Universidad de Concepción
Facultad de Ingeniería
Departamento en Ciencias de la Computación

Mutation Testing Techniques for Mobile Applications

Tesis para optar por el grado académico de
Doctor en Ciencias de la Computación

POR: ISYED DE LA CARIDAD RODRÍGUEZ TRUJILLO

Profesores Guía: PhD. Macario Polo Usaola
Escuela Superior de Informática e
Instituto de Tecnologías y Sistemas de Información
Universidad de Castilla-La Mancha

PhD. Diego Seco Naveiras
Departamento de Ingeniería Informática y
Ciencias de la Computación
Facultad de Ingeniería
Universidad de Concepción

junio 2021
Concepción, Chile

ACKNOWLEDGMENTS

For the development of this thesis, the collaboration and help of many people was necessary. First, I thank my advisor Macario Polo, for being unconditional, for his commitment, for sharing his knowledge and experience, and for being an example as an engineer. I thank the Alarcos group of the Castilla - La Mancha University for hosting and supporting me during my research stays. Also, to the Coruña University where I did my first research stay and to whom I am very grateful for the opportunity they offered me. Especially, to all the professors and members of the Department of Computer Science of the University of Concepción, for their knowledge and support in these years of study. To Professor Diego Seco, for his teachings, dedication in his work and for his friendship. To my fellow Ph.D. students and to the old and new friends I met at Udec. Finally, I thank my parents for being always present in every stage of my life and for their love, Kiki for being a wonderful person and my husband Jose Luis for being my support and my strength.

Thank you all

SUMMARY

Mutation testing has the important drawback of its high computational cost, especially in the testing of mobile software, due to implications in dealing with mutants (i.e., compile, link, deploy and execute the mutated versions on the mobile device are high-cost tasks). So far, works on mutation testing for mobile mainly focus on the development of new mutation operators. However, to our best knowledge, none of them has evaluated cost reduction techniques in this context to deal with the problem of execution time. The main contributions of this research are related to (1) how several well-known cost reduction techniques help to the effective improvement of testing time and (2) the suitability of the proposed mathematical models for describing the execution time of test cases in mutation testing. In addition, we present the design and architecture of *BacterioWeb v.2* in a distributed environment, to enable mutation testing for teams of testers and contribute to the transition of mutation testing from academic to industrial application.

TABLA OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As it is well known, exhaustive testing of software systems (i.e., testing the system with its possible inputs and environmental conditions) is, in practice, impossible, since it is unfeasible to reproduce all the running situations. So, the tester must make decisions about which parts of the system must be tested and the degree of depth of the test cases.

When a test engineer is testing a software artifact, he/she needs to know which portions of the system are running their test cases. For this, coverage criteria are used because they allow to know the "amount of product" that is being tested (number of lines of code, methods, decisions, conditions, etc.). If the tester knows the coverage that a set of test cases reaches on the SUT (the *System Under Test)*, he/she will be able to determine which portions of the system are not being covered by the test cases:

- If the coverage is lower than a prefixed threshold, the tester must add more test cases to cover the SUT more in depth, to force the execution of the unexplored system areas.
- If the coverage is greater than that prefixed threshold and the test cases do not find any error, then the SUT has a very high quality.

Figure 1 shows a mutation testing process that combines error detection with coverage measurement. This process is a modified version elaborated by Polo and Reales [1] of the mutation testing proposed by Offut [2]. In the refined process in Figure 1, the tester evaluates the correction of the input program for the initial test suite. Once the tester has a process model to follow, the next point of interest is reducing costs in the “create

mutants," "run T on each alive mutant," and "threshold reached" boxes of the figure. Eliminating ineffective test cases can occur during test case execution or after, by applying a test-suite-reduction algorithm based on mutation [1].

Mutation testing builds on discovering the artificial faults inserted in copies of the System Under Test (SUT) which are called mutants. Mutation testing *subsumes* several test criteria by incorporating appropriate mutation operators [3]. A test criterion C_1 subsumes another C_2 if for every program, any test set T that satisfies C_1 also satisfies C_2. Indeed, since mutants can be placed anywhere in the code, mutation coverage can be used as surrogate for almost any other form of structural coverage. Some studies (i.e., [3]) have discussed how discovering all faults seeded by mutation operators might subsume several coverage criteria widely accepted (such as decision, condition, condition/decision, and modified decision/condition). From here, the *Mutation Score* (coverage criterion used in mutation testing) is considered as an adequate-coverage criterion if good mutation operators are

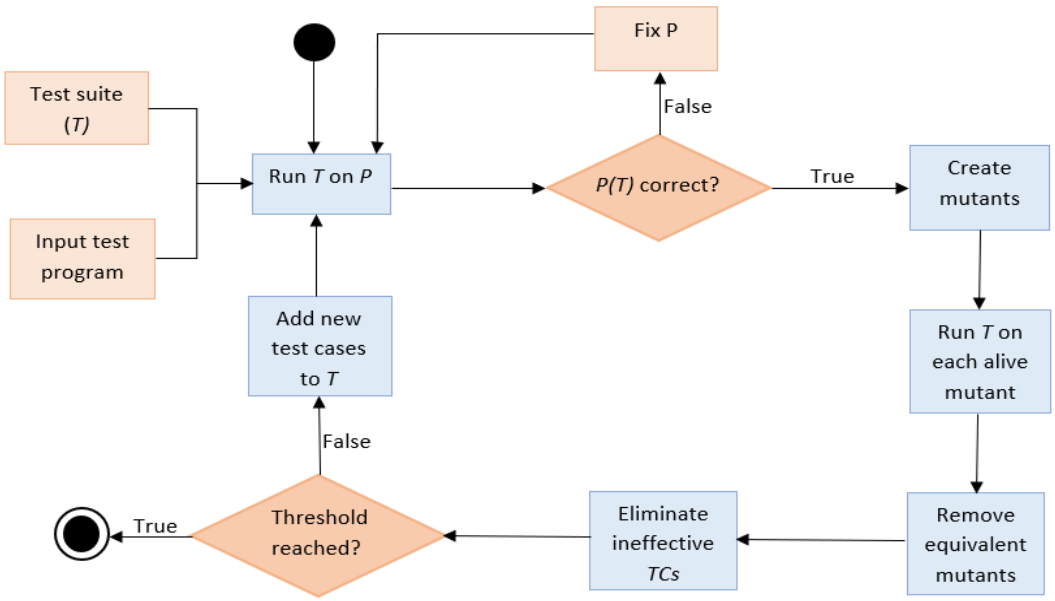


Figure 1. "Traditional mutation testing process", where T is the test suite and P is the program under test and TC is a test case

applied [3], [4]. So, this makes mutation testing a highly generic and flexible approach to software testing.

One of the results of this thesis redefines this process proposed by Polo and Reales [1] for the context of mobile applications.

The effectiveness of mutation testing has been demonstrated in many empirical studies [5], although it has the important drawback of its high computational cost, which is closely related to the number of mutants generated. Thus, cost reduction in mutation testing is a very active research matter [1], [6]–[14].

The situation is especially hard in the testing of mobile software, for several reasons:

1. Testing traditional programs is different from testing mobile applications, due to their specific features. For example, their sensibility before context events such as: location changes, screen orientation, phone calls, and many other events. In addition, mutation testing in mobile applications has implications in dealing with mutants, especially regarding the time needed to compile, link, deploy and execute the SUT and its mutated versions on the mobile device.
2. As Deng et al. point out [15], “for a variety of technical reasons, test execution [in Android] tends to be quite slow”, which “is particularly troublesome for Android testers”. These authors report that “a single iteration of an experiment required more than 20 hours”. In the

experimentation carried out in this research, we have often far exceeded that value.

Related to the latter, this is due to the nature of instrumented Android test cases. In Android, test suites can be composed of "unit" or "instrumented" test cases:

- Unit test cases can be executed as classical Junit tests, directly on the developer's computer, and their execution is relatively fast.
- Instrumented test cases simulate interactions of the user with the application or use Android-specific resources (sensors, for example). These tests require the application under test to be deployed and installed on an emulator or mobile device.

Instrumented tests turn compilation, deployment, and installation into highly costly tasks, slowing down the whole mutation process. This is also highlighted by Escobar-Velásquez et al. [14], for whom "Time is an issue in mutation testing, for both generation and testing time".

As shown in Epigraph 2.2.3 several researchers that have applied mutation testing to mobile software. To our best knowledge, all of them are focused on the proposal and development of mutation operators for this specific context, but none deals with the problem of execution time. Nonetheless, all of them mention it as one of the biggest obstacles in mobile mutation testing.

1.1 GOALS

The **main goal** of this thesis is: To contribute to reducing the cost of mutation testing for mobile applications.

In order to fulfilling it, we must also reach the following subgoals (SG):

SG 1. To elaborate the theoretical conceptual framework of this research from the study of the main concepts, phases, and problems of mutation testing.

SG 2. To study the state of the art of the most used techniques in mutation testing.

SG 3. To know and analyze the state of the art of mutation testing in mobile applications.

SG 4. To develop a framework to automate the mutation testing process in mobile applications.

SG 5. To model combinations of cost reduction techniques that directly influence execution time and directly impact mobile technology.

SG 6. To validate the research applying techniques of Empirical Software Engineering to real mobile.

1.2 RESEARCH METHODS

Different research methods are used during this thesis:

- An Experiment is a procedure carried out to support, refute, or validate a hypothesis. An experiment deals with an independent variable of the environment or phenomenon under study and measures its effect on other dependent variable [16]. Wohlin et al. [17] have described a process for Software Engineering experiments with five steps: scope definition, scheduling, operation, analysis & interpretation, and presentation & dissemination.
- A Case Study in Software Engineering is an empirical research that uses different sources of evidence for researching an instance of a phenomenon inside its real context [18]. According to Runeson et al. [18], case studies: (1) are flexible (since they deal with complex and dynamic characteristics of actual phenomena), (2) their qualitative and quantitative conclusions are based on a clear set of evidences, taken from multiple sources in a planned, consistent way and (3) add

knowledge to the pre-existent, based on a previously established theory or setting up a new one.

1.3 STRUCTURE OF THE THESIS

Chapter 2 Theoretical Framework and State of the Art is divided into two sections. First, the main concepts related to mutation testing and the structure and testing of Android applications are described. Second, some relevant related works, which are part of the state of the art in mutation testing and mobile mutation testing, are discussed.

Chapter 3 Contribution to the improvement of mobile mutation testing forms the core of the research, describes the development of mutation testing techniques with different approaches and implementations, details the tool we have developed for supporting the whole mutation testing process of mobile software, defines various mathematical models of the applied cost reduction techniques, and raises the research questions.

Chapter 4 Experimentation and Results describes the experiments performed to answer the research questions in terms of: mutation testing techniques applied, mutation operators used, target mobile applications for the execution of the tests. Then, the results obtained are analyzed and a set of best practices resulting from the experimentation carried out during the research are listed.

Finally, the *Future work* section describes some lines of work that could drive future research and the *Conclusion* section summarizes the main contributions of this thesis.

CHAPTER 2

THEORETICAL FRAMEWORK AND STATE OF THE ART

This chapter is divided into two key aspects. The first (Background) details the mutation-based testing process and the main concepts associated with mobile applications, specifically Android applications. The second (Related work) presents (1) a state of the art in terms of the works and techniques that have addressed the different problems of mutation testing and (2) a state of the art on mutation testing in mobile applications.

2.1 BACKGROUND

2.1.1 Mutation testing

Mutation Testing goes through three main stages [19]:

- **Mutant Generation:** It consists in creating mutants of the original program using mutation operators.
- **Test case execution:** It consists in executing the test cases against the original program and the mutants.
- **Results Analysis:** It consists in (1) analyzing the results of the executions of the test cases on the mutants (comparing them with the same test case results on the original) and (2) in calculating the mutation score to evaluate the quality of the test cases.

A mutant M of a program under test P is a copy of P that contains a small code change that is interpreted as a fault. These faults are introduced by mutation operators. Consider a simple instruction such as $return\ a\ +\ b$

(where a and b are integers): mutation operators can mutate it in at least 10 different ways *($a - b$, $a \times b$, a / b, $a + b{+}{+}$, $-a + b$, $a + - b$, $0 + b$, $a + 0$, $|a| + b$, $a + |b|$)*, depending on each operator. Thus, the number of mutants generated even for a medium-size program can be very large.

Once the mutation operators generate the mutants, test cases are run. A test case *T* finds the error inserted in a mutant *M* when the test case result is different for the original program *P* and for the mutant: in this case it is said that the mutant is *killed*; otherwise, the mutant is *alive*. The goal of mutation testing is *to kill all the mutants* (i.e., to find all the artificial errors): a test suite that kills all the mutants is *mutation adequate*. Since a mutation adequate test suite finds all the artificial faults, it is expected that it finds also all the natural faults (those inadvertently inserted by the programmer). In the sense of Figure 1, if the test suite finds all the artificial faults and does not find any fault in the SUT, it is very likely that this one is free of them (the artificial faults must be "good" faults, representative enough of those that could be committed by programmers).

Many mutants that remain alive will never be killed because they are *equivalent mutants* and will always produce the same output as *P* for any test case. The fault introduced in equivalent mutants is not a fault but an optimization or de-optimization of the code (for example, the Java instructions *return a* and *return a++* provide the same result). Equivalent mutants are really "noisy" and make difficult analyzing test case execution results. Taking into account the set of equivalent mutants, the following equation gives the quality of a test suite (measured in terms of the number of mutants killed) and defines the *mutation score* (MS) [20]:

$MS(P,T) = \frac{K}{M - E}$	Where P is the program under test; T is the test suite; K is the number of mutants killed; M is the number of mutants generated; and E is the number of equivalent mutants.

Figure 2. Mutation Score

For example: The first row of the following of Table 1, shows the code of a *max(int a, int b, int c)* function that returns the maximum of the three numbers it receives as arguments. Four of the many possible mutants that can be obtained are also shown in the Table 1: in Mutant 1, the *AND* operator (&&) has been replaced by *OR* (||); in the second, a pre-increment operator has been inserted before returning *a;* in the third, a post-increment is introduced, which produces an equivalent mutant (i.e., a mutant whose behavior is indistinguishable from the behavior of the original program).

Version of program	Code

Original	`public int max (int a, int b, int c){`
	`  if (a >= b && a >= c)`
	`  return a;`
	`  if (b >= a && b >= c)`
	`  return b;`
	`  return c;`
	`}`

Mutant 1
Operator LOR: (Logical Operator Replacement)
	`public int max (int a, int b, int c){`
	`  if (a >= b || a >= c) ***`
	`  return a;`
	`  if (b >= a && b >= c)`
	`  return b;`
	`  return c;`
	`}`

Mutant 2
Operator UOI: (Unary Operator Insertion, pre-increment)
	`public int max (int a, int b, int c){`
	`  if (a >= b && a >= c)`
	`  return ++a; ***`
	`  if (b >= a && b >= c)`
	`  return b;`
	`  return c;`
	`}`

Mutant 3
In this case is the same UOI operator, but introducing a post-increment
	`public int max (int a, int b, int c){`
	`  if (a >= b && a >= c)`
	`  return a;`
	`  if (b >= a && b >= c)`
	`  return b;`
	`  return c++;  ***`
	`}`

Table 1. A possible original program and three mutants [3]

Table 2 shows a possible test suite for the mentioned *max* function. Inner cells show the results returned by each test case in every program version: the $tc1$ test case only kills mutant 2. Test cases $tc2$ and $tc3$ both

kill mutant 1. However, none of three test cases kill mutant 3, because this mutant always produces the same result than the original program. This is an example of an equivalent mutant. If no mutant would have been killed, then the tester should have to design manually new test cases for killing them.

	Data of Test Cases (tc_j)		
Version	$tc_1\ (3,2,1)$	$tc_2\ (1,2,1)$	$tc_3\ (2,2,3)$
Original	**3**	**2**	**3**
Mutant 1	3	1	2
Mutant 2	4	2	3
Mutant 3	3	2	3

Table 2. Results produced when executing the test cases on the original program and the mutants

2.1.2 Structure and Test of Android Applications

A mobile application (*app)* is a software application that runs into a small mobile device, such as a cellular phone or a tablet.

According to a very recent report of IDC[1] (International Data Corporation), Android and iOS are the most widely used operating systems across the world. The iOS operating system only runs on iPhone devices, whilst Android is more open and is the native operating system of many mobile devices' brands. With some exceptions, Android devices are in general much cheaper than iPhones. Based on these facts and in the wide availability of tools for dealing with Android devices and with the Android operating system, we will use this platform for the more technical aspects of this thesis.

[1] Available at (October 23, 2020), https://www.idc.com/promo/smartphone-market-share/os

In general, Android native applications are developed with Android Studio. In this IDE (Integrated Development Environment), every project (Gradle project) has at least one *build.gradle* file that drives the compilation, testing and deployment of the app. Android Studio structures the projects in a set of fixed directories (Figure 3). A project is hosted on a directory (see the *ModernTriType* folder in Figure 3, which holds the homonymous project) and may have several modules, each one located in a subdirectory. Android apps have an *app* directory where there are:

- The *build.gradle* file.
- The *src* folder, which has the Java source code and the resource files.
- The *build* folder, which has two significant subdirectories:
    - *intermediates* (where the Java compiler leaves the *.class* files corresponding to the source files).
    - *outputs* (whose *apk* subfolder saves the *.apk* files that will be pushed and installed onto the mobile device).

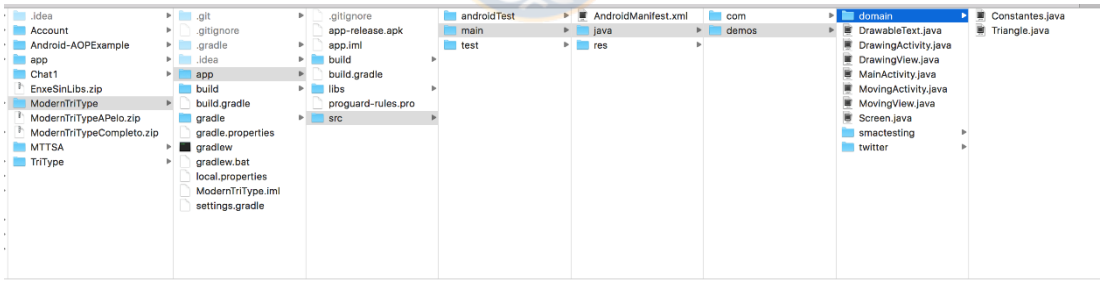


Figure 3. Some folders and files in an Android project

Android apps are usually written in Java or Kotlin and packaged for installation on the device as an *.apk* file. When compiled, the *.java* or *.kt* source files are translated into their corresponding *.class* files (made up of a Java Virtual Machine compatible bytecode) and, from there, a second stage of compilation translates them into *.dex* files. Together with the resource files, the *.dex* files are packaged into an *.apk* file which contains the app. On

the physical device, *.dex* files are interpreted either by the *Dalvik* or by the ART (*Android Runtime*) virtual machines, depending on the Android version.

Given an app with test cases, the gradle project usually has at least two tasks for compiling:

- *gradlew assembleDebug*, which builds the *apk* file corresponding to the app. Typically, this file is called *app-debug.apk*.
- *gradlew assembleDebugAndroidTest*, which builds a different apk file with the test cases. By default, this file is called *app-debug-androidTest.apk*.

Both compilation tasks invoke the required internal operations of the Android SDK for generating the .class files, *.dex* files and to produce the final *.apk* file. This is, there are two compilation steps (from .java to .class and from .class to *.dex*) and one packaging step *(.apk*) before the application is installed on the device.

Since every mutant is a slightly modified version of the SUT, all this whole process described previously should be done with each mutant (i.e., the generation of a different *.apk* with each program version), what would require an additional, significant cost in this type of technology. This particularity of the packaging step requires to use an efficient mutation testing technique or the combination of several techniques.

Regarding the test cases of an Android application, suppose an app (*app.apk)* composed of three classes (*Screen1* and *Screen2*, which conform the user interface and are specializations of *Activity*, and *DomainObject*, which does not have any relation with the Android libraries). In the example we have two different types of tests (Figure 4):

- *Unit tests*, which exercise the business logic. These test cases do not need any special Android resource. These test suites do not need the construction of an *apk* and can be executed without any device.
- *Instrumented tests*, whose test cases require special Android resources for interacting with the user interface (clicking, writing…), using sensors, etc. Its execution requires the production of a separated *apk* file (*testApp.apk* in the example), which must be installed on the device.

Although the *testApp.apk* file only needs one deployment, a classic mutation approach [2], [1] requires that a different version of the *app.apk* is compiled, packaged and installed on the device for each mutant. The costs of compiling, packaging, and installing are so high that testing a mobile app with a classic mutation testing process becomes almost completely impracticable. Thus, mobile software testing is an especially suitable context for applying cost reduction techniques.

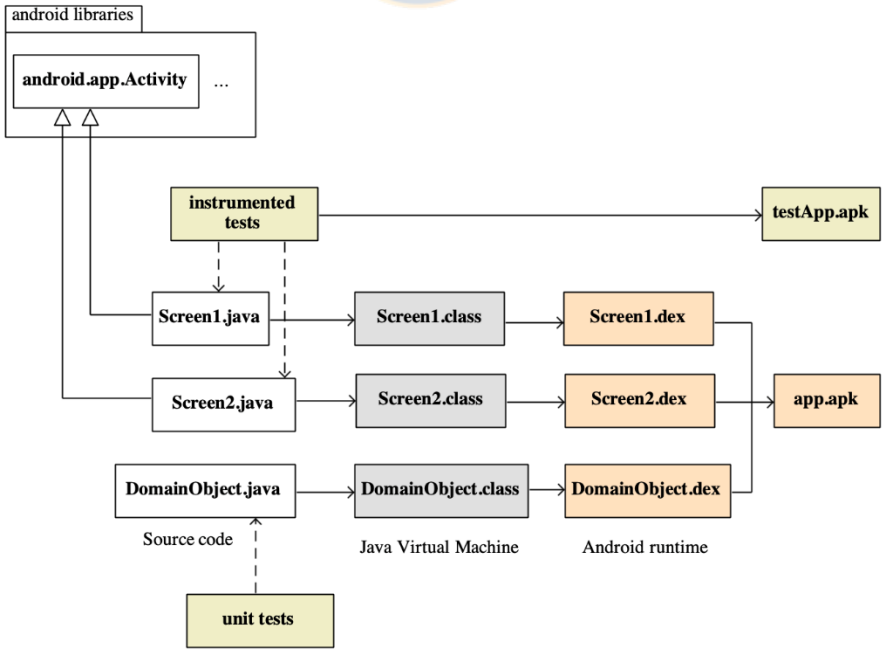


Figure 4. Compilation and packaging of an Android app

Figure 5 shows part of the actual organization of the test files in *WordPress*, which is one of the apps we have used in our experiments. Instrumented and unit tests are respectively located under the *androidTest* and *test* folders. Files in these folders can be auxiliary classes (mocks, for example) or test suites. When the tests are to be executed from the official IDE (Android Studio), this one:

- For the instrumented test cases, the IDE creates a *testApp.apk* and one *app.apk* file. Both apks are pushed and installed on the mobile device (either an emulator or a physical device). Then, the IDE opens a virtual terminal on the device (whose operating system is based on Linux) and sends a command for running the tests (i.e., *adb shell am instrument -w -r -e…*). If all the test files are selected, the device iterates on each file and, inside each file, on each test method.
- For the unit tests, the IDE directly calls the *gradlew test* command on the folder where the project is located (there can be variations of the command). It launches a compilation of the project and the execution of the tests under the *test* folder. These test cases do not need any connected device.

Figure 5. Organization of tests in WordPress

There are frameworks, such as *Robolectric* [21], that allow to execute instrumented test cases without any connected device (physical or emulated). *Robolectric* also mentions the execution time as its main reason for existence: *"Running tests on an Android emulator or device is slow! Building, deploying, and launching the app often takes a minute or more"*. The main problem with this framework is that it does not support all the functionalities of real devices.

2.2 RELATED WORK

2.2.1 Mutation Operators

Mutation operators try to imitate common errors that programmers make. It relies on two hypotheses:

- The *Competent Programmer Hypothesis* states that a program written by a competent programmer may be incorrect, but it will differ from the correct version by relatively simple faults [4]. Therefore, mutation testing, only introduces faults consisting in simple syntactical changes, which represent the faults that are made by "competent programmers".
- The *Coupling Effect* states that a test suite that detects all simple faults in a program is so sensitive that it also detects more complex faults [1], [4], [22].

The suitability of mutation testing for detecting faults has been widely demonstrated along many years. Currently, the main research concerns are related with: (1) the application of mutation in new contexts and paradigms and (2) the reduction of its execution cost and time [20]. These two points are quite important for our research.

