FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Cost Reduction Technique for Mutation Testing

Francisco Bernardo Azevedo

Mestrado em Engenharia de Software

Supervisor: Ana Paiva

Second Supervisor: João Bispo

October 28, 2020

Cost Reduction Technique for Mutation Testing

Francisco Bernardo Azevedo

Mestrado em Engenharia de Software

October 28, 2020

Abstract

Software testing is a fundamental part of the software development process. This is the process that tests if the software is properly developed and should always test for not only the functionality but also test for common mistakes. This is the importance of Mutation testing.

Mutation testing is a type of testing that tests not only the developed software, but most importantly, the developed software tests. This testing works by injecting common defects on the software (Mutations) and expect the test to fail. Failed tests mean that the test takes into account that common issue, and therefore the tests are prepared for it.

The problem with this approach however, is that it is very resource consuming, specially on time. This happens as a new version of the software is needed for every new code injection. This is where this new technique tries to improve the approach, by instead of creating a single version for every mutation it creates only a single version for all the mutations. This without losing the software functionality nor the capability of testing just one mutation at a time.

Keywords: Mutation Testing, Mutation Testing Process, Software Quality, Software Testing, Code injection, Code Defects

Resumo

Testar o Software é uma parte fundamental do processo de desenvolvimento do software. Este é o processo que testa se o Software está desenvolvido da forma correta e deve testar sempre não só a funcionalidade, mas também por erros comuns. E é nesta última parte que entram os Testes de Mutação.

Testes de Mutação é um tipo de testes que testa não só o Software desenvolvido, mas mais importante, os testes desse Software desenvolvido. Este método de testes funciona ao injetar erros comuns no Software (Mutações) e esperar que os testes falhem. Testes falhados significa que esse teste toma em consideração esse erro comum e, por isso, os testes estão preparados para esse erro.

O problema deste método, no entanto, é que consome muitos recursos, especialmente tempo. Isto acontece porque por uma versão nova do software é precisa para cada injeção de código. É aqui que esta nova técnica tenta trazer alguma contribuição nova, ao invés de criar uma versão e código para cada mutação, criar apenas uma versão com todas as mutações. Isto tudo sem nunca perder a funcionalidade original nem a possibilidade de executar apenas uma mutação de cada vez

Keywords: Testes de Mutação, Processo de Teste de Mutação, Qualidade de Software, Testes de Software, Injeçao de código, Defeitos de código

Acknowledgements

This thesis could not have been completed without the help and incentives, and therefore the acknowledgement, of the following:

The professors Ana Paiva and João Bispo, for their sharing of knowledge, their help, and overall support.

The TVVS students who helped developing the initial version of the mutation operators.

My family, for all the things I got them trough and the support they gave, for me to be able to deliver this thesis.

My friends, for the support they gave so I could achieve this goal.

For the persons at work, that sometimes without even knowing helped me during this phase.

And for everyone else, that in one way or another contributed for this thesis.

Francisco Azevedo

“The best way to predict your future
is to create it..”

Abraham Lincoln

Contents

List of Figures

List of Tables

Abbreviations

AST	Abstract Syntax Tree
GUI	Graphical User Interface
CLI	Command Line Interface
API	Application Programming Interface
XML	Extensible Markup Language
MUID	Mutation Unique Identifier

Chapter 1

Introduction

The Software industry is an ever-growing industry that can be found almost everywhere nowadays. This software however, is still made by humans, and that means its still prone to mistakes. That's why on Software development there is a kind of code that its only purpose is to test the final Software.

This testing process works by executing portions of the original code, and compare its results with expected results, this is called unit testing. This results can range from expected values, strings, database changes, exceptions or more, depending on the testing framework in use.

This tests can however, fail to cover common problems that a developer might make without even noticing. For instance, its easy for a developer to instead of coding a ">" to code a "<" when comparing two values as both are in the same keyboard key. If this issue occurs on a hard to notice place in the code, and the tests do not verify it, it might stay undetected and cause issues when deployed. This then, calls for a way to test the before mentioned unit tests for common issues, to see if they are testing for said issues.

For this Mutation Testing was introduced. It works by changing a copy of the original code, injecting common defects, commonly known as mutations, and then execute them against the original tests. The tests are then expected to have at least one failed one. This failure will mean that the failed test is taking that common mistake, on that particular place, into consideration.

Having several examples of already implemented tools for source code mutation testing, this methodology can also be applied to models [2], execution traces [14], web testing [1], amongst other processes. The scope still remains the same, checking whether the test suite of said processes tests for common failures of the process, having the only change being that instead of mutating the source code, the mutations will be on its equivalent for the process.

1.1 Context and Problem

Mutation testing however, has an intrinsic problem. On large programs, that have thousands of lines of code, and equivalent number of tests for that same code, it might require long portions of time and processing power to execute. This happens as a program this big will have most likely also have thousands of mutations, every one of which will require to have its own dedicated version to test, so the place that is not properly tested can be pinpointed. Also most tools fail to have a proper interface or documentation, making harder for the user to optimize its execution and integration on the testing process.

This is where this research work comes in, the creation of a tool that tackles the before mentioned issues by having an higher degree of configurability, and, that uses, a new and more optimized process for mutation testing, as defined on chapter 4.

This process will try to tackle the problem of the performance, therefore improving the process in a way that is more feasible. This is possible by trying to reduce the two most time consuming tasks, files generation and compilation times by only requiring to do both tasks only once in the process, instead of a number of times equivalent to the number of mutants.

To tackle the configurability problems, that are further identified in chapter 2 with an in-depth analysis of the existing tools, we designed an framework, that uses an existing tool as a base, and that will be available for the usedt to use a set of pre-made mutation operators or make his own operators, with little to no extra knowledge required. This is possible as we designed an interface for our tool that requires the user to have knowledge of the widespread programming language *JavaScript*, and an easy to learn API for our framework.

1.2 Document structure

This document will be structured in 7 chapters where we try to explain our process, they are as follows.

The first chapter is this introductory one, where we expose the problem and we will try to solve the said problem.

The second chapter, a State of the Art, where we identify another existing tools, analyze them and identify their problems so we can better understand what is required to do.

The third chapter is where we mention an interface we developed achieve better configurability, and a background on the existent processes and both how we implement the selected process and how the tool is developed.

The Fourth chapter is how we try to achieve better performances, by initially identifying the classic mutation process and how it works, and then our own process and how we perform it.

The fifth chapter where we describe the tool we developed and how it works, so we could test both our proposed interface and our process.

The sixth chapter that, with the knowledge gained and the tool developed on the previous chapters, we made a comparison on the mutations processes, to see how they perform.

And finally the seventh chapter, where we take our conclusions and mention future work.

Chapter 2

State of the art

The ever-growing Software world is constantly being flooded with new and more complex software. Software which most of the times is not being properly tested. This failure on testing often leads to undiscovered failures, that, when found, might cause significant damage. Often those failures have the same common mistakes, mistakes that can be tested and checked before deployment. This verification is performed with tests that may or may not have been properly implemented.

The tests however might not be properly implemented still. That is where mutation testing comes in. It is a way to check if the software testing phase tests for common issues, by injecting a replication of those issues on the existing code and expect the tests to detect them . These changes on the code are called mutations, and there are two ways to apply them, more specifically, on the source code, or on some approaches, the compiled byte-code itself.

Afterwards the same application test suite is run on the mutated code and expected to fail, and with this failure, detecting the injected failures. This way the tool can ensure that the test suite developed for the application, does indeed, detect the common failures the tool injects. Although not standardized, at the end, a report with relevant information is often generated with useful information such as the number of mutations generated, and more importantly, which of them passed on all the unit tests. With this report the developers know which common issue is not being tested, and therefore, know which new unit tests or refactor of existing ones, should be performed.

2.1 Existing Mutation Tools

As mentioned on 2, there are indeed some mutation testing tools, but they all lack something, either documentation, better user interfaces, few mutators and amongst other problems, most have who just stopped being maintained. Also most are not made taking into account what is becoming the biggest software market, the mobile applications, and as such, not prepared to tackle its common mistakes. These issues are shown in Figure 2.1, found in "*Enabling Mutation Testing for Android Apps*" [10], which mentions at least 151 types of bugs just specific for Android.

So, given that the market is slowly changing, we are separating the following tools, which will later be compared on 2.2, in Mobile Application and, due to the base language being common, Java Language Specific. These tools were selected given their relevance and available literature.

2.1.1 Mobile Application Specific

2.1.1.1 Edroid

Edroid is a tool developed with the intention to mutate Android apps, more specifically its XML layout and configuration files, having the goal to add the Java code mutators in the future. Future which might not appear, considering the only information found is 2 year old conference paper [11] and there is no place to obtain the tool and confirm its updates.

It works firstly by asking the user for the source files of the application to be tested, after which it selects the mutation operators to use, between the 14 available. This selection process is also uncertain if it is possible to be done manually, as it states that a subset of the mutants can be used, but it also states that the mutants cannot be removed. It also does not verify the existence of equivalent mutators (Mutators that have the same behaviour as the original code), which may impact the overall performance. Having done this two steps and pressing the "Generate" button the process begins.

The process begins by identifying the XML files to mutate and dividing them into Layout and Configuration files. It preforms that by opening the files as text files and then perform a search for keywords. After which 11 layout mutants of the 14 mutators on the tool might be applied. For the configuration files the remaining operators will be applied. Having the mutators being implemented its runs a test suite on the compiled APK (Android Package) for each mutation and for the original code as well. A comparison of the obtained mutations results and the original results is made, returning a mutation score. [11]

2.1.1.2 MDroid+

Mdroid+ is a mutation tool developed specifically to be only used on an Android platform. As such this tool, according to its documentation, has a total of 38 mutators, all of them specific for android, working either on the AST (Abstract Syntax Tree), for the Java language files, and on text files, for the layout and properties XML files.

This tool, has relatively good mutator documentation, being only a few mutators that are not completely explicit, for instance, the mutators where code is replaced by new one. We don't know how they chose the new code, just that on some, but not all, its randomly generated in a non explicit way [10].

Also, it allows the user to select the mutators to be used, as this tool can only be run on a CLI (Command Line Interface), one would assume that the configuration could be made through execution flags, which is not the case. This tool, to be configured needs the user to manipulate the properties file, which might not be the most friendly way to be made, due to the learning curve needed to know the syntax of said properties files [13].

Furthermore, even though it has future work described on its open-source repository, the last update is more than a year old, leading us to believe that this tool is not being maintained anymore [24].

2.1.1.3 MuDroid

MuDroid is a mutation tool developed in 2015 as a bachelors final project [32], to work on the Dalvik virtual machine bytecode, the android specific Java Virtual Machine (JVM). This allows MuDroid to have the BlackBox testing advantages, only needing the final Android Package(APK) of the software being tested, But it also brings the disadvantages of harder to code mutators, even though in their approach, a parse of the bytecode to more readable code is made using Smali, a Dalvik bytecode assembler/disassembler.

Analyzing its mutants, when looking for android specific mutators, we came out unsuccessful as the mutations are not android specific as claimed. The described mutants only mutate the generic Java Operators (for instance replacing the '==' to '!=', or '+' to '-'), perform constants replacement and change the return output of the functions to 0 or *null*. All of which can be language agnostic.

This tool also has very specific requirements to be ran, the test suite must be a device/emulator with a 1920*1200 resolution. Also it is recommended to use just a certain device, the 2013 Asus Google Nexus 7 tablet. Furthermore, at the time of writing, this tool is not being maintained anymore, having its repository not being updated in more than 4 years [27].

2.1.2 Java Language Specific

2.1.2.1 JavaLanche

JavaLanche is a 2009 Java mutation tool that has, as its main purpose, to efficiently and automatically allow mutation testing on Java applications. [15] It tries to perform mutation testing efficiently by:

- Only using a small set of mutation operators, 4 to be exact, highly reducing the set mutants to be tested after without actually removing a lot of potential fault detection by using more embracing mutation operators [4]. This could still allow for a higher degree of coverage if the mutants would still perform a higher order mutation, covering therefore smaller mutation operators. However this is not the case, because as shown in its coverage score, described in the Table 2.2.3 in the Comparison 2.2, it has an equal performance as most.
- By working on the Java bytecode, therefore avoiding compiling every mutation, which is highly costly in terms of time and performance.

- By only testing the mutations that are covered by the tests, which coupled with the previous one can result in a significant reduction of time and processing power needs
- By the use of parallelism on the testing phase, which consists of every test being ran in a separate system task, allowing to perform an undetermined number of tests simultaneously being only limited by the number of processing threads on the system being used

JavaLanche is also an open source Java application, which could allow it to be platform independent. But according to its execution manual on its repository [20] it was made to be ran only on Unix based systems, as the procedure to perform its installation, per documentation, is running a Unix Shell script. The open source repository has also not been updated since March 2012 and has open issues from that epoch, making us believe it is not being maintained anymore.

2.1.2.2 Jumble

Jumble is a mutation tool developed by Reel Two, a New Zealand tech company. Firstly released in 2007 and updated until 2015 [21], it was developed with the intention to test the company projects unit tests, which at the time of first development were greatly increasing due to a methodology change. It works by following these steps:

1. Run the test cases on the original source code and record their time of execution, allowing therefore to later detect infinite loops and also to order the test cases.
2. Perform bytecode mutation, to prevent the time and resource costly compilations
3. Order the tests by using 3 heuristics, giving precedence to test cases that it is known to kill the mutants (Only works if it is not the first run), giving precedence to the test cases that killed a mutant on the same method due to the probability to kill another mutant is greater, and, finally by ordering all of them by the first execution time in an ascending order
4. Run the test cases on the mutated code and expect them to fail. These test cases are ran on a child JVM (Java Virtual Machine), so when a infinite loop is detected by taking too long to run or a heavy memory usage between tests is detected, the JVM can be safely killed and started again on for the next test.
5. Generation of a report, where the mutation score and other information is displayed.

