatorArgumentsList": ["+", "-"],
6        "operator": "BinaryMutator",
7        "isAndroidSpecific": false
8      },
9      "mutationLine": 383,
10     "filePath": "lib/src/main/java/com/auth0/jwt/algorithms/Algorithm.java"
11   },
12   {
13     "mutantId": "/binaryMutator_Main_id_6a2c9a9f_4320_49b6_967a_f6c0c617f7aa",
14     "mutation": {
15       "mutationOperatorArgumentsList": ["+", "*"],
16       "operator": "BinaryMutator",
17       "isAndroidSpecific": false
18     },
19     "mutationLine": 383,
20     "filePath": "lib/src/main/java/com/auth0/jwt/algorithms/Algorithm.java"
21   }
22 ]

```

Listing 5.1: Example of the content on the *MutationInfo.json* file

5.4.1 Classpath Configuration

The division of each Java file of the project into different LARA environments posed a challenge. Since each LARA environment created operates independently, it became necessary to find a way to locate and load all the imports, dependencies and classes needed for creating the AST within each individual thread. Without this loading mechanism, the AST would not be created correctly, resulting in code that would not compile afterwards. Kadabra already supports setting the classpath, an environment variable used by the Java Virtual Machine (JVM) that specifies the location of all the necessary dependencies and classes. These dependencies are then loaded at runtime.

In Java projects that use Maven, the external dependencies can be obtained by executing the command `mvn dependency:copy-dependencies` that resolves all the dependencies. By adding the following flag, `-DoutputDirectory=/dependenciesDirectory`, the dependencies would be copied to the specified folder (in this example, it was chosen the folder `dependenciesDirectory`).

Gradle, contrary to Maven does not have any built-in function to obtain all the dependencies into a folder. In Java projects that use Gradle the solution was to create a task named `downloadDependencies` that would add this behaviour on the `build.gradle` file. This task uses the `configurations.compileClasspath` configuration, which includes all the libraries and dependencies required

to build and compile the project. With the help of the Gradle wrapper and by executing the command `./gradlew downloadDependencies` all the dependencies are copied to the specified folder in the form of JAR files. The defined task can be seen below [5.2](#).

```

1 task downloadDependencies(type: Copy) {
2     from configurations.compileClasspath
3     into '/dependenciesDirectory'
4 }

```

Listing 5.2: Java Gradle downloadDependencies Task

Despite the fact that the majority of Android Applications use Gradle, the process of getting all the dependencies has some differences. Instead of using the configuration *configurations.compileClasspath*, it was necessary to use the *configurations.implementation*. This configuration is used during the build process to resolve the required libraries, classes, and resources in the APK. The task is executed in the same way with the help of the Gradle wrapper. All the dependencies are copied to the specified folder in the form of JAR and AAR files. The defined task can be seen below [5.3](#).

```

1 android {
2     configurations {
3         resolvedImplementation.extendsFrom(implementation).canBeResolved = true
4     }
5
6     task downloadDependencies(type: Copy) {
7         from configurations.resolvedImplementation
8         into '/dependenciesDirectory'
9     }
10 }

```

Listing 5.3: Android Gradle downloadDependencies Task

As Kadabra does not currently support AAR files, they must be uncompressed. Finally, the classpath variable can be defined after having all the dependencies in one folder. All the paths to all necessary JAR and class files are combined using a semicolon. Even though this process can be automated, it has some failures, especially on Android applications, requiring resolving dependencies manually. As the primary focus of this research was not to find all the dependencies of all classes, the process was simplified as much as possible. It was decided that the user would be responsible for defining the classpath and passing it to the application when making the request to apply the mutations. Then the application passes this property into Kadabra.

5.4.2 Mutation Process

The mutation process followed by each individual LARA environment can vary based on the specific mutation type (traditional or schemata) and the target project (if is a traditional Java application that uses Gradle or Maven, or if is an Java Android application that uses Gradle). In the below sections, all the differences are explained.

5.4.2.1 Traditional Mutation

The traditional mutation process remains independent of the build system being used, whether it's Maven or Gradle, and is also unaffected by the use of Android-specific Java code. The process starts with the call to the method *runTreeAndGetMutantsTraditionally()*. This method uses the function *Query.root().descendants* from the *weaver.Query* API to iterate over all the joinpoints from the AST. Then, for each joinpoint and for all mutant operators that were passed by the main LARA environment, check if the mutation can be applied or not by calling the method *addJp()*. At the end of the iteration of the AST, each mutant operator will be storing the corresponding joinpoints that they can mutate. The next step is the execution of method *applyTraditionalMutation()*, which iterates over each mutant operator and checks if it has any mutation points by calling the method *hasMutations()*. If the mutant operator has any, the mutation is applied by calling the method *mutate()*, which applies the mutation to the AST. Then, the method *saveFile()* is called to create a new folder following the naming scheme stated above in figure 5.7. It then proceeds to duplicate the entire project substituting the original file with the mutated version. At the end of the copy, the method *restore()* is called to return the AST to its original form. The process is repeated for all operators. The code can be seen in appendix A.2.

5.4.2.2 Java Mutant Schemata

The process of creating mutant schemata is similar to whether Java or Android-specific code is being used and is unaffected by what build system is being used. Our process of implementing mutant schemata on Java applications uses the *if*, *else if* and *else* control flow statements to differentiate between mutants and the original project. These statements need a Boolean expression to determine which flow is going to be executed, expressions that differ between on whether Java or Android-specific code is being used and from the build system. The Java class *System.getProperty* was utilised to manage each execution of native Java projects that use Maven or Gradle build systems. This class allows the storage of information like local system properties and configurations. Then, we created a property called Mutation Unique Identifier or *MUID* to control the flow. This property is then passed during runtime and serves as a control mechanism for each execution.

The mutation process starts with the call to the method *runTreeAndApplyMetaMutant()*. This method also uses the function *Query.root().descendants* to iterate over all joinpoints from the AST. Opposite to the traditional process, when applying mutant schemata, the mutations are applied alongside the iteration. For each joinpoint, all mutant operators check if the mutation can be applied or not by calling the method *addJp()*, and the number of mutations to be applied is stored in a variable named *mutationPoints*. This variable is used to define how many control flow statements are going to be needed. The first flow statement is always an *if* statement, and the last is always an *else*. Additional *else if* statements can be added depending on the number of mutations. In a joinpoint where three mutations are applied, four control statements are going to be needed, one *if* statement for the first mutation, two *else if* for the remaining two mutations and an *else* statement for the original code (as evidenced in listing 5.4). The next step is the execution of the method

hasMutations() for each mutation operator. If the mutant operator has any, the mutation is applied by calling the method *mutate()*, and the corresponding control flow statement is inserted. For each mutation, the control statement is generated with a unique identifier composed of the mutant operation name plus the filename where the mutation was applied and a newly generated UUID. Then, the *restore()* method is called to guarantee that the else statement is always the original code. In the mutant execution folder generated by the main LARA environment, the mutated file overrides the original one at the end of the execution. The code can be seen in appendix A.3.

```

1  if (System.getProperty("MUID") != null && System.getProperty("MUID").equals("
    BinaryMutator_JsonNodeClaim_id_02632f92_c118_40c3_87d0_4f4b7de9a7a1")) {
2      // Executes Mutant 1
3  }else if (System.getProperty("MUID") != null && System.getProperty("MUID").equals("
    BinaryMutator_JsonNodeClaim_id_85c8e255_9340_4394_90b4_7dfb247ab6dc")) {
4      // Executes Mutant 2
5  }else if (System.getProperty("MUID") != null && System.getProperty("MUID").equals("
    BinaryMutator_JsonNodeClaim_id_ea4b4090_446d_4c4b_be12_c76d85ce7afb")) {
6      // Executes Mutant 3
7  }else{
8      // Executes Original Code
9  }
```

Listing 5.4: Example of the control flow of mutant schemata in a Java Application

This mutant schemata implementation offers the advantage of facilitating code inspection. As explained in detail in section 2.3.3.3, some implementations can make it challenging to identify where a mutation was applied and what specific mutation was made. However, in this case, the original code that was mutated will always reside within the else statement, while each mutation of that statement will be on the preceding *if* and *else if* statements. This clear separation aids in better understanding and analysis of the applied mutations.

5.4.2.3 Android Mutant Schemata

Two different ways of managing the execution were implemented in Android. The first is through the use of a *buildConfigField*. This property is defined on the *build.gradle* file of the Android project as evidenced in listing 5.5. When the Android application is built, a new MUID field is generated in the BuildConfig class with the value obtained from the *system.getProperty("MUID")*. The APK is then generated and deployed to the Android device. This method improves the compilation process, as only the BuildConfig class needs to be compiled between different mutants. However, it still requires the newly created APK to be deployed to the device. The code can be seen in appendix A.4. An example of the resulting expressions on the control flow can be seen in listing 5.6.

```

1 android {
2     defaultConfig {
3         buildConfigField 'String', 'MUID', '"' + System.getProperty("MUID") + '"'
4     }
5 }

```

Listing 5.5: build.gradle file with buildConfigField

```

1 if (BuildConfig.MUID != null && BuildConfig.MUID.equals("
    findViewByIdReturnsNullOperatorMutator_PreferencesActivity_id_64b0974")) {
2     // Executes Mutant 1
3 }else{
4     // Executes Original Code
5 }

```

Listing 5.6: Example of the control flow of mutant schemata in an Android Application using the buildConfigField

Due to the limitations in the implementation of mutant schemata using the *buildConfigField* in Android compared to traditional Java applications, a decision was made to introduce a second type of schemata for Android. The first challenge encountered was that recent Android versions do not support direct modification of system properties out of the box due to security concerns. To overcome this obstacle, our testing device was rooted, granting the necessary permissions to access and modify system properties. The next step was to define a Java function that could get a system property. That was done with the creation of a new process, which executes the command *getprop MUID*. This command obtains the specific value associated with the *MUID* property. The created function *getMUID()* can be seen in listing 5.7.