Mutation testing evolution has led to the proposal and development of multiple operators for all kind of testing levels, programming languages, paradigms and platforms. Thus, for example:

- First works about mutation testing targeted individual functions and methods of Fortran programs, in a kind of unit testing [2], [4], [22], [23].
- Later, mutation operators for integration testing were developed [24], [25]. These operators reproduce common faults that programmers commit in software units' interactions.
- Ma, Offutt and Kwon [26] proposed specific operators for object orientation, and implemented them in the MuJava tool, which are method-level operators and class-level operators.
- Reales et al. [27] defined several mutation operators for testing multi-class systems at the integration and system levels. The authors group them into five categories, depending on the faults they can insert.
- Different authors have proposed specific operators for several other programming languages: C [24], [28], C# [29], [30], C++ [31], SQL [32], [33], Aspect programs [34]–[36] or Python [37].
- There are also mutation operators for other contexts: relational databases [38], the ATL model transformation language [39] or BPEL [40].
- Besides using mutation testing at the software implementation level, it has also been applied at the design level to test the specifications or models of a program. For example, there are operators for Finite State Machines [41], [42], Network protocols [43], [44], Web Services [45], [46] and Security Policies [47], [48].

Thus, the variety of work is large, due to the diversity of systems, platforms, and environments. Mutation operators for a certain type of system, paradigm or programming language are responsible of inserting the common faults that programmers and developers commit when they build the system: the *Virtual Modifier Insertion Operator* for C++ [31], for example, cannot be applied to a BPEL or a PHP specification. It is worth

noting this fact: the suitability of mutation operators for testing programs depends on the programming language.

Android mutation operators are mentioned in Epigraph 2.2.3 (Mutation Testing in Mobile Applications).

2.2.2 Mutation Testing Problems and Techniques

Although Mutation Testing can effectively assess the quality of a test suite; despite this, it presents some drawbacks.

The main drawback is the high computational cost, since even a huge number of mutants can be generated from a medium-sized program. Let us suppose a system under test from which 500 mutants are generated and a test suite with 50 test cases (what is a relatively small system). If each test requires 0.5 seconds to be executed, the execution of all the test cases against the original system and the mutants will require 50*0.5 + 500*50*0.5 = 25+ 12500= 12,525 seconds = 208.75 minutes = 3.47 hours, which is too much time just to get the execution results.

Equivalent mutants are another problem, since they involve additional human effort to identify them [49] and an useless waste of computational time when they are executed [50]. Also, when the mutation score established is not reached, the tester needs to design new test cases, which also implies additional human effort.

Many works have focused on the development of techniques or strategies for cost reduction, but these problems are not totally closed yet. Besides, due to technological development, the number of platforms and applications is growing, and testing techniques need to be adapted to these changes.

Next, we review the main cost-reduction techniques proposed in mutation testing for its three main steps (mutant generation, test case execution and results analysis).

2.2.2.1 *Mutant Generation*

The number of mutants is the main cost factor in mutation testing, since the further steps completely depend on them. So, several mutant reduction techniques have been proposed for this first step:

- **Mutant Sampling** [51]: is a simple approach that randomly chooses a small subset of mutants from the entire set, according to a predefined percentage. Wong and Mathur [52] conducted an experiment using a variable selection rate x from 10% to 40%. The results of this study shown that Mutant Sampling is valid with an x value higher than 10%. De Millo et al. [53] and King and Offutt [23] also evidenced these results. Recently, Derezińska and Rudnik [54] proposed different mutant sampling criteria based on equivalence partitioning with respect to object-oriented program features. Based on the results, class random sampling and operator random sampling are recommended for OO in standard mutation testing, since the mutant sampling technique is easily applicable in comparison to other cost reduction techniques.
- **Mutant Clustering** [55]–[57]: Instead of selecting mutants randomly, mutant clustering chooses a subset of mutants using clustering algorithms. Hussain's empirical results [55] suggest that Mutant Clustering is able to select fewer mutants but still maintaining the same mutation score.
- **Higher Order Mutation**: is a form of mutation testing introduced by Jia and Harman [58]. This technique combines two or more mutants into the same mutated program (a higher order mutant, HOM). The empirical results of Polo et al. [59], [60] suggest that applying second order mutants reduces the test effort by approximately 50%, without

much loss of test effectiveness. Langdon, Harman an Jia [61] build higher order mutants that are harder to kill than any first order mutant. More recently, Abuljadayel and Wedyan [9] present an approach to generate higher order mutants using a genetic algorithm, also harder to kill than first order mutants.

- **Selective Mutation**: firstly suggested by Mathur [62] and later extended by Offutt, Rothermel, and Zapf [63], states that the number of mutants can be reduced by applying a subset of the mutation operators. Therefore, the objective is to find a small set of mutation operators that generates a subset of all possible mutants without a major loss of test efficiency. Some of the most recent works in this line of research are [64]–[67].

In this very same category (mutant generation), there are other strategies that accelerate the process of mutant generation or test case execution:

- **Mutation at bytecode level**: consists in injecting the changes directly in the compiled code, avoiding the cost of mutant compilation. This technique has been used by tools such as MuJava[26], Javalanche [68] and Bacterio [69].
- Bogacki and Walter [70], [71] introduced an alternative approach to reduce compilation cost using **Aspect-Oriented Programming**: mutants are implemented as aspects that introduce changes in the behavior of the SUT methods. Unfortunately, these researchers abandoned this research works after some very preliminary results.
- **Mutant Schema** [72]: is designed to reduce the total cost of mutation testing. The basic idea of this technique is to compose different programs into a *metaprogram* (all program versions are included in a single file). To determine which of the program versions included in

the schema must be executed, some type of control mechanism must be implemented.

For illustrating the Mutant Schema approach, consider the example shown in Table 3.

Original	Mutant 1

`class ClassA {`
` public int foo (int a, int b){`
`  for (b < 10) {`
`  a++;`
`  b = b + 2;`
`  }`
`  return a;`
`  }`
` }`
	`class ClassA {`
	` public int foo (int a, int b){`
	`  for (b < 10) {`
	`  a++;`
	`  b = b - 2; ***`
	`  }`
	`  return a;`
	`  }`
	`}`

Mutant 2	**Mutant 3**

`class ClassA {`
` public int foo (int a, int b){`
`  for (b < 10) {`
`  a++;`
`  b = b * 2; ***`
`  }`
`  return a;`
`  }`
`}`
	`class ClassA {`
	` public int foo (int a, int b){`
	`  for (b < 10) {`
	`  a++;`
	`  b = b / 2; ***`
	`  }`
	`  return a;`
	`  }`
	`}`

Table 3. A small program and three mutants

To our best knowledge, the first work about Mutant Schema is that of Untch, Offutt and Harrold [72], who created a *mutant schema generator* for Fortran. They used *metamutants* and *metaprocedures*. A *metamutant* contains all the mutants in a single file as a set of *metaprocedures*, which are functions that gather the different changes introduced by mutation operators. In Table 4, we have written a *metaprocedure* in Java for the AOR (Arithmetic Operator Replacement) operator.

Inside the AOR function, a *switch* statement asks for the mutant version that must be executed depending on the *mutant descriptor* passed by a test driver, which invokes the *metamutant* and directs which mutants are to be instantiated. The code of the original class is substituted by a call to the

corresponding *metaprocedure*. Actually, the paper [72] does not give too many implementation details.

Mutant Schema (*metaprocedure*)

```
class ArithmeticOp{
  public static int AOR (int a, int b, String mutantDescriptor){
    switch (mutantDescriptor){
       case "SUB" : return a - b;
       case "MUL" : return a * b;
       case "DIV" : return a / b;
       default: return a + b;
    }
     return a;
  }
}
```

Possible mutant code

```
class ClassA{
  public static int foo (int a, int b){
    for (b < 10){
      a++;
      b = ArithmeticOp (b, 2, MUTANT_DESCRIPTOR);
    }
     return a;
  }
}
```

Table 4. Mutant Schema (adapted from Untch, Offutt and Harrold [72])

That work, of 1993, has inspired other researchers:

- Ma, Offutt and Kwon [26] adapt the idea to Java programs in the MuJava tool, also automating the *metamutant* generation. These authors create *metaprocedures* for the object-oriented characteristics, such as inheritance, polymorphism, and instantiation overhead. Some of these authors reuse this very same approach (Kim, Ma and Kwon, in [73]).
- Papadakis and Malevris [74] apply the original Untch et al.'s approach, but adapting it to symbolic execution.
- Reales and Polo ([20], [75]) include *metamutants* instrumenting the original Java bytecode with the insertion of *if-else* statements. Table 5 shows the metamutant for the example

shown in Table 3. It includes several conditional statements to know what mutant must be executed by means of the *exec* function. In the same way, these authors do not explain how that *exec* function drives the mutant instantiation.

Mutant Schema

```
class ClassA{
  public static int foo (int a, int b){
    for (b < 10){
      a++;
      if(exec(m1)){
          b=b-2;
      } else if(exec(m2)){
           b = b * 2;
         } else if(exec(m3)){
              b = b / 2;
          } else {
             b = b + 2; // original statement
             }
     }
     return a
   }
}
```

Table 5. Mutant Schema (adapted from Mateo and Usaola [20])

Discussion: In traditional mutation testing, the execution environment must load, for each mutant, the class containing the mutated statement (because each mutant is a version different of class) and then to execute the test cases. As we have pointed out, one of the bottlenecks in mobile mutation testing is the deployment of the application into the device: at a first glance, it could be required to deploy the application once per mutant. Thus, the use of Mutant Schema technique may help to package all mutants in just an application version, reducing the number of deployments to one. However, as noted above, these Mutant Schema approaches do not provide many technical details neither about the construction of the Mutant Schema structure nor about the controller in charge of assigning the current mutant. Anyway, these approaches require to analyze the program code to (1) detect

the statements to be changed, (2) substitute the original statements by calls to the *metaprocedures*, (3) create the *metaprocedures* and (4) implement the test driver.

Bytecode translation is also a very interesting technique: since Android applications are written in Java and, before being compiled into *.dex* files, are compiled into *.class* classes, we will explore the introduction of mutation operators in this intermediate step.

With respect to techniques such as High Order Mutation, Mutant Sampling and Mutation Selective, they are focused on reducing the number of mutants generated, but we have decided to leave them out of the scope of this research because execution time is the most worrying factor in mobile mutation tests.

2.2.2.2 ***Test Case Execution***

The strategy used for test execution also has a strong impact on the total testing time [2], [1]. In fact, in the most primitive model, the tester executes all test cases against all mutants, although it is possible to reduce the number of executions if each test case is only launched against those mutants remaining alive: suppose a system with 7 mutants ($m_1, \dots, m_7$) and 5 test cases ($t_1, \dots, t_5$). Suppose also that the killing matrix obtained after executing all tests against all mutants is the one appearing in Table 6: as shown, 40 executions (5 test cases against 7 mutants plus the original program) are required to complete the process.

	m1	m2	m3	m4	m5	m6	m7
t1	X			X	X		X
t2	X			X	X	X	X
t3	X			X	X		
t4		X				X	
t5		X	X			X	X

Table 6. An "all against all" killing matrix for a supposed system

However, if test cases are launched against mutants that, after each iteration, remain alive (i.e., the test suite does not attempt to "kill twice" the same mutant), the number of executions may be lower whilst the mutation score is preserved: in Table 7, t_2 is not executed against the mutants that t_1 has already killed and, in general, t_{n+1} is not launched against the mutants killed by $t_1..t_n$. In this example, 20 executions are required (15 + 5 of the original).

- t_1 kills m_1, m_4, m_5 and m_7, which are removed from the mutant suite (there are 7 executions at this point).
- Then, when the next case t_2 is to be executed, is not executed against the mutants that t_1 has already killed. Hence, t_2 is directly executed against m_2, m_3 and m_6 (3 executions), and m_6 is removed from the mutant suite because is killed by t_2.
- Then, t_3 is launched only against m_2 *and* m_3 (2 executions). They are not removed from the mutant set because they are not killed.
- In general, t_{n+1} is not launched against the mutants killed by $t_1..t_n$.

In this example, only 20 test case executions are required (15 + 5 of the original) instead of 40, and the test suite can be reduced to 4 test cases.

	m1	m2	m3	m4	m5	m6	m7
t1	X			X	X		X
t2						X	
t3							
t4		X					
t5			X				

Table 7. An "Only against alive" killing matrix for a supposed system

Discussion: In the example, this technique shows that the final test suite is formed by $\{t_1, t_2, t_4, t_5\}$, because it reaches 100% as mutation score. However, the Table 8 shows that $\{t_1, t_5\}$ is also a mutation-adequate test suite. Overall, for *regression testing* (i.e., the execution of an existing test suite against a SUT after this one is modified), a smaller test suite is better than other with more test cases.

Although the problem of minimizing a test suite (the *optimal test-suite reduction problem*) has been shown to be NP-hard [76], several approaches present greedy algorithms for its solution. Gupta et al. [77] have worked intensively in this area. However, these greedy algorithms require the complete execution of all test cases against all the mutants. Though, since testing is often programmed as an unattended, nightly batch process, the complete execution and further application of a greedy algorithm is a good choice to deal with test suites reduction and obtain a test suite like the one in the Table 8.

	m1	m2	m3	m4	m5	m6	m7
t1	X			X	X		X
t5		X	X			X	X

Table 8. A reduced test suite obtained from Table 6

Weak Mutation [73], [78]–[80]: in strong mutation, the decision about if a mutant is killed or alive is taken at the end of the execution of the test case: with this approach, a mutant is killed when the three *RIP* conditions are got: *R*eachability (the mutated statement is reached), *I*nfection (once the statement has been reached, the test case causes an erroneous state on the mutant), and *P*ropagation (the erroneous state is propagated to the output). In weak mutation, the mutants are checked immediately after the point where the mutation has been introduced, rather than checking the mutant output after execution ends. So, weak mutation only requires the two first conditions (reachability and infection). The main drawback is that detected faults may not be observable in strong mutation, so accepting faults of less quality.

Discussion: The advantage of weak mutation is that test case execution can be analyzed before its completion since the difference of states between the SUT and the mutants can be immediately checked after the mutated statement is executed. The main drawback is that detected faults may not be observable in strong mutation, so accepting faults of less quality. Moreover, the concept of equivalent mutant may be different now, also because of the non-observability of the output.

Another strategy is to use advanced execution environment, such as **parallel execution** [81]–[84]. This technique executes mutants in parallel processors, reducing the total time of execution with no loose of effectiveness. However, it is necessary to have a good infrastructure.

2.2.2.3 *Results Analysis*

The most important obstacle in the result analysis stage is the presence of equivalent mutants. From a formal point-of-view, the problem of detecting all equivalent mutants is undecidable [5]. Annotating the SUT with constraints [85] may help in the automatic discovering, but this technique is never used in practice.

The best strategies for reducing costs in result analysis consist in diminishing the number of mutants generated with some of the techniques reviewed in Section 2.2.2.1.

2.2.3 Mutation Testing in Mobile Applications

The massive development of software for mobile devices is so recent, and the platforms and operating systems evolve so quickly, that techniques and tools that could be valid a few years ago may not be longer applicable today: as Kirubakaran and Kasthikeyani pointed out [86]. This section reviews some relevant works related to mutation testing on mobile software.

The special characteristics of mobile software (Figure 6) have a direct influence on testing. Most research works on mobile testing have focused on the proposal of new mutation operators for injecting faults based to reproduce the problems associated to these characteristics. The most significant research works about this area focus on: (1) analyzing the suitability of classic mutation operators to mobile software and (2) the proposal of new mutation operators to reproduce common faults in this environment.

```
(1) Connectivity and mobility with multiple network connections with
different bandwidths.
(2) Different screens sizes, resolutions and orientations.
(3) Resource constraints (memory, processor).
(4) Context awareness and multiple input channels (users,sensors,
networks).
(5) Potential interaction with other applications.
(6) Security and vulnerability.
(7) Finite energy source.
(8) Double nature of apps (native and web).
(9) Short development life cycle (to gain competitive advantage).
(10) Performance.
(11) Multiple devices and operating systems.
```

Figure 6. Special characteristics of mobile software

Deng et al. [15] proposed 11 mutation operators for simulating faults in the parts of the code that use the new programming features of Android (Table 9), although they leave for the future the implementation of others, specially related to the context-awareness of applications: "*... we have not yet considered all aspects of Android apps. For instance, one important distinct characteristic of mobile apps is that they are context-aware*". Their source of faults is some Google's technical documentation for testers and some significant characteristics of Android apps (event-driven nature, configuration saved in XML files, null values and screen orientation). They analyze the quality of their operators generating mutants for several apps and execute tests against the mutants. It is worth noting that Deng et al. [15] emphasize the excessive cost of tests execution, because their mutant generation tool builds an *.apk* file for each mutant.

Deng et al.'s [15] mutation operators	
Category	**Operator**
Intents	Intent Payload Replacement Intent Target Replacement
Activity Lifecycle	Lifecycle Method Deletion
Event Handler	OnClick Event Replacement OnTouch Event Replacement
XML	Activity Permission Deletion Button Widget Deletion EditText Widget Deletion Button Widget Switch
Common faults	Fail on Null Orientation Lock

Table 9. Android mutation operators

In a very recent article, Escobar-Velásquez et al. [14] extended a previous study of 2017 [87]. They analyzed 2,023 fault reports taken from six different sources (bug reports of open-source Android apps, bug-fixing commits of Android open-source apps, Android-related Stack Overflow

discussions, the Exception hierarchy of the Android APIs, Crashes and bugs described in previous studies and Reviews posted by users of Android apps on the Google Play Store). Their analysis shows that 65% of the bugs are typical of any Java application, and that the remaining 35% are directly related to Android-specific characteristics. They classify the bugs using a taxonomy with 14 high-level categories of faults. Some categories (e.g., "Collections and Strings") only contain Java faults, others (e.g., "Activities and Intents") Android-specific faults, and others (such as "Input/Output") contain a mix of both. The authors propose 38 mutation operators covering 10 out of the 14 categories. Some of the operators are specifically designed for Android. Besides the taxonomy and the operators, an additional and interesting contribution is their *MutAPK* tool that directly inserts the faults into the compiled and packaged APK file. They also describe MDroid+ [8], another tool that generates the faults from the source code.

Deng et al. [7] apply 17 mutation operators specifically designed for Android applications and compare their ability to detect faults with other four techniques. These authors conclude that mutation is effective at detecting faults in Android applications, although more research effort is needed.

Polo et al. [88] describe a generic architecture for the development of mutation operators for Android. All operators are specializations of a set of abstract classes. Their goal is to make easy the development of new operators and to decouple the operators' implementation from the external libraries used for manipulating the Java bytecode and injecting the faults.

Jabbarvand and Malek [89] search "energy anti-patterns" and, from them, they build many operators that, for example, increase the frequency of the location update requests or do not switch off the Bluetooth. Their testing framework is called *μDroid*. The *μDroid* generates mutants and, to determine whether a mutant is killed, it compares its power consumption with the

original program. There are no details about how test cases are executed. Regarding the test execution time, the authors only report about the mean times for determining whether mutants are killed (i.e., comparing the traces of energy consumption). The mean time is 11.7 seconds in the 9 apps used in their experiments.

More recently, Paiva et al. [12] describe 3 mutation operator to test the specific behavior of mobile applications (UI patterns). The iMPAcT tool is designed to deal with the mutants generated by these operators [90].

2.3 PARTIAL CONCLUSIONS

According to the state of the art, we can conclude that research in Mutation Testing can be classified in five items:

1. Definition of mutation operators for specific context and technologies.
2. Reduction of the number of mutants generated, since it is the most influencing factor in the general cost of the process.
3. Reduction of the cost of test cases execution.
4. Designing ways to reduce the cost of result analysis.

With respect to *Mobile Mutation Testing*, there is a common agreement related to the need of having specific mutation operators that reproduce common faults in Android applications, and it is evidenced the adequacy of the mutation score as a valid coverage criterion for mobile software.

However, to our best knowledge, there is a lack of works analyzing, adapting, or proposing the use of specific techniques to reduce the cost of mutation testing in this concrete environment. Deng et al. [15] are the only who point out that performance should be improved with parallel execution, using fewer mutants or building a faster test framework. This is one of the main goals of this research.

CHAPTER 3

A CONTRIBUTION TO THE IMPROVEMENT OF MOBILE MUTATION TESTING

Our objective is to address the cost of mutation testing in mobile applications. The main cost factor of mutation testing is the number of mutants generated, which influences on generation, execution and result analysis times. In this regard, techniques such as mutant scheme and parallel execution are effective. In the case of mobile applications, mutation testing has implications when dealing with mutants, especially regarding the time needed to compile, link, deploy and execute the SUT and its mutated versions on the mobile device. Our first attempt for improvement the mutation testing on mobile applications began with (1) the mutant generation at Java bytecode level and (2) the implementation of a novel strategy of Mutant Schema using wrappers, thus introducing the same mutations as the classical form. But this first attempt was not successful and is explained in detail in the next section. However, it allowed us to understand in depth mutation testing in mobile applications, reuse the good practices acquired and use the knowledge learned to: improve the generation of mutants in terms of design and form, redefine the Mutant Schema approach, redesign the implementation and architecture of the mutation tool used. In addition, the extensive experimentation, design, and implementation of different approaches and techniques, allowed to build a mathematical model to estimate the execution cost of a mutation testing cycle.

3.1 First Attempt to Improve Mutation Testing on Mobile Applications

3.1.1 Mutant Generation at Bytecode Level

Mutation operators are designed to introduce artificial errors into the SUT, these can be errors at the source code level or at the java bytecode level. In this section, we describe and discuss mutant generation at the bytecode level (code embedded in the *.class* files).

3.1.1.1 *Bytecode manipulation*

BacterioWeb v.1 is a web tool for the mutation testing of mobile applications (Epigraph 3.1.2), which we have developed as an evolution of *Bacterio* [69]. *BacterioWeb v.1* introduces mutants in the Java bytecode of the SUT using the ASM library. ASM is a powerful API to directly manipulate the bytecode produced by the Java compiler [91]. With ASM, a *.class* file can be loaded into a *ClassNode*, an object that wraps the class, holds all the information required to know the wrapped class details and offers all kind of operations to manipulate it. Thus, a *ClassNode* has the collections of fields and methods in two respective lists of *FieldNode* and *MethodNode*. Besides other information (name, annotations, exceptions, etc.), every *MethodNode* has its bytecode instructions in an *InsnList* object, which implements a doubly linked list of instructions.

Every bytecode instruction is an instance of $AbstractInsnNode$, an abstract class with so many specializations as instructions categories showed in Figure 7. *AbstractInsNode* has an *opcode,* which determines the concrete type of operation it performs. This field may take one of the values defined in the constants in the *Opcodes* interface, which represent the Java Virtual Machine assembler instructions.

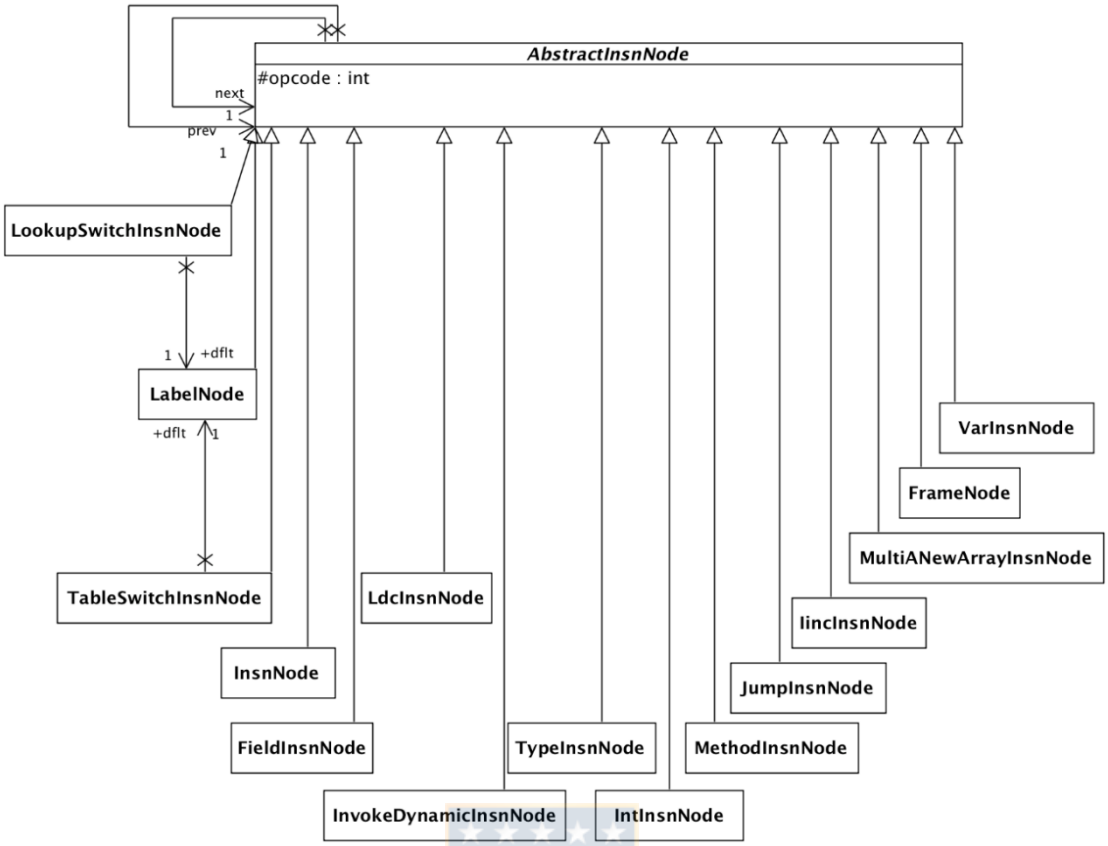


Figure 7. Class hierarchy of instructions

Table 10 shows the Java source code of a simple method of the *Figures* project, that assigns a parameter value to a field and two bytecode translations: one with and another with no debugger information. From the first one we observe that:

- *ALOAD* 0: the first local variable (index 0) is loaded onto the stack.
- *ILOAD* 1: the first parameter of the method is loaded onto the stack. Moreover, it is an integer due to the "*I*" prefix of the instruction name.
- *PUTFILED Figures.i : I* : assigns the *i* field of this *Figures* object the integer value in the top of the stack.