Jumble also provides 7 types of mutators, which can be excluded from being ran on a certain method either using the parameter *-x* followed by the list of method to ignore or *–exclude=METHOD* when running it on the CLI (Command Line Interface). In case of using the plugin or using the tool as an external package, the addition of the annotation *@JumbleIgnore* to the methods, is the way to achieve the same functionality.

Also 5 of those 7 mutators are disabled by default, having the need to activate them using the appropriate command line parameters for each disabled mutator. It is not explicit whether this behavior replicates on the other possible interfaces or not [22].

This tool was also firstly developed to run from Java 1.3 to 1.6 with it now supporting till Java 8. Also it can be used in one of 3 ways being it, executing it on the command line, by executing it as an Eclipse IDE plugin or adding it as a dependency to the the Java project.

The company where it has been developed also achieved their goal, by implementing it on a Integration testing system. That system works, amongst other functionalities, by running the unit tests of the newly committed code every 15 minutes, a process that was already implemented in the integration system before. Overnight however things change, the remaining part of the tools process is ran, allowing therefore to a more efficient use of the system resources. It Is also documented that by using this process, extra motivation was given to the developers to implement better unit tests of their code due to having a mutation score [7].

2.1.2.3 Major

Major is Java specific mutation tool that was developed to be integrated inside the Java compiler, trying to do do mutation testing with the following changes to the compiler:

- Non-invasive implementation
- Configurable through a DSL(Domain specific language), which allows to also extend its functionality
- 3 mutation operator groups that can be selected through compiler options
- Efficient analysis due to its compiler embedded nature

During its development, due to its nature of changing another the compiler to run, some precautions were taken. The compiler must not change its original behaviour, changes must be externalized, and required changes to the application must be kept to a minimum to mitigate the before mentioned behavioural changes.

It works by analysing the AST (Abstract Syntax Tree) and encapsulating the mutants on the same basic block. It also collects additional information about the coverage of the test cases, so it can avoid testing mutants that have no way of being detected.

This tool only provides a CLI(Command Line Interface) which is integrated into the java compiler, it is used by adding an additional parameter to the compiler, more specifically *-XMutator*. It also allows to select which of the mutation operators are ran on the code, by adding their identifier to the execution parameter, for instance, *-XMutator:AOR,ROR*. It also supports configuration scripts, to configure existing mutants, written on its specific DSL [9].

2.1.2.4 MuJava

MuJava is a mutation tool that initially appeared in 2005, developed with the objective to implement object oriented mutants. At the time of writing, it had its last repository update on August 2016 [24].

It has a GUI (Graphical User Interface) that provides three tabs as following:

- Mutants Generator: A tab that allows the user to select the files to be mutated, and also, which operators to implement to be consequently ran. These operators are divided in behavioural and structural, reason being that they perform different kind of mutations, source code in case of behavioural and bytecode in case of structural.
- Mutants Viewer: A tab that allows the user to visualize the mutants that the tool generated on the Mutants Generator tab. As some of the mutants are generated on the bytecode some might not be completely visible.
- Test Executor: A tab that allows the user to select the tests to be executed and consequently do so. These tests must be functions that start with "*test*" and that return "*void*". Also these test are not ran automatically, the user must do so by hand.

This tool mutation process changes depending on the kind of mutator being applied because, as mentioned before, there are two types of mutation operators. For the behavioural mutants, as they are performed on the source code the process begins by selecting mutating only the selected files, after which they are compiled. For the structural mutants however, as they are bytecode based, first the compilation is performed and only after the mutants are applied.

On the test phase, although they differentiate the executor names, the process is the same for both. It begins by having the compiled mutant loaded on the executor, after which the classes are instantiated and ran against the user selected tests. Tests that, as mentioned before, must be purposely made for the tool as support testing frameworks such as JUnit is not mentioned [12].

2.1.2.5 Pit

Pit is a tool that started its open source development in 2010 [30], and is the only tool being analysed that is still being maintained at the time of writing [31]. It has the aim to perform mutation testing in a well integrated, fast and clear way.

The integration aim is being tackled by allowing the users to integrate the tool with their build tools such as Maven, And and Gradle. Allowing to integrate the tool with the IDE's with plugins, such as InteliJ and Eclipse. And also allowing to integrate the tool with code analysis tools such as SonarQube.

It tries to be fast by running on the bytecode to generate the mutants, reducing the compilation times to being just the time to compile the project once, instead of the number of mutations. It also tries to be faster by analysing which tests execute the mutants and only trying those, therefore reducing testing times by not executing the whole test suite. And lastly, it tries to run a subset of mutants at a time, reducing even more the executions times. A thing that makes it slower however, is running the tests on a separate JVM(Java Virtual Machine), but it is a must, so the tests are all ran in the same state, making the testing environment more sterile. At the time of writing the number of documented mutants is 29 [29].

And at last, the way that the tool tries to make it clear is by giving the user a clear web page with the report that can be navigated through allowing the user to see which tests were ran, which mutants were created and with a color code identifying which mutants were or not killed [3].

2.2 Comparison

To compare the tools a set of metrics should be defined. First and foremost there is a need to evaluate the quality of the documentation, afterwards, how up-to-date are they (Are they being maintained? Did they stop being maintained long ago or just recently?), and finally how do the tools perform amongst each other.

2.2.1 Documentation Quality

The tools that were analysed almost all of them have some sort of problem on the documentation, ranging from lack of documentation, making it harder to analyse, to not being available anymore, passing through conflicting versions. So in this documentation analysis the quality of the said documentation is going to be analysed. First a brief individual summary about the documentation of each tool, after which a comparison is going to be made on Table 2.1 with 4 metrics and a overall rating being the average of the 4 that range from 0 to 5, being 0 Non existent, 1 Very Bad, 2 Bad, 3 Acceptable, 4 Good and 5 Very Good. These 4 categories are:

- Availability: A metric to assess how easy to obtain the information is.
- Quantity: A metric to assess the quantity of different information about the tool is found, not to confuse with the number of sources.
- Quality: A metric to assess the overall quality of the sources, discarding the information itself as it is on another metric.
- Clarity: A metric to assess how easy it is to get the information is from the sources

So starting on the *Mobile Application Specific 2.1.1* tools, and keeping the order of 2.1, the initial documentation analysis is as follows:

1. Edroid: The only documentation found for Edroid is a paper [11] which by itself gives some information, but sometimes confusing, such as the question is the tool configurable, it is not clear either it is or not. In this case it starts by stating that a subset of the mutator operators

can be used, but it also states that it is not manually configurable. Also their claims about the tool, and inconsistencies like the one just mentioned, cannot be checked due to the lack of access to the tool.

2. MDroid+: MDroid+ is a tool that actually has a relatively good documentation, but not found right away, as if searching by the tool itself the documentation found is paper is an minor overview of the tool [13] and its repository [24]. Repository which, at the time of writing, has on the *README.md* file an indication that for citations of the tool an article with a more extensive review of the same should be used that has the required information [10].

3. MuDroid: MuDroid documentation is only a bachelors thesis [32] and the tools repository [27], which by themselves provide enough and decent information, but as it has no articles the documentation is not easy to come by, not only that but the said repository only has information regarding on how to execute it and its last update date, for more information on that source, a code analysis is required.

4. JavaLanche: JavaLanche is one of the cases where the only issue found is the lack of some information being more explicit, for instance on the regard of the mutation operators, there's only a minor mention on "*(Un-)Covering Equivalent Mutants*" [4] and some of them is not very explicit what they do, just that they do it. Excluding that, JavaLanche documentation is good, it has its own website [28], a open source code repository [20] and two papers, where one has an overview on how the tool works [15] and one where a more extensive overview of the same. [4]

5. Jumble: Jumble is a case where most of the information came, not from a scientific article [7], but from its own website [22] and repository [28]. This is due, to we believe, this tool being enterprise made, and therefore most of the documentation is available through themselves. However the documentation being provided by themselves does not guarantee the best documentation, this due to the sometimes, lack of clarification, for instance, this tool has 5 of its 7 mutators disabled by default, indicating configuration, but this was only found by chance on the website as the tool interface is not properly documented.

6. Major: Major's documentation is good, it has 2 papers [9] [8] and a website [23] where the tool can be downloaded. However its documentation is sometimes confusing, for instance, on one of the papers [9], where the configuration was mentioned, it explicitly states that there were only 3 parameters, but on the command example the 3 parameters were shown

followed by a "**...**" indicating that more configurations are allowed.

7. MuJava: MuJava, the oldest tool being analysed, when looking for its documentation on the conventional search tools such as *Scopus* some documents are initially shown, but the original host for said documents does not list them as available. This made us look for another ways to obtain them. After a new search, this time more broad, the tool repository [26] is found and with it a link to the original tool website [25]. Website which has the documents that were missing on the source. These documents are also not that great, for instance, on one of them [12] when referring to the only existing two tables, the same sentence refers not to the tables but to a third nonexistent one. Also some non explicit parts exists, such as stating that it provides an API to easily change the behaviour of a program during the execution, but it doesn't refer if its during the execution of the tool or the application being tested.

8. Pit: Pit is a tool that provides an article but not in the sense of providing the full details of the tool but more a demo of said tool [3], as most of the information is on the tools website [29] or the tools repository [31]. Also there are some minor problems on the documentation such as typos, for instance, '*asses*', instead of '*assess*' we believe, due to the context. And some information that wasn't properly explicit, such how does some operators work, for instance at the time of writing, the mutant operator "*Arithmetic Operator Deletion Mutator (AOD)*" is documented on the website that it replaces an arithmetic expression with one of its members, and they give an example that shows with two variables. But what happens with more than two variables still remains, will it delete just one? Or will it choose just one? And if it just deletes one, will it generate a mutant for the remaining variables?

Table 2.1: Documentation comparison

Tool	Availability	Quantity	Quality	Clarity	Overall Rating
Edroid	4	3	3	3	3.25
MDroid+	3	4	4	4	3.75
MuDroid	3	4	4	4	3.75
JavaLanche	4	4	4	3	3.75
Jumble	4	4	3	3	3.5
Major	4	4	3	2	3.25
MuJava	2	5	2	3	3
Pit	5	4	3	4	4

So as seen in the individual description and in the Table 2.1, all of the tools stand between the average and good documentation, with most having only an acceptable rating.

2.2.2 Maintainability

As already seen when the tools were initially analyzed in 2.1 most of them are not being maintained for more than one year, making their functionality more out of date as time passes. To compare the tools Table 2.2 was made for a better comparison, with the initial year of known development, and last year of known updates. For a final comparison a final rating depending on the last update was also made, where 5 points represents if the tool is being maintained (considered if last update was in 2019 or 2020), 4 points if it just isn't maintained for more than 2 years (2018), 3 points if for no more than 5 years (2015-2018), 2 points no more than 10 years (2010-2015), 1 point no more than 15 years (2005-2010), and 0 points for more than 15 years (2004 or older).

Table 2.2: Maintainability comparison

Tool	Initial Year	Last Update	Overall Rating
Edroid	2018	2018	4
MDroid+	2017	2018	4
MuDroid	2015	2016	3
JavaLanche	2009	2012	2
Jumble	2007	2015	3
Major	2011	2018	4
MuJava	2005	2016	3
Pit	2010	2020	5

2.2.3 Performance

Performance amongst tools can be measured in several ways, but for our case study we are going to restrain to measure their coverage of the mutation operators, their efficiency and at last their configurability.

For the coverage we are going to guide ourselves with the mutation operators defined on the *Taxonomy of Android Bugs* shown in Figure 2.1 of "Enabling mutation testing for android apps" from M. Linares-Vásquez [10], the values will also be obtained by analysing the same paper, as, at the time of writing, this is the most complete paper on the Android Mutation subject and one that considered most of the tools being analysed.

For efficiency we considered the methods used for the generation of the mutants and how the tests are executed are the the parameters taken into account as those two are the thing that impact efficiency the most.

At last, for a comparison on how configurable a tool is is. This is also taken in account here, because although it may not directly impact the performance, it may impact how the user learns to operate the tool and therefore if the the user can operate it efficiently.

The Table 2.3 is where the performance comparison is being made. For the *Overall Rating* a sum of the ratings was made, being *Android Bugs Coverage* attributed the value of the percentage divided by 100, for the *Byte-Code* 1 and due to this method having a better performance impact over the *AST* this latter on was attributed a value of 0.5, *Selection Through Coverage* has a value of 1 in case of Yes, as does have *Parallel Execution*. For the *Configurability* a value of $\frac{1}{3}$ is attributed in case of no interface for the configuration, a value of $\frac{2}{3}$ in case of a CLI (Command-Line Interface) and a value of 1 in case of a GUI (Graphical User Interface). Overall a tool can have a maximum overall rating of 5.

2.2.4 Final Comparison

For the final comparison an average of the results obtained in the analysis of the Documentation 2.2.1, Maintainability 2.2.2 and Performance 2.2.3 is made. The values represent as mentioned on 2.2.1 from 1 to 5 being 1 Very Bad, 2 Bad, 3 Acceptable, 4 Good and 5 Very Good.

As seen in Table 2.4 according to our metrics the existing tools are mostly on the Bad! Only three tools have a score greater than 3, being only Pit that has a rating good enough to qualify to be a Good tool. This is highly due to the Performance Rating given to the tools, because the tools with a score lower than 3 all had a Performance Rating lower than 2, and vice versa.