```

1 public class PreferencesActivity extends AegisActivity implements
    PreferenceFragmentCompat.OnPreferenceStartFragmentCallback {
2     //... remaining code
3
4     @Override
5     protected void onCreate(Bundle savedInstanceState) {
6         if (getMUID().equals("findViewByIdReturnsNullOperatorMutator_
    PreferencesActivity_id_3c1339dd_a398_4204_97d7_e2d4f5933680")) {
7             // Mutated Code
8             setSupportActionBar(null);
9         }else{
10             // Original Code
11             setSupportActionBar(findViewById(R.id.toolbar));
12         }
13     }
14
15     // New function getMUID()
16     public static String getMUID() {
17         String propertyValue = null;
18         try {

```

```

19         Process process = Runtime.getRuntime().exec("getprop MUID");
20         InputStream inputStream = process.getInputStream();
21         BufferedReader reader = new BufferedReader(new InputStreamReader(
    inputStream));
22         propertyValue = reader.readLine();
23         reader.close();
24         inputStream.close();
25     } catch (IOException e) {
26         Log.e("ERROR", String.valueOf(e));
27     }
28     return propertyValue;
29 }
30 }

```

Listing 5.7: Example of the control flow of mutant schemata in an Android Application using the *getMUID()* function

In order to incorporate this function into the control flow expression, it was required to create a copy for each mutated file. This was an additional step that was done before overriding the mutated file. While this implementation may not be considered optimal, it was chosen to ensure automation. A possible improvement could involve creating a singleton class that encapsulates the *getMUID()* function. This class could also serve as a cache, storing the retrieved value and only executing the process when needed for the first time.

5.5 Mutation Operators

Our goal was to implement mutation operators that would be agnostic to wherever they are being applied in the mutant schemata format or the traditional. We encountered the problem that mutation operators that did mutate variable declaration would create code that does not compile. Considering the example of a binary mutation applied to a variable declaration *int a = b + c*. Suppose the declaration of the variable *a* is inserted inside the control statement. In that case, the scope is only inside it, and if the variable is used on another part of the program, it will result in a failed compilation.

One possible solution to this problem is for the mutation operators to consider this case, resulting in the function *addJp()* ignoring joinpoints that included a variable declaration. This solution would reduce the number of mutations applied when using the schemata, something that was not wanted.

The solution was to transform the original code, separating the variable declaration from the variable initialisation. A method implemented by Ana Veiga [41] iterates over the AST and separates all the variable declarations from the variable initialisations. This method called *changeVarDeclarations()* is always called when applying mutant schemata.