<table>
<tr><th>Source code</th><th>Bytecode with debug info</th></tr>
<tr><td>public void setI(int v) {
this.i = v;
}</td><td rowspan="2">public setI(I) V
L0
LINENUMBER 14 L0
ALOAD 0
ILOAD 1
PUTFIELD Figures.i : I
L1
LINENUMBER 15 L1
RETURN
L2
LOCALVARIABLE this Figures; L0 L2 0
LOCALVARIABLE v I L0 L2 1
MAXSTACK = 2
MAXLOCALS = 2</td></tr>
<tr><td>Bytecode with no debug info</td></tr>
<tr><td>public setI(I)V
ALOAD 0
ILOAD 1
PUTFIELD Figures.i :I
RETURN
MAXSTACK = 2
MAXLOCALS= 2</td></tr>
</table>

Table 10. Source code of a simple setter method and two bytecode translations

Besides having an *opcode,* each instruction is an instance of one of the concrete subtypes of Figure 7. For example:

- *ALOAD* and *ILOAD* belong to the *VarInsnNode* type.
- *PUTFIELD* is an instance of *FieldInsnNode*.
- *RETURN* is of the *InsnNode* subtype.

Besides the inherited *opcode* field, subtypes have additional fields. When we write, for example, *v* + + or *v* - - (being *v* an *int* variable), the compiler translates it into the instruction: *IINC* 1 or *IINC* 1 - 1, where the first argument is the index of the variable to be incremented or decremented, and the second is the amount to be summed to the variable. *IINC* is an instance of the *IincInsnNode*, and its constructor has in fact these two arguments: the index of the variable and the amount. Thus, inserting a pre or post increment or decrement involves: (1) the construction of the adequate *IincInsnNode* with the suitable variable index, (2) the insertion of this object in the right place of the instructions' list (i.e., in the *InsnList*) that is being mutated.

Continuing with the previous example, applying the *Unary Operator Insertion* (*UOI*) to the assignment of *source code* showed in Table 10 , where

v is *integer*, may produce at least five mutants, some of which appear in Table 11. In the bytecode column, we have underlined the change introduced by the Java compiler.

Source code (mutant)	Bytecode mutant	Instruction subclas

`i = - v;`	`ALOAD 0`
	`ILOAD 1`
	`INEG`
	`PUTFIELD Figures.i : I`
	`RETURN`
		`VarInsnNode`
		`VarInsnNode`
		`InsnNode`
		`FieldInsnNode`
		`InsnNode`

`i = v ++;`	`ALOAD 0`
	`ILOAD 1`
	`IINC 1 1`
	`PUTFIELD Figures.i : I`
	`RETURN`
		`VarInsnNode`
		`VarInsnNode`
		`IincInsnNode FieldInsnNode`
		`InsnNode`

Table 11. Some UOI mutants and their bytecode translation (*v* integer)

Although, the set of required instructions to insert the mutation changes depending on the data type of the variable. For example, the same assignment and mutants of Table 11, if the variable *v* is *double*, the changes normally involve more than one instruction. For example, the addition of the unary minus only needs the insertion of the *DNEG* instructions, but the pre and post increments and decrements need four instructions.

For example, *v* ++ requires:

- *DUP2*, to duplicate the two words on the stack.
- *DCONST_1*, to put the 1 number (as a *double*) on the stack.
- *DADD*, to sum 1 and the values in the stack.
- *DSTORE* 1, to store the result in the local variable 1.

Source code (mutant)	Bytecode mutant	Instruction subclas

`i = - v;`	`ALOAD 0`
	`ILOAD 1`
	`DNEG`
	`PUTFIELD Figures.i : D`
	`RETURN`
		`VarInsnNode`
		`VarInsnNode`
		`InsnNode`
		`FieldInsnNode`
		`InsnNode`

`i = v ++;`	`ALOAD 0`
	`ILOAD 1`
	`DUP2`
	`DCONST 1`
	`DADD`
	`PUTFIELD Figures.i : D`
	`RETURN`
		`VarInsnNode`
		`VarInsnNode`
		`InsnNode (92)`
		`InsnNode (15)`
		`InsnNode (99)`
		`VarInsnNode (57)`
		`FieldInsnNode`
		`InsnNode`

Table 12. Some UOI mutants and their bytecode translation (*v* double)

3.1.1.2 *Mutable Instructions and Mutant Generator*

The behavior of a mutant generator may consist in going through every mutation operator and asking it to get the mutants of the class to mutate. Supposing (for the shake of clarity) that only constructors and methods can be mutated, the operator goes through every operation in the class, and, for each operation, it goes in turn over all its instructions to determine whether it can or cannot mutate the method.

Figure 8 shows the pseudocode of an implementation of a *generateMutants(c: Class*) method that belongs to the Operator class: as observed, it adds to a *mutableMethods* collection all the methods in c that it can mutate. For every mutable method, it calls an additional *mutate(c: Class, m: Method)* function, that applies the mutation operator to the method passed as parameter. The behavior described in the pseudocode of Figure 8 is common for all the mutation operators: thus, even though the *Operator* class must be abstract (because the change implementation obviously depends on the self-operator), this operation may be concrete.

The function called by *generateMutants* (i.e., *mutate(c: Class, m: Method)*) goes over the instructions of m and gets the corresponding mutants: if the operator can produce *p* mutants for a given instruction and there are *q* mutable instructions in the method, the operator must generate *p x q* mutants.

```
let be c the class to mutate
let be mutants = ∅
let be mutableMethods = ∅

for each method m in c
   if m is mutable then
      mutableMethods = mutableMethods ∪ { m }
   end
end
for each method m in mutableMethods
   mutants = mutants ∪ mutate(c, m)
end
```

Figure 8. Pseudocode of Operator::generateMutants(c: Class)

Thus, for each mutable instruction in m, *mutate(c, m)* calls *mutate(c, m, instruction)*, that:

1. Gets the list of changes applicable to the *instruction* passed.
2. For each *change*, performs the mutation by calling *performMutation(method, instruction, change)*.

Obviously, both getting the list of changes and performing the mutation depend on the concrete operator.

3.1.1.3 ***Operators Architecture***

We define a reusable architecture to easily implement mutation operators, we have defined an abstract Operator class that holds as many concrete methods as posible. In Figure 9:

- Each operator has two fields: the class file name (which is used to process its bytecode with ASM) and the *family*, which is used to group the operators by categories in the web user interface. Some values of the *family* field can be "Traditional" (in the sense of the classification

given in [26]) or "Android" (meaning that the operator is designed to Android apps).

- Since we want to give the tool a plugin architecture (i.e., new operators can be added, loaded at applied at runtime), the class constructor is protected and is not visible from the outside. To instantiate and load the operators, the tool will look for all the concrete specializations of *Operator* and, over every one, it will call its constructor with a reflective call to its *newInstance* method (inside the *java.lang.Class*).
- *getName* returns the class operator name, and it is the acronym shown in the user interface. For example, if the *AOR* operator is implemented in the *AOR.class* file, it reflectively returns the "AOR" string.
- *getDescription* is abstract, because it returns a textual description of the operator. For AOR, for example, it returns "Arithmetic Operator Replacement".
- Both *mutate* methods implement the tasks described in the previous subsection, and they are concrete.
- *instructionIsMutable* is abstract, since its implementation depends on the concrete operator.
- *performMutation* modifies the method and instruction whose indexes are passed as parameters. The change may be a single instruction (substituting machine instructions *IADD* by *ISUB*, for example) or a list of instructions: thus, the third parameter is a list of instructions (i.e., an instance of *InsnList*). This is the method that builds up each mutant, returning it as a *ClassNode* object with its bytecode.

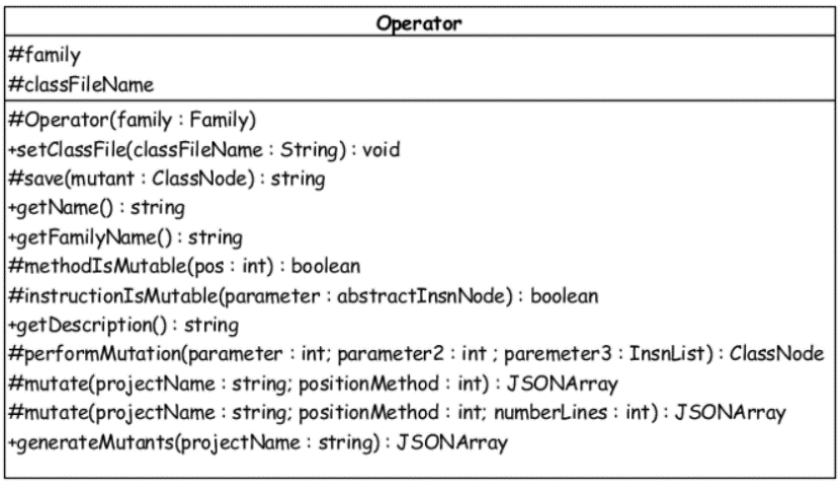


Figure 9. Structure of the abstract Operator

The *Operator* class has three direct, abstract specializations shown in Figure 10 (*InsertionOperator*, *ReplacementOperator* and *DeleteOperator)*.

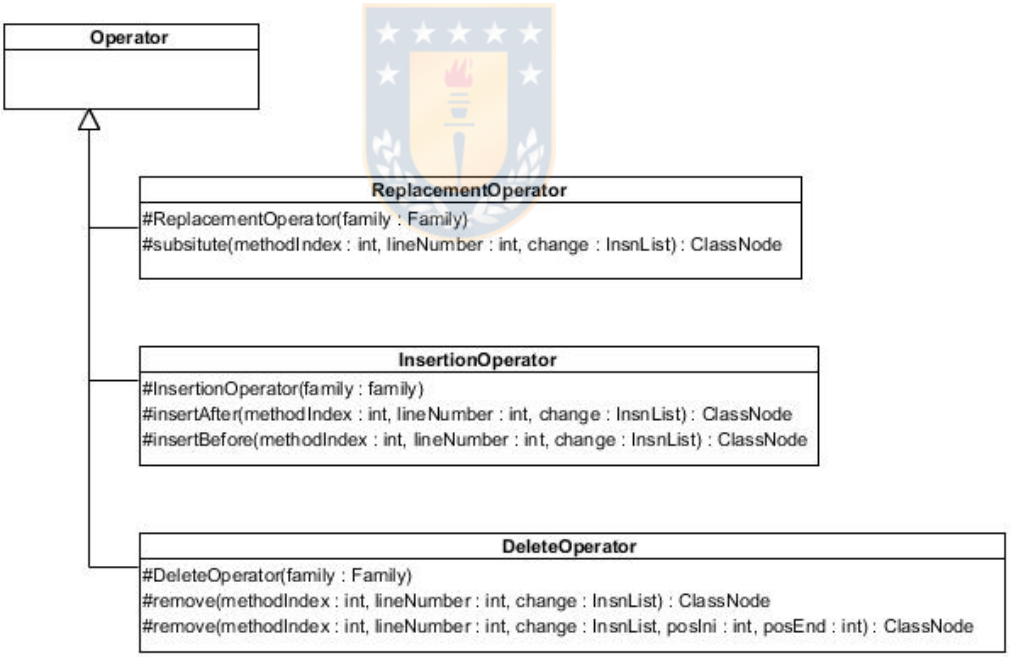


Figure 10. Three abstract specializations of Operator

The implementation of classic operators and Android operators is supported by means of the *methodIsMutable* operation defined in *Operator*, the root of the hierarchy. Overriding this method makes possible to build operators for specific operations of the system under test. *methodIsMutable*,

together to *instructionIsMutable*, are especially useful for the characteristics of mobile applications.

For implementing the mutation operators at bytecode level, is require an external library to manipulate the bytecode. Both inheritance and external libraries increase the system coupling. Inheritance introduces *Content coupling*, probably the most dangerous of all, since the structure and behavior of all subclasses have a complete dependence on all their ancestors; thus, the modification of a superclass affects all its descendants. If the superclass is implemented in a third-party component, then the evolution of our system becomes completely dependent on the evolution of such external system.

Type use is a "not so bad" type of coupling. It occurs when "component *A* uses a data type defined in component B" [92]. If *B* is the external library and this does not evolve according to *A's* requirements, *A* must be modified, maybe with the substitution of *B* by a new library. This type of coupling is better than content coupling because the structure and behavior of *A* is actually implemented in *A* itself, being under the control of *A's* developer.

Due to these risks (ASM is an external library), the development of operators in *BacterioWeb v.1* uses *Type use* coupling (Figure 11) and the dependence on changes of ASM is not as strong as with *Content coupling*.

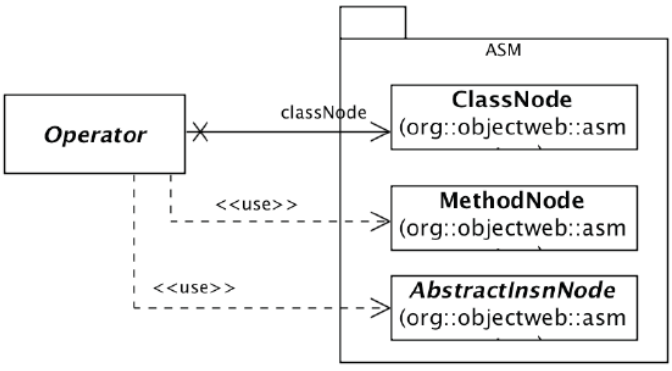


Figure 11. Dependencies of our Operator with respect to ASM

3.1.1.4 *Partial conclusions*

In traditional java projects, it is known that the insertion of the errors in *.class* files avoids a recompilation of the mutated file [26]. For that reason, we believed that in mobile technology, this type of error insertion could reduce the high cost of packaging applications on the device. However, when we applied mutation testing techniques on mobile apps, e.g., Mutant Schema technique, the *.class* files (under test) required to be decompiled into its corresponding *.java* files, so that Android Studio could translate them into *.dex* files. As compilation tasks take the source files as input (*.java* files) and directly generate the *.apk* as output, the mutant generation at bytecode level are not usable for Android testing (we reached the above conclusion after several experiments and implementations).

3.1.2 BacterioWeb v.1: First version of Android Mutation Testing Tool

In mutation testing, the generating, executing, and analyzing the results are costly tasks, so is a necessity the automation. In this sense, we develop *BacterioWeb v.1*, a mutation testing tool for mobile applications. *BacterioWeb v.1*, runs on a web server and can be executed with any browser. The user (a tester) uploads his/her Android projects to the server. Figure 12 describes, as use cases, the main functionalities of the tool and its relationships with external actors: the projects are saved in MongoDB databases (therefore making the projects accessible from any place) and in the server's local file system. The tool oversees generating the mutants, composing the Mutant Schema (if required by the user), compiling the mutated system, executing the test cases, and collecting the results.

The communication with the external devices (physical devices or emulators) is achieved by means of the creation of operating system processes. For example, the execution of a command on a device requires

creating an operating system process for sending the order. All communication between the frontend and the backend is based on the *websocket (ws)* protocol to keep the user informed of the progress of mutant generation, test execution, etc.

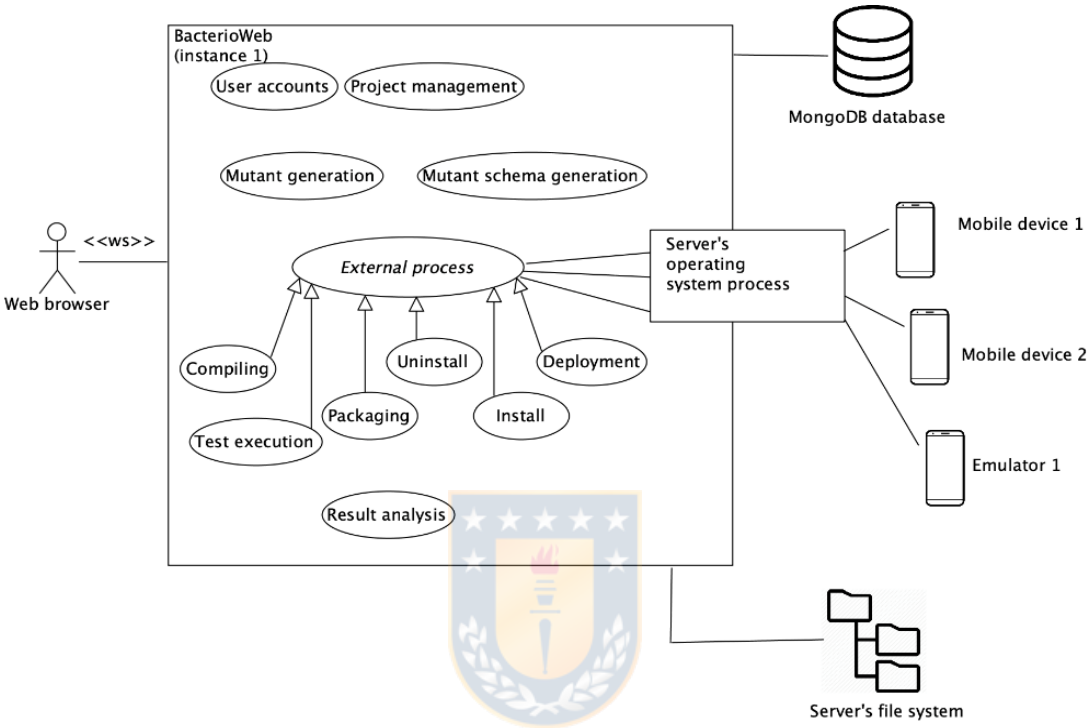


Figure 12. General structure of *BacterioWeb v.1*

Moreover, and to push forward the parallelism, *BacterioWeb v.1* may communicate with other instances of *BacterioWeb v.1* running on other servers and this in turn helps the use of the devices they are not using at some point: in Figure 13, instance 1 can send mutants and test cases for execution on the devices connected to instances 2 and 3.

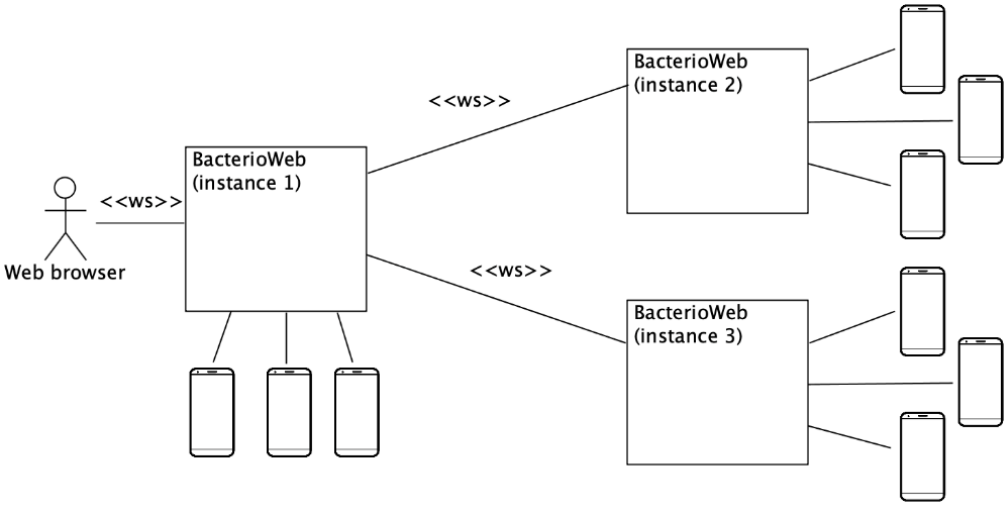


Figure 13. Collaboration among different instances of BacterioWeb v.1

3.1.3 Mutant Schema Using Wrappers (MSW)

This approach systematizes the creation of the Mutant Schema using Wrappers with a set of relatively simple transformations of the original program structure. This section gives all implementation details about both the creation of the Mutant Schema structure and the test and mutant execution control.

3.1.3.1 *Structure of the MSW*

To illustrate the structure, let us consider the simple system shown in Figure 14: it contains an implementation of the classic *Triangle* class (which in this case is the class under test) and a *ManualTestCases* class, which holds two test cases.

When mutants are generated, they are placed in a specific *mutants* package and numbered from 1 to *n*, so being *Triangle_1*, *Triangle_2*, etc. the mutants of the class under test. Additionally, a slightly modified, *adapted copy* of the original Triangle class, called *Triangle_0*, is placed in an *originals* package (Figure 15).

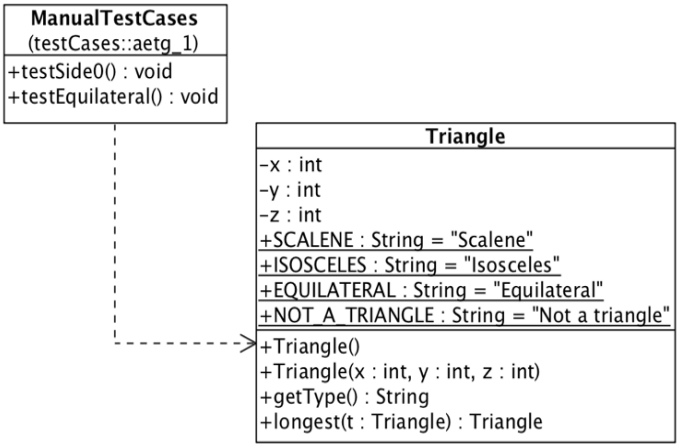


Figure 14. A simple system

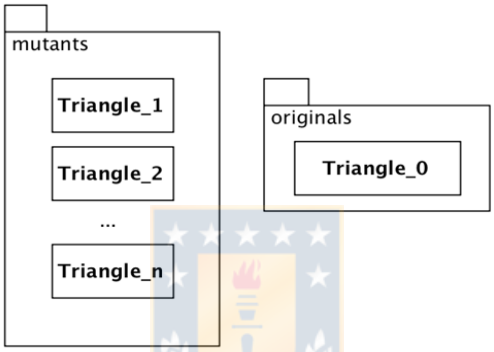


Figure 15. Mutants and originals packages

Test cases (those located at the *ManualTestCases* class in Figure 14) call the methods offered by the original *Triangle* class. Since now the logic has been moved to the copies (*Triangle_0* to *Triangle_n)*, the original *Triangle* class is replaced by a wrapper (also called *Triangle)* that will forward the calls from the tests to the adequate mutant. To avoid compilation problems, this wrapper offers the test suite the same methods and constructors, with the same signature. So, the two constructors and the two methods are included in the wrapper, but obviously with a different implementation: (1) constructors, instead of directly assigning the parameter values to the instance fields, create an instance of the corresponding mutant version; (2) in the same way, *getType* and *longest* call the homonym methods

in the mutant. An exception to this *same-signature rule* is in those methods that include the class under test in their return type or in their parameter types, such as *Triangle longest(Triangle t)*: in order to preserve the compatibility with test cases and with the whole system, these types are changed to *Triangle_Mutant,* an interface that is explained below.

The wrapper asks for the creation of every mutant to a *Triangle_Controller* class (Figure 16), which holds as many *createMutant* methods as there are constructors in the class under test. Each *createMutant* method returns the adequate version of *Triangle* depending on the value of its *currentMutant* field. This value is updated by *loadCurrentMutant()* according to the value saved in the *mutantFileName* file. This file is explicitly required by mobile applications: during the iterations that launch the tests against the original and the mutants, a separated Android process creates this file and, then, a second process launches the test cases. In this way, we avoid modifying the application permissions for writing and reading the internal storage of the device.