Table 2.4: Final Comparison and Rating

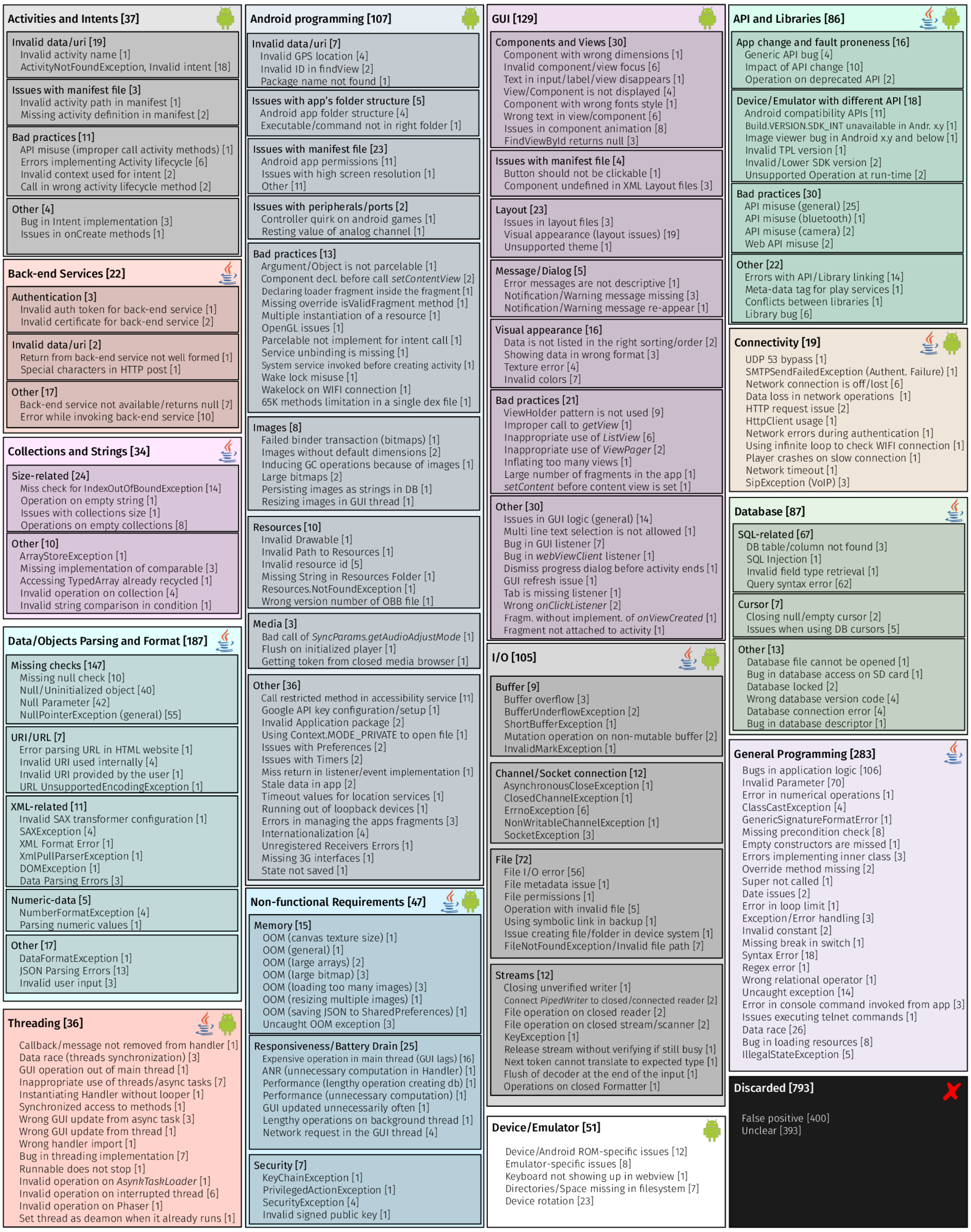
Tool	Documentation Rating	Maintainability Rating	Performance Rating	Final Rating
Edroid	3.25	4	0	2.42
MDroid+	3.75	4	1.21	2.99
MuDroid	3.75	3	1.75	2.83
JavaLanche	3.75	2	3.77	3.17
Jumble	3.5	3	1	2.5
Major	3.25	4	2.3	3.18
MuJava	3	3	1.09	2.36
Pit	4	5	4.13	4.38

2.3 Summary

Mobile mutation tools are in a need for a refresh, no Mobile Specific tool gets to the Acceptable rating, being only MDroid+ close to achieving it, this may reflect on the quality of the tests of the applications and the applications themselves as there's no way to ensure that an application is being properly tested.

One of the highest impact on the Final Rating on Table 2.4 is the Performance Rating, having most of the tools with a very bad or bad results. As seen on Table 2.3, one of the most impacting metrics for this result the Android Bugs Coverage, this is an aspect that requires a change for a better future result.

When analyzing the Android Bugs Coverage, according to the source of the values [10], the union of the tools that they analysed reached a overall coverage of 50%. This although only being half of the identified bugs, is still significantly better than the 38% obtained by the highest ranking



Activities and Intents [37]
- Invalid data/uri [19]
  - Invalid activity name [1]
  - ActivityNotFoundException, Invalid intent [18]
- Issues with manifest file [3]
  - Invalid activity path in manifest [1]
  - Missing activity definition in manifest [2]
- Bad practices [11]
  - API misuse (improper call activity methods) [1]
  - Errors implementing Activity lifecycle [6]
  - Invalid context used for intent [2]
  - Call in wrong activity lifecycle method [2]
- Other [4]
  - Bug in Intent implementation [3]
  - Issues in onCreate methods [1]

Back-end Services [22]
- Authentication [3]
  - Invalid auth token for back-end service [1]
  - Invalid certificate for back-end service [2]
- Invalid data/uri [2]
  - Return from back-end service not well formed [1]
  - Special characters in HTTP post [1]
- Other [17]
  - Back-end service not available/returns null [7]
  - Error while invoking back-end service [10]

Collections and Strings [34]
- Size-related [24]
  - Miss check for IndexOutOfBoundException [14]
  - Operation on empty string [1]
  - Issues with collections size [1]
  - Operations on empty collections [8]
- Other [10]
  - ArrayStoreException [1]
  - Missing implementation of comparable [3]
  - Accessing TypedArray already recycled [1]
  - Invalid operation on collection [4]
  - Invalid string comparison in condition [1]

Data/Objects Parsing and Format [187]
- Missing checks [147]
  - Missing null check [10]
  - Null/Uninitialized object [40]
  - Null Parameter [42]
  - NullPointerException (general) [55]
- URI/URL [7]
  - Error parsing URL in HTML website [1]
  - Invalid URI used internally [4]
  - Invalid URI provided by the user [1]
  - URL UnsupportedEncodingException [1]
- XML-related [11]
  - Invalid SAX transformer configuration [1]
  - SAXException [4]
  - XML Format Error [1]
  - XmlPullParserException [1]
  - DOMException [1]
  - Data Parsing Errors [3]
- Numeric-data [5]
  - NumberFormatException [4]
  - Parsing numeric values [1]
- Other [17]
  - DataFormatException [1]
  - JSON Parsing Errors [13]
  - Invalid user input [3]

Threading [36]
- Callback/message not removed from handler [1]
- Data race (threads synchronization) [3]
- GUI operation out of main thread [1]
- Inappropriate use of threads/async tasks [7]
- Instantiating Handler without looper [1]
- Synchronized access to methods [1]
- Wrong GUI update from async task [3]
- Wrong GUI update from thread [1]
- Wrong handler import [1]
- Bug in threading implementation [7]
- Runnable does not stop [1]
- Invalid operation on *AsynkTaskLoader* [1]
- Invalid operation on interrupted thread [6]
- Invalid operation on Phaser [1]
- Set thread as deamon when it already runs [1]

Android programming [107]
- Invalid data/uri [7]
  - Invalid GPS location [4]
  - Invalid ID in findView [2]
  - Package name not found [1]
- Issues with app's folder structure [5]
  - Android app folder structure [4]
  - Executable/command not in right folder [1]
- Issues with manifest file [23]
  - Android app permissions [11]
  - Issues with high screen resolution [1]
  - Other [11]
- Issues with peripherals/ports [2]
  - Controller quirk on android games [1]
  - Resting value of analog channel [1]
- Bad practices [13]
  - Argument/Object is not parcelable [1]
  - Component decl. before call *setContentView* [2]
  - Declaring loader fragment inside the fragment [1]
  - Missing override isValidFragment method [1]
  - Multiple instantiation of a resource [1]
  - OpenGL issues [1]
  - Parcelable not implement for intent call [1]
  - Service unbinding is missing [1]
  - System service invoked before creating activity [1]
  - Wake lock misuse [1]
  - Wakelock on WIFI connection [1]
  - 65K methods limitation in a single dex file [1]
- Images [8]
  - Failed binder transaction (bitmaps) [1]
  - Images without default dimensions [2]
  - Inducing GC operations because of images [1]
  - Large bitmaps [2]
  - Persisting images as strings in DB [1]
  - Resizing images in GUI thread [1]
- Resources [10]
  - Invalid Drawable [1]
  - Invalid Path to Resources [1]
  - Invalid resource id [5]
  - Missing String in Resources Folder [1]
  - Resources.NotFoundException [1]
  - Wrong version number of OBB file [1]
- Media [3]
  - Bad call of *SyncParams.getAudioAdjustMode* [1]
  - Flush on initialized player [1]
  - Getting token from closed media browser [1]
- Other [36]
  - Call restricted method in accessibility service [11]
  - Google API key configuration/setup [1]
  - Invalid Application package [2]
  - Using Context.MODE_PRIVATE to open file [1]
  - Issues with Preferences [2]
  - Issues with Timers [2]
  - Miss return in listener/event implementation [1]
  - Stale data in app [2]
  - Timeout values for location services [1]
  - Running out of loopback devices [1]
  - Errors in managing the apps fragments [3]
  - Internationalization [4]
  - Unregistered Receivers Errors [1]
  - Missing 3G interfaces [1]
  - State not saved [1]

Non-functional Requirements [47]
- Memory [15]
  - OOM (canvas texture size) [1]
  - OOM (general) [1]
  - OOM (large arrays) [2]
  - OOM (large bitmap) [3]
  - OOM (loading too many images) [3]
  - OOM (resizing multiple images) [1]
  - OOM (saving JSON to SharedPreferences) [1]
  - Uncaught OOM exception [3]
- Responsiveness/Battery Drain [25]
  - Expensive operation in main thread (GUI lags) [16]
  - ANR (unnecessary computation in Handler) [1]
  - Performance (lengthy operation creating db) [1]
  - Performance (unnecessary computation) [1]
  - GUI updated unnecessarily often [1]
  - Lengthy operations on background thread [1]
  - Network request in the GUI thread [4]
- Security [7]
  - KeyChainException [1]
  - PrivilegedActionException [1]
  - SecurityException [4]
  - Invalid signed public key [1]

GUI [129]
- Components and Views [30]
  - Component with wrong dimensions [1]
  - Invalid component/view focus [6]
  - Text in input/label/view disappears [1]
  - View/Component is not displayed [4]
  - Component with wrong fonts style [1]
  - Wrong text in view/component [6]
  - Issues in component animation [8]
  - FindViewById returns null [3]
- Issues with manifest file [4]
  - Button should not be clickable [1]
  - Component undefined in XML Layout files [3]
- Layout [23]
  - Issues in layout files [3]
  - Visual appearance (layout issues) [19]
  - Unsupported theme [1]
- Message/Dialog [5]
  - Error messages are not descriptive [1]
  - Notification/Warning message missing [3]
  - Notification/Warning message re-appear [1]
- Visual appearance [16]
  - Data is not listed in the right sorting/order [2]
  - Showing data in wrong format [3]
  - Texture error [4]
  - Invalid colors [7]
- Bad practices [21]
  - ViewHolder pattern is not used [9]
  - Improper call to *getView* [1]
  - Inappropriate use of *ListView* [6]
  - Inappropriate use of *ViewPager* [2]
  - Inflating too many views [1]
  - Large number of fragments in the app [1]
  - *setContent* before content view is set [1]
- Other [30]
  - Issues in GUI logic (general) [14]
  - Multi line text selection is not allowed [1]
  - Bug in GUI listener [7]
  - Bug in *webViewClient* listener [1]
  - Dismiss progress dialog before activity ends [1]
  - GUI refresh issue [1]
  - Tab is missing listener [1]
  - Wrong *onClickListener* [2]
  - Fragm. without implement. of *onViewCreated* [1]
  - Fragment not attached to activity [1]

I/O [105]
- Buffer [9]
  - Buffer overflow [3]
  - BufferUnderflowException [2]
  - ShortBufferException [1]
  - Mutation operation on non-mutable buffer [2]
  - InvalidMarkException [1]
- Channel/Socket connection [12]
  - AsynchronousCloseException [1]
  - ClosedChannelException [1]
  - ErrnoException [6]
  - NonWritableChannelException [1]
  - SocketException [3]
- File [72]
  - File I/O error [56]
  - File metadata issue [1]
  - File permissions [1]
  - Operation with invalid file [5]
  - Using symbolic link in backup [1]
  - Issue creating file/folder in device system [1]
  - FileNotFoundException/Invalid file path [7]
- Streams [12]
  - Closing unverified writer [1]
  - Connect *PipedWriter* to closed/connected reader [2]
  - File operation on closed reader [2]
  - File operation on closed stream/scanner [2]
  - KeyException [1]
  - Release stream without verifying if still busy [1]
  - Next token cannot translate to expected type [1]
  - Flush of decoder at the end of the input [1]
  - Operations on closed Formatter [1]

Device/Emulator [51]
- Device/Android ROM-specific issues [12]
- Emulator-specific issues [8]
- Keyboard not showing up in webview [1]
- Directories/Space missing in filesystem [7]
- Device rotation [23]