```

1 // Original code
2 int a = b + c
3
4 // Mutant Schemata without the changeVarDeclarations()
5 if (System.getProperty("MUID") != null && System.getProperty("MUID").equals("
    BinaryMutator_JsonNodeClaim_id_02632f92_c118_40c3_87d0_4f4b7de9a7a1")) {
6     int a = b - c; // Mutated code
7 }else{
8     int a = b + c; // Original code
9 }
10
11 // Mutant Schemata with changeVarDeclarations()
12 int a;
13 if (System.getProperty("MUID") != null && System.getProperty("MUID").equals("
    BinaryMutator_JsonNodeClaim_id_02632f92_c118_40c3_87d0_4f4b7de9a7a1")) {
14     a = b - c; // Mutated code
15 }else{
16     a = b + c; // Original code
17 }

```

Listing 5.8: Differences between the resulting code with and without the *changeVarDeclarations()* method call

Ana Veiga also implemented 14 general-specific operators, 14 Java-specific operators and 25 Android-specific operators. A detailed specification of each mutation operator is available on [41]. All mutation operators follow the same structure of the Mutator API, with the addition of 2 methods. The method *isAndroidSpecific()* defines whether the mutation operators are Android-specific or not. The method *toJson()* is used to obtain information on the mutation operation being applied. It returns information like mutator name, mutator arguments and if the mutator is Android-specific or not.

At this point, implementing the mutation operator Not Serialisable only works in traditional mutation. This operator selects a serialisable class and removes the "implements Serializable". This mutation can only be done using traditional mutation, as our implementation of mutant schemata does not generate syntax-correct code.

5.6 Test Execution

The test execution is an essential step in mutation testing. When done manually, it is a tedious process, especially in projects that generate many mutants. Currently, our tool only supports the automatic execution of all tests sequentially. This process followed can have some differences depending on the type of project that is being used.

In Java Maven projects, a process executes the command *mvn surefire-report:report* on the local folder of the mutated project. When mutant schemata is used, it is necessary to add the flag *-DMUID=BinaryMutator_JsonNodeClaim_id_02632f92_c118_40c3_87d0_4f4b7de9a7a1* to set

Table 5.1: List of Operators - General Specific [41]

Category	Operators
General Specific	Arithmetic Operator Bitwise Operator Comparison Operator Logical Operator Assignment Operator Arithmetic Deletion Operator Bitwise Deletion Operator Comparison Deletion Operator Logical Deletion Operator Assignment Deletion Operator Constant Operator Unary Operators Unary Logical Negation Operator Unary Deletion Operators

Table 5.2: List of Operators - Java Specific [41]

Category	Operators
Java Specific	Constructor Call Remove Conditional Non Void Call Nullify Input Variable Nullify Return Value Return Value Operator Invalid Date Invalid Method Call Argument Null Method Call Argument Not Serializable Fail On Null String Argument Replacement String Call Replacement Conditional Expression Replacement

the system property *MUID* with the corresponding mutant to execute. This is automatically done by the tool, with the information that is already on the database from the generation process. The surefire report is a Maven plugin that generates a report of the test execution in the project's build folder. That report is a combination of XML files with all information about the test execution (e.g. failed tests, time executing each test, skipped tests). The elapsed time starts counting before the execution of the command and only stops after the execution finishes. Then, the *SurefireReport-Parser*, a XML parser for surefire-reports is used to parse the resulted report files to Java objects. Finally, the test execution time and the mutation score are calculated, and each test's information is persisted in the database.

In Java Gradle projects, the process is similar. The process executes the command *./gradlew*

Table 5.3: List of Operators - Android Specific [41]

Category	Operators
Android Specific	Buggy GUI Listener Lengthy GUI Listener Lengthy GUI Creation Find View By Id Returns Null View Component Not Visible Invalid View Focus Invalid ID FindView Null Intent Random Action Intent Definition Intent Target Replacement Invalid Key Intent Null Value Intent PutExtra Intent Payload Replacement XML Edit TextWidget Invisible XML ViewGroup Widget Invisible XML Button Widget Invisible XML EditText Widget Deletion XML Button Widget Deletion XML TextView Widget Deletion XML Invalid Color XML Button Widget Change Appearance XML EditTextWidget Change Appearance XML ViewGroup Widget Change Type Null Bluetooth Adapter Null GPS Location

test on the local folder of the mutated project. When using mutant schemata, the flag *-DMUID* also needs to be set with the unique identifier of the mutant to be executed. This result of executing the command is the execution of the tests, with the creation of XML files in the build folder with the results of the tests. Then, with the help of the package *javax.xml* all the files are parsed into Java objects and stored in the database.

The process is more complex on Java Android projects compared to traditional Java projects due to the two different test suits present. One test suit comprises instrumented tests that verify the correct behaviour of the UI, and the other comprises the unit tests. Mutations that only change Android-specific code are only tested with the instrumented tests, and the mutations that change Java-specific code are only tested with the unit tests. In traditional mutation, only the instrumented tests are executed when an Android-specific operator is used. These tests are initiated by a new process that executes the command *./gradlew connectedAndroidTest* on the folder of the mutated project. When a Java-specific operation is used with traditional mutation, only the unit tests are executed. To do so, the command *./gradlew test* is executed on the folder of the mutated project. On mutant schemata projects that use the *buildConfigField*, the instrumented tests are executed with the command *./gradlew connectedAndroidTest* and the unit tests are executed

with the command `./gradlew test`, both with the addition of the flag `-DMUID`. On mutant schemata using the function `getMUID()`, the instrumented tests are executed with the command `./gradlew connectedAndroidTest`. Between each execution is created a process that sets the Android device system property `MUID`. This property is set with the execution of the command `adb shell set-prop MUID $value` with `$value` being the mutation unique identifier. Finally, with the help of the package `javax.xml` all the files from the execution are parsed into Java objects and stored in the database.

5.7 Application Interaction

In the initial stage of this research work, we intended the implementation of a GUI for the users to interact with the application. This was not the main focus of this research work, and due to time constraints, the application in its current state can only be interactive through HTTP requests. These requests can then be called by a front-end application or by an application like Postman, a platform for building and using APIs.

It was used Swagger, an open-source framework that provides a set of specifications for building and documenting REST APIs. It was used the OpenAPI Specification to describe the API. The API is composed of multiple requests that allow to add, get, update and delete projects, create, get and delete mutated projects and finally, create, get and delete test executions from projects. All the requests can be seen in figure 5.8.

project-controller		^
PUT	/project/updateProject/{projectId}	Update a Project
POST	/project/addNewProject	Add a new project
GET	/project/getProjectVersions/{projectId}	Get Project Versions
GET	/project/getMetricsBetweenVersions/{projectId}	Get Changes Between a Project Versions
GET	/project/getAllProjects	Get all projects
DELETE	/project/removeProject/{projectId}	Remove a Project
test-controller		^
POST	/test/{projectMutantGenerationId}/executeAllTests	Execute All Tests For a Project
GET	/test/getAllTestResults/{projectMutantId}	Get All Test Execution for a Mutant Project
DELETE	/test/{testExecutionId}/	Deletes a Test Execution For A Project
mutation-controller		^
POST	/mutate/{projectVersionId}	Create Mutant Project
GET	/mutate/getAllMutationProjects/{projectVersionId}	Get All Mutated Projects From a Project
DELETE	/mutate/delete/{mutantId}	Delete a Mutated Project

Figure 5.8: Tool API Description

Chapter 6

Empirical Evaluation

To ensure a thorough and accurate comparison, we carried out several experiments, each isolating the maximum number of variables possible. We believe this was the only way to measure the performance benefit of our tool. Each experience was done using a traditional Java project (that uses Gradle as its build tool) and an Android application. The criteria for selecting the projects were the following:

- The project would need to have any development in the last six months and more than 40 versions publicly available.
- The project would need to have a minimum of 50 tests.
- The project would need to have at least 50 Java files.
- The project can not have any tests that fail.

As many applications fit the criteria, two applications were chosen randomly from the list of possible applications that fit the criteria. The Java project used was *java-jwt*, which is currently available on the GitHub¹. The project contains 76 Java files totalising 10303 lines of code. The total number of files is 106, and the original project size is 2696743 bytes (2.7 megabytes). The number of tests is 676.

The Android project used was *Aegis*, which is currently available on the GitHub². The project contains 218 Java files totalising 19918 lines of code. The total number of files is 474, and the original project size is 207352086 bytes (207.35 megabytes). The total number of Java-specific tests is 56, and the total number of Android-specific tests is 23.

Each experiment was divided into two phases. The first phase was the generation of the mutants. The measures taken were the total generation time (in seconds) and the total size (in bytes). The second phase was the build and testing phase. Here two times were taken, the testing time

¹<https://github.com/auth0/java-jwt>

²<https://github.com/beemdevelopment/Aegis>

(time executing all tests in seconds) and the total elapsed time (total time in seconds of the test execution time plus the time to build, compile and deploy the project if needed plus the time of setting the system variable MUID if needed).

The experiments were conducted on a single machine with the following specifications: a Ryzen 5 2600X CPU, 32GB of memory, and Crucial P1 1TB SSD. The operating system was Ubuntu 22.04.2 LTS. To ensure a correct collection of data, all the background programs were disabled during the execution, with only the application, the database and the emulator running. The background programs were disabled by executing the command *ps* to list all processes, and *kill [id]* to kill each one.

6.1 Impact of Mutation Operators

To test the impact of different mutation operators, it was decided to isolate each one, and compare the times of generation of the mutations, the test execution time and the elapsed time. The goal was to assess how each mutation operator affected the performance.

6.1.1 Impact of Mutation Operators On Java Projects

It was chosen five random Java specific operators from the ones that were implemented, the ConstructorCallOperatorMutator, the FailOnNullMutator, the RemoveConditionalMutator, the ReturnValueMutator and the BinaryMutator. Each operator was generated four times, with two instances using the traditional approach and two using the mutant schemata approach. The average values were calculated for each approach, and the results are present in the table 6.1.

Table 6.1: Generation Results for Java Operators

Mutation Operator	Average Generation Time (s)	Average Generation Size (bytes)	Average Number of Mutants	Type of Generation
ConstructorCallOperatorMutator	2,038	2.591.517	27	Schemata
FailOnNullMutator	2,061	2.568.916	4	Schemata
RemoveConditionalMutator	2,185	2.633.216	59	Schemata
ReturnValueMutator	1,881	2.587.720	25	Schemata
BinaryMutator	2,022	2.589.327	27	Schemata
ConstructorCallOperatorMutator	6,892	65.049.738	27	Traditional
FailOnNullMutator	2,161	9.771.061	4	Traditional
RemoveConditionalMutator	14,561	245.249.025	59	Traditional
ReturnValueMutator	5,146	60.230.718	25	Traditional
BinaryMutator	8,041	65.039.141	27	Traditional

By analysing the generation time results, it becomes evident that both the average generation time (figure 6.1) and the average generation size (figure 6.2) in the traditional approach are directly proportional to the number of mutants created. With the mutant schemata, all generations did have similar times. Furthermore, it was also observed that the mutation operator had no significant

impact on the generation time or the total size. This was evident as both approaches, using the ConstructorCallOperatorMutator, ReturnValueMutator and BinaryMutator operators, each with 27, 25 and 27 mutations, respectively, exhibit similar generation times and generation sizes.

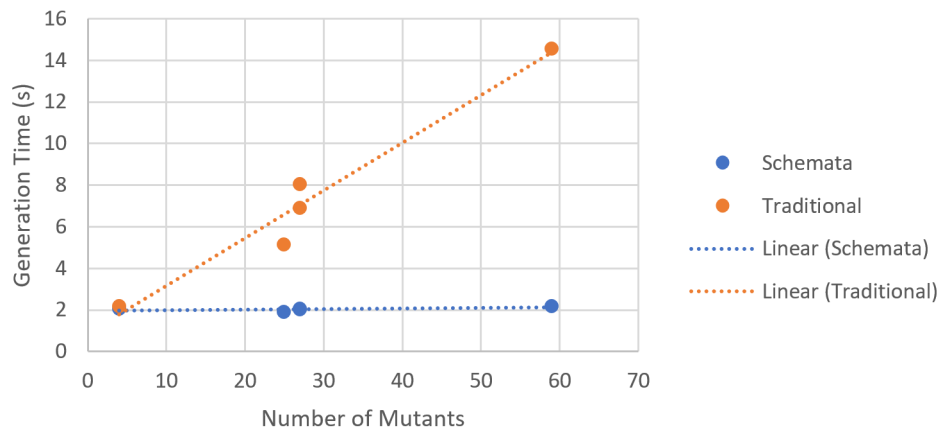


Figure 6.1: Generation Time For Each Java Mutation Operator

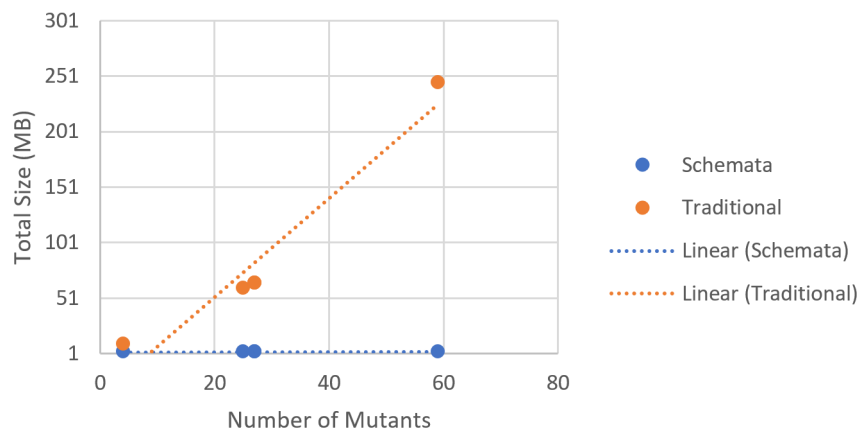


Figure 6.2: Total Size For Each Java Mutation Operator

At this point, the benefit of adopting the mutant schemata approach became evident. The generation time was much higher in traditional due to the number of copies needed. Additionally, it was evident that the speed of the disk played a significant role in the generation process of the traditional approach. As the traditional approach, for n mutants n copies, are generated, the disk speed is the bottleneck. The traditional mutation process would have been further prolonged if a slower-speed disk had been used.

The next step of our experience was to execute the tests for each mutation operator. Each test execution was done twice and the averages were calculated. The results are present in the table 6.2.

It is possible to verify that even with a limited number of mutations, the execution of tests took longer when utilizing the mutant schemata (figure 6.3). This can be attributed to the increased

Table 6.2: Test Execution Results

Mutation Operator	Average Test Execution Time (s)	Average Elapsed Time (s)	Generation Type
BinaryMutator	6,943	9,806	Schemata
ConstructorCallOperatorMutator	7,041	9,931	Schemata
FailOnNullMutator	7,541	10,690	Schemata
RemoveConditionalMutator	7,257	10,136	Schemata
ReturnValueMutator	7,180	10,053	Schemata
BinaryMutator	6,573	10,859	Traditional
ConstructorCallOperatorMutator	6,951	11,320	Traditional
FailOnNullMutator	7,469	11,788	Traditional
RemoveConditionalMutator	7,018	10,812	Traditional
ReturnValueMutator	7,151	11,536	Traditional

complexity introduced by each control flow statement. However, the mutant schemata implementation successfully reduced the overall elapsed time, as it eliminated the need for continuous compilations. It was evident that the execution time of the test suite varied depending on the mutation operator used. The FailOnNullMutator consistently resulted in longer test suite execution times, while the BinaryMutator proved to be one of the fastest. This difference is attributed to the impact of the applied mutation on the code behaviour.

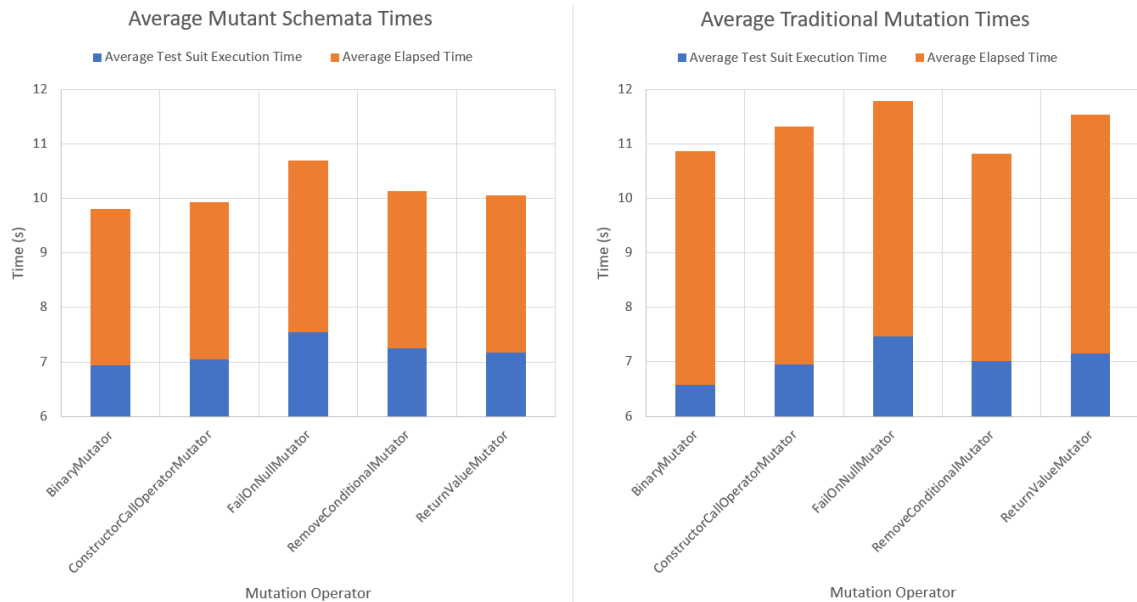


Figure 6.3: Average Time for each mutant execution using Traditional Mutation and Mutant Schemata

It was also analysed the total test suit execution time and the total elapsed time (figure 6.4). In all cases, an improvement of the total elapsed time was made when applying the mutant schemata over the traditional mutation. The improvements vary from 6% on the RemoveConditionalMutator

to 12% on the ReturnValueMutator.

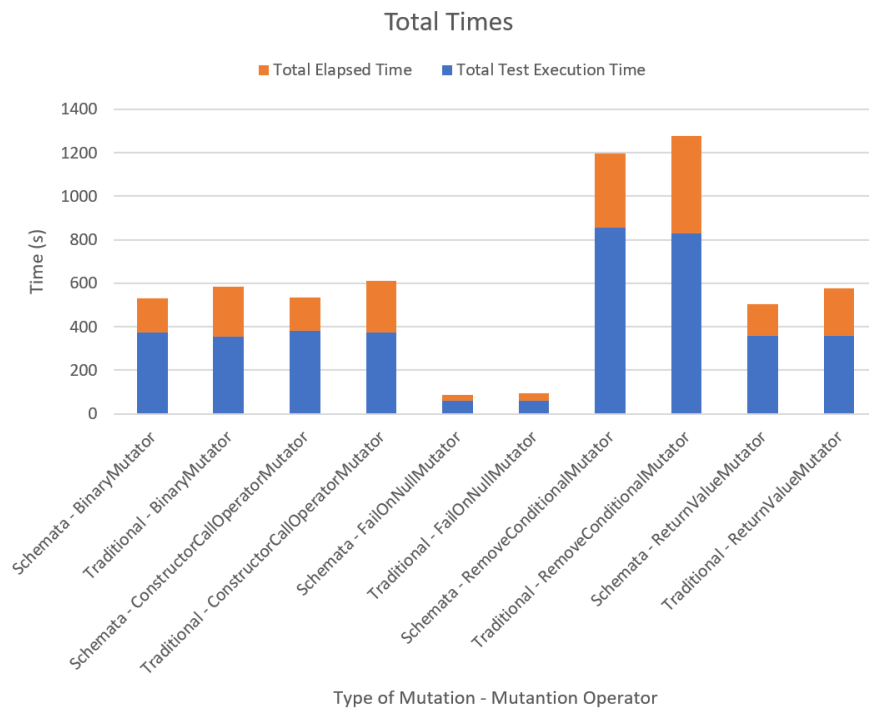


Figure 6.4: Execution Times for Java Traditional Mutation and Mutant Schemata

6.1.2 Impact of Mutation Operators On Android Projects

The same experience was also conducted for the Android-Specific mutations. It was chosen five random operators from the ones that were implemented, the BuggyGUIListenerOperatorMutator, the LengthyGUICreationOperatorMutator, the NullIntentOperatorMutator and the RandomActionIntentDefinitionOperatorMutator. Each operator was generated six times, with two instances using the traditional approach, two instances using the mutant schemata with the *buildConfigField* variable and two using the mutant schemata using the method *getMUID()*. The average values were calculated for each approach, and the results are present in the table 6.3.

Once again, by analysing the generation time results, it becomes evident that both the average generation time (figure 6.5) and the average generation size (figure 6.6) in the traditional approach are directly proportional to the number of mutants created. Furthermore, it was also observed that the different types of mutant schemata did not significantly impact the generation time or the generation size.

The next step of our experience was to execute the tests for each mutation operator. The findings revealed that the mutation operators did not significantly affect the generation time or size of the generated mutants.

Each test execution was done twice and the averages were calculated. The results are present in the table 6.4.

Table 6.3: Generation Results for Android Specific Operators

Mutation Operator	Average Generation Time (s)	Average Generation Size (bytes)	Average Number of Mutants	Type of Generation
BuggyGUIListenerMutator	172,492	829.415.046	4	Traditional
LengthyGUICreationMutator	132,833	414.708.217	2	Traditional
NullIntentMutator	147,549	829.415.810	4	Traditional
RandomActionIntentDefinitionMutator	232,636	2.280.893.958	11	Traditional
Mutator	128,646	207.431.292	4	<i>getMUID()</i>
LengthyGUICreationMutator	132,810	207.429.919	2	<i>getMUID()</i>