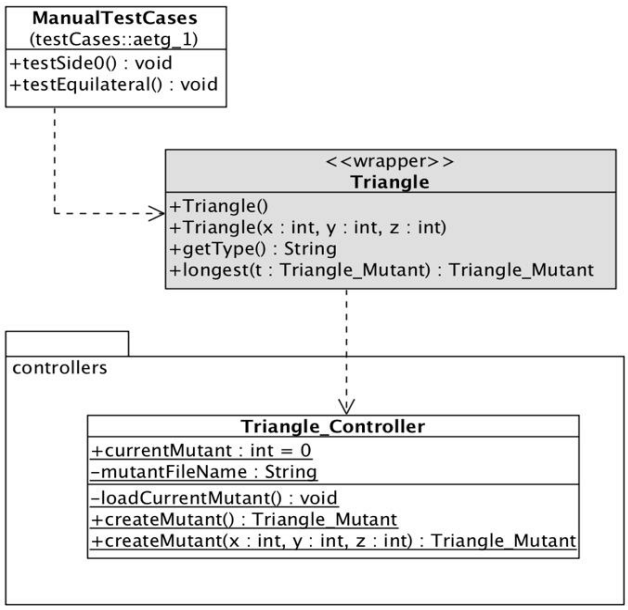


Figure 16. Inclusion of a controller

As we have said, when a test case asks for the creation of a *Triangle*, it still continues calling the *Triangle* constructor that, in turn, calls the corresponding *createMutant* method in the controller. As shown in Figure 17, *createMutant* returns an instance of the *Triangle_Mutant* interface according to the value saved in the *mutantFileName* file. *Triangle_Mutant* is implemented by the wrapper (*Triangle)*, by all the mutants (*Triangle_1…)* and by the *adapted copy* of the original class (*Triangle_0)*. In this way, the wrapper indirectly knows the actual mutant instance by means of the interface, which has been instantiated by the controller: the association from *Triangle* to *Triangle_Mutant* in Figure 17 points to that actual instance of the mutant.

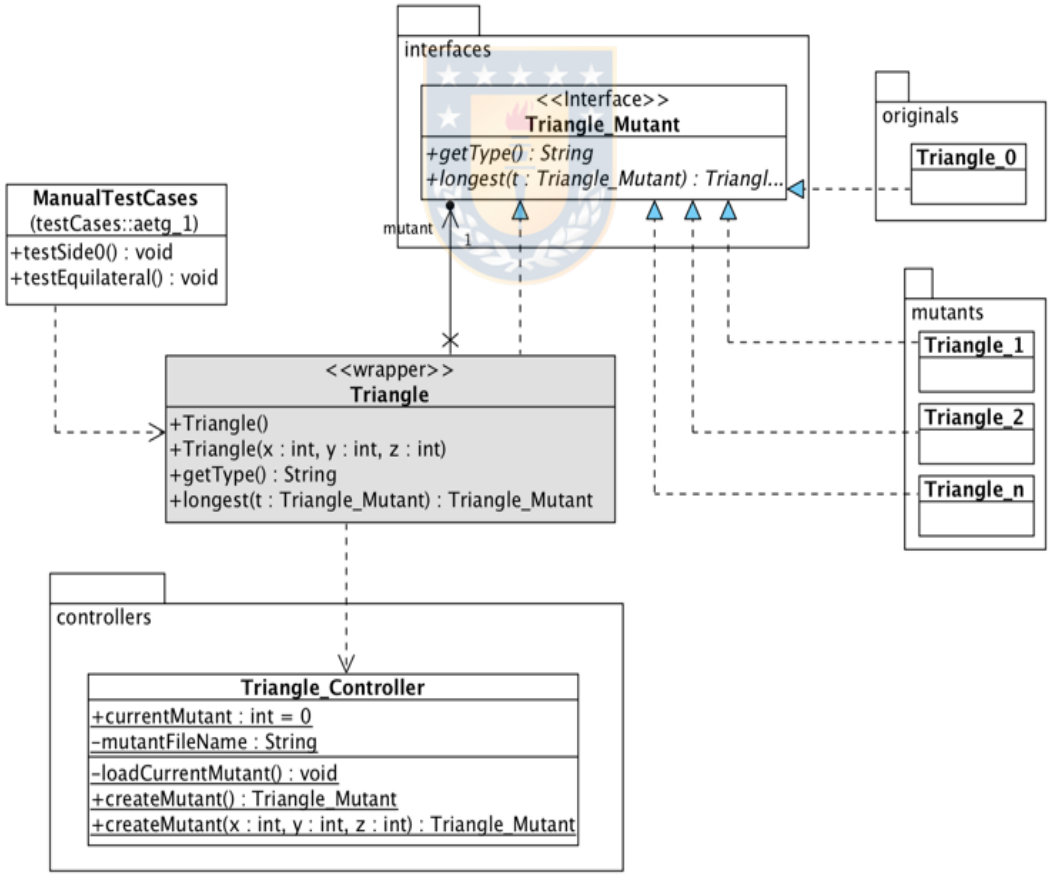


Figure 17. Final structure of the Mutant Schema

3.1.3.2 *Implementation Details*

For the implementation of MSW we use the ASM library [91], which allows the direct manipulation of the bytecode of compiled Java files. With ASM, a class is handled as a tree via a *ClassNode*, which offers an API for accessing and changing the wrapped class. Below we explain how to get the structure shown in Figure 17.

3.1.3.3 *Creating the Interface*

An interface is created for every class under test (the *Triangle_Mutant* interface, in our example). All interfaces have the same structure and, so, they proceed from the template shown in Figure 18, which has two tokens:

- *#CLASS#* is replaced by the class under test name.
- *#METHODS#* is substituted by the signature of all the methods in the class under test, preserving in most cases the return type, parameters, and exceptions. It is important to note that all type names will be fully included and, therefore, no *import* statements are required.

```
package edu.mutantSchema.interfaces;
public interface #CLASS#_Mutant {
   #METHODS#
}
```

Figure 18. Interface template

Thus, the *Triangle_Mutant* interface remains as in Figure 19. The *longest* method illustrates how to deal with methods whose return type or some parameter type coincides with the class under test (*Triangle)*: it compares the perimeter of *this* triangle with another passed as parameter,

returning the longest one. Observe that the return and the parameter types (which were *Triangle* in the original) have been changed to *Triangle_Mutant*.

```
package edu.mutantSchema.interfaces;
public interface Triangle_Mutant {
   String getType() throws Exception;
   Triangle_Mutant longest(Triangle_Mutant t);
}
```

Figure 19. Triangle_Mutant interface

3.1.3.4 *Creating the Adapted Copy of the Classes Under Test*

Once the interface has been created, we generate an almost exact copy of the original class, which is placed in the *originals* package. The changes introduced are detailed in Figure 20: (1) the declared package is changed to *originals;* (2) it implements the corresponding interface (*Triangle_Mutant* in the example); (3) the class name is suffixed with *_0*, also changing the constructor names and the types of the local variables whose type is the class under test; and (4) in methods, the return types and parameters whose type is the class under test are changed to the interface (see the longest method in Figure 20).

```
package edu.mutantSchema.originals;

public class Triangle_0 implements
Triangle_Mutant {
  private int x, y, z;
   ...

  public Triangle_0(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public String getType() throws Exception {
    ...
  }

  public Triangle_Mutant
longest(Triangle_Mutant t)
       throws Exception {
    ...
  }
```

```
}
```

Figure 20. Copy of the original (Triangle_0)

3.1.3.5 *Creating the Controller*

There is a controller for each class under test. They have the same name than the original class, but suffixed with *_Controller*, and are placed in the *controllers'* package.

Controllers are generated from the template shown in Figure 21:

- The *#CLASS#* token is replaced by the original class name.
- *#CURRENT_MUTANT_FILENAME#* is replaced by a file name, which saves the *id* (a number from 0 to *n*) of the mutant to be executed. The location of this file depends on the type of project, for mobile projects, it is saved directly on the device, so that the *.apk* (actually, the *loadCurrentMutant*) can access the file at runtime. The mutant *id* is unique, even for different classes under test of the project.
- *#CREATE_MUTANTS#* is replaced by so many static *createMutant* methods as constructors there are in the class under test, with or without parameters depending on the constructor signature. This method executes the *loadCurrentMutant()* method, which reads the mutant *id* from the *mutantFileName* mentioned: depending on the read value, it returns the corresponding mutant instance.

```
package edu.mutantSchema.controllers;

import java.io.FileInputStream;
import edu.mutantSchema.interfaces.#CLASS#_Mutant;
import edu.mutantSchema.mutants.*;
import edu.mutantSchema.originals.#CLASS#_0;

 public class #CLASS#_Controller {
   public static int currentMutant = 0;
   private static String mutantFileName=
        "#CURRENT_MUTANT_FILENAME#";

   private static void loadCurrentMutant(){
       try{
        FileInputStream fis=
       new FileInputStream(mutantFileName);
        byte[] b = new byte[fis.available()];
        fis.read(b);
        fis.close();
        String s=new String(b);
        currentMutant=Integer.parseInt(s.trim());
       }
       catch(Exception e){ currentMutant =0; }
   }

#CREATE_MUTANTS#
}
```

Figure 21. Controller template

Since the *Triangle* class has two constructors (one with three parameters and another one with none), the replacement of the tokens produces the file shown in Figure 22: note the presence of two *createMutant* methods, that return either the instance corresponding to the mutant read, or the original (*Triangle_0)* if the mutant *id* does not match with any value in the *switch* statement. The *mutantFileName* points to a location in the mobile device (*/data/local/tmp/currentMutant.txt*).

```
package edu.mutantSchema.controllers;

import java.io.FileInputStream;
import edu.mutantSchema.interfaces.Triangle_Mutant;
import edu.mutantSchema.mutants.*;
import edu.mutantSchema.originals.Triangle_0;

 public class Triangle_Controller {
   public static int currentMutant = 0;
   private static String mutantFileName=
      "/data/local/tmp/currentMutant.txt";

   private static void loadCurrentMutant(){
       ...
   }

   public static Triangle_Mutant createMutant() {
      loadCurrentMutant();
      switch (currentMutant) {
         case 1 : return new Triangle_1();
         case 2 : return new Triangle_2();
         ...
      }
      return new Triangle_0();
   }

   public static Triangle_Mutant createMutant(
                     int x, int y, int z) {
      loadCurrentMutant();
      switch (currentMutant) {
         case 1 : return new Triangle_1(x, y, z);
         case 2 : return new Triangle_2(x, y, z);
         ...
      }
      return new Triangle_0(x, y, z);
   }
}
```

Figure 22. Wrapper generated for Triangle

3.1.3.6 *Creating the Wrapper*

Wrappers substitute the mutated classes. They offer test cases the same operations than the original classes, but they forward calls to the corresponding controller which, in turn, send them to the respective mutant.

For creating a wrapper, we use the ASM library to analyze the bytecode of the class under test and, then:

- We preserve the package of the original class, reading it from the *ClassNode.name* field, which proceeds from the *.class* original file.
- The wrapper implements the generated interface (*Triangle_Mutant* in the example).
- All the non-static fields of the original class are removed.
- An additional *mutant* field is added, whose type is the interface. This field is initialized in the constructor, calling the *createMutant* included in the controller.
- We add to the wrapper a method for each one of the methods in the original class with the same signature, but whose body is replaced by calls to the homonym methods of the mutant class. Moreover, if the method return type or any of its parameters' types are of the class under test, then they are changed to its interface.

Figure 23 shows the code of the *Triangle* wrapper: (1) the addition of the *Triangle_Mutant* field; (2) how this field is instantiated by means of calls to *Triangle_Controller* in the constructors; (3) *getType* preserves the signature in the original class; and (4) the *longest* method signature has been adapted from *Triangle longest(Triangle t)* to *Triangle_Mutant longest(Triangle_Mutant t).*

```
package demos.com.moderntritype.domain;

import edu.mutantSchema.interfaces.Triangle_Mutant;
import edu.mutantSchema.controllers.Triangle_Controller;

public class Triangle implements Triangle_Mutant {
  Triangle_Mutant mutant;

  public Triangle() {
    this.mutant=Triangle_Controller.createMutant();
  }

  public Triangle(int x, int y, int z) {
    this.mutant=Triangle_Controller.createMutant(x,y,z);
  }

  public String getType() throws Exception {
    return this.mutant.getType();
  }

  public Triangle_Mutant longest(Triangle_Mutant t)
          throws Exception {
    return this.mutant.longest(t);

  }
}
```

Figure 23. Wrapper for Triangle

3.1.3.7 *Test Execution*

In the original project, test cases send call the methods offered by the classes under test. After generating the Mutant Schema structure, these calls are received by the wrappers, which forward them to the controllers. In this way, there are no compilation problems at all.

Test cases are executed against the *n* versions of the system under test (original and mutants). To tell the test suite which mutant must be executed, the *mutantFileName* file with the *id* of the corresponding mutant is created before launching the test suite.

The pseudocode in Figure 24 illustrates how test cases are executed: each iteration writes the *id* of the mutant in the afore mentioned file, launches the test cases, collects the results of the execution and completes the killing matrix. The value saved in the *mutantFileName* file is read by the

controller's *loadCurrentMutant()* method: depending on its value, the controller instantiates either the copy of the original class (*Triangle_0)* or the suitable mutant (*Triangle_1*, etc.).

```
km : KillingMatrix = ∅
for i=0 to |mutants|
   write i in mutantFileName
   results = execute tests against the i-th
mutant
   build(km, results)
next
```

Figure 24. Pseudocode of the execution engine

In mobile projects, the execution process is launched from the server to the device by opening an Android console (through the *adb* command). In the first iteration, an Android process saves a 0 (for *Triangle_0)* in the *mutantFileName* file and, with a new Android process, launches the test cases against the original (*Triangle_0).* When the test suite execution has finished, the output of the console is processed to fill in the killing matrix. The process continues with every mutant, saving *1, 2…* and instantiating *Triangle_1*, *Triangle_2...*

3.1.3.8 *Disadvantages of Mutant Schema Using Wrappers*

- **Preprocessing Classes with Inheritance**

  This approach requires a preprocessing classes with inheritance, as long as the classes under test (mutated classes) have inheritance. Suppose our system has an abstract *Figure* superclass with two *Triangle* and *Quadrilateral* specializations (Figure 25). Remind that the fields of the class under test are removed in the wrappers and substituted by a *mutant* field (whose type is its corresponding interface). If the superclass is mutated, subclasses will not be able to access the attributes

in the superclass. So, *get* and *set* methods must be included in the superclass. Moreover, in the subclasses, direct accesses to the superclass fields are replaced by calls to the corresponding *get* and *set* methods.

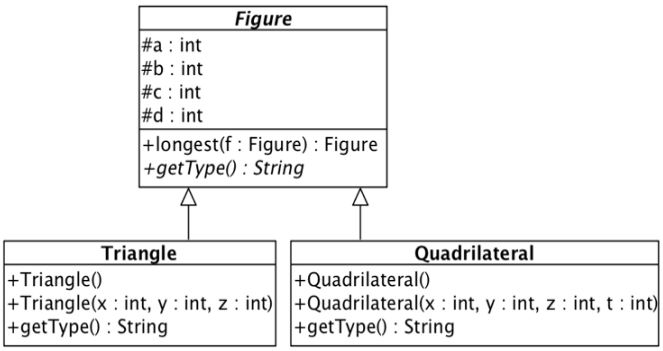


Figure 25. A simple system with inheritance

Therefore, before creating the Mutant Schema, a preprocessing of the system may be required only when a superclass is selected to be mutated:

- Superclasses are preprocessed for ensuring that they have *get* and *set* methods for accessing all their attributes. This task is done using the ASM library.
- In subclasses, direct accesses to the superclass attributes are replaced by calls to the corresponding *get* or *set* methods, also manipulating the bytecode with ASM.
- In addition, all methods of the superclass must be reproduced in their subclasses. Hence, inheritance is not preserved in the mutated system.

Figure 26 shows the result of the constructor preprocessing in one subclass. Methods *setA*, *setB* and *setC*, which did not exist in the superclass, have also been added.

Before

```
public Triangle(int x, int y, int z) {
  this.a=x;
  this.b=y;
  this.c=z;
}
```

After

```
public Triangle(int x, int y, int z) {
  this.setA(x);
  this.setB(y);
  this.setC(z);
}
```

Figure 26. The Triangle, before and after the preprocessing

In short, this implementation strategy of the Mutant Schema has the main disadvantage that it does not preserve the object-oriented characteristics of the original program (inheritance, polymorphism, etc.).

Some of the newest characteristics of Java programs have not been tested in depth, such as the *lambda* expressions.

3.1.3.9 *Partial conclusions*

Mutant Schema using Wrappers (MSW) is a different implementation strategy of Untch's idea [72]. MSW is independent on the type of operator used, so it can be combined with other complex mutation operators and other techniques, such as Higher order mutants. MSW implementation is time consuming, but tracking mutations is easy (e.g., analysis of equivalent mutants). This implementation of Mutant Schema, introduce the same mutations as the classical form, since a copy of the SUT is created for each mutation (i.e: Triangle_1, Triangle_2) (see Epigraph 3.1.3.1) and therefore the same number of trivial mutants as the classical mutation. However, this approach has two drawbacks: (1) the processing of the classes with inheritance of the SUT and (2) the mutant schema structure is implemented by manipulating the bytecode (therefore, to apply it in mobile applications,

the manipulated *.class* files should be decompiled into *.java* files). For these reasons, we develop another approach to the Mutant Schema (see Section 3.2.2).

3.2 Successful Approach to Mutation Testing on Mobile Applications

3.2.1 Mutant Generation at Source Code Level

A recent study by Hariri et al. [13] has shown that mutation testing at source level produces much fewer mutants than at bytecode level, so making test execution less expensive.

For this approach we implement two mutant generation engines:

- The "classic engine" creates as many copies of every mutated file as mutants there are.
- The "Untch's engine" groups all the mutants corresponding to a file in a *metamutant* file, according to the Untch's mutant schema idea [72].

The tester selects the files to be mutated in the Project screen Figure 27. At this point, if she/he clicks the *Generate mutants* button, the tool shows the screen for choosing the mutation operators and generating the mutants.

We discuss in [88], the design of the operators of several mutation tools: one of the main problems highlighted is the dependency of the internal operators' design with respect to the libraries used for parsing the source of the bytecode of the system under test. In *BacterioWeb v.2*, the architecture of the generation engines has been carefully designed to avoid coupling with respect to external libraries and make the tool easily extensible thinking both in the easy addition of new operators as in the processing of other programming languages, such as Kotlin.

This decoupled design is shown in Figure 28: the tester (using her/his user agent) asks for the *WSMutantsGeneration websocket* to generate the mutants, sending in the message the selected operators and files. The

websocket passes in turn the order to the CWS's Manager, who in turn instantiates the two mutant generators (classic and Untch's).

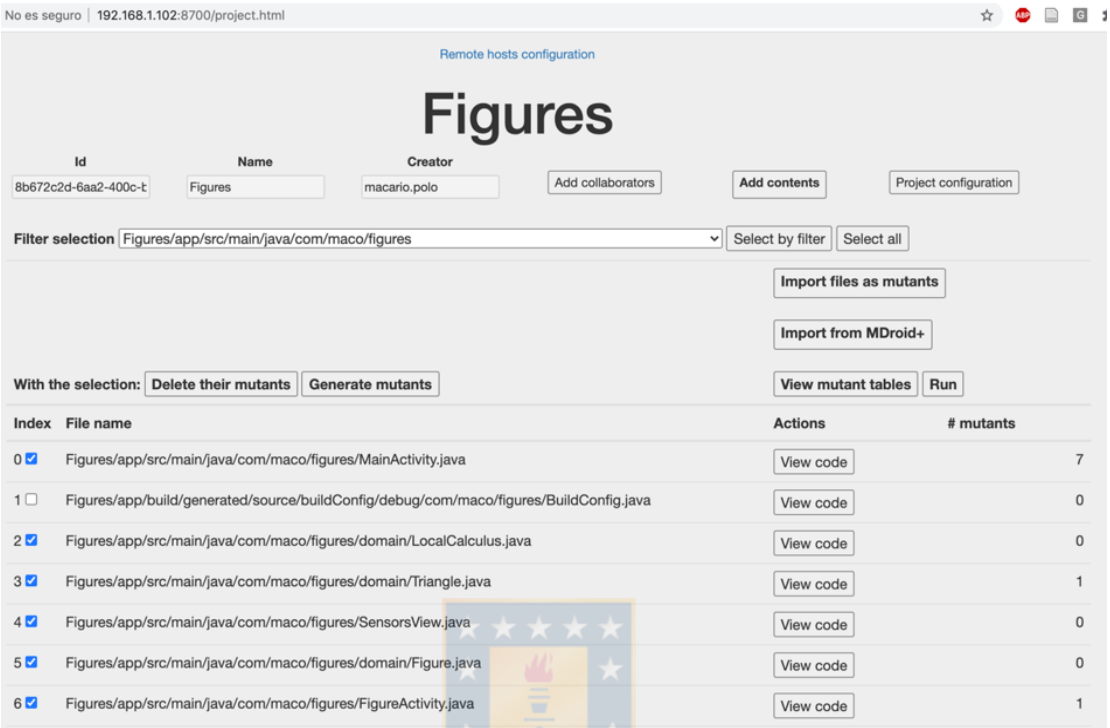


Figure 27. The Project screen is used to generate mutants, run tests, etc.

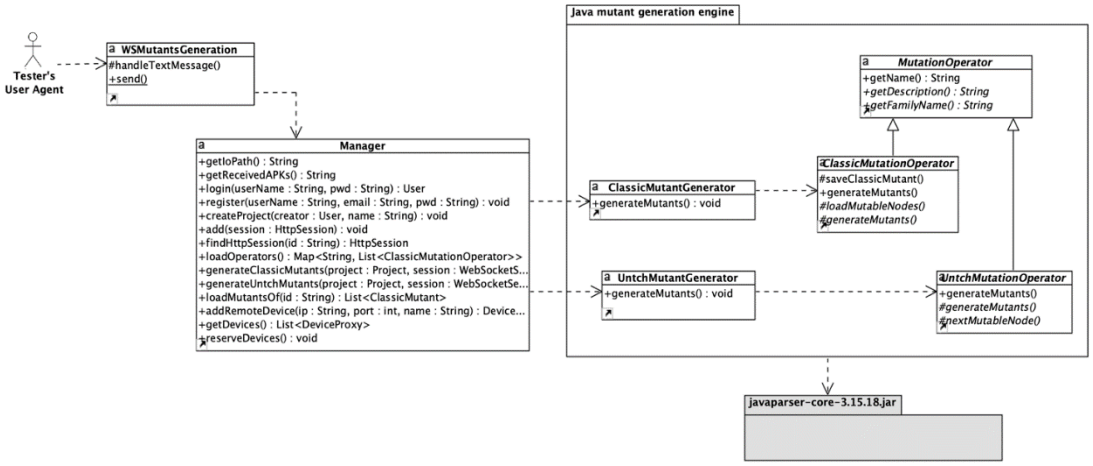


Figure 28. Decoupled design of the mutant generation engine

The processing of the Java source files is made with the *javaparser* library [93]. As it can be seen in the Figure 28, only the classes in the package labelled as Java mutant generation engine have dependency from this third-party library.

ClassicMutationOperator and *UntchMutationOperator* are abstract classes that have so many concrete specializations as mutation operators are implemented. Figure 29 shows an excerpt of them. Note there is a parallel structure in the specializations of both superclasses, although it is not mandatory that all the operators under Classic are also implemented under Untch: for example, there exist *AOR* and *UOI* in both subsystems, but the *NotParcelable* operator [14] (removes 'implements Parcelable' and '@Override' annotations) has only classic implementation.

Mutation operators are loaded at runtime using the Java Reflection API. Thus, if new operators are added, *BacterioWeb v.2* will load and show them without touching its code.

BacterioWeb v.2 iterates on the selected mutation operators and on the selected files. For each source file, *BacterioWeb* v.2 recovers its AST (abstract syntax tree) from the database. Then, it checks whether the current operator is applicable:

- If the operator is "classic", it performs a change in the AST, saves the resulting code in the *classic_mutant* table and, after, undoes the change to continue with the next mutable statement (if the change is not undone, the last mutant would have accumulated all the changes introduced).
- For "Untch's" operators, firstly a copy of the AST of the original file is made. If the operator is applicable to a given statement, the generator replaces the statement in the AST by a new

statement consisting in a call to the corresponding operation of the *MutantDriver* (Section 3.2.2).

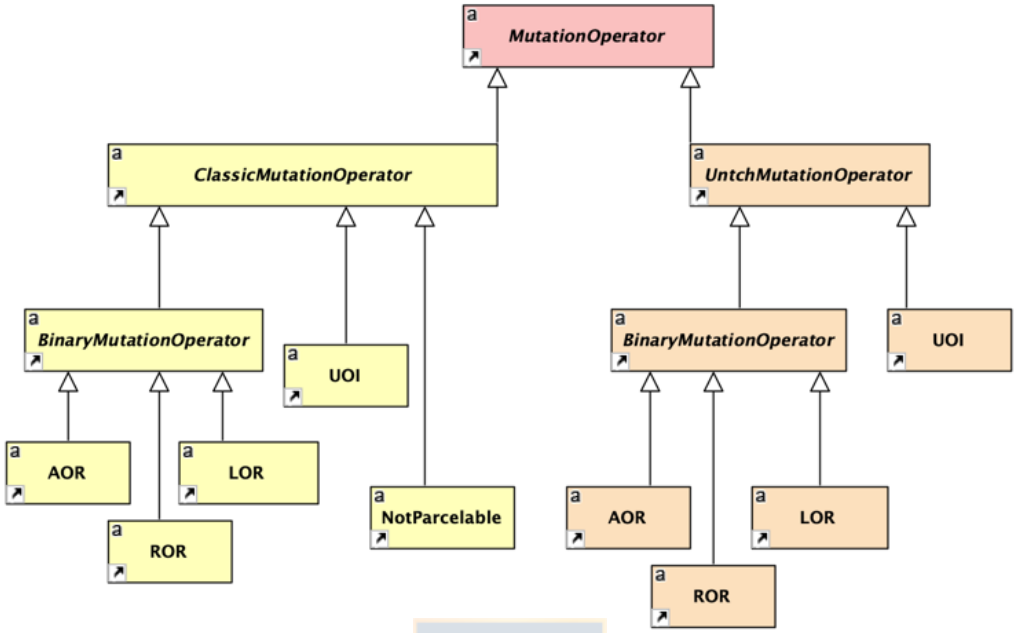


Figure 29. Hierarchical structure of the operators

As an example, Figure 30 shows the code of the *generateMutants* method in the *Unary Operation Insertion* Untch's mutation operator: the first parameter is a Node containing a mutable expression according to this operator; the second one is the counter of mutants generated for this project.