API and Libraries [86]
- App change and fault proneness [16]
  - Generic API bug [4]
  - Impact of API change [10]
  - Operation on deprecated API [2]
- Device/Emulator with different API [18]
  - Android compatibility APIs [11]
  - Build.VERSION.SDK_INT unavailable in Andr. x.y [1]
  - Image viewer bug in Android x.y and below [1]
  - Invalid TPL version [1]
  - Invalid/Lower SDK version [2]
  - Unsupported Operation at run-time [2]
- Bad practices [30]
  - API misuse (general) [25]
  - API misuse (bluetooth) [1]
  - API misuse (camera) [2]
  - Web API misuse [2]
- Other [22]
  - Errors with API/Library linking [14]
  - Meta-data tag for play services [1]
  - Conflicts between libraries [1]
  - Library bug [6]

Connectivity [19]
- UDP 53 bypass [1]
- SMTPSendFailedException (Authent. Failure) [1]
- Network connection is off/lost [6]
- Data loss in network operations [1]
- HTTP request issue [2]
- HttpClient usage [1]
- Network errors during authentication [1]
- Using infinite loop to check WIFI connection [1]
- Player crashes on slow connection [1]
- Network timeout [1]
- SipException (VoIP) [3]

Database [87]
- SQL-related [67]
  - DB table/column not found [3]
  - SQL Injection [1]
  - Invalid field type retrieval [1]
  - Query syntax error [62]
- Cursor [7]
  - Closing null/empty cursor [2]
  - Issues when using DB cursors [5]
- Other [13]
  - Database file cannot be opened [1]
  - Bug in database access on SD card [1]
  - Database locked [2]
  - Wrong database version code [4]
  - Database connection error [4]
  - Bug in database descriptor [1]

General Programming [283]
- Bugs in application logic [106]
- Invalid Parameter [70]
- Error in numerical operations [1]
- ClassCastException [4]
- GenericSignatureFormatError [1]
- Missing precondition check [8]
- Empty constructors are missed [1]
- Errors implementing inner class [3]
- Override method missing [2]
- Super not called [1]
- Date issues [2]
- Error in loop limit [1]
- Exception/Error handling [3]
- Invalid constant [2]
- Missing break in switch [1]
- Syntax Error [18]
- Regex error [1]
- Wrong relational operator [1]
- Uncaught exception [14]
- Error in console command invoked from app [3]
- Issues executing telnet commands [1]
- Data race [26]
- Bug in loading resources [8]
- IllegalStateException [5]

Discarded [793]
- False positive [400]
- Unclear [393]

Figure 2.1: Taxonomy of Android Bugs [10]

Table 2.3: Performance Comparison

Tool	Android Bugs Coverage [1]	Efficiency				Configurability	Overall Rating
		Byte-Code	AST	Coverage Selection	Parallel Execution		
Edroid	Unknown	No	No	No	Unknown	Unknown	0 [2]
MDroid+	38%	No	Yes	Does not test		Yes - Conf. File	1.21
MuDroid	8%	Yes	No	No	No	Yes - CLI	1.75
JavaLanche	10%	Yes	No	Yes	Yes	Yes - CLI	3.77
Jumble	Unknown	Yes	No	No	No	No	1
Major	13%	No	Yes	Yes	No	Yes - CLI	2.3
MuJava	9%	No	No	No	No	Yes - GUI	1.09
Pit	13%	Yes	No	Yes	Yes	Yes - GUI	4.13

[1] The values obtained in "Android Bug Coverage" were obtained from the analysis on "Enabling mutation testing for android apps" [10]

[2] This result is due to the lack of documentation on this regard and the availability of the tool. Therefore a lack of possibility to analyse its performance

tool, which does not perform the hole process. If there's a need for a tool that ensures the hole Mutation Testing process there's a need to restrain to a tool that only has a coverage of 13%, being this not achieved by any Mobile Specific tool.

Analysing the best scoring tool, PIT, if this tool had the Android Bugs Coverage that joins all the tools, a 50% score, this tool would make its final 4.13, a Good performance analysis, rise to a 4.5, and Very Good analysis. This score would then also make the also already Good final score of 4.38 rise to 4.5, making this a very good tool to analyse Mobile Applications. This also makes us believe that a need to develop a tool similar to PIT, that also tackles its identified deficiencies, is necessary to ensure that Mobile Applications are indeed being properly tested

Chapter 3

Framework

As identified on the performance comparison table 2.3 most tools lack on both the efficiency and configurability metrics. For the configurability this is mostly due to the the interface of the analysed tools being either poorly designed, hard to use, or simply non existent.

For this we tried to implement a way that not only could be easy to use, but would also allow the end user to use their already existing knowledge on programming, and, with a very small learning curve, learn how to use our interface.

3.1 Background

Before trying to design our interface however, we needed to better understand what is required to perform the mutation namely which are processes currently being used and if any tools exist that could help us achieve our goal.

3.1.1 Mutation process

To perform mutation testing a key part is generating the mutations, for this there are two main processes that could be used, an AST (Abstract Syntax Tree) based process or Byte-Code based process.

The AST process works by analysing the source code, and, with every entry of code creating a node on a tree. This would in sum create what could be considered a kind of decision tree. An example of this is the image 3.1 that shows an example of an AST for a specific piece of code.

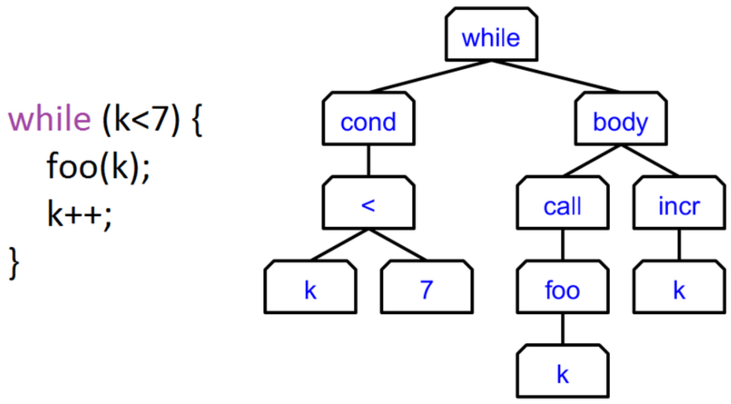


Figure 3.1: Example of an AST

With a then created AST, we can then apply the mutants by changing the nodes on the AST, which, when saving the mutated files based on said AST will create the mutated source code.

The other possible process is using the already compiled source code, or Byte-Code. This process has one major advantage when compared with the previous process, not needing to compile the mutated code, as it works directly on the compiled code. However this approach brings to major setbacks. The possibility of, on a new version of the compiler, the bytecode structure changes, requiring grater maintenance to support in those new versions. And, for us the worst setback, that due to its nature of using the compiled code it is extremely harder to design mutants for the compiled code, as it would add the necessity to learn how to interpret the compiled code.

With these setbacks posted by the ByteCode, the decision to use the AST process was made, as mentioned, on a way to achieve the *Configurability* we wanted, without risking too much the of the *Efficiency* due to our proposed process that is described on chapter 4

The way we achieve the proposed *Configurability* is using the *Kadabra* tool [17] and its *JavaScript* based framework *Lara* [18]. The decision to adopt this tool was made in the early stages of the design phase, when we were deciding on this subject and found out about it, and that the tool was being developed on the same place, FEUP. This allowed us to have a better contact with the developers, therefore not only allowing us to achieve our goal, but also to help them improve their tool. This tool, as is going to be mentioned in more detail on 5, allows us not only to have our own mutants on the tool GUI but also allows the users to implement an already developed interface.

This interface is the base of the configuration of a new mutant, that contains the basic mutation functions to complete the process, identifying the mutation points, performing the mutation and restoring it to the previous state. This method allows us to abstract most of the *Lara* framework additions to the *JavaScript* language, and, with this, the user only requirements are to have some knowledge of *JavaScript* language and to learn how to use only one of the Kadabra libraries, *LaraActions*, and have a common knowledge of how *Abstract Sintax Trees* work.

With this configurability, we achieve our goal of a more configurable tool. Allowing not only to use it without own pre-made mutants, but allowing also the user to use their own mutants, that can be used both on GUI and CLI.

3.1.2 *Lara* and the *Kadabra* tool

As mentioned on the previous section 3.1.1, as a base for our AST process we used the *Kadabra* tool [17] and its *Lara* framework [18]. This tool, maintained by *Special-Purpose Computing Systems (SPeCS)* from FEUP, was created as a Java-to-Java compilation tool for code instrumentation and transformations controlled by the LARA Framework.

As such, not only it allowed us to tackle the issues we identified as mentioned, but it also allowed us to have a close contact with the developers of the tool. This close contact allowed us not only to have a closer integration of the tool, but, helped to provide improvements to the *Kadabra* tool itself, regarding its mutation usage.

The *Lara* framework, the base of the *Kadabra* tool, is a language that, as previously mentioned, is based on the *JavaScript* language. It was developed as a domain-specific, aspect-oriented language with specially defined keywords for navigation on the AST.

The language also defines every element of the AST as a *JoinPoint*, where this *JoinPoints* try to include all the possible types of elements of the AST. This *JoinPoints* are the elements to be identified and where the mutation occurs by replacing, removing or inserting before and/or after.

As mentioned, all these operations are required for the mutations and most of the required learning curve to operate this tools.

A trial for the tool and its languages can be found on `http://specs.fe.up.pt/tools/kadabra/` where an example for mutation of a binary operator *"<"* to *">"* can be found as well as links for.

The *Kadabra* tool as mentioned is a Java-to-Java compilation tool, but it also has another similar tools for another languages such as C/C++ (*Clava*), JavaScript(*JackDaw*) and MatLab(*Matisse*). This family of tools gives another advantage for this approach, that all the language agnostic mutation operators, such as binary operator mutators, can be re-used using these tools, therefore allowing the development of a mutation tool with broader programming language targets.

3.2 The mutation interface

As mentioned, to achieve the desired configurability of the mutation process, an interface was developed for the *Lara* framework, so that most of the mutation process could be abstracted from the tool user. This abstraction on interface works by requiring the end user to develop the bare minimum of tasks for a mutant to work, an way to identify the nodes in AST to mutate, and what does the mutation changed the desired nodes to.

On figure 3.2 we can see the UML diagram of the public interface architecture. This architecture allows to develop two kinds of mutation operators, both of which work similarly. We've got the IterativeMutator that tries to abstract the classic mutation process, and then a CumulativeMutator, that tries to abstract the interface for our new proposed process. Both processes are described in more detail on chapter 4

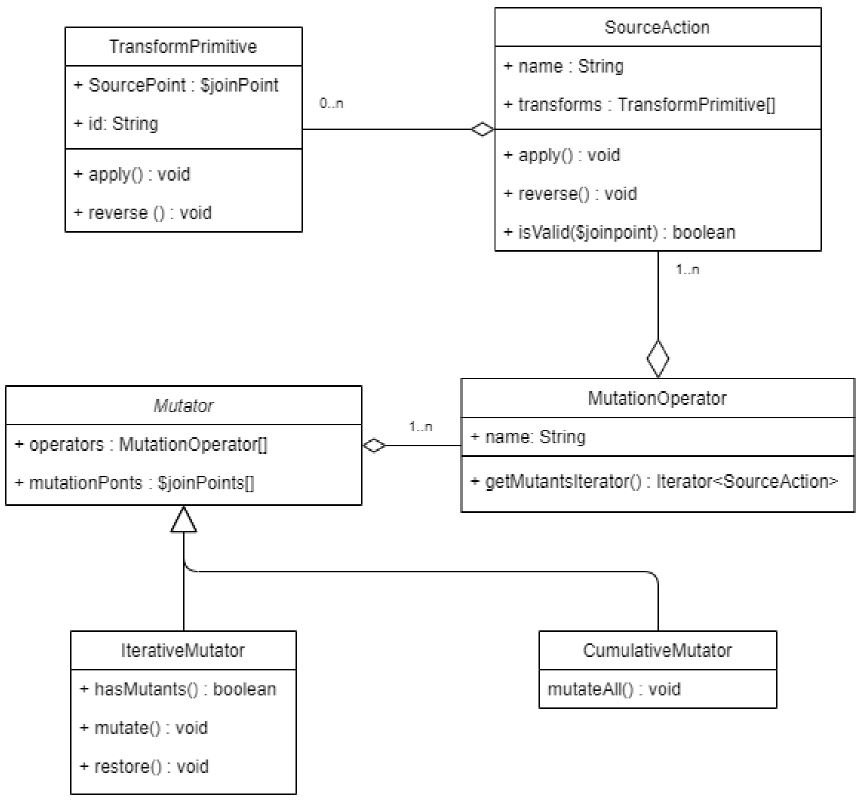


Figure 3.2: Interface Architecture

Mutation operators that use any of the interfaces then require to implement just two *JavaScript* functions, *isMutationPoint($joinpoint)* and *_mutatePrivate($joinpoint)*. The first function is required to identify if the AST node, or joinpoint as identified by *LARA*, is a possible mutation point. These are then saved internally on the interface to be used when using the second function. This second function is where the mutation takes in fact place, as its how the user tells the tool how do they want to perform said mutation.

On image 3.3 we can see an example of both required functions developed for a Binary Mutator, this mutator has as its main purpose to mutate binary operators with other operators on the same class. For instance, replacing a "$+$" with other operators such as "$-$", "$/$", "$*$" or "$\%$".

To implement the mutation, knowladge of only two *Lara* Framework API's are required to implement a mutation. They are the LaraActions and KadabraNodes API's. These two API's are developed to work as a *JavaScript* library, allowing therefore to have only a small learning curve in case the end user desires to develop his own mutation operators.

```
BinaryMutator.prototype.isMutationPoint = function($jp) {
    // Check if binary expression
    if(!$jp.instanceOf("binaryExpression")) {
        return false;
    }

    if(this.originalBinaryOperator === undefined)
        // Check if it not the same operator
        if($jp.operator === this.binaryOperator) {
            return false;
        }
    } else {
        // Check if it is the target operator
        if($jp.operator != this.originalBinaryOperator) {
            return false;
        }
    }

    return true;

}

BinaryMutator.prototype._mutatePrivate = function($jp) {

    // Create a BinaryExpression node
    var $newOp = KadabraNodes.binaryExpression(this.binaryOperator, $jp.lhs.copy(), $jp.rhs.copy());

    // Replace given node
    var replaceAction = LaraActions.replaceJp($jp, $newOp);


    return replaceAction.getPoint();
}
```

Figure 3.3: Cumulative Mutator implementation example

3.3 Conclusion

After developing this interface we expect the end users, which will be mainly Developers and Software Testers, to be able to develop their own mutation operators to be used in their pipelines without a big learning curve, as it will mostly be similar to already known technologies. This will then help us achieve the Configurability we wanted for our tool without compromises.