NullIntentMutator	131,875	207.430.645	4	<i>getMUID()</i>
RandomActionIntentDefinitionMutator	136,322	207.435.163	11	<i>getMUID()</i>
BuggyGUIListenerMutator	131,044	207.364.875	4	<i>buildConfigField</i>
LengthyGUICreationMutator	126,533	207.363.432	2	<i>buildConfigField</i>
NullIntentMutator	135,227	207.364.228	4	<i>buildConfigField</i>
RandomActionIntentDefinitionMutator	139,461	207.368.991	11	<i>buildConfigField</i>

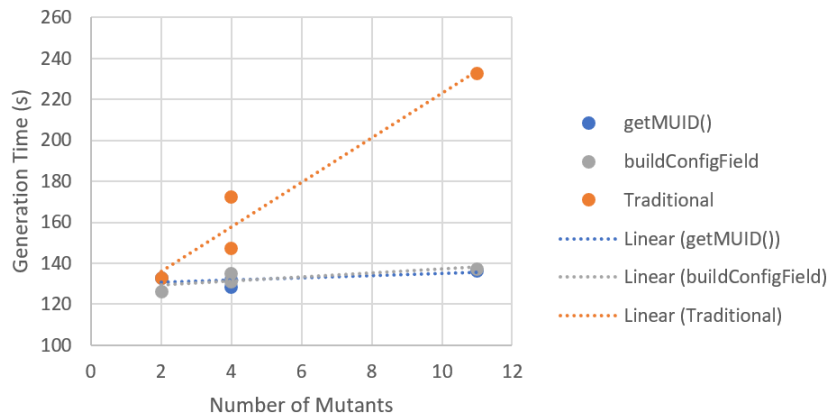


Figure 6.5: Generation Time For each Android Mutation Operator

It was possible to verify that the mutant schemata implementation with the *getMUID()* method was better than using the traditional mutation or schemata with the *buildConfigField* (figure 6.7). It is also possible to verify that *buildConfigField* schemata is better than the traditional. The actual test execution time is similar across all three generation types. it was observed that the advantage of using mutant schemata was only during the deployment of the test execution. However, in all cases, it was the test execution that did take most of the time.

6.1.3 Summary

This experiment provided insights to address RQ4. It was possible to verify that the mutation operators do not impact the generation time or size but do impact the time executing the tests and subsequently impact the elapsed time.

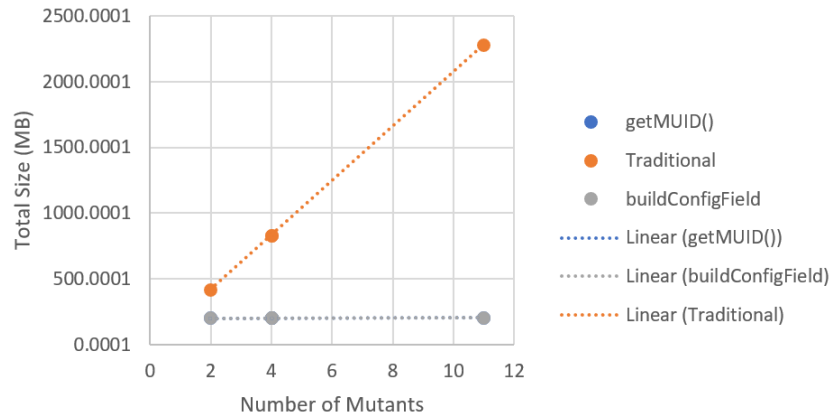


Figure 6.6: Total Size For Each Android Mutation Operator

Table 6.4: Test Execution Results Android

Mutation Operator	Test Execution Time (s)	Elapsed Time (s)	Generation Type
BuggyGUIListenerMutator	650,578	836,995	<i>getMUID()</i>
LengthyGUICreationMutator	343,623	434,141	<i>getMUID()</i>
NullIntentMutator	643,442	809,210	<i>getMUID()</i>
RandomActionIntentDefinitionMutator	1.785,988	2196,501	<i>getMUID()</i>
BuggyGUIListenerMutator	649,545	856,374	<i>buildConfigField</i>
LengthyGUICreationMutator	446,573	540,191	<i>buildConfigField</i>
NullIntent	643,204	827,784	<i>buildConfigField</i>
RandomActionIntentDefinitionMutator	1.781,463	2272,726	<i>buildConfigField</i>
BuggyGUIListenerMutator	649,372	864,317	Traditional
LengthyGUICreationMutator	446,211	548,790	Traditional
NullIntentMutator	645,727	853,733	Traditional
RandomActionIntentDefinitionMutator	1.775,061	2323,165	Traditional

By analysing the experiment results, it became evident that some mutation operators introduced code variations that required additional time to execute the tests. The execution time of the tests varies due to the changes induced by the mutation operator.

As a result, the overall elapsed time was also influenced by the choice of mutation operators. It was observed that certain mutation operators led to longer execution times for the tests compared to others.

6.2 Impact of Mutant Schemata

A thorough examination was conducted to assess the benefits of mutant schemata to analyse how the number of mutations applied would affect the performance. Different mutations were applied,

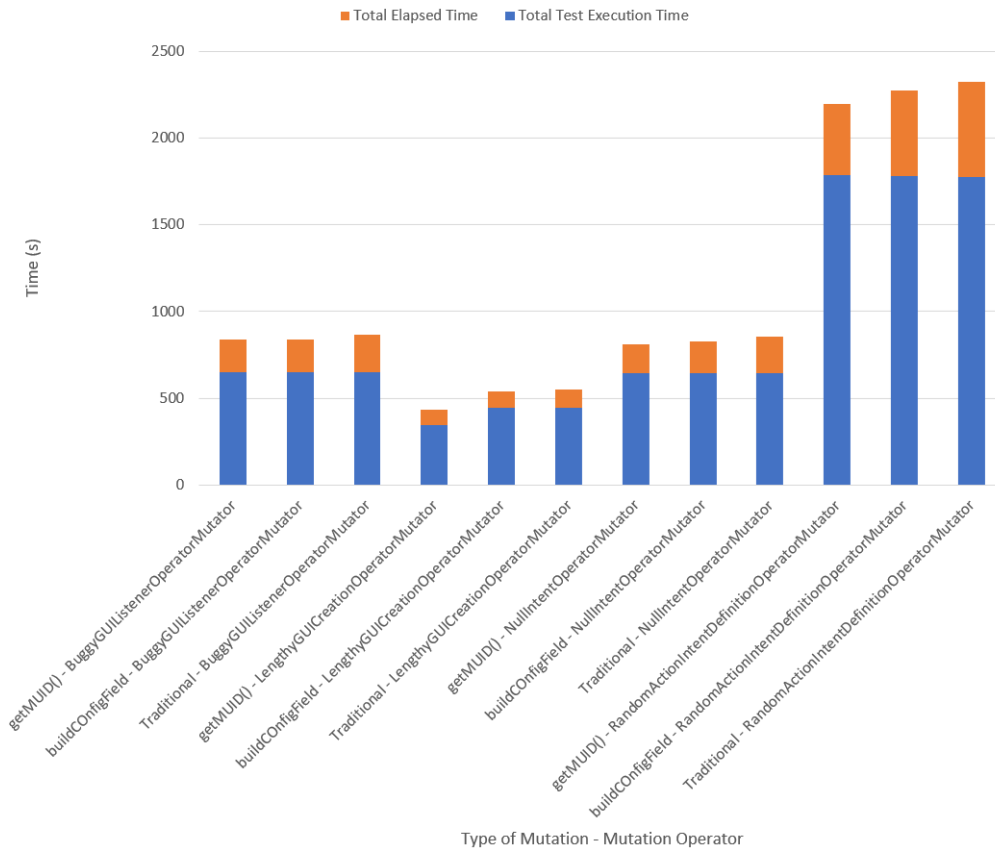


Figure 6.7: Execution Times for Android Traditional Mutation and Mutant Schemata

but only the original code was tested. This approach was chosen to assess the performance degradation caused by each control flow statement of the mutant schemata. In the initial experiment, it became evident that the mutation operators impacted the test execution times. Executing the mutated code could result in different program flows compared to the original version, limiting the ability to accurately evaluate the actual performance of the mutant schemata with the increasing number of mutations. By simply testing the original code, it was possible to isolate the influence of the mutation operators, providing a better assessment of the performance benefits of the mutant schemata.

6.2.1 Impact of Mutant Schemata on a Java Application

It was chosen random operators to generate four mutant schemata projects with different number of mutations. The tests were executed three time for each mutant schemata and the average values were calculated for each one. The results are present in the table 6.5.

Through the analysis, it was possible to verify that the execution time of the tests increased proportionally with the number of mutations applied (figure 6.8). This is an expected behavior, as increasing the number of mutations introduces more control flow statements, resulting in a slower test execution. In the particular case of the mutant schemata with 498 mutations, the tests executed 6.6% slower compared to with the original project.

Table 6.5: Schemata Performance Degradation in Java

Number of Mutations	Total Test Execution Time (s)	Total Elapsed Time (s)
498	7.473	10.563
291	7.263	10.239
237	7.192	10.152
136	7.153	10.147
0	7.010	10.081

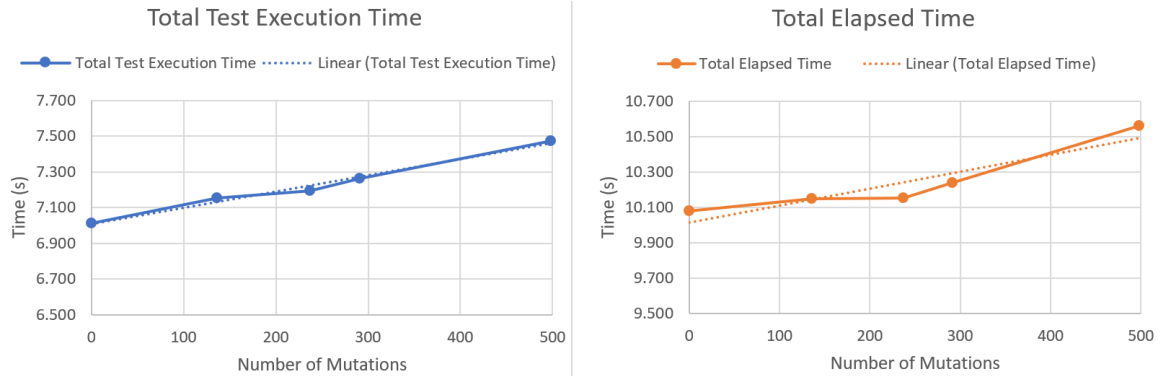


Figure 6.8: Total Execution and Elapsed Time for the Different Mutant Schemata Projects

6.2.2 Impact of Mutant Schemata on an Android Application

It was chosen random operators to generate six mutant schemata projects, three using the *getMUID()* and three using the *buildConfigField* with different number of mutations. Both the unit tests and the instrumental tests were executed three time for each mutant schemata and the average values were calculated. The results are present in the table 6.6.

Table 6.6: Schemata Performance Degradation in Android

Number of Mutations	Generation Type	Unit Test Execution Time (s)	Android Test Execution Time (s)	Total Elapsed Time (s)
431	<i>buildConfigField</i>	200.130	201.04	463.258
238	<i>buildConfigField</i>	16.218	173.080	238.058
130	<i>buildConfigField</i>	15.315	167.420	227.087
431	<i>getMUID()</i>	199.660	200.910	455.468
238	<i>getMUID()</i>	15.550	162.400	231.245
130	<i>getMUID()</i>	14.778	162.470	221.488
0	Original	14.318	153.320	210.182

Contrary to the linear increase of time verified on the Java application 6.2.1, the time to execute unit tests on Android increased exponentially in both versions of mutant schemata (figure 6.9) only on the unit tests. This was not expected and required a deeper analysis. The first step was to detect where the mutations were being applied. To our surprise, the total number of mutations on

the file *Salsa20Engine.java* was 196. Inside the file is present a method *salsaCore()* that implements the algorithm *Salsa20*. All of the mutations were inside of the *for* loop. This resulted in a significant reduction of performance on the algorithm and, subsequently, a considerable increase in time to execute the unit tests. The only unit tests affected were the *testTrailingNullCollision* and *vectorsMatch* that executed the method *salsaCore()*, taking a total of 187 seconds to execute.

Additionally, it was possible to verify that the test execution of Android Instrumented tests also increased with the number of mutants present in the schemata.

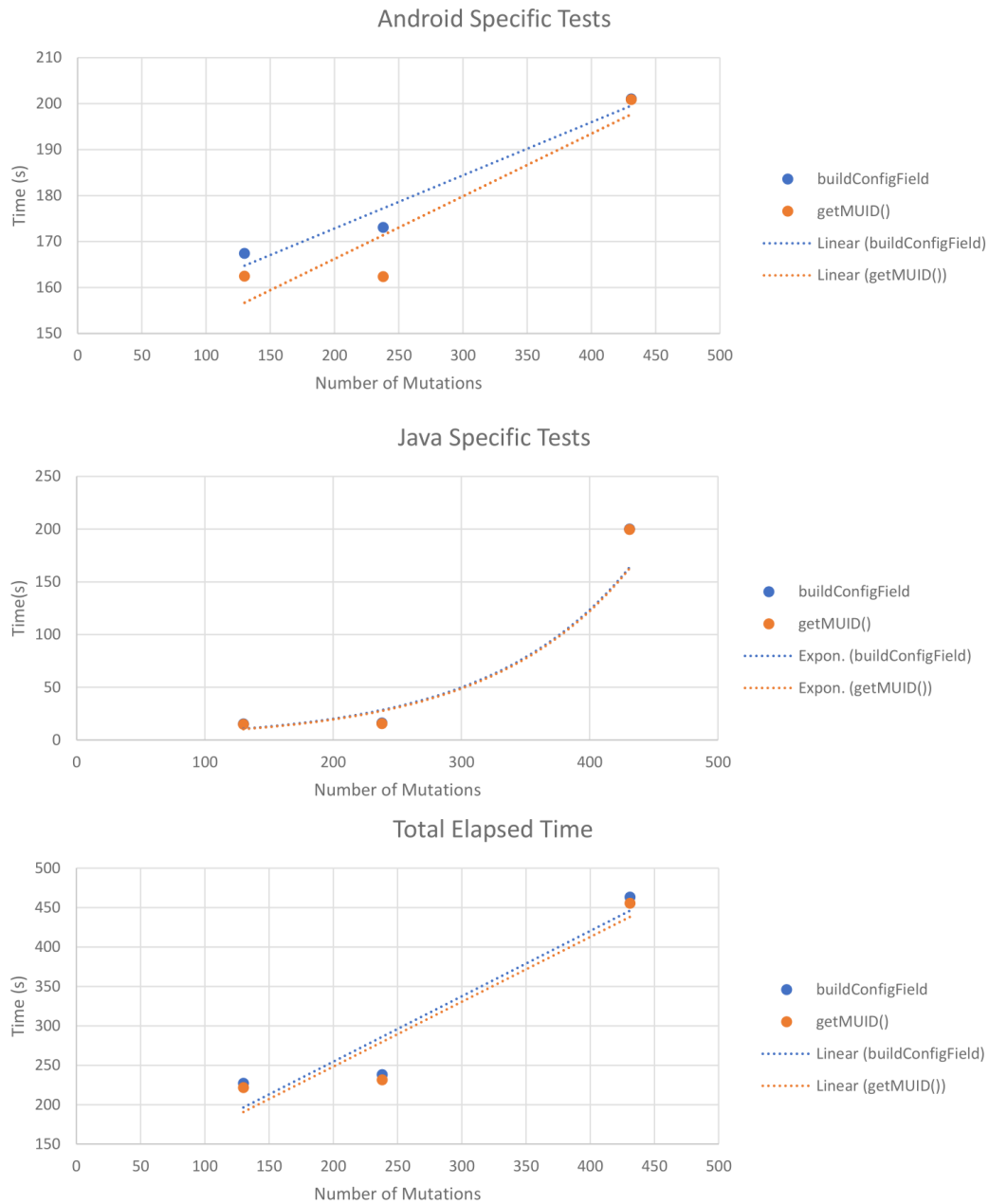


Figure 6.9: Total Test Execution Times and Elapsed Times for the Different Mutant Schemata Projects in Android

6.2.3 Summary

This experiment provided insights to address RQ5. These insights provide empirical evidence of the correlation between the number of mutations and the performance of tests, indicating that an increased number of mutations within the same project leads to slower test execution time.

Consequently, a threshold exists beyond which mutant schemata is no longer the optimal choice. This threshold is reached when the performance decrease caused by the high number of mutations exceeds the time required to compile and deploy the test execution.

The point at which this threshold is crossed depends on various factors, including the project's complexity, the number of tests, the time taken to execute the tests, and the compilation time.

6.3 Impact of Git Versioning Reduction

We conducted an experiment to assess the advantages of applying mutations exclusively to the changed files between Git versions. This experiment was conducted on the Aegis Android application using six distinct releases. For each version, we collected the total number of files, the total number of Java files, the total number of changed files compared to the previous versions, and the total number of changed Java files compared to the previous versions. This information can be seen in table 6.7.

Table 6.7: File Metrics From Different Versions of Aegis

Version	Total Number of Files	Total Number of Java Files	Total Number of Changed Files	Total Number of Changed Java Files
v2.0.2	464	178	46	10
v2.0.3	466	178	7	2
v2.1	530	211	237	126
v2.1.1	535	215	107	50
v2.1.2	535	215	15	1
v2.1.3	538	216	51	5

By examining the collected metrics, we can observe the evolution of the Android project across different versions. The variations in the number of changed files indicate the degree of modification and potential focus areas for mutation testing. We verified that only version v2.1 of Aegis introduced significant changes in files, with more than 50% of Java files changing. Then, we proceeded to apply the mutations to both the entire project and to only the changed files. In both cases, it was used the mutant schemata with the *getMUID()* implementation. The mutations were applied with 3 different operators, the *RandomActionIntentDefinitionMutator*, *BuggyGUIListenerMutator* and the *BinaryMutator*. The results of applying the Git improvement are present in table 6.8 and the results of applying to the entire project are present in table 6.9.

One notable observation is the consistent number of mutants, which remains at 442 throughout the traditional approach of applying mutations to the entire project. This occurrence can be

Table 6.8: Generation Metrics of Mutant Schemata with Git Improvement

Version	Total Number of Mutations	Total Generation Time (s)
v2.0.2	3	26,212
v2.0.3	0	20,478
v2.1	336	50,911
v2.1.1	2	35,310
v2.1.2	0	20,940
v2.1.3	0	22,699

Table 6.9: Generation Metrics of Mutant Schemata Without Git Improvement

Version	Total Number of Mutations	Total Generation Time (s)
v2.0.2	442	133,382
v2.0.3	442	118,919
v2.1	442	130,186
v2.1.1	442	118,179
v2.1.2	442	120,155
v2.1.3	442	119,739

attributed to the mutation operators employed, as they did not generate any additional mutations in the modified code between versions. As a result, almost all of the 442 mutants are equal in the different versions of Aegis. Our approach aims to reduce this number of duplicated mutants. However, it does not eliminate them entirely. This limitation arises from applying mutations to all changed files rather than solely focusing on the changed code. Only when applying mutations to the changed code will ensure that each mutant is unique.

It is also possible to verify that the traditional way of applying mutations to the entire project creates a much higher number of mutants than only applying to the changed files. If mutation testing was applied to all versions of Aegis, a total number of 2652 mutants would be created. When applying mutation to only the changed files, that number reduces to 341 mutants. This represents a reduction of 87% in the number of mutants. Considering that the tests also would need to be executed 2652 times, it would not be feasible the execution of mutation testing in the context of an evolving software project.

To address RQ6, our findings suggest that we maintain the effectiveness of fault detection when applying mutation testing exclusively to the changed parts of the code. This approach results in a significant reduction of duplicated mutants between versions.

Chapter 7

Conclusions and Future Work

Even though the benefits of mutation testing have already been shown, it is still not widely adopted in the software industry due to the high computational cost. A high number of changes that simulate defects can be applied, and subsequently, many test executions need to be executed to detect those changes. In the case of Android applications, the computational cost associated with mutation testing is further magnified. In traditional mutation testing, every Android-specific code change requires generating a new APK that needs to be deployed on a device to execute the tests. This challenge highlights the need for efficient strategies that mitigate the computational cost associated with mutation testing on Android applications.

During this research, we conducted a comprehensive literature review on the significant techniques that reduce the cost of mutation testing in both Java traditional projects and Java Android projects. Some techniques try reducing the number of mutants while maintaining the same mutation score. This is the case of mutant sampling, mutant clustering, selective mutation and high-order mutation. Mutation testing also can create mutants that are semantically different from the original program but maintain the same behaviour. This detection typically needs to be done manually by the tester. It was also possible to verify strategies that tackle this problem. Finally, it was observed that specific strategies aim to reduce the cost of mutation testing by optimising the mutant execution process. This is the case of strong, weak, and firm mutation, run-time optimisation techniques, and mutant schemata.

After the comprehensive literature review, we analysed existing mutation tools. It became evident that Java-specific tools incorporated more cost-reduction techniques than Android-specific tools. Java-specific tools primarily operated at the byte code level, while Android-specific tools focused on the AST. Some Java tools offered equivalent mutant detection, while no Android tool provided this feature. Most Android tools lacked updates, integration, documentation, and support.

Following the analysis of the tools, we present two innovative approaches to mutation testing. The first approach suggests the independent application of mutation operators, where they alter the AST without knowing whether traditional or schemata-based mutations are being employed.

The second approach addresses the fact that software undergoes continuous updates and improvements, resulting in different versions that require testing. Despite efforts to reduce the cost of mutation testing, each new release generates equal mutations for unchanged code, requiring additional compilation, testing, and analysis time. Our approach utilises Git versioning to identify what files should be mutated.

Finally, we presented the developed tool, outlining the architectural decisions made. We discussed its integration with Kadabra and Git, highlighting the version control capabilities. The tool incorporates various cost reduction techniques, including selective mutation, mutant schemata, and parallelization aimed at improving the efficiency and effectiveness of mutation testing.

7.1 Results

To evaluate the performance improvements of our tool, multiple experiments were conducted. First, the impact of mutation operators was analysed. It was discovered that with our implementation, mutation operators have no direct impact on generation time or size. Mutation operators with similar number of generated mutants had similar generation times and sizes. However, mutation operators significantly affect test execution time and overall elapsed time. Some mutation operators introduced code variations that increased test execution time, resulting in longer overall elapsed time.

Secondly, it was analysed the performance benefit of mutant schemata. On the generation of the mutants, it was clear that using schemata brings enormous benefits in generation time and size. However, the advantages of mutant schemata can be compromised by slower test execution times. Depending on the project and the specific mutation operators employed, a mutant schemata with high number of mutants can exhibit slower performance than the traditional approach. These findings are in accordance with the findings of Diego Naveiras [27].

Finally, the Git optimization was analysed. In summary, applying mutations to the entire project generates a significantly higher number of mutants compared to applying mutations only to the changed files. The reduction from 2652 mutants (all versions) to 341 mutants (changed files only) represents an 87% decrease. We concluded that our approach makes it feasible to perform mutation testing on different versions of an evolving software project.

7.2 Further Work

Like any project developed within a limited timeframe, there is room for further improvement and extension in this project. The developed tool is regarded as being in the early stages of its full potential, leaving ample room for enhancements and advancements.

In the context of mutant generation, further experiments are required to validate the performance benefit introduced by creating multiple LARA environments for each file. It would be interesting to compare this approach against a single LARA environment that mutates the entire project. Furthermore, the performance impact of applying the code transformation in the schemata

format was not measured. It would be interesting to measure the overhead created by this transformation. Also, the generation of the classpath automatically would be a welcome feature.

The tool can also be extended in the context of test execution. Currently, all the tests are executed, but optimisations like executing until a first failed test can be implemented to reduce the test execution time further. Additionally, as our tool stores all the information of each test execution on a database, learning algorithms can be implemented to process that data and define the order of each individual test. The ultimate goal would be executing tests most likely to kill the mutant first. With sufficient data, these learning algorithms could determine what would be most beneficial for each project if applying traditional or schemata mutation.

Finally, it would be an excellent addition to the tool the creation of a front-end application that would make more accessible the interaction of users.

Appendix A

Lara Environments Code

All LARA environments code can be found in this appendix. This code is also publicly available on GitHub ¹.

A.1 Main Lara Environment Code