Suppose that *node* contains the expression *x* (which is an *int* variable): in its second line, the method recovers the unary operators applicable to this expression (since *x* is a number, the operators are the *unary minus*, prefix increment, prefix decrement, postfix increment and postfix decrement). Then, it builds a string (the *sbTransformation* variable) consisting in a call to the *MutantDriver.UOI* operation: the first parameter of this call is the original expression (*x*), and it adds as remaining parameters the counter of mutants generated (so many as operators). The built string (*newExpr*) is, for this example, *MutantDriver.UOI(x, 27, 28, 29, 30, 31)*, where the numbers

27 to *31* will represent, when the tests are executed, the current mutant under execution.

Finally, *newExpr* is translated (using the *javaparser* library) into a valid *Expression* of the abstract syntax tree. After all operators have been applied to this file, it is saved in the *untch_mutant* table of the database.

```
@Override
protected Expression generateMutants(Node node, int[] projectCounter) {

   Expression originalExpression = (Expression) node;

   Operator[] otherOperators = getOtherOperators();

   StringBuilder sbTransformation=
        new StringBuilder("MutantDriver.UOI(" +
             originalExpression.toString() + ", ");

   for (int i=0; i<otherOperators.length; i++)
        sbTransformation.append((++projectCounter[0]) + ", ");

   String newExpr=sbTransformation.substring(0,
        sbTransformation.length()-2) + ")";

   Expression mutatedExpression =
        StaticJavaParser.parseExpression(newExp);
   return mutatedExpression;
}
```

Figure 30. Implementation of the UOI operator in Untch's

In addition, we have added a new mutant import function. To our best knowledge, *BacterioWeb* v.2 is the only mutation tool offering this interesting functionality (***Import of mutants***). Other tools only deal with the mutants that they generate themselves or, at least, do not have any automated process to incorporate third-party mutants.

BacterioWeb v.2 has a special importer for MDroid+ [8], the tool generates Android-specific mutants, and a general importer for uploading versions of a given file. When MDroid+ generates mutants, it creates a folder per mutant. In this folder there is an exact copy of the original application, only differing in the file mutated (Figure 31).

The *log* file generated contains a description of every mutant that includes the mutant index, name of the mutated file, operator applied and

line number. Our MDroid+ importer analyses this file and uploads to the database only the mutated files, including the operator's name (see the *classic_mutant* table in the database schema in Figure 40).

The "general importer" takes a folder as input, searches all the source files it contains and, if the file name coincides with some of the project files, adds it, as a mutant, to the database. In this case, the mutation operator column is set to "Imported".

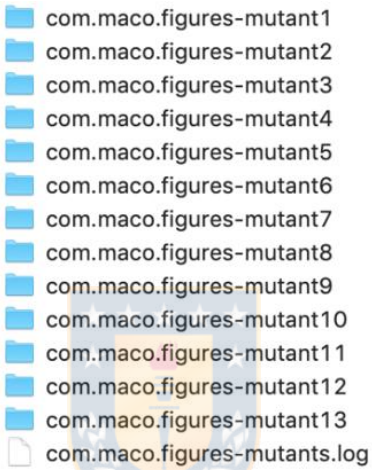


Figure 31. Mutants generated by MDroid+

As an example, Figure 32 shows the count of mutants per operator for a sample application: ROR, UOI and LOR mutants have been generated by *BacterioWeb v.2*; MuJava has generated the 22 mutants labelled as "Imported"; the remaining ones proceed from MDroid+.

	count(*)	operator
▶	205	ROR
	100	UOI
	22	Imported
	12	LOR
	3	NullValueIntentPutExtra
	1	NullIntent
	3	InvalidKeyIntentPutExtra
	1	DifferentActivityIntentDefinition
	1	NotParcelable
	2	LengthyGUICreation

Figure 32. Mutants for a sample application

3.2.2 Untch Mutant Schema (UMS)

The Mutant Schema are generated from the source code following the Untch's idea [72]. This approach creates *metamutants* that contains all the mutants in a single file and *metaprocedures* that have the different changes that the mutation operators can introduce in the program. This approach was adapted for Android mobile apps.

Every Java source file is processed with the *javaparser* library [93]. This library builds the abstract syntax tree (AST) of each processed file. In the relational database we save the source code and the serialized AST, using the own *javaparser* serialization functionalities.

The Mutant Schema generator iterates trying to apply each selected mutation operator to the considered file. For example, the traditional *AOR* operator takes all the binary expressions in the file and, if the corresponding operator is +, -, *, / or %, modifies the original statement by a call to *MutantDriver.X*, where *X* is the name of original operator.

Consider the statement *a+b+c:* in prefix notation it can be written as *+(a, +(b, c))*. If we substitute the operator by a call to a *PLUS* method in a *MutantDriver*, the statement can be rewritten as:

MutantDriver.PLUS(a, MutantDriver.PLUS(b, c))

In this case, every operator can be replaced by the other four operators. This is: + is replaced by -, *, / and %; – is replaced by +, *, / and %, etcetera.

Suppose the first mutated binary expression is *b+c*. In order to represent the four possible mutants, the statement *MD. PLUS(b, c)* will be written as:

MutantDriver.PLUS(b, c, 1, 2, 3, 4)

The four indexes (1-4) reference the *mutant index*.

Then, the second binary expression (*a+b+c*, that now is *a+ MutantDriver.PLUS(b, c, 1, 2, 3, 4)*) is mutated. The whole expression remains as follows:

MutantDriver.PLUS(a, MutantDriver.PLUS(b, c, 1, 2, 3, 4), 5, 6, 7, 8)

The *MutantDriver* implements the *PLUS(int x, int y, int… indexes)* metaprocedure as the method shown in Figure 33: the first two parameters are the numbers to be added; the others are the indexes of the applicable mutants (note that Java allows to pass a variable number of parameters with the suspension points, which can be processed as an array).

Consider the following situations:

a. We are executing the test suite against the original program, which has 0 as mutant index: in this case, the implementation of *PLUS* reads the value of *currentMutant* from a file. Since its value is zero, it returns the result pointed by the first *if: a+b*, which is the same than in the original program.
b. We are now executing the mutant index with value 3, that is the 3rd mutant proceeding from *b+c*. This value has been saved in the aforementioned file and is assigned to *currentMutant* in the *loadCurrentMutant* method. This value (3) is searched in the array passed in the variable parameters set (it was [*1, 2, 3, 4]*) and found with *location=2*. Then, the method returns *a/b*.

c. If we are executing the test suite against the mutant index number *100*, since this value is not found in the array, the method returns the expression in the first *if*.

```
public static int PLUS(int a, int b, int... indexes) {
  loadCurrentMutant();
  int location =
        Arrays.binarySearch(indexes, currentMutant);

  if (currentMutant == 0 || location < 0) return a + b;
  if (location == 0) return a - b;
  if (location == 1) return a * b;
  if (location == 2) return a / b;
  if (location == 3) return a % b;
  return a + b;
}
```

Figure 33. Implementation of PLUS in the MutantDriver

As an additional example, Figure 34 shows the implementation we have given to the *ITR* mutation operator described by Deng et al. [15] and by Escobar-Velásquez et al. [14].

```
public static Intent ITR(Context ctx, Class<?> activityClass, int index) {

    loadCurrentMutant();
    if (index == currentMutant)
        return new Intent(ctx, Activity.class);
    return new Intent(ctx, activityClass);
}
```

Figure 34. Implementation of the ITR operator in the MutantDriver

The *ITR* operator replaces the target activity of an *Intent*. Consider the statement:

Intent i = new Intent(this, MainActivity.class)

Supposing the mutant to be generated is the 15th, the statement is changed to:

Intent i = MD.ITR(this, MainActivity.class, 15)

When the mutant is to be applied (i.e., *currentMutant==15)*, the program will behave as if the programmer would have written:

Intent i = new Intent(this, **Activity.class**)

3.2.2.1 *Disadvantages of UMS*

- **Triviality of schema mutants**

Often, schema mutants are much easier to kill than traditional ones. Consider the small, selected, fragment of the *Kuar* app (one of the projects used in our experiments) shown in Figure 35: a *SlidingBoard* holds a collection of *Square* instances.

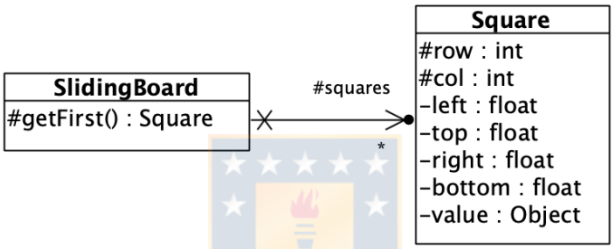


Figure 35. A small excerpt of the Kuar design

Figure 36 shows a small piece of code of the *getFirst* method of the *SlidingBoard* class. Note that it takes the *Square* instance located at the coordinates *(row, col)* and, if it is not *null*, reads its value and casts it as an *Integer*. Since the whole decision is evaluated in short-circuit, if the instance is *null*, the second condition is not evaluated.

```
if (squares[row][col].getValue() != null &&
  (Integer) squares[row][col].getValue() == 1) {
        ...
}
```

Figure 36. A fragment of the code in the SlidingBoards' getFirst method

The statement in Figure 36 works rightly both in the original program as in a classic mutant that replaces, for example, the && by applying the *LOR* operator.

Consider however the effect produced by the same operator with Mutant Schema, that appears in Figure 37: the original infix expression (with the form *A && B)* is translated into a prefix expression *MutantDriver.AND(A, B, 2),* where the last value references the mutant index.

In this second case, the decision is not evaluated in short-circuit and, then, both conditions are evaluated. Suppose that the *Square* instance is *null:* in the first time, the instance is compared to *null.* With independence of the result returned, the second condition is checked and, since the instance is *null*, the program cannot cast it to an *Integer* and crashes, producing what is usually known as a *trivial mutant* [14].

According to Escobar-Velásquez et al. [14], a *trivial mutant* is a mutant that always or frequently crashes at runtime. Trivial mutants introduce "noise" in the result analysis phase and may lead to misinterpret the mutation score.

```
if (MutantDriver.AND(squares[row][col].getValue()!= null,
       (Integer) squares[row][col].getValue() == 1, 2)) {
         ...
}
```

Figure 37. One of the schema mutants of getFirst in SlidingBoard

In general, these operators are more frequent with schema than with traditional mutants, although Deng et al. [15], who do not use Mutant Schema, also are aware of the problem they represent ("we need to make our tool generate fewer mutants that immediately crash […]").

- **Legibility of schema mutants**

The example in Figure 38 proceeds from the *Kuar* app too: the original statement checks whether a point is inside the area of a board. The statement is a decision with four conditions and is mutable in several ways.

Suppose that the tester wants to investigate why one of this statement's mutants remains alive: clearly, this task is much easier comparing the original code in rows 2 and 3, than with the too confusing call to the Mutant Driver of the last row.

Original program **if** (x >= board.getLeft() && x <= board.getRight() && y >= board.getTop() && y <= board.getBottom()) {
LOR mutant **if** (x >= board.getLeft() || x <= board.getRight() && y >= board.getTop() && y <= board.getBottom()) {
ROR mutant **if** (x >= board.getLeft() && x != board.getRight() && y >= board.getTop() && y <= board.getBottom()) {
Mutant Schema combining LOR and ROR **if** MutantDriver.*AND*(MutantDriver.*AND*(MutantDriver.*AND*(MutantDriver.*GREATER_EQUALS*(x, board.getLeft(), 214, 215, 216, 217, 218), MutantDriver.*LESS_EQUALS*(x, board.getRight(), 219, 220, 221, 222, 223), 176), MutantDriver.*GREATER_EQUALS*(y, board.getTop(), 224, 225, 226, 227, 228), 177), MutantDriver.*LESS_EQUALS*(y, board.getBottom(), 229, 230, 231, 232, 233), 178)) {

Figure 38. A piece of code, two classic mutants and a Mutant Schema

3.2.2.2 *Partial conclusions*

Untch Mutant Schema (UMS) is "our own" Untch's implementation, because no author gives enough technical details to have a faithful reproduction of their method. The works [83], [26], [74] that refer to the use of this approach are limited to referencing Untch et al. [72] and in some cases give details only of the *metaprocedures*. However, none of them provides how to implement or how to work with the driver that invokes the

metamutants and which indicates which mutants should be instantiated. UMS requires to analyze the program code to (1) detect the statements to be changed, (2) substitute the original statements by calls to the *metaprocedures*, (3) create the *metaprocedures* and (4) implement the test driver, all without making any copy of the SUT. However, on many occasions some mutants overlap (which depends on the order and selection of the operators), therefore the number of mutations that are generated with UMS does not always coincide with the number generated in a traditional way. On the other hand, UMS implementation is easy, the structure mounting is fast, but the mutated code readability is cumbersome, and it also often produces what are known as trivial mutants.

3.2.3 BacterioWeb v.2: Improved version of Android Mutation Testing Tool

BacterioWeb v.2, is a web tool for the mutation testing of Android mobile applications in a distributed environment, we have developed it as an evolution of *BacterioWeb v.1*. To our best knowledge, *BacterioWeb v.2* is the first mutation testing tool which is available on the web. All mutation tools we know ([26], [68], [69], [89], [94], [8], [95]) are "standalone" or desktop tools, that operate with projects saved on the same machine where the mutation tool is installed. Existing tools make it difficult to share a mutation testing project among a team of testers. Also, to take advantage of the power of these and other mutant generation tools, *BacterioWeb v.2* offers the possibility of importing mutants (Section 3.2.1). Moreover, *BacterioWeb v.2* implements mutation techniques that allow tester to perform mutation testing with reasonable cost and time, such as (1) Selective Mutation, to select the mutation operators to apply, but we have not made any analysis to check possible subsumptions among operators, (2) Mutant Schema, to reduce the packaging time of the mutants, (3) Execution mutants parallel, (4) Only Alive strategy, test cases are launching only against those mutants

remain alive and (5) Mutant Sampling, to randomly choose a small subset of mutants from the entire set.

BacterioWeb v.2 consists of two web applications (Figure 39): The Central Web Server (CWS) is the entry point for testers. It allows to create and manage projects and to access the mobile devices offered by the Device Web Servers. The Devices Web Server (DWS) offers the CWS a web interface for managing the SUT on the devices it has connected. The CWS saves the whole project on a relational database, which is also known by the DWS's. Mutants generation is performed in the CWS. Test execution is launched and coordinated by the CWS, although the tests run on the devices connected to each DWS.

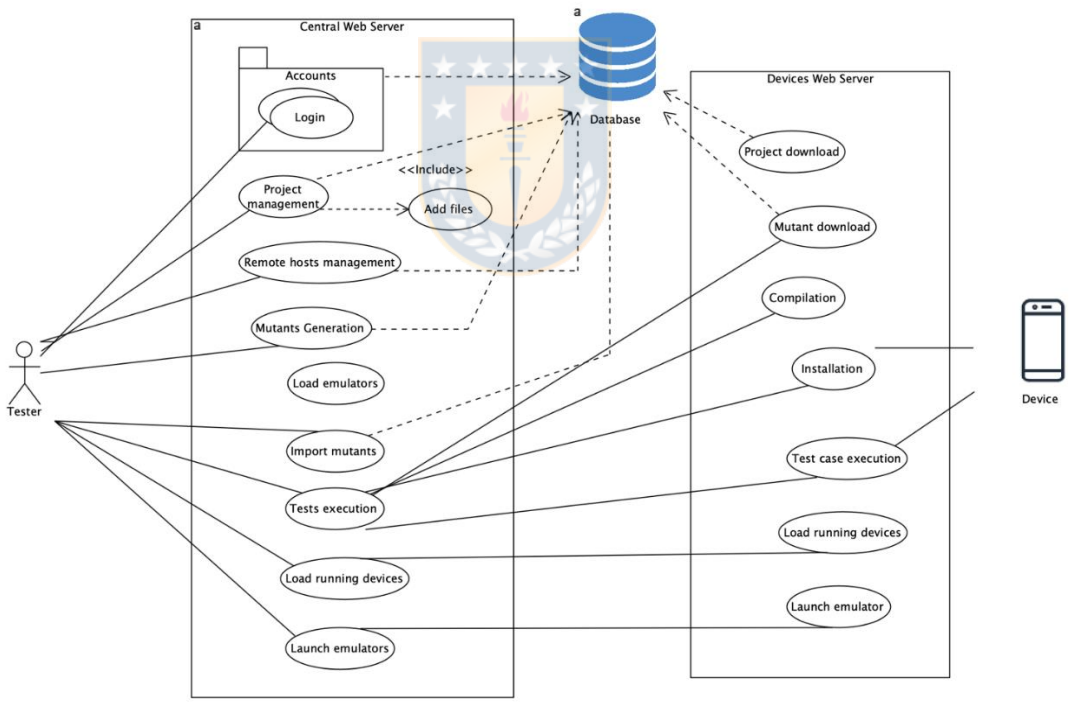


Figure 39. Functional view of *BacterioWeb v.1*

For both systems we have created a web resource for every of the use cases shown in Figure 39. Some of them are implemented as http resources, whilst others are offered via websockets. In both systems too, all the web

resources pass their received requests to a Manager, which is a singleton that holds all the knowledge about the business objects, which are placed in the internal layers.

3.2.3.1 ***Project Creation***

The tester uploads the folder containing the Android project to the CWS. This one copies the folder in its own file system and uploads every of its files to the *project_file* table of the database (Figure 40). The file contents are saved in binary format in the contents column.

Each time a *.java* file is uploaded, we use the javaparser library to build its abstract syntax tree (AST), which is serialized and saved in the *compilation_unit* column of the *project_file* table. Keeping the AST together to the file avoids reanalyze the file when mutants are to be generated.

Moreover, source files are analyzed to check whether they are test classes. In this case, they are separately saved in the *test_class* table, and their test cases in the *test_method* table.

Once the project has been uploaded, the CWS calculates some parameters that will be needed to compile, install, and execute the tests. These values are saved in the *project_configuration* table and are modifiable by the tester.

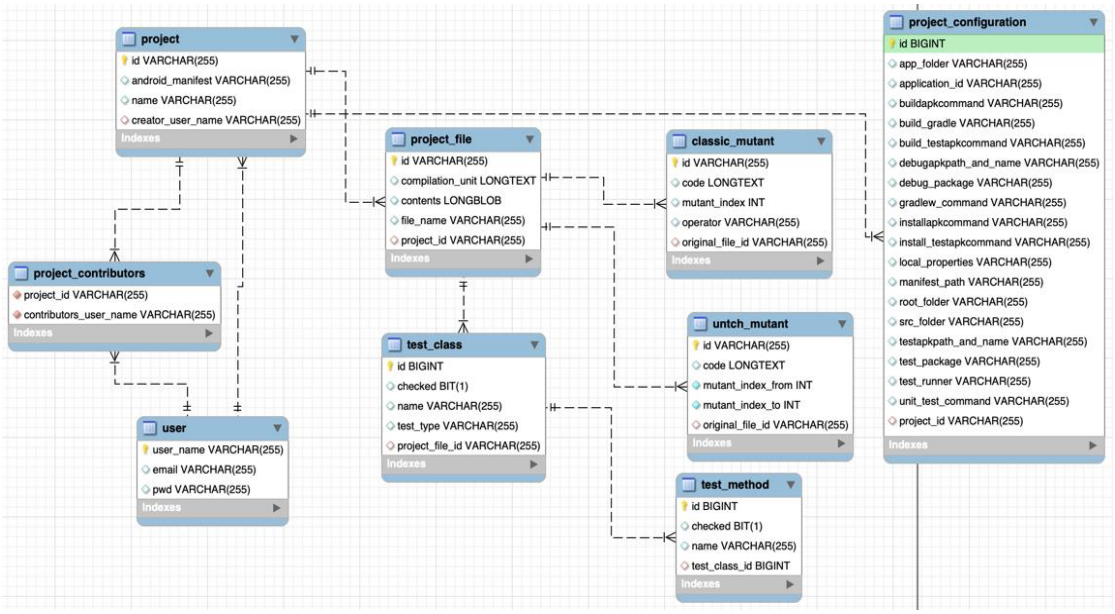


Figure 40. Main tables in the database

3.2.3.2 *CWS-DWS communication*

When the browser's tester arrives to the *Run* screen in the CWS, this one contacts via *http* requests with all the DWS's, whose ip's are saved in the database. Each connected DWS returns the list of its available devices and the list of emulators it may offer.

Figure 41 is showing that the Device Web Server at *192.168.1.11* offers one emulator (*Pixel_2_API_30,* which is not currently running) and has one available physical device (that one with *id* 94769a87); the DWS at *192.168.1.60* offers three emulators and has one device running (*4bd3f236)*.

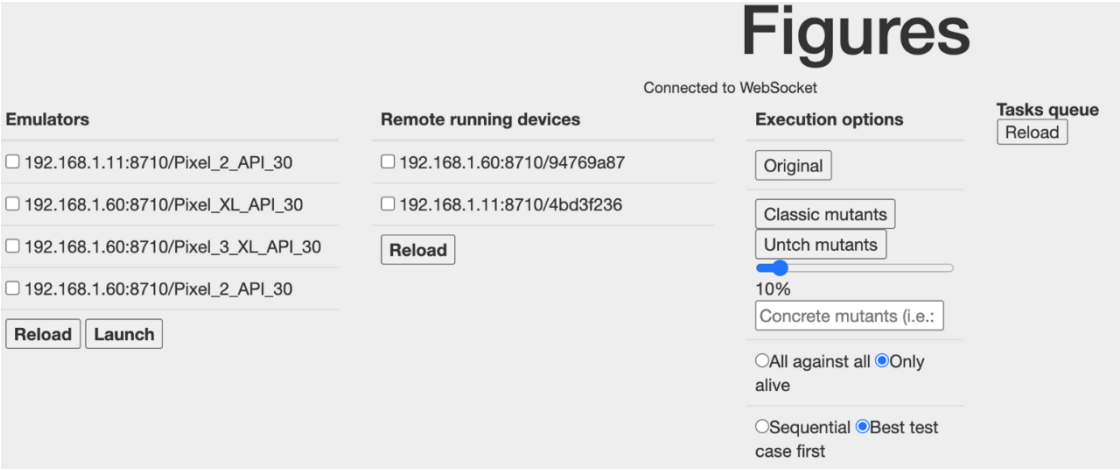


Figure 41. Connection to the remote Device Web Servers

If the tester selects one or more emulators (Figure 42), she/he can launch them in order to make them available as more *remote running devices*: after launching the three selected emulators in Figure 42, the tester has 5 available Android devices (2 physical devices and 3 emulators) to execute the tests (Figure 43).

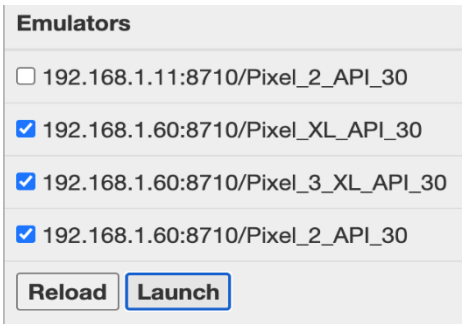


Figure 42. Launching remote emulators

Figure 43. Available devices after Figure 42

In the Central Web Server, every remote device is represented as an instance of a *DeviceProxy* object (Figure 44), which offers the tester all the required operations for executing the tests: *downloadAndInstallNextMutant, push, install, compile, runNextText...* The *DeviceProxy* sends all these operations to the Device Web Server where the corresponding physical device is located via *http* requests. These ones are cached in the DWS by a *RunController* that, passing by a *manager*, routes the request to a *Device* instance. Finally, this one sends the command to the physical device through an operating system process (Figure 47).