Chapter 4

Proposed Process

As mentioned in the previous chapter, we want to tackle not only the Configurability issues but also its Efficiency. This has been tried several times, one example can be the testing process changing from an AST based approach to the faster, but less configurable, ByteCode process.

Another example of this can be trying to reduce the number of mutation operators, that has been the most recent focus. This can be seen by L. Sousa et al. [16], where, in 2019, a reduction of the number of mutants is tried by reducing the number of semantically similar mutants, or by M. Guimarães et. al. [6], where, in 2020, the reduction of mutations is tried by removing redundant versions.

However, no example that we can find tried to tackle the part of the process that takes longer to execute, the compilation. To tackle this a new process was required, an architecture that takes into account the compilation and run times and tries to reduce them exponentially, which we propose in this chapter.

4.1 Classic Process

Before explaining our proposed process we first need to explain the Classic Process, which is the process that is commonly used in all mutation tools. It was created with the purpose to create a new version of the code for each mutation, and for that, it follows the following steps:

1° Search the source code for the next mutation point (First if none were found yet)

2° Change the source code on the identified mutation point, creating the mutated code

3° Compile the mutated code

4° Execute the mutated code

5° Save the results

6° Repeat the process

This process can take a lot of time, specially the compilation phase. This is due to a need of compiling a new version of the code every time a new mutant is created, as every copy only has a mutation.

Performance improvements however were already tried with on this process, either by running every step as Phases, executing only the next one when all the mutants went the current one, or running every mutant in parallel so it takes less time.

All of these improvements however, do not tackle the most time expensive part of the process, the compilation time. This, for larger and more complex programs, can rise exponentially as more mutants, and therefore more compilations, are required.

There is also a process that does not require the compilation phase altogether as it applies the mutations directly on the compiled code. But for this, two issues rise. First, that a knowledge of how the code is compiled is required and as the code is already compiled is harder to change. But, the worst problem for this approach, is how the mutants are found and applied might require a complete refactoring process, for every language version update, that may or may not occur regularly.

4.2 Proposed Process

On our new proposed process we then try to tackle the lack of improvements on the compilation times of the mutants without compromising on the rest by reducing the need to perform one compilation per mutant to needing to compile only once in the whole process. With this change. we propose a change of the process to follow the following steps:

1° Search the source code for all the mutation points

2° Apply the mutations to the identified mutation points, on a single copy of the source code.

3° Compile the mutated code

4° Execute the mutated code, once for every mutation

This new process allows us to not only reduce time in the compilation phase, by not having the need to compile the program as many times as the number of generated mutants, but also theoretically reduce the mutant generation times, as only one new version of the files needs to be created

Also, depending on the testing framework, it can also have a positive impact on the testing phase, as a new instantiation of the project dependencies is not required for another mutant execution.

4.2.1 Mutant generation and identification

As stated, all the mutants are on the same source code. To achieve this, we took the approach of enclosing each of the mutations on an *If Else* statement, with an *Mutation Unique Identifier (MUID)* for each mutant.

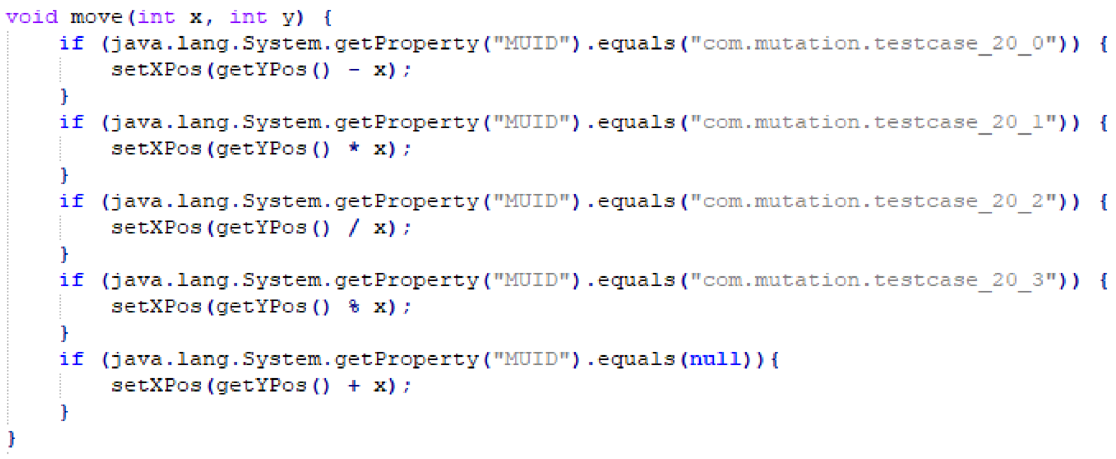
This MUID will be used an execution parameter allowing us to execute the mutated version only once for each mutation, and therefore, having the same advantages of running once for each version without the impact of the costly multiple compilation for each mutant.

A proposed way to generate the MUID, and the way we generate one on our tool, is to use the name fully qualified class name followed by the line number for where the mutant is located on the file and the number of mutations till that point.

For instance, the MUID for the 7th mutation of the tool, on the 20th line of the file *Main.java*, located on the package *com.mutation.testcase* would look like:

com.mutation.testcase.Main_20_7

This way we can guaranty the the MUID is unique, which is a requirement if we want to execute only one mutant at a time. One code example of how this can be applied is shown on Figure 4.1, that represents an code example of this process with 4 mutants along with the original code. Note the in the last if clause, that the it compares the *MUID* with *null*, this is the encapsulation of the original mutation point, required so the application retains its original functionality, and, for when testing, to only execute the mutants.



```
void move(int x, int y) {
    if (java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_0")) {
        setXPos(getYPos() - x);
    }
    if (java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_1")) {
        setXPos(getYPos() * x);
    }
    if (java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_2")) {
        setXPos(getYPos() / x);
    }
    if (java.lang.System.getProperty("MUID").equals("com.mutation.testcase_20_3")) {
        setXPos(getYPos() % x);
    }
    if (java.lang.System.getProperty("MUID").equals(null)){
        setXPos(getYPos() + x);
    }
}
```

Figure 4.1: Proposed Process Code Example

4.2.2 Incompatible Mutants

At the time of writing, the only disadvantage found for this approach, is the impossibility of certain mutants to work as intended. This is due to our approach, of using an *If* clause to allow the enclosure of all the mutants on only one source code.

For example, it would not be capable of working with mutants that require to be inserted in the exact beginning of a function, or mutants that change the signature of functions.

For this disadvantage however, we also have a solution, that instead of using the original *class* and inject the mutant inside it, we would copy the entire object and do the required changes as in the classic way. Repeating this process for each mutation point.

This slight change on the approach, for the these mutants, would allow us to whenever the original object was used, to do the same *If* enclosure and use the mutated object instead. One example of this can be found on the following example of a mutator that changes function parameters from *boolean* to *int*, in this case a constructor.

```
//***********************************
//Original Code
//***********************************

public class TestClass{
    TestClass(boolean test){ //Mutation Point
        //Do Something...
    }
}

public class MainClass{}
    public static void main(String[] args) {
        TestClass testClass = new TestClass(true); //Where IfClause is
            placed
        //Do something...
    }
}

//***********************************
//Mutated Code Using Proposed Process
//***********************************

public class MutatedClass extends TestClass{ //Extends main class so no
    further changes are required
    MutatedClass(int i){
        //Do Something...
    }
}

public class TestClass{
    TestClass(boolean test){
        //Do Something...
    }
}

public class TestClass{ //Original class remains untouched
    TestClass(boolean test){ //Mutation Point
        //Do Something...
    }
}

public class MainClass{}
    public static void main(String[] args) {
```

```
                TestClass testClass; //In case the If Clause is placed in variable
                    declaration, the variable must be declaration must be separated

                if(System.getPropery("MUID").equals("<Mutant identifier>")){
                    testClass = new MutatedClass(true); //Won't fail as
                        MutatedClass extends TestClass
                }
                if(System.getPropery("MUID").equals(null)){ //Compares to null to
                    access original funtionality
                    testClass = new TestClass(test);
                }
                //Do something...
            }
        }
```

In the previous example we can find several of the things that we when developing this process had to take into consideration so the process wouldn't fail. First, and a thing that we also need to take into consideration with compatible mutants, when the point where the *If Clause* is inserted is a point where a new variable is declared, the variable declaration needs to be extracted from said *If Clause*. This is due to, if the variable remains enclosed in the *If Clause*, it won't be accessible from outside said enclosure, having the potential to change the original functionality of the source code being tested.

Another thing to take into account, and the main of the solution for incompatible mutants, is that the mutated class needs to extend the original class. This is so there is no necessity for further changes, such as functions that use the original class as a type for one of its parameters. Extending the original class will also allow the mutated class to be used in such cases.

This solution however has a specific case that will not work as well. The case is further depicted on one of the implemented mutation operators defined on Mutation Operators 5.2, with the mutation operator in question being *Super Call Deletion* 5.2.2.9. This mutation operator depicts a mutation that deletes the calls to the parent class, namely the *super()* calls. These calls need to be the first line in the constructor of the target class, and, when deleted, they are added by the constructor it self. In this case, if a mutation class is created, it will still call the original constructor, which includes a call for the parent class which we want to delete. Further description of the mutation operator in question, and a case specific solution, can be found on *Super Call Deletion* 5.2.2.9

4.3 Conclusion

With this new process we then expect to achieve our proposed objectives to achieve better performance without compromising on the configurability. This process however needs to take in account a lot of variables so the compilation is successful, as noticed on the *Incompatible Mutators* example.

This process, with a few changes to the *MUID* could also allow for a new way of performing mutation testing, since having all the mutants on one single version of the code, might allow for multiple mutant execution and how they interact with each other.

Chapter 5

The tool

As defined in the previous chapters, we defined our strategy and process to improve both the Configurability and Efficiency, to test the strategy a tool is still needed to experiment our proposed theories.

For that we created a Java based tool that uses as a base for the Kadabra tool and its GUI API, this allowed us not only to do a quick user interface to abstract the Kadabra tool, but also implement in a native way both the designed interface on chapter 3 and our proposed process on 4.

5.1 Tool Design

The tool design is more focused on the functionality rather than its looks. Also, it was developed to take into consideration our before mentioned concerns about the Configurability and Efficiency. It is designed to have two pages that works as follows.

The first, and initial page, is the screen that allows selecting an existing configuration file, as defined on the second screen, as well as displaying the results on a white box. It is also where it allows the user to start the mutation testing execution, with the proper configurations, as well as to stop it.

The main screen also allows on the top left corner to select the current page. The default is the main screen, that shows as "*Program*". There is also another page, "*Options*", that is shown on Figure 5.1, where not only allows to change the configurations but allows the creation of a new configuration file.

The configurations allow to change a number of things, that are as follows:

- Selecting the project to be mutated, by inserting its complete or relative path to the tool on the "*Project Path*" option.
- Where the user wants to store the mutated code, inside an "*Output*" folder. This option, "*Output Path*", by default has the tool location as its directory. This functionality is of importance, to give the possibility to the user to analyze the mutants, either the tests detected them or not.

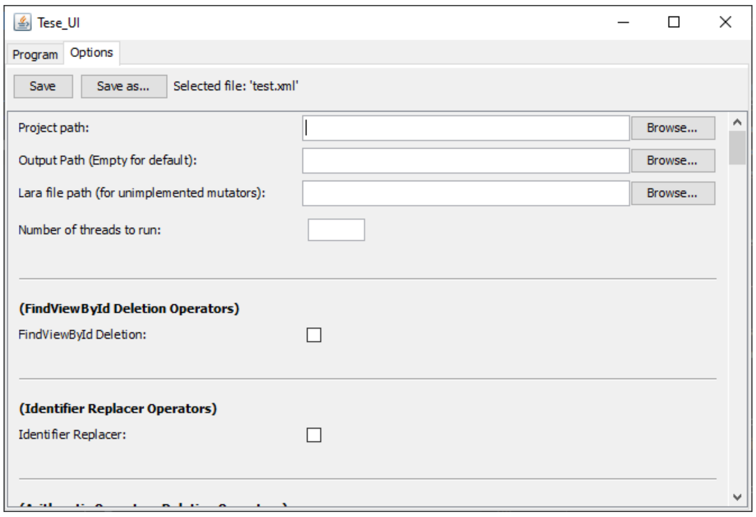
- Custom mutation operator. As mentioned on chapter 3, one of our contributions is that we want to allow the user to develop his own mutations. After developing said mutations this configuration, "Lara file path", is where the the user indicates his custom mutation operator. Note that currently it only allows for one new Lara file, but with use of the *Lara* imports functionality that file can contain more than one mutation operator.

- How many simultaneous executions we want. This functionality exists to improve the execution time of both the creation and execution phases of the mutation testing process. Currently it only supports the mutation creation phase, as the tool works by executing *Kadabra* once for every file. The inserted number will represent how many simultaneous executions of *Kadabra* will exist. If left empty the number of executions is the number of files the tool detected that are targeted for mutation.