```
1 laraImport("lara.io");
2 laraImport("lara.Strings");
3 laraImport("weaver.Query");
4 laraImport("Arguments");
5
6 const outputPath = laraArgs.outputPath;
7 const traditionalMutation = laraArgs.traditionalMutation;
8 const projectPath = laraArgs.projectPath;
9 const debugMessages = laraArgs.debugMessages;
10 const folderToIgnore = laraArgs.folderToIgnore;
11 const folderToIgnoreAndroid = laraArgs.folderToIgnoreAndroid;
12 const operatorNameList = laraArgs.operatorNameList;
13 const operatorArgumentList = laraArgs.operatorArgumentList;
14 const projectExecutionName = laraArgs.projectExecutionName;
15 const includesFolder = laraArgs.includesFolder;
16 const isAndroid = laraArgs.isAndroid;
17 const classpath = laraArgs.classpath;
18 const useIncompleteClassPath = laraArgs.useIncompleteClassPath;
19 const mutationType = laraArgs.mutationType;
20 const useGitVersioning = laraArgs.useGitVersioning;
21 const gitFiles = laraArgs.gitFiles;
22
23 main();
24
25 function main() {
26     //Shows additional prints
27     if (debugMessages) {
```

¹<https://github.com/specs-feup/mutation-testing-v2>