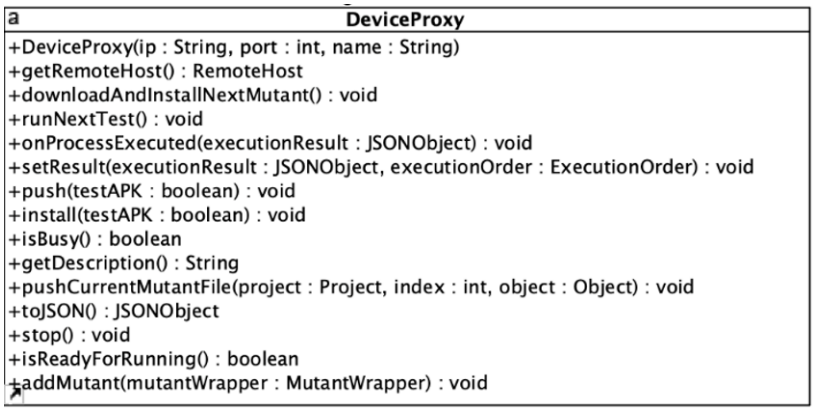


Figure 44. A DeviceProxy interacts in the CWS with a remote device, located at the DWS

3.2.3.3 *Starting test execution*

The tester launches the execution of the tests by pressing one of the three buttons *Original, Classic mutants* or *Untch mutants* shown in Figure 41. She/he must also select the remote devices where the tests will be executed. Moreover, she/he has other choices:

1) *All against all* or *Only Alive*: the first one executes all the test cases against all the mutants. The second one executes the test cases only against the mutants remaining alive.
2) *Sequential* or *Best test case first:* with the first one, each device executes one test case after the other, in a given order. With the second one, the *Device* instance asks for the Central Web Server which one is the test case that, in all the devices, has killed more mutants, and executes it. At this point, it is worth noting that, after executing every test case, the DWS sends back the result to the CWS.

The tester may also:

3) Either select a *percentage* of mutants to be executed. In this case, the CWS makes a random selection of mutants and distributes them equally among the remote devices.
4) Or select some *concrete mutants*, which is a list of numbers (the mutant indexes) separated by commas or tabs.

When one of the afore-mentioned execution buttons is pressed, the browser sends a message with all the execution configuration to a WebSocket listening at the Central Web Server, which creates an object (*TestRunnerGroup*) for controlling the progress of the execution.

Figure 45 illustrates the interactions between four of the five types of systems involved in the starting of a test execution scenario (we have taken out the database for clarity): after pressing the desired execution button, the tester's browser sends a message to the CWS's WebSocket. After checking

the data received, creates an instance of *TestRunnerGroup.* This oner recovers from the database the available list of remote hosts: as well as a *DeviceProxy* represents a remote device in the CWS, a *RemoteHost* instance is a proxy used to send *http* messages to a DWS.

The *TestRunnerGroup* asks each *RemoteHost* to download the project. The *RemoteHost* instance sends (message 5) a *http* request to the DWS's *RunController.* This one asks for the DWS's *manager* (who knows the database) to download it.

When it has been downloaded, the *RunController* sends back a *200/OK http* response to the CWS's *RemoteHost,* who sends in turn a new *http* request for compiling both the project APK and its test APK. Note that both downloading and compiling do not depend on the device.

When the DWS has successfully compiled the project and the tests, sends back a new 200/OK response to the *RemoteHost.* Each time a remote device has successfully installed both the app and the tests, its corresponding *DeviceProxy* instance tells the *TestRunnerGroup* that the remote device is ready (message 28, the last one in Figure 45).

When all devices are ready, the *TestRunnerGroup* asks every *DeviceProxy* instance to *push* and *install* both APK's (the app and the tests) on the physical remote devices. These requests (that now contain the *device name*) are sent to the same *RunController,* who sends the corresponding call to the *Device* instance. This one creates an operating system process that sends the corresponding *adb* command to the physical device.

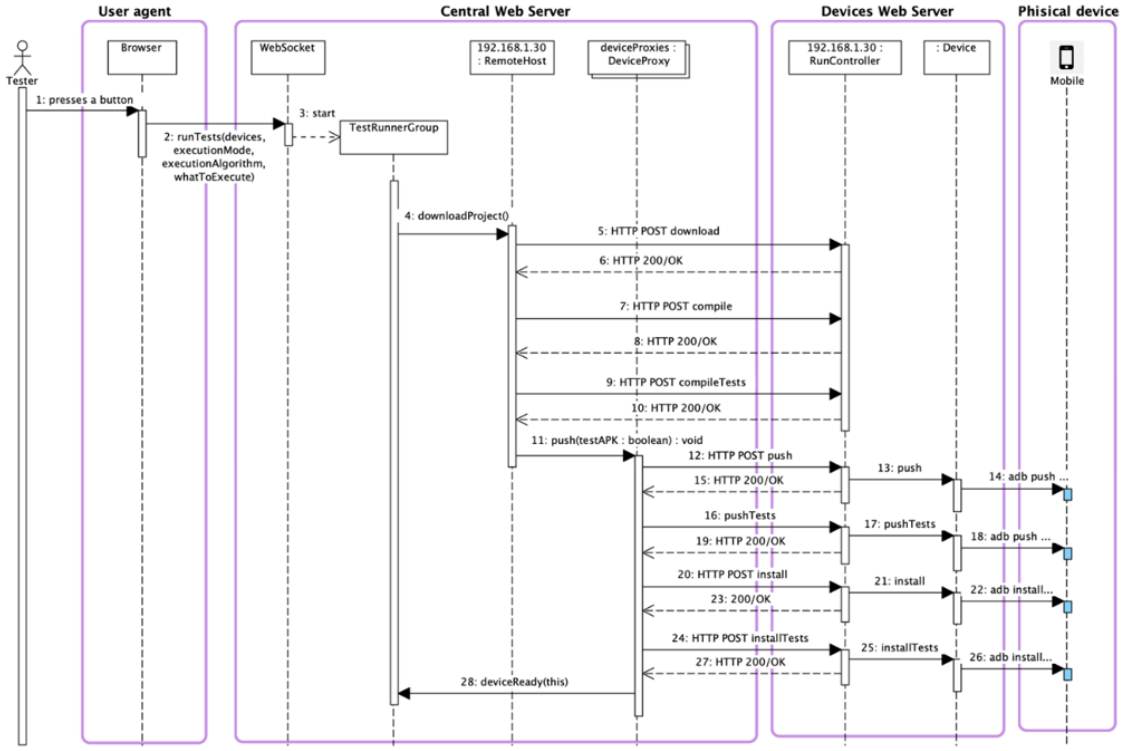


Figure 45. Sequence of operations for starting a test execution

If any of the remote hosts returns a fault in the downloading or compilation, or a remote device in the pushing or installation, the process is stopped, and the tester is reported. But if all the remote devices are ready, the *TestRunnerGroup* equally distributes the mutant ids among the remote devices and sends them the starting execution message.

At first glance, all the operations performed so far will be only useful if the tester has pressed the *Original* button: in fact, the original app does not need to be compiled or installed when mutants are to be run. However, to perform these previous steps are useful to check that the app is valid for all the remote devices (an app compiled with Android version 8 cannot be installed on a device with Android version 4), that it has space enough, that there are no compilation errors, etc.

3.2.3.4 ***Test execution***

The *TestRunnerGroup* asks for every *DeviceProxy* to start the test execution when all the devices have installed both the app and the tests.

The *DeviceProxy* sends the corresponding *http* request to the *RunController,* who locates the target device by means of the *manager.* The execution of a test case from the console consists in writing and executing a command like that in Figure 46: it launches the Android's *adb* program with several parameters, some of which are the device name, the test class, the test method in this test class and the *test execution runner*. The *adb* program may be located at different locations in each remote host. Thus, the *remote_host* table in the database holds the Android SDK and AVD paths. The *test execution runner* is specified in the *gradle* file of the project (during the project uploading).

Therefore, the CWS's *DeviceProxy* sends the corresponding test case execution request to the DWS's *RunController,* who in turn locates the *Device* instance corresponding to the target physical device. Running a test case on the *Device* consists in creating an operating system process (see Figure 47) imitating the one shown in Figure 46. The output of the process is saved in a file on the disk of the DWS, which is sent back to the *DeviceProxy* (in the CWS). The *DeviceProxy* sends the result to the *TestRunnerGroup* that, through the WebSocket, updates the tester's browser.

```
platform-tools — -bash — 97×37
MBP-                :platform-tools                    $ /Users/                    /Library/Android/sdk/pla
tform-tools/adb -s 94769a87 shell am instrument -w -r -e debug false -e class 'com.maco.figures.T
estTriangleWritingLocal#testWL33' com.maco.figures.test/androidx.test.runner.AndroidJUnitRunner
INSTRUMENTATION_STATUS: class=com.maco.figures.TestTriangleWritingLocal
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=
com.maco.figures.TestTriangleWritingLocal:
INSTRUMENTATION_STATUS: test=testWL33
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS: class=com.maco.figures.TestTriangleWritingLocal
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: test=testWL33
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_RESULT: stream=

Time: 5.656

OK (1 test)
```

Figure 46. Console command for executing a test case and its result

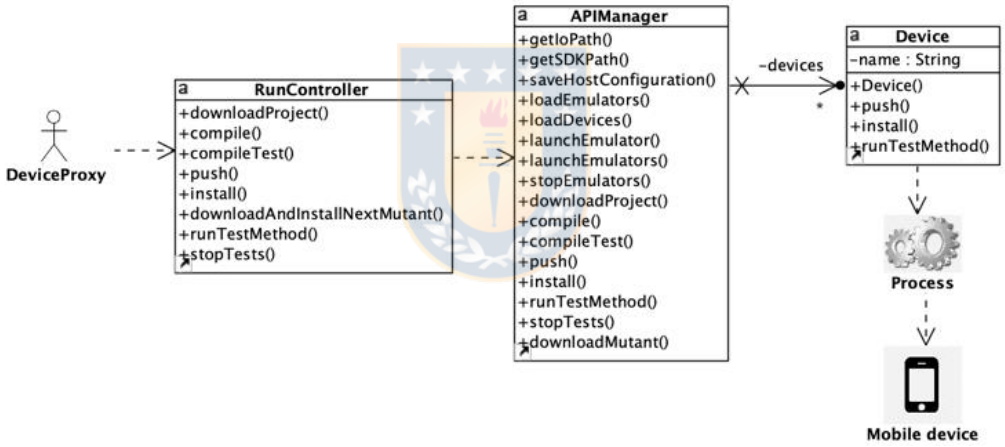


Figure 47. Connection between a *DeviceProxy* (on the CWS) and its physical device (on the DWS)

Besides knowing (through the WebSocket) the tester's browser, the *TestRunnerGroup* holds also an instance of a *KillingMatrix* class, which is updated each time a device sends a test case result. If the execution options the tester did select included the option *Best test case first,* the *TestRunnerGroup* asks the *KillingMatrix* for the best test case (i.e., that one that kills more mutants), and sends it to be run in the remote device.

3.2.4 Mathematical Models of Cost Reduction Techniques

As described above, we have implemented, designed, and experimented with different mutant generation techniques and mutant schema approaches. In addition, we have performed several complete cycles of mutation testing on mobile applications using the *BacterioWeb v.2* tool, which has allowed us to build mathematical models that model the theoretical savings of the cost reduction techniques used in the experimentation of Chapter 4.

From the execution time point of view, the worst situation is when no cost-reduction technique is applied: neither Mutant Schema nor Parallel Execution. In this situation, all test cases are executed against all mutants. Although the absence of cost-reduction techniques is obviously unadvised, it is useful to take it as a baseline for estimating the cost reduction achieved. This idea is similar to that of Grindal and Offutt [96] in their paper about combinatorial test generation: although *All combinations* is not a good technique (it produces many test cases, many of which are redundant), "it is often used as a benchmark with respect to the number of test cases".

Setting aside the result analysis step, the total time required for executing *T* (a set of test cases) against *M* (a set of mutants) is the sum of the times for mutant generation (T_{gen}) and for the required steps for executing the tests (T_{exec}) (Eq. 1).

$$T = T_{gen} + T_{exec} \qquad (1)$$

Both with or without Mutant Schema, and independently of the execution algorithm, the mutant generation time (T_{gen}) is equal and it does not depend on the approach. So, we will not consider it in the next equations.

3.2.4.1 *Mathematical model of T_{exec} without any cost-reduction technique*

With respect to the execution time, it depends on:

(1) The number of test cases and the number of mutants ($|M|$).

(2) The nature of the test suite (*unit* or *instrumented*). The execution of *instrumented* test suites requires compiling the app, packaging it into an *apk*, pushing it onto the device and executing the tests. *Unit* tests only need the compilation and the execution.

(3) The execution algorithm: in this research we distinguish between executing all test cases *against all the mutants* ("All against all", such as in the example of Table 6) and *only against the mutants remaining alive* ("Only Alive", like in Table 7).

Table 13 summarizes the tasks to be performed depending on the type of test.

Type of test	Tasks
Unit	Compile Run tests
Instrumented	Compile (and build APK) Push APK onto devices Install APK Run tests

Table 13. Execution tasks depending on the type of tests

In the case of *instrumented* tests, the tester must build, push, and install an *apk* with the change that corresponds to every mutant. The tests are launched against all the mutants (M), as shown in Eq. (2). For *unit* test cases, $T_{push}=T_{install}=0$.

$$T_{exec}^{NoTech} = |M| \cdot (T_{compile} + T_{push} + T_{install} + T_{run}) \quad (2)$$

In Eq. (2):

- $T_{compile}$ is the time required to compile one version of an app.

- T_{push} is the time required for pushing the *apk* file corresponding to an app from the computer to the device.
- $T_{install}$ is the time required for installing an app on a device. The *apk* file has been previously deployed onto that device.
- T_{run} is the time required for executing the test suite against an app.

3.2.4.2 *Mathematical Model of T_{exec} with Mutant Schema*

The same steps are required for Mutant Schema. There is however a previous step to generate and mount the schema (T_{ms}), but there is only one compilation and, for *instrumented* tests, only one pushing and one installation. As in the previous case (we are considering the *All against all* execution), all the tests are launched against all the mutants, as shown in Eq. (3).

$$T_{exec}^{MS} = T_{ms} + T_{compile} + T_{push} + T_{install} + |M| \cdot T_{run} \qquad (3)$$

T_{ms} is negligable in most cases (7-8 milliseconds in almost all projects). Even in the case of WordPress (one of the selected applications for our experiment, which has 538 Java source files and 109,991 lines of code), the generation of the mutant schema is almost insignificant Figure 48. Thus, we will remove T_{ms} from our equations.

Times (in milliseconds)		
Untch schema mounting	APK build	
542	51699	<— Total time
54	*5170*	<— Mean time
58	5085	
55	5189	
53	5299	
51	5417	
53	5415	
52	4958	
52	5402	
50	4999	
67	4926	
51	5009	

Figure 48. Ten measures of *Tms* and *Tcompile* for WordPress and its 538 Java files

3.2.4.3 *Mathematical Model of Parallel Execution*

Without Mutant Schema and with *n* devices, the execution time is directly reduced in *1/n* (Eq. 4): every task is made |*M*| times but distributed in parallel on the *n* connected devices.

$$T_{exec,n}^{NoTech} = \frac{|M|\cdot(T_{compile}+T_{push}+T_{install}+T_{run})}{n} \qquad (4)$$

Mutant Schema still requires only one compilation; the system must be uninstalled and installed on all the devices but, since these tasks are executed in parallel, it is like performing them only once. T_{run} is reduced in *1/n* (Eq. 5):

$$T_{exec,n}^{MS} = T_{compile} + T_{push} + T_{install} + \frac{|M|\cdot T_{run}}{n} \qquad (5)$$

Equations (4) and (5) substitute Equations (2) and (3) with the introduction of the new parameter, *n* (*n=1* when there is no parallel execution).

3.2.4.4 *Mathematical Model of Only Against Alive (OA) with Mutant Schema*

The success of *Only against alive* depends on the better or worse "luck" in the execution order of the test cases:

- The best situation happens if the first test case is able to kill all the mutants, since no more test cases need to be executed (i.e., the killing matrix would have only one row with all the mutants killed). This is illustrated in Table 14, where the first test finds all the artificial faults. In this case, there is a cost reduction factor (we call it ρ) of *1/*|*T*|.
- The worst situation is when none of the tests kills any mutants (all the cells in the matrix would be empty) or if the last test case executed is the only one that kills mutants Table 15. In any of these two

situations, all the tests are executed against all the mutants. There is no cost reduction in this case, so $\rho = 1$.

	m1	m2	m3	m4	m5	m6	m7
t1	X	X	X	X	X	X	X
t2							
t3							
t4							
t5							

Table 14. The first test kills all the mutants (ρ = 1/5)

	m1	m2	m3	m4	m5	m6	m7
t1							
t2							
t3							
t4							
t5	X	X	X	X	X	X	X

Table 15. The last test kills all the mutants (ρ = 5/5)

Without Mutant Schema, the total time is given by Eq. (6):

$$T_{exec,n}^{NoTech,OA} = \frac{|M| \cdot (T_{compile} + T_{push} + T_{install} + \rho\, T_{run})}{n} \qquad (6)$$

With Mutant Schema, the total time is (Eq. 7):

$$T_{exec,n}^{MS,OA} = T_{compile} + T_{uninstall} + T_{install} + \frac{|M| \cdot \rho \cdot T_{run}}{n} \qquad (7)$$

In the *worst case*, the reduction factor is $\rho = 1$, being in this case the times equal to those in Eqs. (4) and (5).

In the *best case*, the reduction factor is $\rho = 1/|T|$, where $|T|$ is the number of test cases. This behavior occurs if all the mutants are killed by the first test case executed.

In the *average case*, the reduction factor takes intermediate values. Therefore: $\frac{1}{|T|} \leq \rho \leq 1$

3.2.4.5 *Improvement factor*

The improvement in the execution time with respect to our benchmark (non-using any technique, Eq. 2) can be described as a quotient. Thus, the "improvement factor" (*IF*) of applying Mutant Schema with Parallel execution on *n* devices and an arbitrary reduction factor (ρ) is given by Eq. (8). Note that, if $\rho=1$, this equation is valid for the *All against all* execution algorithm.

$$IF = \frac{T_{exec,n}^{NoTech,OA}}{T_{exec,n}^{MS,OA}} \quad (8)$$

We develop the equation replacing (4) and (5) in (8):

$$IF = \frac{\frac{|M|\cdot\left(T_{compile}+T_{push}+T_{install}+\rho\cdot T_{run}\right)}{n}}{T_{compile}+T_{push}+T_{install}+\frac{|M|\cdot\rho\cdot T_{run}}{n}} \quad (9)$$

Removing *n* from the numerator:

$$IF = \frac{|M|\cdot\left(T_{compile}+T_{push}+T_{install}+\rho\cdot T_{run}\right)}{n\cdot\left(T_{compile}+T_{push}+T_{install}\right)+|M|\cdot\rho\cdot T_{run}} \quad (10)$$

As $|M|$ grows up, *IF* improves, although it tends to asymptotically stabilize towards a maximum (Eq. 11).

$$\lim_{|M|\to\infty} IF = \frac{T_{compile}+T_{push}+T_{install}+\rho\cdot T_{run}}{\rho\cdot T_{run}} = 1 + \frac{T_{compile}+T_{push}+T_{install}}{\rho\cdot T_{run}}$$

(11)

3.2.4.6 *Assumptions*

The comparison of the times and the calculus of the limit in Eq. (11) requires that the different times can be compared. Thus, we assume that all of them are equal: this is, the times required to compile ($T_{compile}$), pushing (T_{push}) and installing ($T_{install}$) an app are equal with or without mutant schema or parallel execution. In the same way, we assume that T_{run}, which is the time

required to execute 1 test case against an app is also the same, independently on the use of mutant schema.

The execution time of all test cases against all mutants is formaly defined as: $\sum_{i=0,j=1}^{i=M,j=T} T_{run}\ i,j$, where M (a set of mutants), T (a set of test cases) and $T_{run}i,j$ is the execution of test case j on mutant i. We can say that its upper bound is $M \cdot T_{run}i,j$ where $T_{run}i,j$ is the longest execution time of a test case j on a mutant i; i.e: $\sum_{i=0,j=1}^{i=M,j=T} T_{run}\ i,j$ is $O(M \cdot T_{run}i,j)$. For obtaining a simpler model, we assume that the longest $T_{run}i,j$ is T_{run} so, the mathematical models *roughly* estimate how long it may take to run mutation tests in a mobile app.

3.2.5 Research Questions

For checking the goodness of our mathematical models to estimate the execution cost of a mutation testing cycle, we combine three cost reduction techniques (Mutant Schema, Parallel Execution and Only Alive) both isolated and in different combinations, as well as the non-use of any cost reduction technique at all. The fit of the mathematical model is validated with a set of mobile apps, and it could be extended to consider other cost-reduction techniques. Although the model is applicable to any other context, our motivation for developing it has been mobile-software testing because of its high cost. To do so, the following questions were defined:

RQ1: Can we accept the assumptions made in the Epigraph 3.2.4.6 (Assumptions) of Section 3.2.4 (Mathematical Models of Cost Reduction Techniques)? This is, the times for compiling, pushing, installing, and running are the same, independently of the use or not of Mutant Schema?

RQ2: How good are the Mathematical models to predict the mutation testing time?

RQ3: Does the combination of Parallel Execution, Only Alive strategy and Mutant Schema improve the mutation testing time?

RQ4: How does the number of mutants influence on the improvement factor?

CHAPTER 4

EXPERIMENTATION AND RESULTS

In this chapter, we answer the research questions with experiments. Initially, we refer to the tool used, describe the target mobile applications and the operators used in these experiments. Then, we analyze the results obtained after experimentation.

4.1 MUTATION TESTING TOOL

The tool used for the experimentation is *BacterioWeb v.2* (described in Section 3.2.3). Figure 49 shows a general view of *BacterioWeb v.2*: testers access the system via a Central Web Server (CWS), which manages a database that saves the projects. This CWS knows a set of Devices Web Servers (DWS), which, via *http* and websocket (*ws*), offer their mobile devices and emulators to the CWS to execute the tests.

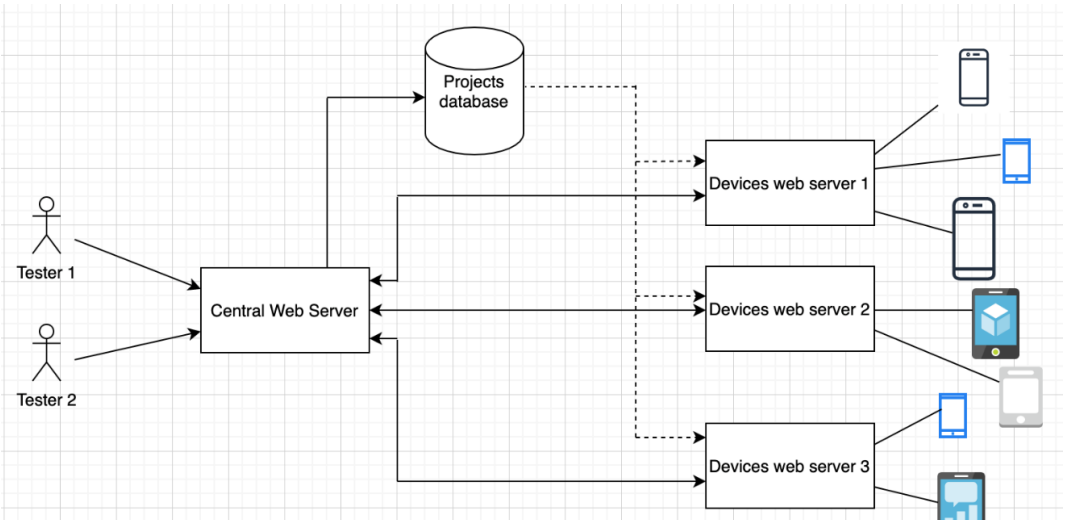


Figure 49. General view of *BacterioWeb v.2*

4.2 TARGET ANDROID APPS AND MOBILE DEVICES

The experiments have been run on the following eight Android apps, which were selected according to the following criteria:

- To make use of common mobile functions (such as touch events).
- They must have available test cases implemented by the developers.
- Their source code must be also available.

1) *WordPress* is the biggest application used in this study. It is used for creating web sites and blogs. It is published in the Google Play Store, it has over 10 million downloads and almost 150,000 comments. Its source code is available at github.
2) *Figures* is a project implemented during the development of *BacterioWeb v.2* for testing some of its characteristics. It is an app that calculates the perimeter and type of a triangle or of a quadrilateral. The calculus can be done locally or by querying a remote web service (Figure 50a). Moreover, the lengths of the sides can be: (1) directly written (Figure 50b), (2) calculated from the coordinates where the tester clicks (Figure 50c) or (3) calculated by clicking on the measures from different sensors (Figure 50d). The selected combination is saved in a set of preferences. It is not a complex project, but it holds the logic of the triangle-type determination problem (proposed by Myers [97] and typically used in many papers about testing), dealing with user events, with web communication, with sensors' data and with preferences.