- Selecting the mutation operators. The list of Mutation Operators is defined on the next section, 5.2, but as we can see on Figure 5.1, the first two of the allowed mutations are mutations that are allowed to either be executed or not. Some other mutations allow for extra configurability, such as the *BinaryOperators*, that allow the user to select with which operator they want the original to be replaced with.

- Save or create a copy of the configurations. This is where the file loaded by the first page is saved. It allows for both a quick save to the current file being edited as well as saving to a new file. This is a required step before changing to the initial page, having the risk of losing the configurations in case this step is skipped.

All these configurations, when saved, are all stored on an *Extensible Markup Language (XML)* file, which is required to run the tool successfully. This file is also required to run the tool on CLI mode, as it is where the configuration is stored.



Selected file: 'test.xml'

Project path:

Output Path (Empty for default):

Lara file path (for unimplemented mutators):

Number of threads to run:

(FindViewById Deletion Operators)

FindViewById Deletion: ☐

(Identifier Replacer Operators)

Identifier Replacer: ☐

Figure 5.1: GUI Configuration interface

5.2 Mutation Operators

On our tool we have a set of 19 mutation operators that can be used, having some of which that allow further configuration. The selection of these operators was made by analysing other tools and replicating their mutation operators, and for the two Android specific operators by further analysing the Figure 2.1. The selection criteria for this operators was based on the information that we had that allowed us to replicate said mutants and the feasibility of implementing them given the available time.

These operators were implemented using a primitive design of the interface mentioned on 3.1.2 and work with the Classic Architecture. For the new proposed Architecture, some of the mutants are incompatible with the it, requiring to be adapted with the method mentioned on 4.2.2.

To present the operators a division will be made into three different categories, General Operators, Java Methods, and Android Specific.

5.2.1 General Operators

Under General Operators fall all the mutation operators that operate over a set of general set of language operators, mostly language agnostic.

5.2.1.1 Arithmetic Operators

The Arithmetic Operators are binary operators that perform arithmetic operation, under these operators fall the $+$, $-$, $/$, $*$ and the $\%$ operator.

The mutation operation, in this case, is performed each time one of these arithmetic operators is found, and it works by replacing said arithmetic operator with one other arithmetic operator. Which arithmetic are searched for, and with which its replaced with, can be defined on the GUI and/or configuration file.

5.2.1.2 Arithmetic Operator Deletion

This operator, defined on the PIT tool [19], operates on the before mentioned Arithmetic Operators, but instead of performing an replace function, it deletes the arithmetic operator, creating two mutations. For example, on an operation of $var3 = var1 + var2$, deleting the arithmetic operator $+$ will result in two mutators, $var3 = var1$ and $var3 = var2$

5.2.1.3 Bitwise Operators

Bitwise operators are the operators that operate on the bit value (1's and 0's representation) of the variable. There are 3 types of Bitwise operators, the ones that perform bitwise shifts, the ones that perform boolean operations on the variable bits and one single operator that performs a complement of the bits.

The bitwise shift operators are, $<<$ that shifts a bit pattern to the left and the operator $>>$ shifts a bit pattern to the right. The operator $>>>$ however shifts a zero into the leftmost position, while the leftmost position after $>>$ depends on sign extension.

There are also Bitwise operators that perform boolean operations AND(&), OR(|) and XOR($\wedge$) on the variable bits.

A last Bitwise operator, is the complement operator($\sim$), this operator replaces all the binary elements with their complement. In other words replaces the 0's with 1's and vice-versa.

All these operators can be interchanged between them on our tool except the complement operator that can only be removed. Also, with the correct verification to use our purposed process, can be interchanged with the Arithmetic Operators.

5.2.1.4 Conditional Operators

There are two types of Conditional operators, there are the binary ones, that compare two variables, and there is a unary one, that negates the boolean value of either a boolean value or boolean expression. On this Mutation Operator only the binary operators are taken into account, and they are the Conditional-AND(&&) and the Conditional-OR(||), which are used when creating a boolean condition. On our tool these operators can only be interchanged between them.

5.2.1.5 Conditional Operators Deletion

As mentioned on the previous Mutation Operator, the *Conditional Operators*, there are two types of Conditional mutators, the binary and unary operators. On this mutation operator the only *Unary Conditional Operator*(!) is used.

This operator negates every boolean variable or expression and in this Mutation Operator, as the name suggests, detects every *Unary Conditional Operator* and removes every occurrence of it.

5.2.1.6 Conditional Operators Insertion

Like on the previous Mutations Operator, the *Conditional Operators Deletion*, this operator uses the only the *Unary Conditional Operator*(!). However, unlike ON the previous mutation operator, it inserts this operator in every boolean variable or expression.

This change will cause similar results to the previous mutation operator in cases where the *Unary Conditional Operator* is already present, as per boolean logic, however, on places where the *Unary Conditional Operator* is not present, it will create different results.

Given this, on our tool we allow both the *Conditional Operators Deletion* and *Conditional Operators Insertion*, but in case this last one is present it will be the only one used

5.2.1.7 Constant Operations

The constant operations mutation operator is an slightly changed version of the operator that is defined on the Pit Tool [19] that searched for inline constants and tries to change them on one of 6 different ways. Our version of the mutator tries to apply the same 6 types of mutations, however, searches not only for inline variables but also searches for constant variables or variable assignments.

The 6 types of mutations are as follows:

- Replacing the constant with a 1
- Replacing the constant with a 0
- Replacing the constant with a -1
- Reverse the sign of constant
- Adding 1 to the constant
- Subtracting 1 to the constant

All of this types of the mutations are independent and can be configured in our tool, either on the GUI or the XML configuration file.

5.2.1.8 Unary Operators

Unary operators are the language operators that operate on one single variable at a time, for instance, the right/left hand increments or decrements (++_, _++, –_, _–), which can be important to test, as, the order on which the increments/decrements happen can influence behaviour.

Also other unary operators can be the + or - unary operators that evaluate an expression, or the operator that inverts the value of a boolean expression.

Our mutation operator works by replacing an occurrence of either of the unary operators with a different one, which are selected by the user when configuring on the GUI/XML configuration file. The only unary operator that cannot be interchanged is the operator that is only able to be removed.

5.2.2 Java Methods

The mutation operators under this category are operators that are specific for the Java language, and, although some can still be reused by similar languages, are all limited by similarities.

5.2.2.1 Constructor Call

This mutation operator, as the name indicates, operates on a constructor call, and works by deleting said constructor call and replacing it with a *null* value. For example, the following line of code:

```
        TestObject o = new TestObject();
```

Will get the constructor deleted replaced like so:

```
        TestObject o = null;
```

This Java Method operation can also be considered an operator for Object Oriented languages, as it it is not just Java specific.

5.2.2.2 Fail on Null

The Fail on Null mutation, as defined on *Mutation operators for testing Android apps* from Deng L. et. al. [5], is a mutator that inserts a verification to all variables if they are null before their usage. Verification which will cause a failure in case the variable is indeed null.

This mutant has the purpose to alert the users (developers) of variables that are not properly being checked if they are null, as when defining a variable with data retrieved form somewhere the user has no control (being files, web, API's, etc), this variable is prone to cause a *NullPointerException*

5.2.2.3 Literal Operations

Literal Operations, or as defined on the Pit Mutations list, Inline Constants [19], its a set of sub-mutants that perform certain replacements of literals with certain values, depending on the type, as demonstrated on table 5.1.

	boolean		*integer byte short*				*long*		*float*			*double*	
Original	false	true	1	-1	5	literal	1	long	1.00	2.00	Any	1	Any
Mutated	true	false	0	1	-1	literal+1	0	long+1	0.00	0.00	1.00	0	1

Table 5.1: Literal Mutation Operators

Also, with this mutator, we have a sub-mutator to change every literal *String* to an empty literal *String*.

All this mutations can be activated or deactivated on our tool GUI and/or XML configuration file.

5.2.2.4 Non Void Call

This Mutation Operator, defined in the Pit tool [19], is a mutation operator that replaces the call for of a function depending on its return type as defined on the Table 5.2.

Type	Default value
boolean	false
int byte short long	0
float double	0.0
char	’\u0000’
Object	null

Table 5.2: Non Void Call Mutation Operator Mappings

One example is the following code:

```
        //***********************************
        //Original Code
        //***********************************
        public boolean exampleFunction() {
            return true;
        }

        public void testFunction() {
            boolean i = exampleFunction();
            if(i){
                // Do something
            }
        }

        //***********************************
        //Mutated Code Using Classic Process
        //***********************************
        public boolean exampleFunction() {
            return true;
        }

        public void testFunction() {
            boolean i = false;
            if(i){
                // Do something
            }
        }

        //***********************************
        //Mutated Code Using Proposed Process
        //***********************************
        public boolean exampleFunction() {
            return true;
        }

        public void testFunction() {
```

```
37          boolean i; //Needs to be extracted from the declaration so it can be
                used outside the If clause
38          if(System.getPropery("MUID").equals("<Mutant identifier>")){
39              i = false;
40          }
41          if(System.getPropery("MUID").equals(null)){
42              i = exampleFunction();
43          }
44          if(i){
45              // Do something
46          }
47      }
```

In this example the mutation point is located on Line 9, where a call for the *exampleFunciton* is made. As this function of return type *boolean*, and checking on Table 5.2, the mutation opertaor replaces it with false.

This can be found on Line 23 on the Classic Process, and on Line 39 for our proposed process. Note that on the proposed process, not only the original behaviour can be found on Line 42 and executed when no *MUID* is set, but we also had to separate the variable declaration from the assignment. This is so that the variable can be used outside the If Clause and does not impact the original behaviour.

5.2.2.5 Nullify Input Variable

This mutation operator, as the name suggests, has as its main purpose test if the user methods are verifying if its inputs are null, and this is done by replacing the parameters of a function with a null value. An example is provided in the following code:

```
1       //**********************************
2       //Original Code
3       //**********************************
4       public boolean exampleFunction(String test1, String test2) {
5           return test1.equals(test2);
6       }
7
8       public void testFunction() {
9           boolean i = exampleFunction("Test1", "Test2");
10          if(i){
11              // Do something
12          }
13      }
14
15      //**********************************
16      //Mutated Code Using Classic Process
17      //**********************************
18      public boolean exampleFunction(String test1, String test2) {
```

```
19            return test1.equals(test2);
20        }
21
22        public void testFunction() {
23            boolean i = exampleFunction(null,"Test2");
24            if(i){
25                // Do something
26            }
27        }
28        //New version required for second operator
29
30        //**********************************
31        //Mutated Code Using Proposed Process
32        //**********************************
33        public boolean exampleFunction(String test1, String test2) {
34            return test1.equals(test2);
35        }
36
37        public void testFunction() {
38            boolean i;//Needs to be extracted from the declaration so it can be
                  used outside the If clause
39            if(System.getPropery("MUID").equals("<Mutant1 identifier>")){
40                i = exampleFunction(null,"Test2");
41            }
42            if(System.getPropery("MUID").equals("<Mutant1 identifier>")){
43                i = exampleFunction("Test1", null);
44            }
45            if(System.getPropery("MUID").equals(null)){
46                i = exampleFunction("Test1", "Test2");
47            }
48            if(i){
49                // Do something
50            }
51        }
```

Like on the previous mutation operator, one mutation point is found on Line 9, however on the same line there is a second mutation point, as the function being mutated has two parameters that can be nullified.

The difference of both processes can be analyzed here, as on Line 23 we can see a mutation for the proposed process, for the second mutation point a new version is required. For our proposed process however, we can see both mutations on the same version, on both lines 40 and 43, all while continuing to allow the original functionality to be used by line 46. Note that both mutations have different *MUID*'s, as well as the original is set to *null* so both mutations and the original functionality can all be executed separately.

5.2.2.6 Nullify Return Value

Similar to the previous Mutation Operator, *Nullify Input Variable*, this mutation operator has as its main purpose to replace a variable/constant with null, this time however, instead of being in a call for a method it will be on the method return value. An example can be as follows:

```java
        //***********************************
        //Original Code
        //***********************************
        public boolean exampleFunction(String test) {
            return test == null;
        }

        public void testFunction() {
            boolean i = exampleFunction("Test");
            if(i){
                // Do something
            }
        }

        //***********************************
        //Mutated Code Using Classic Process
        //***********************************
        public boolean exampleFunction(String test) {
            return null;
        }

        public void testFunction() {
            boolean i = exampleFunction("Test");
            if(i){ //Will throw NullPointerException here
                // Do something
            }
        }

        //***********************************
        //Mutated Code Using Proposed Process
        //***********************************
        public boolean exampleFunction(String test) {
          if(System.getPropery("MUID").equals("<Mutant identifier>")){
            return null;
          }
          if(System.getPropery("MUID").equals(null)){
            return test == null;
          }
        }

        public void testFunction() {
            boolean i = exampleFunction("Test");
```

```
43              if(i){//If testing the mutation it will throw NullPointerException here
44                  // Do something
45              }
46          }
```

On this example the mutation point can be found on Line 5, where the *exampleFunction* has a *return*. This *return* is then mutated to return a *null* instead of a *boolean* value as expected. This is a good way to check that when the function being used (Line 9) is also being checked for null returns, to prevent a *NullPointerException* that may occur in this case on Line 10.

The difference here on both processes can be seen on the mutation point, that on our proposed process the capability of testing the original functionality still exists, and as supposed, it will work as the classic process, also triggering a *NullPointerExcecption* on Line 43, in case the mutant is used (Line 24 on the Classic Process).