```
28     setDebug(true);
29 }
30
31 //makes the project copy if it's not being used traditional mutation
32 if (!traditionalMutation) {
33     Io.copyFolder(
34         projectPath,
35         outputPath + Io.getSeparator() + projectExecutionName,
36         true
37     );
38 }
39
40 let filesToUse = [];
41 // Get only java files without the test files
42 if (useGitVersioning) {
43     filesToUse = gitFiles;
44 } else {
45     filesToUse = getFilesToUse();
46     println("Files to use: " + filesToUse);
47 }
48
49 //Creates the arguments for each kadabra parallel execution
50 args_final = [];
51 for (i in filesToUse) {
52     args = {
53         outputPath: outputPath.trim(),
54         filePath: filesToUse[i].toString(),
55         projectPath: projectPath,
56         debugMessages: debugMessages,
57         operatorNameList: operatorNameList,
58         operatorArgumentList: operatorArgumentList,
59         projectExecutionName: projectExecutionName,
60         isAndroid: laraArgs.isAndroid,
61         mutationType: laraArgs.mutationType,
62     };
63
64     let args_kadabra = new Arguments(
65         (outputPath + Io.getSeparator() + projectExecutionName).trim(),
66         JSON.stringify(args),
67         filesToUse[i],
68         traditionalMutation,
69         includesFolder,
70         classpath,
71         useIncompleteClassPath,
72         isAndroid
73     ).getList();
74
75     args_final.push(args_kadabra);
76 }
```

```

77
78 //Kadabra Parallel execution
79 let result = Weaver.runParallel(args_final, args_final.length);
80
81 //Writes the output formatted to a file
82 writeExecutionInfo(result);
83 }
84
85 function getFilesToUse() {
86   let filesToUse = [];
87
88   //Checks what files to use
89   let allJavaFiles = Io.GetFiles(projectPath, "*.java", true);
90   let javaFilesToRemove = Io.GetFiles(folderToIgnore, "*.java", true);
91   println("AllJavaFiles: " + allJavaFiles);
92   println("Folder yo Ignore: " + folderToIgnore);
93   println("javaFilesToRemove: " + javaFilesToRemove);
94
95   if (
96     folderToIgnore != null &&
97     folderToIgnore != "" &&
98     folderToIgnore.replace(projectPath, "") != ""
99   ) {
100     for (i in allJavaFiles) {
101       for (j in javaFilesToRemove) {
102         if (allJavaFiles[i].equals(javaFilesToRemove[j])) {
103           break;
104         }
105         if (j == javaFilesToRemove.length - 1 && !j.includes("build")) {
106           filesToUse.push(allJavaFiles[i]);
107         }
108       }
109     }
110   } else {
111     filesToUse = allJavaFiles;
112   }
113
114   if (folderToIgnoreAndroid != null && folderToIgnoreAndroid != "") {
115     let javaFilesToRemove = Io.GetFiles(folderToIgnoreAndroid, "*.java", true);
116     let filesToUseFinal = [];
117
118     for (i in filesToUse) {
119       for (j in javaFilesToRemove) {
120         if (filesToUse[i].equals(javaFilesToRemove[j])) {
121           break;
122         }
123         if (j == javaFilesToRemove.length - 1 && !j.includes("build")) {
124           filesToUseFinal.push(filesToUse[i]);
125         }

```