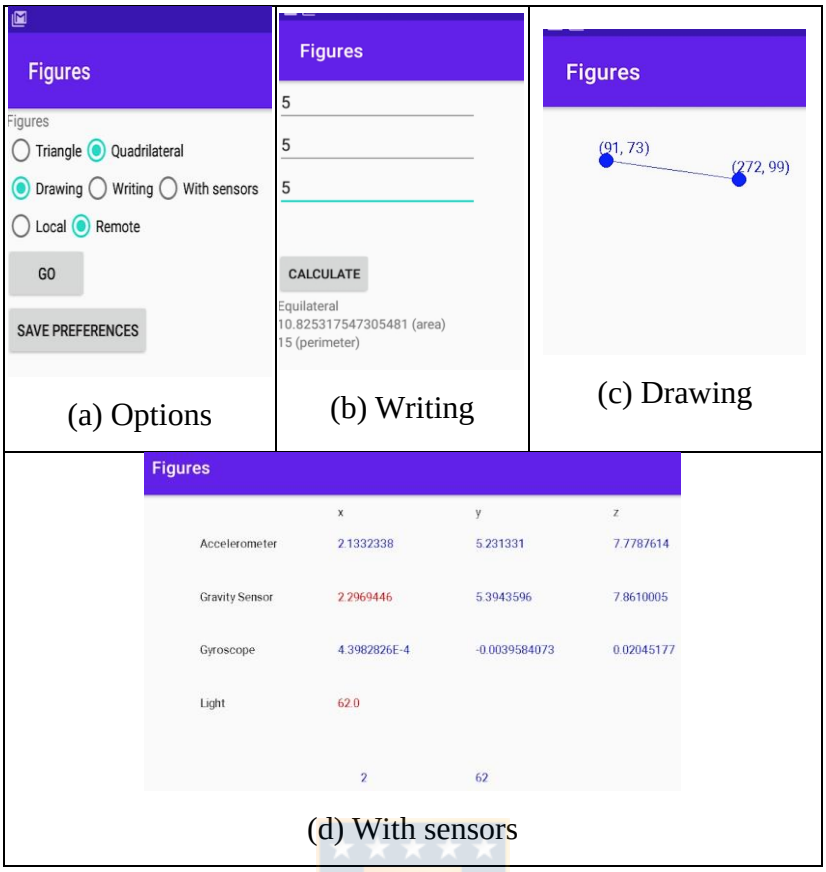


Figure 50. Screenshots of Figures

3 and 4) *AlarmKlock* and *JustSit* are two of the apps used in Deng et al.'s [15] work on mutation operators for Android. *AlarmKlock* allows to set up alarms for different days and hours; *JustSit* is a single app that shows the user two time counters, one in seconds and the other in minutes.

5) *Tourism* has been developed by a company for a Spanish regional government. It uses Google services to plan touristic visits to cities. The user adds points of interest to a touristic route and constraints the time and budget; the app gives in return a route fitting the user's input.

6) *Mangosta* is an open source, open standard, XMPP/Jabber client. Its code is available at github.

7) *Dexter* is an Android library that simplifies the process of requesting permissions at runtime. It is also available at github.
8) *Kuar* is a game developed some years ago by Macario Polo Usaola. The user must consecutively order the numbers on a board using as few movents as possible.R

Table 16 shows some quantitative data of the apps: the *#Files* and *LOC* columns correspond to the number of Java files in each project (excluding interfaces) and their number of lines of code. Last columns contain the number of test cases in each project and the maximum number of mutants that *BacterioWeb v.2* can generate. We wrote the test cases for *Figures, Kuar* and *JustSit*. Test cases for *Tourism* were provided by its developer. The other projects have test cases in their respective repositories.

App	Mutated classes		# Tests	Mutants
	#Files	LOC		
WordPress	538	109991	120	29554
Figures	13	6450	31	1331
Alarm Klock	92	6608	12	3239
JustSit	4	483	10	171
Tourism	54	4902	15	851
Mangosta	59	9902	31	1579
Dexter	30	2872	10	216
Kuar	37	3580	12	3225
Total	**827**	**144788**	**241**	**40166**

Table 16. Some characteristics of the apps

The deployments (push, installing) of these apps and the test execution were carried out on two identical physical devices: two *Samsung* tablets, model SM-T590 running Android 8.1.0 with 3 Gb RAM.

4.3 APPLIED MUTATION OPERATORS

In this thesis, our goal is not to validate mutation operators for Android, our main goal is to measure the influence of several techniques in the reduction of the time required for mutation testing in Android projects. Obviously, using one or another operator with so many mutants and executions does not have any significant impact on the results of our experiments, which are not concerned with the quality of operators, but with the time required for mutation testing. However, we have implemented the traditional operators most used in the literature for java projects [26] and some Android-specific operators proposed by Deng et al. [15] and Escobar-Velásquez et al. [14]. The set of operators will introduce both traditional errors (typical of the Java language) as specific errors for Android (as event handlers).

Traditional operators:

LOR (Logical Operator Replacement), which substitutes a logical operator by another one (|| by &&, |, etc.).

ROR (Relational Operator Replacement), which replaces a relation operator by another one (e.g., && by ||).

UOI (Unary Operator Insertion), which inserts predecrements, postdecrements and the unary minus in numeric variables.

AOR (Arithmetic Operator Replacement), that replaces some arithmetic operators by others (+ by -, *, /, etc.).

IMCA (Invalid Method Call Argument) is one of the Escobar-Velásquez et al.'s operators [14]. It randomly mutates a method call argument of a basic type.

Android mutation operators:

The Android operating system makes available to developers, different mechanisms for storing data in files: *SharedPreferences* files are key-value

tables that allow the application to store small data sets in a simple way. All changes made in an editor are batched, and not copied back to the original *SharedPreferences* until a call to *commit()* or *apply()* is executed. We define 3 mutation operators to reproduce errors that can occur when working with *SharedPreferences* files:

FEC (Forget Editor Commit): This operator simulates that the developer forgets calling *commit*. The values are set with some of the *putX* methods, but the changes are not materialized. So, this operator removes the statement *editor.commit()*. For killing these mutants, test cases need either to include an oracle to check that the preferences have been save (what is unusual), or to execute a long scenario that makes use of the previously saved preferences: therefore, this operator confirms one of the conclusions of Gordon and Gargantini in [98], who observed that is preferred to have a few long test cases than many short test cases. This operator is quite similar to the *CPSE* operator proposed by Paiva [12].

FEA (Forget Editor Apply): This operator is similar to FEC, but in this case it removes the call to the *apply()* method. It works exactly in the same way as the FEC mutation operator.

RSPE (Replace Shared Preferences Editor): A typical error in the use of *SharedPreferences.Editor* type files is to mismanage the keys entered in the file. This operator mutates the key-value pairs of the statements *putInt(…), putBoolean(…), putString(…), putLong(…), putFloat(…)* of different ways.

MDL (Lifecycle Method Deletion) is one of Deng et al.'s operators [15]. It deletes each overriding activity method to force Android to call the version in the super class.

ETR (OnTouch Event Replacement) [15]: it searches and stores all event handlers that respond to *OnTouch* events in the current class. Then, it replaces each handler with every other compatible handler.

KER (Key Event Replacement): Key events contain information about keys pressed. This operator replaces some key events with other equivalent key events: for example, it replaces *KeyEvent.ACTION_UP* by *KeyEvent.ACTION_DOWN*.

IEC (Interchanges the Event's Coordinates): This operator modifies motion event's location through interchanges and replacement of axis values. So, if the user clicks on (100, 200), this mutation operator sends the event to (200, 100)

IPR_E (Intent Payload Replacement Extension): This operator is an extension of the IPR operator (Intent Payload Replacement) proposed by Deng et al. [15]. An *Intent* can send different types of data from one activity to another, as key-value pairs. The *putExtra(...)* method takes the key name as the first parameter, and the value as the second parameter. IPR_E includes all mutations of IPR proposed in [15], but it also adds a mutation for the first parameter (*empty String*) and different mutations for the key-string pairs.

ITR (Intent Target Replacement) [15]**, also called InvalidActivityName** in [14]: This operator mutates the *Intent* target object (an activity), changing the target activity. This idea also is included in the *NACT* operator proposed by Paiva [12].

ORL_M (Orientation Locked Modified): This operator is a modification of the Deng et al.'s ORL operator [15]. The original ORL freezes the orientation of an activity to be in portrait or landscape, and this is done by inserting a locking statement into the source. Our modification preserves the same idea, but the mutants freeze the orientation of an activity

to be in portrait or landscape through insertion or replacement of *setRequestedOrientation(…)* statement into the source.

MJP (Modify JSON Put): This operator inserts small changes into the key-value pairs of the different *put(...)* methods of the *JSONObject* class. JSON is a widely used format for message interchange. Developers tend to copy and paste calls to *put(...)* or to build unexpected hierarchies of JSON objects. This operator modifies keys and values in *put* calls.

IQ (Incorrect Query): The SQLite database allows the developer to introduce SQL statements as strings. This operator mutates the query passed as parameter in calls to *SQLiteDatabase.RawQuery(…)*. This operator is similar to the *InvalidSQLQuery* proposed by Escobar Velásquez et al. [14].

RAQ (Replace read-write Access to a database Query): This operator mutates the calls allowing the iteration through the result set returned by a database query. It changes *moveToFirst*, *moveToLast*, *moveToNext* and *moveToPrevious* by the others.

Currently, there are many specific mobile software operator [14], [15], [89]. However, the similarity between them is high and their mutations are like some mutations of the traditional operators; that is why we propose as future work, a study on subsumption of android mutation operators (Section 5.1).

4.4 EXPERIMENTAL SETUP

For each application under test, we have carried out the following steps:

- Mutant generation. This task generates and saves the mutants in the relational database.
- Second, we execute the test suites against the mutants with the combinations of techniques shown in Figure 51. Excepting for

WordPress, each test suite has been executed against all the mutants 3 times to minimize bias. For WordPress we have taken a sample (10%) of mutants, since otherwise the execution time is huge. Maybe 3 times does not seem too much, but some complete executions with the combination 1 require around a week. At this point, it is important to note that, the times of connection to the database, writing, reading, etc. are removed from the time calculus

1	(NoTech, All against all, 1 device)
2	(NoTech, Only Alive, 1 device)
3	(NoTech, All against all, 2 devices)
4	(NoTech, Only Alive, 2 devices)
5	(Mutant Schema, All against all, 1 device)
6	(Mutant Schema, Only Alive, 1 device)
7	(Mutant Schema, All against all, 2 devices)
8	(Mutant Schema, Only Alive, 2 devices)

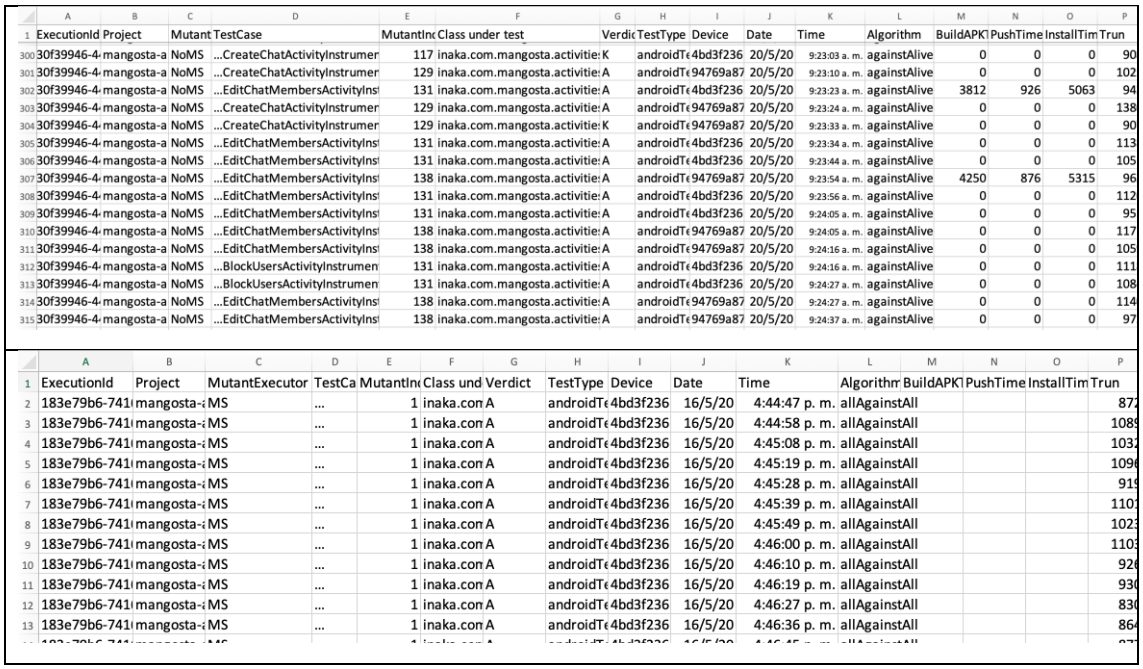
Figure 51. Execution combinations

- Third completing a cycle of mutation testing, we obtain a list of the best test cases applying a greedy algorithm such as the described in Epigraph 2.2.2.2.

BacterioWeb v.2 saves much information in a set of comma-separated files: in one of them (*global.txt)*, it accumulates all the execution data of all the projects; in the other, it saves the same data, but creating a single file for each test suite execution. Figure 52 shows an excerpt of *global.txt*: each row holds the data of the execution of one test case against one mutant. The columns contain: (A) the unique *id* of this test suite execution, (B) the project under test, (C) the test executor used (*NoMS* or *MS*), (D) the test case, (E) the mutant index, (F) the current class under test, (G) the verdict (*A* or *K*

depending on whether the mutant is alive or killed), (H) the test case type, (I) the device where the mutant has been installed, the date (J) and time (K), the execution algorithm (*All against all* or *Only against alive)* in column *L*, the compile, push and install execution times (M, N, O) and the run time (P) required by this test case with this mutant.

Figure 52 corresponds to two different executions of a test suite against the *Mangosta* project. The execution in the top row took place in the morning (column *K*) of May 20, 2020 (column *J*). The run times always appear on column *P*. However, the build, push and install times only appear when these tasks are effectively performed: for example, the mutant in the first row (mutant number *117*, column *E*) was deployed to the device *4bd3f236* (column I). Since the test case kills this mutant (see the *Verdict* in column *G*) and we are executing with *Only Alive* (column *L)*, the execution of the mutant is interrupted and, on row 302, the deployment of the next mutant (number 131) to the *4bd3f236* starts.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	ExecutionId	Project	Mutant	TestCase	MutantIn	Class under test	Verdic	TestType	Device	Date	Time	Algorithm	BuildAPK	PushTime	InstallTim	Trun
300	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumer	117	inaka.com.mangosta.activitie	K	androidTe	4bd3f236	20/5/20	9:23:03 a. m.	againstAlive	0	0	0	90
301	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumer	129	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:23:10 a. m.	againstAlive	0	0	0	102
302	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:23:23 a. m.	againstAlive	3812	926	5063	94
303	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumer	129	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:23:24 a. m.	againstAlive	0	0	0	138
304	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumer	129	inaka.com.mangosta.activitie	K	androidTe	94769a87	20/5/20	9:23:33 a. m.	againstAlive	0	0	0	90
305	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:23:34 a. m.	againstAlive	0	0	0	113
306	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:23:44 a. m.	againstAlive	0	0	0	105
307	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:23:54 a. m.	againstAlive	4250	876	5315	96
308	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:23:56 a. m.	againstAlive	0	0	0	112
309	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:24:05 a. m.	againstAlive	0	0	0	95
310	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:24:05 a. m.	againstAlive	0	0	0	117
311	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:24:16 a. m.	againstAlive	0	0	0	105
312	30f39946-4	mangosta-a	NoMS	...BlockUsersActivityInstrumen	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:24:16 a. m.	againstAlive	0	0	0	111
313	30f39946-4	mangosta-a	NoMS	...BlockUsersActivityInstrumen	131	inaka.com.mangosta.activitie	A	androidTe	4bd3f236	20/5/20	9:24:27 a. m.	againstAlive	0	0	0	108
314	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:24:27 a. m.	againstAlive	0	0	0	114
315	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie	A	androidTe	94769a87	20/5/20	9:24:37 a. m.	againstAlive	0	0	0	97

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	ExecutionId	Project	MutantExecutor	TestCa	MutantIn	Class und	Verdict	TestType	Device	Date	Time	Algorithm	BuildAPK	PushTime	InstallTim	Trun
2	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:44:47 p. m.	allAgainstAll				87
3	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:44:58 p. m.	allAgainstAll				108
4	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:45:08 p. m.	allAgainstAll				103
5	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:45:19 p. m.	allAgainstAll				109
6	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:45:28 p. m.	allAgainstAll				91
7	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:45:39 p. m.	allAgainstAll				110
8	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:45:49 p. m.	allAgainstAll				102
9	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:46:00 p. m.	allAgainstAll				110
10	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:46:10 p. m.	allAgainstAll				92
11	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:46:19 p. m.	allAgainstAll				93
12	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:46:27 p. m.	allAgainstAll				83
13	183e79b6-741	mangosta-a	MS	...	1	inaka.com	A	androidTe	4bd3f236	16/5/20	4:46:36 p. m.	allAgainstAll				86

Figure 52. Two excerpts of the *global.txt* file, generated by *BacterioWeb v.2*

The bottom row of Figure 52 corresponds to an execution with Mutant Schema (see column C). So, columns *M*, *N* and *O* are empty because the building, pushing, and installing are performed before launching the test suite execution. Anyway, *BacterioWeb v.2* shows the tester the times spent in its test execution window (Figure 53). Also note the presence of the time devoted to mounting the mutant schema.

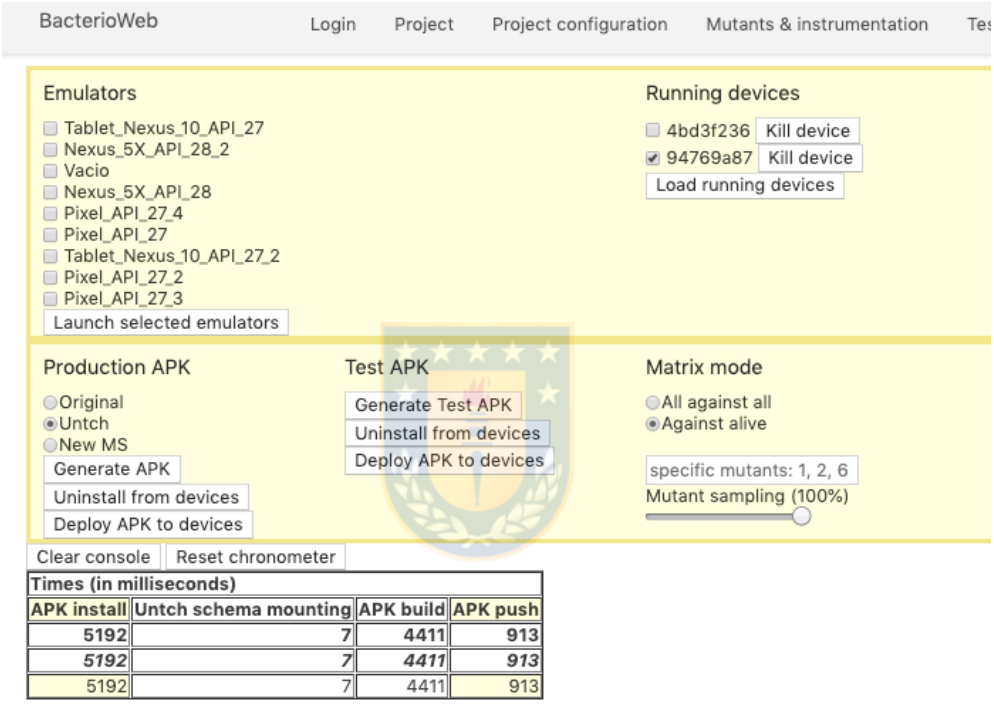


Figure 53. Summary of times in *BacterioWeb v.2*

Therefore, the times collected are:

$T_{compile}$, which is the time required for compiling the application. This time is only applicable for instrumented test cases.

T_{push}, which is the time for pushing the application from the server to the device. It is also only applicable for instrumented test cases.

$T_{install}$, which is the time for installing the application onto the device. This time is only applicable for instrumented test cases.

T_{run}, which is the time spent in executing one test case against one mutant.

4.5 RESEARCH QUESTIONS ANSWERED

RQ1: Can we accept the assumptions made in the Epigraph 3.2.4.6 (Assumptions) of Section 3.2.4 (Mathematical Models of Cost Reduction Techniques)? This is, the times for compiling, pushing, installing, and running are the same, independently of the use or not of Mutant Schema?

To answer the RQ1 each test suite has been executed 3 times in each of the 8 modes shown in Figure 51. This is each test suite has been executed 24 times in every project. These executions have produced a lot of data.

For comparing *Tcompile, Tpush* and *Tinstall* with and without Mutant Schema, we have taken 3,000 random compilations, pushes and installations with No Mutant Schema and 100 compilations with Mutant Schema. The reason of the difference in the sample sizes is that No Mutant Schema compiles, pushes and installs a lot of times, whilst Mutant Schema does them only once.

Then, for each time, we have compared the values got and applied the Student's T. We have two hypotheses:

H0*: Tcompile, Tpush* and *Tinstall* are the same with or without Mutant Schema.

H1*: Tcompile, Tpush* and *Tinstall* are different with and without schema.

H0 can be rejected if *p-value<0.05*. Note however that our assumptions are true (or are not false) if *H0* cannot be rejected.

Table 17, Table 18 and Table 19 respectively summarize the data collected from compiling, pushing and installing in the analyzed projects. Note that all p-values are greater than 0.05, what leads us to not reject *H0.*

Thus, we can accept that there is no significative difference between compiling, pushing, or installing one app with or without the use of Mutant Schema.

Some results may surprise the reader and have surprised the writers. One expects that *Tcompile* should be significantly greater with Mutant Schema, since almost every call, arithmetic operation or comparison is translated into a call to a method in the *MutantDriver*.

	***Tcompile* (milliseconds)**						
	NoTech			**MS**			**p-value**
Project	**Mean**	**Std. Dev.**	**Samples**	**Mean**	**Std. Dev.**	**Samples**	
AlarmClock	-	-	-	-	-	-	
JustSit	2,523	663	3,000	2,486	624	100	0.58
Mangosta	4,373	887	3,000	4,279	911	100	0.30
Figures	1,245	21	3,000	1,241	18	100	0.06
Dexter	1,692	101	3,000	1,684	49	100	0.43
Turismo	2,115	120	3,000	2,095	210	100	0.11
Kuar	1,983	123	3,000	2,001	138	100	0.15
WordPress	5,336	240	3,000	5,301	326	100	0.16

Table 17. *Tcompile* with and without Mutant Schema

	***Tpush* (milliseconds)**						
	NoTech			**MS**			**p-value**
Project	**Mean**	**Std. Dev.**	**Samples**	**Mean**	**Std. Dev.**	**Samples**	
AlarmClock	-	-	-	-	-	-	
JustSit	51	18	3,000	49	23	100	0.28
Mangosta	887	142	3,000	893	151	100	0.68
Figures	102	10	3,000	101	12	100	0.33
Dexter	208	85	3,000	212	106	100	0.65
Turismo	578	186	3,000	588	197	100	0.60
Kuar	281	140	3,000	278	56	100	0.83
WordPress	1,184	452	2,096	1,209	511	100	0.59

Table 18. *Tpush* with and without Mutant Schema

	Tinstall (milliseconds)						
	NoTech			MS			p-value
Project	**Mean**	**Std. Dev.**	**Samples**	**Mean**	**Std. Dev.**	**Samples**	
AlarmClock	-	-	-	-	-	-	
JustSit	979	603	3,000	1,012	611	100	0.59
Mangosta	5,129	189	3,000	5,097	211	100	0.10
Figures	4,431	456	3,000	4,344	480	100	0.06
Dexter	4,540	170	3,000	4,509	206	100	0.08
Turismo	6,185	141	3,000	6,191	143	100	0.68
Kuar	4,798	2,711	3,000	4,526	2,634	100	0.32
WordPress	10,132	2,911	3,000	10,159	10,318	3001	0.53

Table 19. *Tinstall* with and without Mutant Schema

With respect to *Trun*, which is the execution time of a tests case against a mutant, the sample size in both cases is 3,000. This is because we save (remind the *global.txt* file in Figure 52) the execution time of every test case against very mutant and in both cases (with and without Mutant Schema) thousands of executions have been run in each test cycle. Thus, in this case we can compare samples of the same size. As it is seen in Table 20, neither the null hypothesis can be rejected: this is, we cannot distinguish whether a test case execution against a mutant has been executed with or without Mutant Schema.

	Trun (milliseconds)					
	NoTech		MS		Sample	p-value
Project	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**		
AlarmClock	2,528	801	2,489	815	3,000	0.06
JustSit	9,310	3,129	9,397	3,213	3,000	0.15
Mangosta	10,239	3,755	10,242	3,978	3,000	0.98
Figures	7,453	464	7,465	464	3,000	0.32
Dexter	4,047	666	4,051	689	3,000	0.82
Turismo	11,552	2,960	11,486	2,888	3,000	0.38
Kuar	15,432	8,146	15,704	7,870	3,000	0.19

Table 20. *Trun* with and without Mutant Schema

Partial conclusions

Since there is no evidence to reject *H0*, we will assume for the remaining experiments that compiling, pushing, and installing an app onto a device is the same with independence of the use of Mutant Schema.

Note that the veracity of this assumption would allow us to build mathematical models with almost not executing tests.

RQ2: How good are the Mathematical models to predict the mutation testing time?