5.2.2.7 Remove Conditional Operation

Defined on the Pit tool [19], this mutation operator functionality is to replace a conditional expression with the boolean value *true*. Future versions might also allow to change to the boolean value. An example of the implemented version is as follows:

```
1           //************************************
2           //Original Code
3           //************************************
4           public boolean exampleFunction(String test) {
5               return test == null;
6           }
7
8           public void testFunction() {
9               boolean i = exampleFunction("Test");
10              if(i){
11                  // Do something
12              }
13          }
14
15          //************************************
16          //Mutated Code Using Classic Process
17          //************************************
18          public boolean exampleFunction(String test) {
19              return test == null;
20          }
21
22          public void testFunction() {
23              boolean i = exampleFunction("Test");
24              if(true){
25                  // Do something
26              }
27          }
28
29          //***********************************
30          //Mutated Code Using Proposed Process
31          //***********************************
32          public boolean exampleFunction(String test) {
33              return test == null;
34          }
35
36          public void testFunction() {
37              boolean i = exampleFunction("Test");
38
39              if(System.getPropery("MUID").equals("<Mutant identifier>")){
40                  if(true){ //As the mutation point is the if it encapsulates the
                        whole If statement
41                  // Do something
42                  }
43              }
44              if(System.getPropery("MUID").equals(null)){
45                  if(i){
46                      // Do something
47                  }
48            }
49          }
```

This example has the mutation point on Line 10, where an already existing *If* statement exists. As per the mutation operator the conditional statement, in this case just checking if the variable *i* is true, is replaced with the key word *true*, forcing to always enter the *If* clause. This can be seen on the Classic Process on Line 24. On our proposed the mutated point is located on line 40. however we also to take something into account here. As the mutation occurs on the conditional statement of the *If* clause, the entire statement needs to be inside our proposed process own *If* Clause, as noticed on line 39 for the mutated point and Line 44 for the original code.

5.2.2.8 Return Value

Similar to the mutation operator *Nullify Return Value* 5.2.2.6, this mutant functionality is to replace the return value of methods that return *int, short, long, char, float or double* with 0 and functions that return a boolean with true. An example is as follows:

```
//***********************************
//Original Code
//***********************************
public boolean exampleFunction(String test) {
  return test == null;
}

```

```
        public void testFunction() {
          boolean i = exampleFunction("Test");
          if(i){
            // Do something
          }
        }

        //************************************
        //Mutated Code Using Classic Process
        //************************************
        public boolean exampleFunction(String test) {
          return true;
        }

        public void testFunction() {
          boolean i = exampleFunction("Test");
          if(i){
            // Do something
          }
        }

        //************************************
        //Mutated Code Using Proposed Process
        //************************************
        public boolean exampleFunction(String test) {
          if(System.getPropery("MUID").equals("<Mutant identifier>")){
            return true;
          }
          if(System.getPropery("MUID").equals(null)){
            return test == null;
          }
        }

        public void testFunction() {
          boolean i = exampleFunction("Test");
          if(i){
            // Do something
          }
        }
```

As mentioned, this mutation operator is similar to the *Nullify Return Value* 5.2.2.6, and as such, the example is similar. In this case however the return function will not return a null but a standard value depending on its return type, therefore not triggering the *NullPointerException* as the *Nullify Return Value* 5.2.2.6, making the purpose of this mutation operator check if the functionality of the functions is being tested.

5.2.2.9 Super Call Deletion

This mutation operator will try to replicate one possible mistake that might be common when developing a Java application, when developing a constructor for an object that extends another to add extra functionality, a keyword method must be used to call the parent object functionalities, *super()*, also a *this()* method might be used.

These methods might be easily forgotten to add, and therefore, lose functionality. For that this mutant will delete any *super()* or *this()* detected calls to check if those errors are being tested. This mutation operator will only yield results if used when the super call includes parameters, as a *super()* call is added by the compiler to the constructor.

This mutation operation, is an operation that cannot be directly implemented on our proposed process, as both the *super()* calls require to be the first items on the constructor. Therefore, this mutation operator, requires a solution similar to the one proposed on *Incompatible Mutants* 4.2.2, but slightly changed, as the proposed solution extends the original class, method that this mutator, by the reason above mentioned is not possible.

The required difference is that, so we don't require to add an *if* to all the usages of the original class usages, is to create a copy of the constructor with the *super()* call with one extra parameter of an object created by our tool and without the *super()* call. An example for this can be found on the following code:

```
//***********************************
//Original Code
//***********************************
public class ExampleClass extends AbstractExampleClass{
    private boolean example;
    ExampleClass(boolean example){
        super(example);
        this.example = example
    }
    public getExample(){
        return example;
    }
}

public class TestClass{
    public void testFunction() {
        ExampleClass exampleClass = new ExampleClass(true);
        boolean i = exampleClass.getExample;
        if(i){
            // Do something
        }
    }
}

//***********************************
```

```
        //Mutated Code Using Classic Process
        //***********************************
        public class ExampleClass extends AbstractExampleClass{
            private boolean example;
            ExampleClass(boolean example){
                //Deleted super call
                this.example = example
            }
            public getExample(){
                return example;
            }
        }

        public class TestClass{
            public void testFunction() {
                boolean i = exampleFunction("Test");
                if(i){ //Will throw NullPointerException here!
                    // Do something
                }
            }
        }

        //***********************************
        //Mutated Code Using Proposed Process
        //***********************************
        public class ExampleClass extends AbstractExampleClass{
            private boolean example;
            ExampleClass(boolean example){
                super(example);
                this.example = example
            }
            //New Constructor new Parameter and same functionality except the super
                () call
            ExampleClass(boolean example, MutationTest mutTest){
                //super() Removed
                this.example = example;
            }
            public getExample(){
                return example;
            }
        }

        //Creation of a Mutation class, so mutated constructor can be called
        public class MutationTest{
                MutationTest(){}
        }

        public class TestClass{
            public void testFunction() {
```

```
74                ExampleClass exampleClass;
75                if(System.getPropery("MUID").equals("<Mutant identifier>")){
76                    //Added the MutationTest as a parameter to call the mutated
                          constructor
77                    exampleClass = new ExampleClass(true, new MutationTest());
78                }
79                if(System.getPropery("MUID").equals(null)){
80                    exampleClass = new ExampleClass(true);
81                }
82                boolean i = exampleClass.getExample;
83                if(i){
84                    // Do something
85                }
86            }
87        }
```

As mentioned on *Incompatible Mutants* 4.2.2, this mutation operator requires some extra steps to perform a mutation operation using our proposed process due to its nature. In contrast the Classic Process deletes the mutation point on Line 31.

As the mutation point requires to be the first line of the constructor, and due to the compiler adding it in case its missing (without the parameters), the Incompatible Mutants process cannot work. For this mutation operator however, as seen in the example above, it is needed to add a new constructor, similar to the original one (Line 36). This new extra constructor will have however a new parameter, so it is able to be distinguished. This parameter will be of a type defined by the tool (Line 68), to ensure that it is used only by the mutants.

After this process, our proposed process will insert the *If* clauses on every constructor call, in this case Line 17 of the original code.

5.2.3 Android Specific

Under Android Specific mutators fall all the mutators that we developed that are specific for the Android Framework. The two implemented mutators work on the interface between the Java source code for the application and its XML Layout files. These layouts files are where the GUI is designed, and, where all its elements are defined. For each element there is an identifier, which, at the compilation phase, all the identifiers are compiled on several Java classes filled with constants, having each represent an element identifier, so an interface between the Java code and the layout files can be achieved.

5.2.3.1 FindViewById Deletion

To retrieve a layout element the *findViewById()* function is used, this function receives a resource identifier present on the Views class, where the layout elements identifiers are present.

For proper GUI behaviour on the mobile application this is crucial to get right, and can be easily forgot, so this mutation operator finds every occurrence of the *findViewById()* method and replaces it with a *null* keyword, simulating a failure retrieving the layout element.

```
1          //************************************
2          //Original Code
3          //************************************
4          public View exampleFunction() {
5              View view = findViewById(R.id.exampleView);
6              return view;
7          }
8
9          //************************************
10         //Mutated Code Using Classic Process
11         //************************************
12         public View exampleFunction() {
13             View view = null; //Mutated here
14             return view;
15         }
16
17         //************************************
18         //Mutated Code Using Proposed Process
19         //************************************
20         public View exampleFunction() {
21             //Variable declarations are separated from the If Clause or else won't
                   be able to be used. If it is not possible to separate it, it is
                   required to use the solution proposed for incompatible mutation
                   operators
22             View view;
23             if(System.getPropery("MUID").equals("<Mutant identifier>")){
24                  //Although in this example won't make a difference inserting this
                        line here, might make a difference in other cases
25                  view = null;
26             }
27             if(System.getPropery("MUID").equals(null)){
28                 view = findViewById(R.id.exampleView);
29             }
30           return view;
31         }
```

This example is a simple example of a function to get a View element from the XML resources. This view has the id *exampleView*, and, the mutation point works by simulating that the ID is not found. The mutation point is located on Line 5, and the mutated points can be seen on Lines 13 for the classic process and 25 for our proposed process. As usual, in cases where the mutation point is set on a declaration statement, the declaration and assignments are separated in our proposed process so that the variable remains accessible outside the *If* clause.

5.2.3.2 Resource Identifier Replacer

Resource Identifiers however can be found not only for elements of layout files but also for every kind of resource defined on an XML file, such as images, colors, GUI text string (Used mostly for multi language applications), etc.

These identifiers are crucial to get right, having the possibility to change the wrong resource in case of an poorly tested method makes any change using a resource. For this we have implemented this mutation operator that searches for every resource identifier, and, replaces them with a similarly typed resource identifier. An example for the view typed identifier can be as follows:

```
        //************************************
        //Original Code
        //************************************
        public View exampleFunction() {
            View view1 = findViewById(R.id.exampleView1);
            View view2 = findViewById(R.id.exampleView2);
            View view3 = findViewById(R.id.exampleView3);
            //Do something...
        }

        //************************************
        //Mutated Code Using Classic Process
        //************************************
        //It will create a number of version square of the number identified id's (
            Including original). This case 6, will provide an example for the 1st
        public View exampleFunction() {
            View view1 = findViewById(R.id.exampleView2); //Mutated here, next will
                 be R.id.exampleView3
            View view2 = findViewById(R.id.exampleView2); //Same as this one
            View view3 = findViewById(R.id.exampleView3);
            //Do something...
        }

        //************************************
        //Mutated Code Using Proposed Process
        //************************************
        public View exampleFunction() {
            //Variable declarations are separated from the If Clause or else won't
                be able to be used. If its not possible to separate it, it is
                required to use the solution proposed for incompatible mutation
                operators
            View view1; //First Mutation point
            if(System.getPropery("MUID").equals("<Mutant1 identifier>")){
                view1 = findViewById(R.id.exampleView2);
            }
            if(System.getPropery("MUID").equals("<Mutant2 identifier>")){
                view1 = findViewById(R.id.exampleView3);
```

```
33         }
34         if(System.getPropery("MUID").equals(null)){
35             view1 = findViewById(R.id.exampleView1);
36         }
37         //... Similar If Clause process for the two other View objects
38       return view;
39     }
```

This example is a good example to compare both classic and proposed processes, as for this mutant to work, it is required for more than one mutation point to exist, as the *Resource Identifiers* are swapped between them. As such this example has 3 mutation points, creating 6 mutations as calculated using the formula:

*Number of Mutation Points * (Number of Mutation Points - 1 = Number of Mutations*

The 3 mutation points in this example are located on Lines, 5, 6 and 7, on the parameters of the *findViewById* functions. Of the 6 mutated versions however, only part of them are present on the example, this is to reduce the size of the example, but the remaining mutations are similar to the presented ones. For the classic version only the first mutation is present, on line 16, and it replaces the first mutation point with the second, where the remaining mutations produce a similar mutation on a new version of the code.

For the proposes process however, only one mutated version is needed to be created, and as such, there is the possibility to execute any of the 6 mutants or the original version. In the example however there are only present two mutations, the ones referring the first mutation point, and therefore, the first set of *If* Clauses. The number of *If* clauses per mutation point, using this mutation operator, is always equal to the number of mutation points. For every mutation point a similar set of *If* Clauses is produced, exchanging only the order of the *Resource Identifiers*

5.3 Conclusion

Having developed this tool, with this set of mutants, was a great way to test our new process and it limitations. To note for instance the Java Method mutation operators, and more specifically the Super Call Deletion, that required us to further enhance how we tackled the incompatible mutation operators.

It was also a great way to better understand the *Kadabra* tool [17] and its *Lara* framework [18], as developing the tool and the mutation operators required for a deeper understanding of both tools, which later helped creating our interface.

This tool also allowed us to generate the source code for both processes, which is a fundamental step for the next chapter, Comparison 6, as we needed said source codes to test the compilation times for both processes and compare.

Chapter 6

Process Comparison

Having proposed a new process, that theoretically improves significantly the existing classic process, a comparison is required so the improvement claim can be confirmed.

6.1 Used Method

To perform the comparison, and achieve acceptable results, we tried to isolate the most variables possible. This was done so we could have the most precise and accurate results of only the process performance

First thing we did was to isolate the mutation operators, using the same operations on both processes. This is possible as our tool architecture and interface allows a distinction on how the mutation operators are identified and applied.

Second thing we isolated, was the method used to compile and test the testing project. This was made by using the test project build tool *Maven*, which allowed us to have control for what we execute and how, using different *Maven* goals. In our case, to execute the tests the goal "*test*" was used, that if no compiled versions were found it would also compile the project (Required for the classic process).

Third, and to isolate our tool away from the process, and therefore obtain more precise times, we compiled and executed the tests using *PowerShell* console commands and its system integrated stopwatch functionality.

6.2 Results

To obtain our results we tested both processes on a personal project were 14 source files were mutated with three increasing number of mutations. Also the project has 66 unit tests that are ran against every mutation and the original project.

As mentioned on the previous section 6.1, we tested the compile and testing times as one for both the classic process and our proposed process, using the test project build tool and a system stopwatch, three times for each mutant count. The results were as follows;

For the first test, with 142 mutations to test, we've found out, as shown on table 6.1, that our proposed process had a speedup ratio of approximately 2.08x in comparison with the classic process, completing the test 13m40s faster.