```

126     }
127 }
128
129     return filesToUseFinal;
130 }
131
132     return filesToUse;
133 }
134
135 function writeExecutionInfo(result) {
136     let fileData = [];
137
138     for (i in result) {
139         if (result[i]["output"] != "") {
140             try {
141                 let listaAux = JSON.parse(result[i]["output"]);
142                 for (j in listaAux) {
143                     fileData.push(listaAux[j]);
144                 }
145             } catch (error) {
146                 println(result[i]);
147                 println(error);
148             }
149         }
150     }
151
152     Io.writeFile(
153         outputPath +
154         Io.getSeparator() +
155         projectExecutionName +
156         Io.getSeparator() +
157         "MutationInfo.json",
158         JSON.stringify(fileData)
159     );
160 }

```

A.2 Traditional Mutation Lara Environment Code

```

1 laraImport("lara.Io");
2 laraImport("lara.Strings");
3 laraImport("weaver.Query");
4 laraImport("weaver.Script");
5 laraImport("kadabra.KadabraNodes");
6 laraImport("MutationOperators.*");
7 laraImport("MutatorList");
8 laraImport("Decomposition");

```

```

9
10 const outputPath = laraArgs.outputPath;
11 const filePath = laraArgs.filePath;
12 const projectPath = laraArgs.projectPath.trim();
13 const debugMessages = laraArgs.debugMessages;
14 const fileName = filePath.substring(
15   filePath.lastIndexOf(IO.getSeparator()) + 1
16 );
17 const projectExecutionName = laraArgs.projectExecutionName;
18
19 main();
20
21 function main() {
22   //Shows additional prints
23   if (debugMessages) {
24     setDebug(true);
25   }
26
27   //If no mutators were selected
28   if (mutatorList.length === 0) {
29     println("No mutators selected");
30     return;
31   }
32
33   //changeVarDeclarations();
34   println("Traditional Mutation");
35
36   let output = {};
37
38   //Goes to each node and stores the mutation point
39   println("Going through AST for file " + fileName);
40   runTreeAndGetMutantsTraditionally();
41
42   //Goes to each stored mutation point and applies the mutation
43   println("Generating Mutants for file " + fileName);
44   output = applyTraditionalMutation();
45
46   Script.setOutput({ output });
47 }
48
49 function runTreeAndGetMutantsTraditionally() {
50   for (var $jp of Query.root().descendants) {
51     var $call = $jp.ancestor("call");
52
53     // Ignore nodes that are children of $call with the name <init>
54     if ($call !== undefined && $call.name === "<init>") continue;
55
56     for (mutator of mutatorList) {
57       if (mutator.addJp($jp)) {

```

```
58         debug(mutator);
59     }
60 }
61 }
62 }
63
64 function applyTraditionalMutation() {
65     let auxOutputStr = [];
66     for (mutator of mutatorList) {
67         while (mutator.hasMutations()) {
68             let auxLine = mutator.getMutationPoint().line;
69
70             //Applies the mutation
71             mutator.mutate();
72
73             //Saves to a file
74             let path = saveFile(mutator.getName());
75
76             auxOutputStr.push({
77                 mutantId: path,
78                 mutation: mutator.toJson(),
79                 mutationLine: auxLine,
80                 filePath: Io.getRelativePath(filePath, projectPath),
81             });
82         }
83     }
84
85     return JSON.stringify(auxOutputStr);
86 }
87
88 function saveFile(mutatorName) {
89     let relativePath = Io.getRelativePath(filePath, projectPath);
90
91     let aux =
92         Io.getSeparator() +
93         mutatorName +
94         "_" +
95         fileName.replace(".java", "") +
96         "_" +
97         Strings.uuid();
98
99     let newFolder = outputPath + Io.getSeparator() + projectExecutionName + aux;
100
101     Io.copyFolder(projectPath, newFolder, true);
102
103     Io.writeFile(
104         newFolder +
105         Io.getSeparator() +
106         relativePath.replace("/", Io.getSeparator()),
```



```

107     Query.root().code
108 );
109
110     return aux;
111 }

```

A.3 Java Mutant Schemata Lara Environment Code

```

1  laraImport("lara.io");
2  laraImport("lara.Strings");
3  laraImport("weaver.Query");
4  laraImport("weaver.Script");
5  laraImport("kadabra.KadabraNodes");
6  laraImport("MutationOperators.*");
7  laraImport("MutatorList");
8  laraImport("Decomposition");
9
10 const outputPath = laraArgs.outputPath;
11 const filePath = laraArgs.filePath;
12 const projectPath = laraArgs.projectPath.trim();
13 const debugMessages = laraArgs.debugMessages;
14 const fileName = filePath.substring(
15     filePath.lastIndexOf(io.getSeparator()) + 1
16 );
17 const operatorNameList = laraArgs.operatorNameList;
18 const projectExecutionName = laraArgs.projectExecutionName;
19 const isAndroid = laraArgs.isAndroid;
20
21 main();
22
23 function main() {
24     //Shows additional prints
25     if (debugMessages) {
26         setDebug(true);
27     }
28
29     //If no mutatans were selected
30     if (mutatorList.length === 0) {
31         println("No mutators selected");
32         return;
33     }
34
35     changeVarDeclarations();
36     println("Mutant Schemata");
37
38     let output = runTreeAndApplyMetaMutant();

```

```

39
40 //print("Output" + output);
41 Script.setOutput({ output });
42 }
43
44 function runTreeAndApplyMetaMutant() {
45     var mutantList = [];
46     for (var $jp of Query.root().descendants) {
47         var $call = $jp.ancestor("call");
48
49         // Ignore nodes that are children of $call with the name <init>
50         if ($call !== undefined && $call.name === "<init>") continue;
51
52         let mutationPoints = 0;
53         let needElseIf = false;
54         let firstTime = true;
55         for (mutator of mutatorList) {
56             if (mutator.addJp($jp)) {
57                 mutationPoints++;
58                 debug(mutator);
59             }
60         }
61
62         if (mutationPoints >= 2) {
63             needElseIf = true;
64         }
65
66         for (mutator of mutatorList) {
67             while (mutator.hasMutations()) {
68                 let mutantId =
69                     mutator.getName() +
70                     "_" +
71                     fileName.replace(".java", "") +
72                     "_" +
73                     Strings.uuid();
74
75                 mutantList.push({
76                     mutantId: mutantId,
77                     mutation: mutator.toJson(),
78                     mutationLine: mutator.getMutationPoint().line,
79                     filePath: Io.getRelativePath(filePath, projectPath),
80                 });
81
82                 // Mutate
83                 mutator.mutate();
84
85                 if (
86                     mutator.name === "NotSerializableOperatorMutator" ||
87                     mutator.name === "NonVoidCallMutator"

```

```

88         ) {
89             var mutated = mutator.getMutationPoint();
90         } else {
91             var mutated = mutator.getMutationPoint().isStatement
92                 ? mutator.getMutationPoint()
93                 : mutator.getMutationPoint().ancestor("statement");
94         }
95
96         //print(mutator.toJson());
97
98         if (needElseIf) {
99             if (mutationPoints > 1) {
100                 if (firstTime) {
101                     mutated.insertBefore(
102                         'if(System.getProperty("MUID") != null && System.getProperty("MUID
103                         ").equals("' +
104                             mutantId +
105                             '")){\n' +
106                             mutated.srcCode +
107                             ";\n}"
108                     );
109                     firstTime = false;
110                 } else {
111                     mutated.insertBefore(
112                         'else if(System.getProperty("MUID") != null && System.getProperty("
113                         MUID").equals("' +
114                             mutantId +
115                             '")){\n' +
116                             mutated.srcCode +
117                             ";\n}"
118                     );
119                 }
120                 mutationPoints--;
121             } else {
122                 mutated.insertBefore(
123                     'else if (System.getProperty("MUID") != null && System.getProperty("
124                     MUID").equals("' +
125                         mutantId +
126                         '")){\n' +
127                         mutated.srcCode +
128                         ";\n}else{\n\t"
129                     );
130                 mutated.insertAfter("}");
131             }
132         } else {
133             mutated.insertBefore(
134                 'if (System.getProperty("MUID") != null && System.getProperty("MUID").
135                 equals("' +

```

```

133         mutantId +
134         '")){\n' +
135         mutated.srcCode +
136         ";\n}else{\n\t"
137     );
138     mutated.insertAfter(")");
139 }
140 mutator.restore();
141 }
142 }
143 }
144
145 //Saves the file
146 let relativePath = Io.getRelativePath(filePath, projectPath);
147
148 let aux =
149     outputPath +
150     Io.getSeparator() +
151     projectExecutionName +
152     Io.getSeparator() +
153     relativePath.replace("/", Io.getSeparator());
154
155 Io.writeFile(aux, Query.root().srcCode);
156
157 return JSON.stringify(mutantList);
158 }

```

A.4 Android Mutant Schemata Lara Environment Code

```

1 laraImport("lara.Io");
2 laraImport("lara.Strings");
3 laraImport("weaver.Query");
4 laraImport("weaver.Script");
5 laraImport("kadabra.KadabraNodes");
6 laraImport("MutationOperators.*");
7 laraImport("MutatorList");
8 laraImport("Decomposition");
9
10 const outputPath = laraArgs.outputPath;
11 const filePath = laraArgs.filePath;
12 const projectPath = laraArgs.projectPath.trim();
13 const debugMessages = laraArgs.debugMessages;
14 const fileName = filePath.substring(
15     filePath.lastIndexOf(Io.getSeparator()) + 1
16 );
17 const operatorNameList = laraArgs.operatorNameList;

```

```

18 const projectExecutionName = laraArgs.projectExecutionName;
19 const isAndroid = laraArgs.isAndroid;
20 const mutationType = laraArgs.mutationType;
21
22 main();
23
24 function main() {
25     //Shows additional prints
26     if (debugMessages) {
27         setDebug(true);
28     }
29
30     //If no mutagens were selected
31     if (mutatorList.length === 0) {
32         println("No mutators selected");
33         return;
34     }
35
36     changeVarDeclarations();
37     println("Mutant Schemata Android");
38
39     let output = runTreeAndApplyMetaMutant();
40
41     if (output.length > 0) {
42         //print("Output" + output);
43         Script.setOutput({ output });
44     } else {
45         Script.setOutput({});
46     }
47 }
48
49 function runTreeAndApplyMetaMutant() {
50     if (mutationType === "MUTANTSCHMATA") {
51         const auxFunction = `
52         public static String getMUID(){ \
53         String propertyValue = null; \
54         try { \
55         Process process = Runtime.getRuntime().exec("getprop MUID"); \
56         InputStream inputStream = process.getInputStream(); \
57         BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
58         \
59         propertyValue = reader.readLine();\
60         reader.close();\
61         inputStream.close();\
62         } catch (IOException e) {\
63         Log.e("ERROR", String.valueOf(e));\
64         }\
65         return propertyValue;\
66     }`;

```