	1st Run	2nd Run	3rd Run	Average
Classic	26m31s	26m20s	26m11s	26m21s
Proposed	12m49s	12m32s	12m40s	12m40s

Table 6.1: First test complete run-times

On the second test, with 258 mutations to test, both process execution times increased as expected as shown on 6.2 but that increment was different on both, as the Classic process took 10m27s longer comparing with the last test in contrast with the 2m05s from the proposed process. This difference also had an increased speedup ratio of 2.49x with the classic process taking 22m02s extra time to complete

	1st Run	2nd Run	3rd Run	Average
Classic	37m41s	36m31s	36m47s	36m47s
Proposed	14m45s	14m55s	14m41s	14m45s

Table 6.2: Second test complete run-times

At last, on the third test, with 440 mutants to test, we've found out similar gains as on the previous run, as can be seen on the results on the table 6.3. Comparing with the previous results from the second test, the classic process took about 29m14s longer to complete, while the proposed process took only 10m13s longer to complete. In total on this run, the proposed classic process took 41m04s longer than the proposed process to complete testing the 440 total mutants, this then represents a total speedup of approximately 2.64x

	1st Run	2nd Run	3rd Run	Average
Classic	1h05m02s	1h06m02s	1h08m24s	1h06m02s
Proposed	24m58s	25m14s	24m45s	24m58s

Table 6.3: Third test complete run-times

As noticed on the tests, there was an increase in time difference that it took to execute the classic process over our proposed process, this, can be further visualized on Figure 6.1 that has a graphic representation of the increase of the execution times and the difference between them.

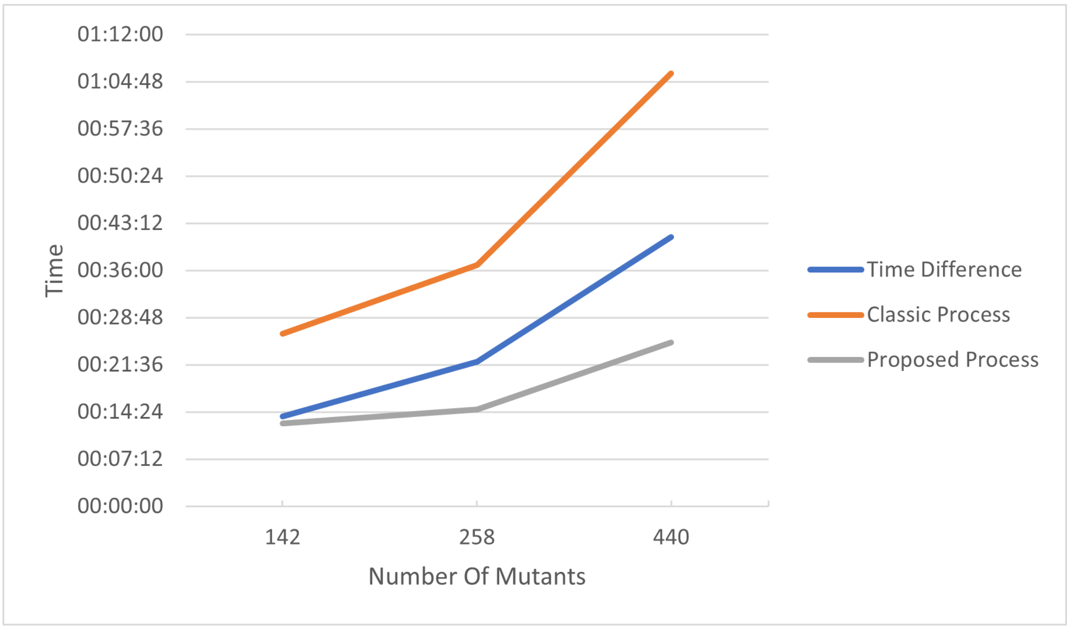


Figure 6.1: Time Difference

6.3 Conclusion

As expected, our proposed process, although it brings an extra layer of complexity to the process, as it might require to check if every mutation can be compiled successfully, it brings a substantial speedup to the compilation and testing phases of the process, as can be seen on figure 6.2. This speedup increases as the number of mutants increase, being that on our tests, that started with 142 mutants, the achieved speedup was of approximately 2.08. This speedup rose to 2.5 with 258 mutants and to 2.64 with 440 mutants.

Although not being scientifically checked, as it was not the main focus of the investigation, the times taken to generate the source files to test were noticed to take significantly longer to generate on the classic process. This was also expected, as, in the proposed process only requires to save the files one single time, heavily reducing this time. In contrast, for the classic process, a creation of a complete copy of the project files is required for every mutant, process which can also be time consuming depending on the project size and drive speed. This was verified on the third test, as for the proposed process it took around 10 to 15 minutes to complete this stage of the mutation testing process, for the classic process, due to its drive usage intensive nature, took around 1h to complete creating the 440 mutated versions o the original source code.

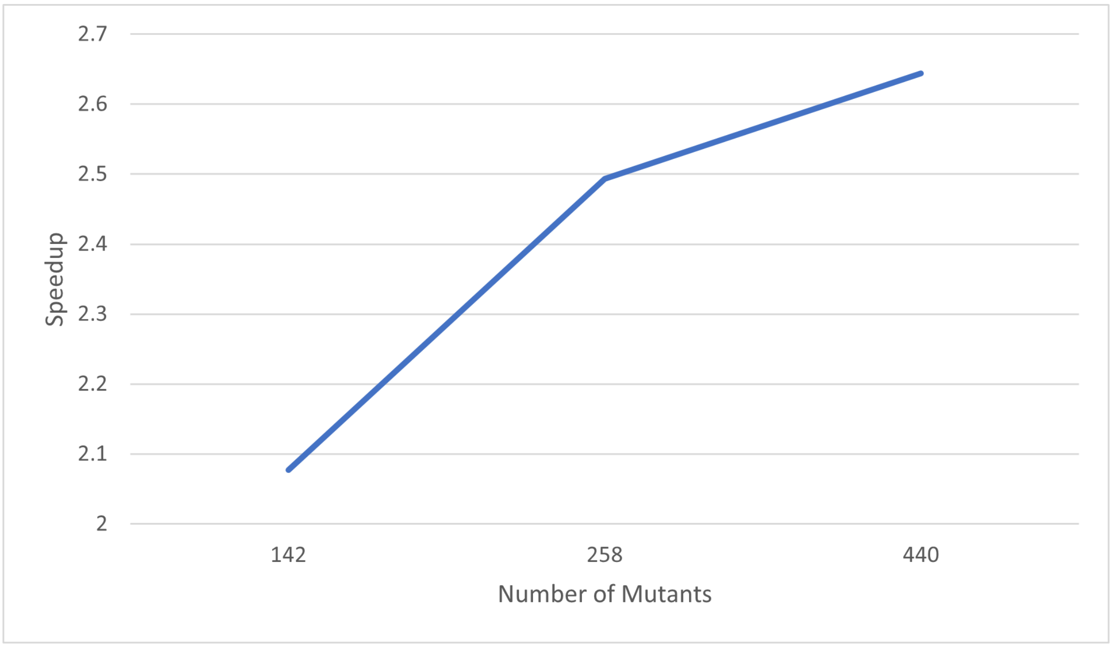


Figure 6.2: Proposed process Speedup by number of mutants

One thing to note is that on our testing method is that due to variables that we could not control, such as background processes on the machine used, we ran all the tests on the same machine, one right after the other for the same, and tried to achieve at least 3 similar results to achieve some consistency and therefore validation. This method was further deemed to be necessary as on our initial testing some results had time differences between tests being greater than expected, indicating something on the background might have been executing and therefore those times were invalidated.

Chapter 7

Conclusions and Future Work

This thesis, with its original purpose of developing an new Mutation Tool for Mobile apps, had its focus refactored to focus more on the new process rather than the tool. This was an important step, as the main goal was to provide a more efficient way to perform mutation testing, and, with this new process, it now can have an even broader application than just the initially considered mobile apps universe.

Its always hard to change focus mid way through any kind of work. This usually means that the thing you were working on stops being the focus, and can even result on what might be considered lost time with the previous work. In this case fortunately, that did not occur, as we started developing our tool the change of the process actually helped on the motivation as that would mean a better contribution.

Not only that, but having already a basis with the old process allowed us to perform the tests with both the classic approach and our new approach, that ultimately helped us to obtain our results, as we wouldn't need to use other tools that might have different approaches which might impact performance.

7.1 Results

As mentioned, the main goal of this thesis was to develop a new process for mutation testing. This goal was achieved, obtaining result of over 2x better performance on our testing, and, with the potential to have even higher gains the larger the application under testing.

Our second goal was also achieved, by developing a tool that not only allows to apply our proposed process, but also tackles one of the identified problems, the configurability, by use of our own framework.

7.2 Further Work

For future work it would be interesting to even extend the tool further, with more parameters to take into account, as for our process it requires to exclude all the mutants that generate compilation errors but are still not identified.

At the time of writing we are also in a phase of re-implementing the Java Method and Android Specific mutators, as it requires to take into account the extra verification's for the proposed architecture, which take longer to implement.

Also it would be nice to add more mutation operators native to it, as well as a report generation, as it would improve its capabilities.

Finally, and most important, would be interesting to perform more testing on the process, on larger applications, to prove the theoretical exponential improvement of performance with our process.

References

[1] Sérgio Almeida, Ana C. R. Paiva, and André Restivo. Mutation-based web test case generation. In Mario Piattini, Paulo Rupino da Cunha, Ignacio García Rodríguez de Guzmán, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technology - 12th International Conference, QUATIC 2019, Ciudad Real, Spain, September 11-13, 2019, Proceedings*, volume 1010 of *Communications in Computer and Information Science*, pages 339–346. Springer, 2019.

[2] Ana Barbosa, Ana C. R. Paiva, and José Creissac Campos. Test case generation from mutated task models. In Fabio Paternò, Kris Luyten, and Frank Maurer, editors, *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011*, pages 175–184. ACM, 2011.

[3] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for Java (Demo). In *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452. Association for Computing Machinery, Inc, jul 2016.

[4] David Schuler and Andreas Zeller. (Un-)Covering Equivalent Mutants. In *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 45–54. IEEE Computer Society, April 2010.

[5] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing Android apps. *Information and Software Technology*, 2017.

[6] Marcio Augusto Guimaraes, Leo Fernandes, Marcio Ribeiro, Marcelo D'Amorim, and Rohit Gheyi. Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations. In *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation, ICST 2020*, pages 198–208. Institute of Electrical and Electronics Engineers Inc., oct 2020.

[7] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, pages 169–175, 2007.

[8] René Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, San Jose, CA, USA, July 23–25 2014.

[9] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 612–615. IEEE, nov 2011.

[10] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume Part F130154, pages 233–244. Association for Computing Machinery, aug 2017.

[11] Eduardo Luna and Omar El Ariss. Edroid: A mutation tool for android apps. In *Proceedings - 2018 6th International Conference in Software Engineering Research and Innovation, CONISOFT 2018*, pages 99–108. Institute of Electrical and Electronics Engineers Inc., feb 2018.

[12] Yu-Seung Seung Ma, Jeff Offutt, Yong Rae Kwon, and Yong Rae Kwon. MuJava: An automated class mutation system. *Software Testing Verification and Reliability*, 15(2):97–133, jun 2005.

[13] 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. In *Proceedings - International Conference on Software Engineering*, pages 33–36. IEEE Computer Society, may 2018.

[14] Ana C. R. Paiva, André Restivo, and Sérgio Almeida. Test case generation based on mutations over user execution traces. *Softw. Qual. J.*, 28(3):1173–1186, 2020.

[15] David Schuler and Andreas Zeller. Javalanche: Efficient Mutation Testing for Java. In *ESEC-FSE'09 - Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 297–298, 2009.

[16] Leonardo Da S. Sousa, Auri M.R. Vincenzi, Marcio Eduardo Delamaro, Igor R. Vieira, Vinicius R.L. Mendonca, and Cassio Leonardo Rodrigues. Reducing the Cost of Mutation Testing Using the Semantic Size of Mutant. In *Proceedings - 2018 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2018*, pages 2675–2680. Institute of Electrical and Electronics Engineers Inc., jan 2019.

[17] Kadabra tool. Available at `http://specs.fe.up.pt/tools/kadabra/`.

[18] Pit mutation testing - mutators. Available at `https://web.fe.up.pt/~specs/projects/lara/doku.php?id=start`.

[19] Pit mutation testing - mutators. Available at `http://pitest.org/quickstart/mutators/`.

[20] Efficient mutation testing for java. Available at `https://github.com/david-schuler/javalanche`, January 2020.

[21] Jumble coverage tool for junit tests. Available at `https://sourceforge.net/projects/jumble/`, January 2020.

[22] Jumble coverage tool for junit tests. Available at http://jumble.sourceforge.net/, January 2020.

[23] The major mutation framework. Available at http://mutation-testing.org/, January 2020.

[24] Mdroidplus. Available at https://bit.ly/2QFAkZT, January 2020.

[25] Mujava home page. Available at https://cs.gmu.edu/~offutt/mujava/, January 2020.

[26] Mutation system for java programs, including oo mutation operators. Available at https://github.com/jeffoffutt/muJava, January 2020.

[27] Mutation testing tool for android integration testing. Available at https://github.com/Yuan-W/muDroid, January 2020.

[28] Mutation testing with javalanche. Available at http://javalanche.org/, January 2020.

[29] Pit mutation testing. Available at http://pitest.org/, January 2020.

[30] Pit repository commit list. Available at https://bit.ly/2ZPxkOW, January 2020.

[31] State of the art mutation testing system for the jvm. Available at https://github.com/hcoles/pitest, January 2020.

[32] Yuan Wei. *MuDroid: Mutation Testing for Android Apps*. Bachelor's final project, Univ. College London, London, U.K., 2015.