```

66
67     const imports = `
68     import java.io.BufferedReader;\n
69     import java.io.IOException;\n
70     import java.io.InputStream;\n
71     import java.io.InputStreamReader;\n
72     import android.util.Log;
73     `;
74
75     for (var $jp of Query.search("class")) {
76         $jp.insertBefore(imports);
77         $jp.insertMethod(auxFunction);
78         break;
79     }
80 } else if (isAndroid) {
81     const imports = `
82     import com.beemdevelopment.aegis.BuildConfig;\n
83     `;
84
85     for (var $jp of Query.search("class")) {
86         $jp.insertBefore(imports);
87         break;
88     }
89 }
90
91 var mutantList = [];
92 for (var $jp of Query.root().descendants) {
93     var $call = $jp.ancestor("call");
94
95     // Ignore nodes that are children of $call with the name <init>
96     if ($call !== undefined && $call.name === "<init>") continue;
97
98     let mutationPoints = 0;
99     let needElseIf = false;
100     let firstTime = true;
101     for (mutator of mutatorList) {
102         if (mutator.addJp($jp)) {
103             mutationPoints++;
104             debug(mutator);
105         }
106     }
107
108     if (mutationPoints >= 2) {
109         needElseIf = true;
110     }
111
112     for (mutator of mutatorList) {
113         while (mutator.hasMutations()) {
114             let mutantId =

```

```

115         mutator.getName() +
116         "_" +
117         fileName.replace(".java", "") +
118         "_" +
119         Strings.uuid();
120
121     mutantList.push({
122         mutantId: mutantId,
123         mutation: mutator.toJson(),
124         mutationLine: mutator.getMutationPoint().line,
125         filePath: Io.getRelativePath(filePath, projectPath),
126     });
127
128     // Mutate
129     mutator.mutate();
130
131     if (
132         mutator.name === "NotSerializableOperatorMutator" ||
133         mutator.name === "NonVoidCallMutator"
134     ) {
135         var mutated = mutator.getMutationPoint();
136     } else {
137         var mutated = mutator.getMutationPoint().isStatement
138             ? mutator.getMutationPoint()
139             : mutator.getMutationPoint().ancestor("statement");
140     }
141
142     if (mutator.isAndroidSpecific() && mutationType === "MUTANTSCHEMATA") {
143         if (needElseIf) {
144             if (mutationPoints > 1) {
145                 if (firstTime) {
146                     mutated.insertBefore(
147                         'if(getMUID().equals("' +
148                         mutantId +
149                         '")){\n' +
150                         mutated.srcCode +
151                         ";\n}"
152                     );
153
154                     firstTime = false;
155                 } else {
156                     mutated.insertBefore(
157                         'else if(getMUID().equals("' +
158                         mutantId +
159                         '")){\n' +
160                         mutated.srcCode +
161                         ";\n}"
162                     );
163                 }

```

```

164         mutationPoints--;
165     } else {
166         mutated.insertBefore(
167             'else if (getMUID().equals("' +
168                 mutantId +
169                 '")){\n' +
170                 mutated.srcCode +
171                 ";\n}else{\n\t"
172             );
173         mutated.insertAfter("}");
174     }
175 } else {
176     mutated.insertBefore(
177         'if (getMUID().equals("' +
178             mutantId +
179             '")){\n' +
180             mutated.srcCode +
181             ";\n}else{\n\t"
182         );
183     mutated.insertAfter("}");
184 }
185 } else {
186     if (needElseIf) {
187         if (mutationPoints > 1) {
188             if (firstTime) {
189                 mutated.insertBefore(
190                     'if(BuildConfig.MUID != null && BuildConfig.MUID.equals("' +
191                         mutantId +
192                         '")){\n' +
193                         mutated.srcCode +
194                         ";\n}"
195                 );
196             }
197             firstTime = false;
198         } else {
199             mutated.insertBefore(
200                 'else if(BuildConfig.MUID != null && BuildConfig.MUID.equals("' +
201                     mutantId +
202                     '")){\n' +
203                     mutated.srcCode +
204                     ";\n}"
205             );
206         }
207         mutationPoints--;
208     } else {
209         mutated.insertBefore(
210             'else if (BuildConfig.MUID != null && BuildConfig.MUID.equals("' +
211                 mutantId +
212                 '")){\n' +

```



```
213         mutated.srcCode +
214         ";\n}else{\n\t"
215     );
216     mutated.insertAfter(")");
217 }
218 } else {
219     mutated.insertBefore(
220         'if (BuildConfig.MUID != null && BuildConfig.MUID.equals("' +
221         mutantId +
222         '")){\n' +
223         mutated.srcCode +
224         ";\n}else{\n\t"
225     );
226     mutated.insertAfter("}");
227 }
228 }
229 }
230
231     mutator.restore();
232 }
233 }
234
235 //Saves the file
236 let relativePath = Io.getRelativePath(filePath, projectPath);
237
238 let aux =
239     outputPath +
240     Io.getSeparator() +
241     projectExecutionName +
242     Io.getSeparator() +
243     relativePath.replace("/", Io.getSeparator());
244
245 Io.writeFile(aux, Query.root().srcCode);
246
247 return JSON.stringify(mutantList);
248 }
```

References

- [1] Allen Troy Acree. *On Mutation*. PhD thesis, USA, 1980. AAI8107280.
- [2] Domenico Amalfitano, Ana C. R. Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. How do java mutation tools differ? *Commun. ACM*, 65(12):74–89, nov 2022.
- [3] and, , Pradeep Kumar Singh, Om Prakash Sangwan, and Arun Sharma. A Study and Review on the Development of Mutation Testing Tools for Java and Aspect-J Programs. *International Journal of Modern Education and Computer Science*, 6(11):1–10, 2014.
- [4] Francisco Bernardo Azevedo. Cost Reduction Technique for Mutation Testing, 2020.
- [5] Adam S. Banzi, Tiago Nobre, Gabriel B. Pinheiro, João Carlos G. Árias, Aurora Pozo, and Silvia Regina Vergilio. Selecting mutation operators with a multiobjective approach. *Expert Systems with Applications: An International Journal*, 39(15):12131–12142, nov 2012.
- [6] Ilona Bluemke and Karol Kulesza. Reductions of operators in Java mutation testing. *Advances in Intelligent Systems and Computing*, 286:93–102, 2014.
- [7] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, USA, 1980. AAI8025191.
- [8] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452, jul 2016.
- [9] R.A. DeMillo and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. cited By 1295.
- [10] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. Towards mutation analysis of android apps. 2015. Cited by: 31.
- [11] Anna Derezinska and Marcin Rudnik. Evaluation of mutant sampling criteria in object-oriented mutation testing. *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017*, pages 1315–1324, nov 2017.
- [12] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. A systematic mapping study on higher order mutation testing. *Journal of Systems and Software*, 154:92–109, aug 2019.
- [13] W.E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, 1982. cited By 307.

- [14] Reyhaneh Jabbarvand and Sam Malek. droid: An energy-aware mutation testing framework for android. volume 2017-January, page 208 – 219, 2017. Cited by: 40; All Open Access, Bronze Open Access.
- [15] Y Jia and M Harman. *Higher Order Mutation Testing (Dissertation)*. PhD thesis, 2013.
- [16] Yue Jia and Mark Harman. Constructing subtle faults using Higher Order mutation testing. *Proceedings - 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, pages 249–258, 2008.
- [17] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
- [18] René Just. *On effective and efficient mutation analysis for unit and integration testing*. Ph.D., Ulm University, 2013.
- [19] René Just. The major mutation framework: Efficient and scalable mutation analysis for Java. *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 433–436, jul 2014.
- [20] Jian Liu, Xusheng Xiao, Lihua Xu, Liang Dou, and Andy Podgurski. Droidmutator: An effective mutation analysis tool for android applications. page 77 – 80, 2020. Cited by: 1.
- [21] Eduardo Luna and Omar El Ariss. Edroid: A mutation tool for android apps. page 99 – 108, 2019. Cited by: 8.
- [22] Yu Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An automated class mutation system. *Software Testing Verification and Reliability*, 15(2):97–133, jun 2005.
- [23] Lech Madeyski and Norbert Radyk. Judy - a mutation testing tool for java. *Software, IET*, 4:32 – 42, 03 2010.
- [24] Pedro Reales Mateo and Macario Polo Usaola. Mutant execution cost reduction: Through music (mutant schema improved with extra code). In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 664–672, 2012.
- [25] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*, pages 604–605, 1991.
- [26] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. Mdroid+: A mutation testing framework for android. page 33 – 36, 2018. Cited by: 28; All Open Access, Green Open Access.
- [27] Diego Seco Naveiras. *Mutation Testing Techniques for Mobile Applications*. 2021.
- [28] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996.
- [29] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. pages 34–44, 05 2001.

- [30] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. *Proceedings - International Conference on Software Engineering*, 1:936–946, aug 2015.
- [31] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, 112:275–378, jan 2019.
- [32] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.
- [33] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. Software testing and Android applications: a large-scale empirical study. *Empirical Software Engineering*, 27(2):31, mar 2022.
- [34] Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João M.P. Cardoso. Aspect composition for multiple target languages using lara. *Computer Languages, Systems Structures*, 53:1–26, 2018.
- [35] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157, nov 2019.
- [36] Macario Polo, Mario Piattini, and Ignacio Garía-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing Verification and Reliability*, 19(2):111–131, 2009.
- [37] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for java. page 297 – 298, 2009. Cited by: 139.
- [38] S Hussain Ms. Th., Kings College London, Undefined Strand, Undefined London, and Undefined 2008. *Mutation clustering*. PhD thesis, 2007.
- [39] Roland H. Untch, Mary Jean Harrold, and A. Jefferson Offutt. Tums: Testing using mutant schemata. *Proceedings - 35th Annual Southeast Regional Conference, ACM-SE 1997*, pages 174–181, apr 1997.
- [40] Macario Polo Usaola, Gonzalo Rojas, Isyed Rodriguez, and Suilen Hernandez. An architecture for the development of mutation operators. page 143 – 148, 2017. Cited by: 7.
- [41] Ana Rita Veiga. Mutation Operators for Android Apps - To be published. 2023.
- [42] M.R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, July 1988.