From the huge amount of data collected by *BacterioWeb v.2*, we have built several tables for each project. As an example, next tables summarize the results for the *Mangosta* project. *Mangosta* has a test suite with 31 test cases and *BacterioWeb v.2* generates 1579 mutants for it.

The first five columns in each row includes the number of devices, a number of mutants, the reached mutation score and the mean of the measured total run time. Last four columns show the execution time (which is the run time plus the time for compiling, pushing and installing): the *Actual* column is the time actually measured, and *Estimated* is the time calculated according to the Mathematical models.

We have executed 3 times all the test cases against the mutants, in the eight variants: for *Mangosta*, for example, we have executed 3 times the 31 test cases against the 1579 mutants using *1* and *2* devices, *Mutant Schema (MS)* and *No Mutant Schema (NoMS)*, *All against all (AA)* and *Only Alive (OA)*. The number of mutants in each row has been randomly selected from the 1579, being exactly the same mutants for each variant.

					Texec (millis)		Texec (hours)	
Devices	**|M|**	**M.Score**	**Executions**	**Total Trun (millis)**	**Actual**	**Estimated**	**Actual**	**Estimated**
1	60	0.98	1860	19533729	20157069	19667880	5.6	5.5
1	100	0.98	3100	32556215	33595115	32779800	9.3	9.1
1	300	0.99	9300	97668645	100785345	98339400	28.0	27.3
1	600	1	18600	195337290	201570690	196678800	56.0	54.6
1	900	0.99	27900	293005935	302356035	295018200	84.0	81.9
1	1200	0.99	37200	390674580	403141380	393357600	112.0	109.3
1	1579	0.99	48949	514062634	530466865	517593042	147.4	143.8
2	60	0.98	1860	9766864	10078534	9833940	2.8	2.7
2	100	0.98	3100	16278107	16797557	16389900	4.7	4.6
2	300	0.99	9300	48834322	50392672	49169700	14.0	13.7
2	600	1	18600	97668645	100785345	98339400	28.0	27.3
2	900	0.99	27900	146502967	151178017	147509100	42.0	41.0
2	1200	0.99	37200	195337290	201570690	196678800	56.0	54.6
2	1579	0.99	48949	257031317	265233432	258796521	73.7	71.9

Table 21. Times with the *Mangosta* project, No Mutant Schema and All against all

Devices	**|M|**	**M.Score**	**Executions**	**Total Trun (millis)**	**Actual**	**Estimated**	**Actual**	**Estimated**
1	60	0.98	1623	16545387	17168727	17241237	4.8	4.8
1	100	0.98	2702	27516217	28555117	28704678	7.9	8.0
1	300	0.99	7861	83320923	86437623	83605479	24.0	23.2
1	600	1	16441	160600645	166834045	174572799	46.3	48.5
1	900	0.99	24107	254476271	263826371	256181673	73.3	71.2
1	1200	0.99	32295	337790730	350257530	343135305	97.3	95.3
1	1579	0.99	43250	446832322	463236553	459240981	128.7	127.6
2	60	0.98	1623	8151993	8463663	8620618	2.4	2.4
2	100	0.98	2702	13365698	13885148	14352339	3.9	4.0
2	300	0.99	7861	42298690	43857040	41802739	12.2	11.6
2	600	1	16441	78483852	81600552	87286399	22.7	24.2
2	900	0.99	24107	126551797	131226847	128090836	36.5	35.6
2	1200	0.99	32295	165803507	172036907	171567652	47.8	47.7
2	1579	0.99	43250	226817787	235019902	229620490	65.3	63.8

Table 22. Times with the *Mangosta* project, No Mutant Schema and Only Alive

					Texec (millis)		Texec (hours)	
Devices	|M|	M.Score	Executions	Total Trun (millis)	Actual	Estimated	Actual	Estimated
1	60	0.98	1860	19273002	19283391	19054929	5.4	5.3
1	100	0.98	3100	32035600	33074500	31751289	9.2	8.8
1	300	0.99	9300	96686426	99803126	95233089	27.7	26.5
1	600	1	18600	195662395	201895795	190455789	56.1	52.9
1	900	0.99	27900	281807454	291157554	285678489	80.9	79.4
1	1200	0.99	37200	390442953	402909753	380901189	111.9	105.8
1	1579	0.99	48949	510347625	526751856	501199200	146.3	139.2
2	60	0.98	1860	9454620	9459814	9527464	2.6	2.6
2	100	0.98	3100	15629881	15635075	15875644	4.3	4.4
2	300	0.99	9300	47783962	47789156	47616544	13.3	13.2
2	600	1	18600	99584353	99589547	95227894	27.7	26.5
2	900	0.99	27900	138850704	138855898	142839244	38.6	39.7
2	1200	0.99	37200	194573006	194578200	190450594	54.0	52.9
2	1579	0.99	48949	247904610	247909804	250599600	68.9	69.6

Table 23. Times with the *Mangosta* project, with Mutant Schema and All against all

					Texec (millis)		Texec (hours)	
Devices	|M|	M.Score	Executions	Total Trun (millis)	Actual	Estimated	Actual	Estimated
1	60	0.98	1608	16129422	16752762	17087652	4.7	4.7
1	100	0.98	2658	27056441	27066830	27225651	7.5	7.6
1	300	0.99	7943	79851108	79861497	81338766	22.2	22.6
1	600	1	15996	157522747	157533136	163793433	43.8	45.5
1	900	0.99	24595	245490830	245501219	251838594	68.2	70.0
1	1200	0.99	32412	321305761	321316150	331876857	89.3	92.2
1	1579	0.99	43025	440782480	440792869	440543364	122.4	122.4
2	60	0.98	1608	8167560	8172754	8237350	2.3	2.3
2	100	0.98	2752	13695917	13701111	14094058	3.8	3.9
2	300	0.99	7851	40159761	40164955	40198389	11.2	11.2
2	600	1	16407	80848459	80853653	84000831	22.5	23.3
2	900	0.99	23548	115252549	115257743	120559180	32.0	33.5
2	1200	0.99	31974	157441585	157446779	163696087	43.7	45.5
2	1579	0.99	43510	223804294	223809488	222754639	62.2	61.9

Table 24. Times with the *Mangosta* project, with Mutant Schema and Only Alive

Figure 54 depicts the data about actual and estimated times shown in the previous tables for 1 device. As it is seen, the adjustment of both curves is almost perfect, with the measured values corresponding almost exactly to the estimates.

It is worth noting that we get similar results in all the analyzed projects.

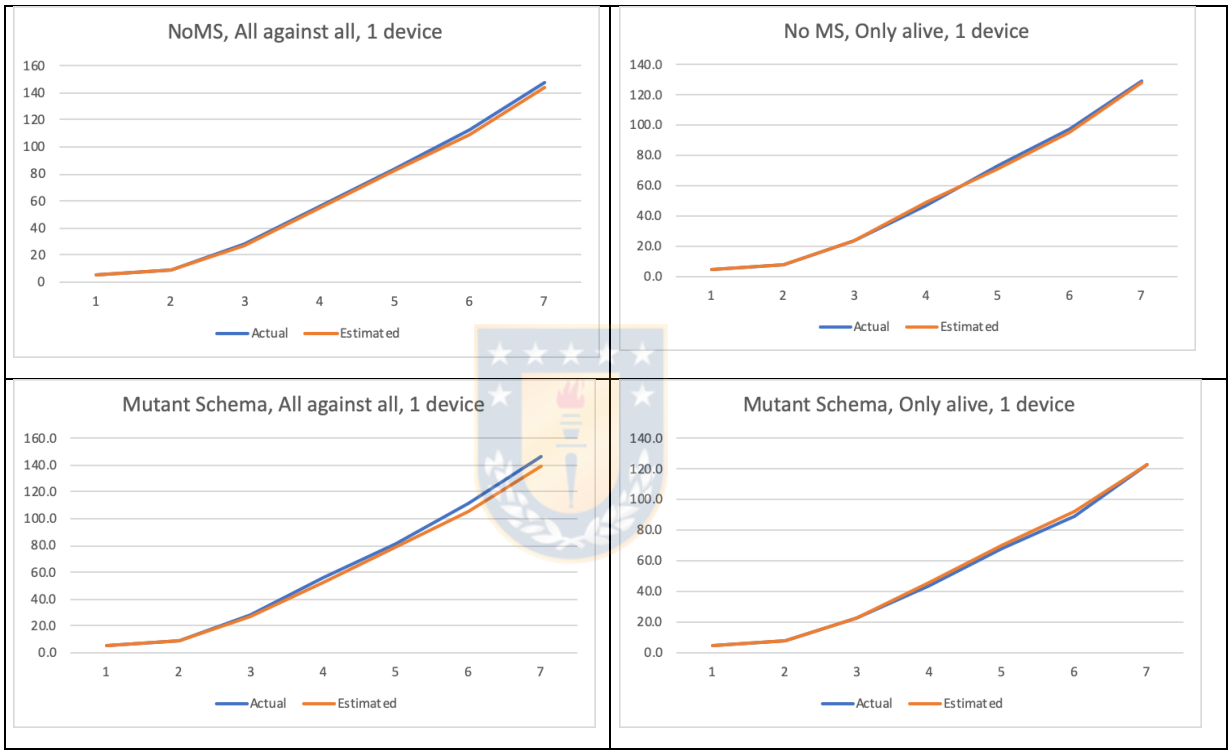


Figure 54. Actual and estimated times in the *Mangosta* project

Partial conclusions

We can conclude that the estimated times from our mathematical models and the real times obtained in the execution of test cases are very similar. Therefore, we can estimate a priori how long it may take to execute mutation tests in a mobile application and, depending on its feasibility, the tester can make decisions about which combination of cost reduction techniques is more convenient.

RQ3: Does the combination of Parallel Execution, Only Alive and Mutant Schema improve the mutation testing time?

The experiments performed to answer to RQ2 also allow RQ3 to be answered. For each apps, we have executed the eight the technique combinations listed in Figure 51. For *Mangosta*, the data is summarized in the Figure 55.

In Figure 55 we can see two groups of combinations; group 1 runs on a single device and group 2 runs on two devices in parallel. As expected, the fastest combination is Mutant Schema, Only Alive and two devices (MS, OA, 2 devices). Also, we can see that all combinations in group 2 are much faster than those in group 1. This indicates that parallel execution is key as a cost reduction technique.

Point out that we obtained the same combination in all the projects analyzed.

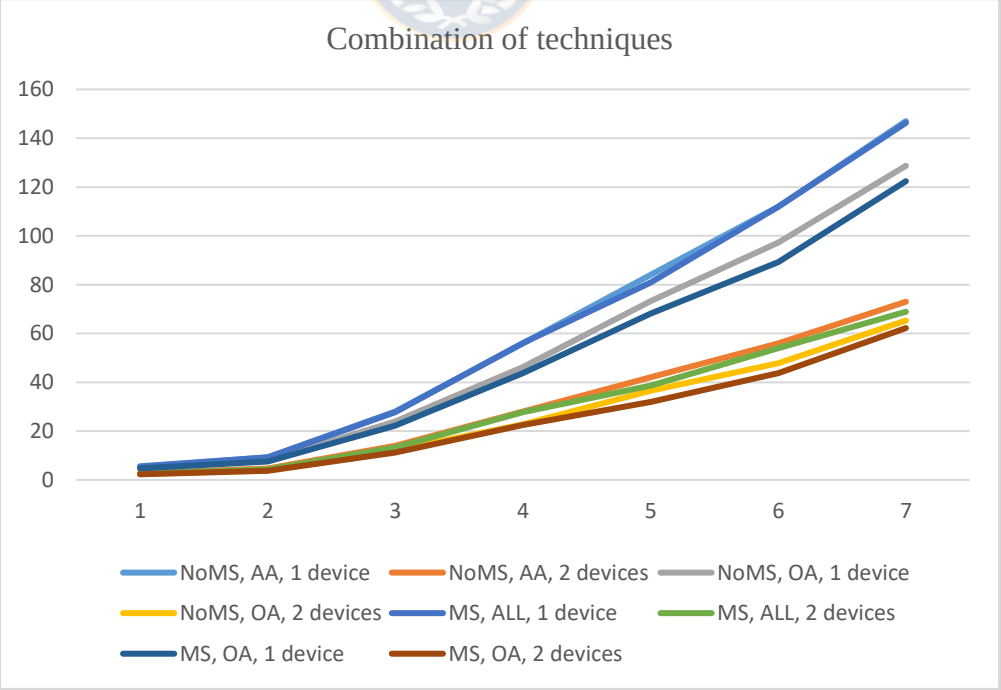


Figure 55. All combination of techniques in the *Mangosta* project

Partial conclusions

The experimental data have shown that: (1) the most efficient combination of the applied cost reduction techniques is: Mutant Schema, Only Alive and Parallel Execution and (2) Parallel Execution is the most cost-savings technique.

RQ4: How does the number of mutants influence on the improvement factor?

In Epigraph 3.2.4.5, we defined *IF*, the Improvement Factor, as the quotient between the time of non-using any technique (i.e., No Mutant Schema and All against all) with the time of using one or more cost-reduction techniques:

$$IF = \frac{|M|\cdot(T_{compile}+T_{push}+T_{install}+\rho\cdot T_{run})}{n\cdot(T_{compile}+T_{push}+T_{install})+|M|\cdot\rho\cdot T_{run}} \quad (10)$$

We also concluded that the improvement factor got by any combination of the analyzed techniques tends to stabilize when the number of mutants grows up. Since the assumptions made in the mathematical models are acceptable, we can draw the tendencies with arbitrary values for the different variables involved in Eq. 10. Top side of Figure 56 shows the tendency of *IF* with different values of ρ and when the number of mutants (horizontal axis) grows up:

- The dotted line evidences that the benefit of using Mutant Schema with *All against all* ($\rho=1$) is low if the number of mutants is high.
- The scatted line shows the tendency in *IF* with $\rho=0.5$: this is, tests are executed with *Only Alive* and there is a "medium good luck" in the execution order of test cases.
- Finally, the solid line is the tendency with $\rho=0.1$. This situation could correspond either to a "very good luck" in the execution

order of test cases or, much better, to a "smart" execution algorithm that prioritizes the test cases with more ability to kill mutants. We will recall this point in the Future works section.

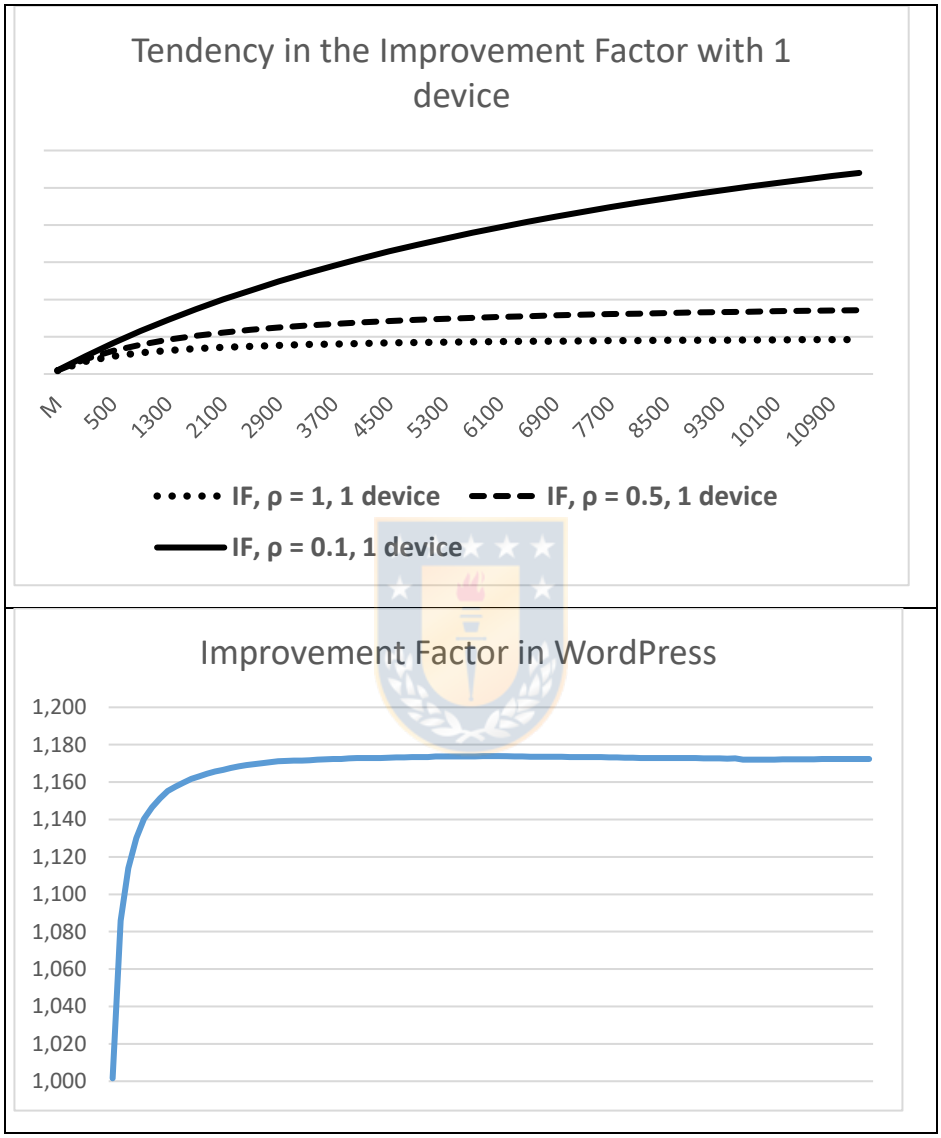


Figure 56. Theoretical tendency in the Improvement Factor with 1 device and different values of ρ (top) and observed tendency in WordPress

The actual tendency observed in WordPress (as in all the Android apps) is as shown in the bottom side of Figure 56 and coincides with the trend of the predictive model.

Partial conclusions

As it was predicted in the Mathematical Model, the experimental data show that Mutant Schema always improves the mutation testing time: the improvement factor rises quickly when the number of mutants is low, but it stabilizes and tends to a constant from a certain number of mutants.

4.6 ANALYSIS OF THE EXPERIMENTS

The experimental data have shown that: (1) the estimated times of our mathematical models and the real times obtained in the execution of test cases are very similar, (2) the most efficient combination of the applied cost reduction techniques is: Mutant Schema, Parallel Execution and Only Live and (3) Parallel execution is the most cost-saving technique. As for Mutant Schema technique, the results obtained are interesting, because although this technique significantly reduces the packaging time of the mutants and therefore improves the total execution time of the mutation testing, two aspects are questionable: (1) its improvement stabilizes and tends to a constant from a certain number of mutants and (2) it much more difficult to find the reason why a mutant remains alive due to a poorer legibility of the code.

Due to the number of applications, mutants, test cases, combinations of techniques and -in order to get reliable measures- the number of repetitions of each task, the process of experimentation and data collection has been very long. From the observation of *BacterioWeb v.1* and *BacterioWeb v.*2 during test execution, from how it fills-in the killing matrixes and from the many results analysis made, we have learned new lessons and provided some ideas to reduce the high cost of mutation testing:

4.6.1 Test suite reduction

The reduction of the test suite size cannot make sense if the SUT's state depends on the test cases previously executed: suppose the test suite

shown in Figure 57, which contains two test cases *test1* and *test2* that respectively insert and delete a customer from a database.

```
public void test1() {
   SUT sut = ...;
   Database db = sut.getDatabase();
   db.removeAllCustomers();

   db.insertCustomer("John", "Smith");

   assertTrue(db.gestCustomers()==1);
}
```

```
public void test2() {
   SUT sut = ...;
   Database db = sut.getDatabase();

   try {
       db.deleteCustomer("John", "Smith");
       assertTrue(db.gestCustomers()==0);
   } catch (Exception e) {
       fail("Customer not found");
   }
}
```

Figure 57. Two test cases in a supposed test suite

If the mutants that *test1* kills are all included in those killed by *test2*, *test2* is selected for the reduced test suite. However, when *test2* is executed in isolation, the test case will always reach the *fail* statement, since the customer inserted in *test1* (that is no longer executed) will not be contained in the database.

We have observed this situation, for example, in *Mangosta*, the project we are using as running example:

- Figure 58 (a) shows a fragment of the killing matrix with 27 of its 31 test cases and some randomly selected mutants. The mutation score is 0.75. Figure 58 (c) shows the summary of the execution: 3 test cases compose the selected reduced test suite: *tryToCreateChatWithoutAddingUsersFirst*, *loadMembersAsAdminAndGoToEdit* and *tryToAddCommentWithoutText.*
- Figure 58 (b) shows some cells of the killing matrix after executing only the reduced test suite. The results are quite different:
    - With the whole test suite, the first test case kills mutants 8, 12, 16, 19, 33 and 35. This test is also the only that kills mutant 24.

- Executing only the reduced test suite, the same test leaves those mutants alive. Moreover, the three test cases kill mutant 24. Note moreover the quite different mutation score of this supposedly equivalent test suite, that is only 0.05 (top-left cell of the killing matrix).

In order to successfully reduce a set of test cases, and for the reduced test suite to kill the same mutants and obtain the same mutation score as the original test suite, it is imperative that each test case be repeatable, autonomous and independent of the other cases in the test suite.

(a) Mutation Score = 0.75

(b) Mutation Score = 0.05

(c) Summary of the execution of the whole test suite

Figure 58. Reducing the test suite does not always produce reliable results

4.6.2 Test case prioritization

(1) In some projects there are test cases whose execution takes much more time than others. The 14 test cases of *AlarmClock*, for example, are grouped in the three files appearing in Table 25. Mean execution times of Alarm Clock's test cases, that also shows the mean execution time of each test case. As it is seen, the last test case (*snoozeAlarm…)* needs more than 1 minute, what slows down the overall execution time very much. Since the mutation process must only start after all the test cases do not find any error in the original system, it is a good idea to organize the execution against mutants in groups of test cases, sorting them by the expected execution time before launching the tests or, even, excluding the longest test cases from the test suite.

Test file	Test case	Mean time (ms)
DaysOfWeekTest	testSaturday…	1900
	testMondayA…	1932
	testSunday…	2095
DurationUtilsTest	testBreakdown	2139
AlarmTest	setRecurringDays…	2014
	snoozeAlarm…	2170
	alarm_RingsAt_... (1)	2205
	alarm_RingsAt_... (2)	2220
	alarm_RingsAt_... (3)	2273
	alarm_RingsAt_... (4)	2293
	alarm_RingsAt_... (5)	2418
	alarm_RingsAt_... (6)	2443
	snoozeAlarm_IsSnoozed_...	62172

Table 25. Mean execution times of Alarm Clock's test cases

The mutation testing processes of Offut [2] and of Polo and Reales [1] include, as an essential technique to reduce costs, the execution of tests only against the mutants remaining alive (what we have called *Only Alive).* Figure 59 redefines the mutation testing process proposed by Polo and Reales [1], with the consideration of the execution time (specially worrying in mobile testing) as a mechanism technique for cost reduction: the tester starts executing the test suite *T* against the SUT, *S.* If there are no errors, s/he separates (node 3) the test cases in several test files ($TF1 \ldots TF_n$) according to the test case execution times. If it is the first execution, mutants must be generated (node 4) and the test files iteratively launched against the mutants, removing the killed mutants (node 6) and analyzing the mutation score: if the prefixed threshold is reached, the process can stop. Otherwise, if there are more test files, the tester launches the next one against the mutants; if there are no more test files, s/he must create a new one to visit and try to kill the mutants remaining alive (node 7). This new test file is executed against the original system on (node 8): if it finds any error, the system must be fixed (and new mutants will have to be later generated because *S* has changed); otherwise, the new file can be directly executed against *M*.

For the case of the *AlarmClock* project, excluding the longest test case from the first execution saves 4 seconds in the execution of every mutant. In the second iteration (node 5), this test case will be launched only against the mutants remaining alive, which are far fewer than before. The risk, with this approach, is that test cases in the first test files kill a small number of mutants.

(2) All test cases under the *androidTest* folder (Figure 5) require the generation of an *apk* file and its installation onto a device. Some

testers leave the unit tests in this folder, what slows down the execution time. It is important to leave the unit tests in its own folder since they are executed much more quickly.

(3) To avoid the execution of test cases against mutants that will not be visited, the tester should not generate mutants with Android-specific operators for executing unit tests.

(4) When *BacterioWeb v.2* fills the killing matrix, it shows a number that indicates the order in which the test case has been executed against every mutant (see Figure 61). This information is interesting for sorting the test suite when facing future regression test cycles.

(5) The process described in Figure 59 can be adapted for regression testing: suppose a system *S* composed by classes *A*, *B* and *C*. Let be *TS* a test suite that (1) does not find any fault in *S*, (2) is mutation-adequate and (3) only contains the best test cases obtained from the application of a test suite reduction algorithm. If one the classes in the system (let be *A*) changes after the addition of, for example, a new functionality, the tester must, in the first time, to re-execute *TS* against thew new version of *S* (let be *S'*) to find possible new faults. If TS does find no faults, then it is recommendable to generate new mutants (i.e., *A1*, *A2*, etc.) only for the classes that have changed and re-execute *TS* only against these new mutants. According to the figure, if *TS* does not reach the mutation score threshold, new test cases should be added to *TS*.

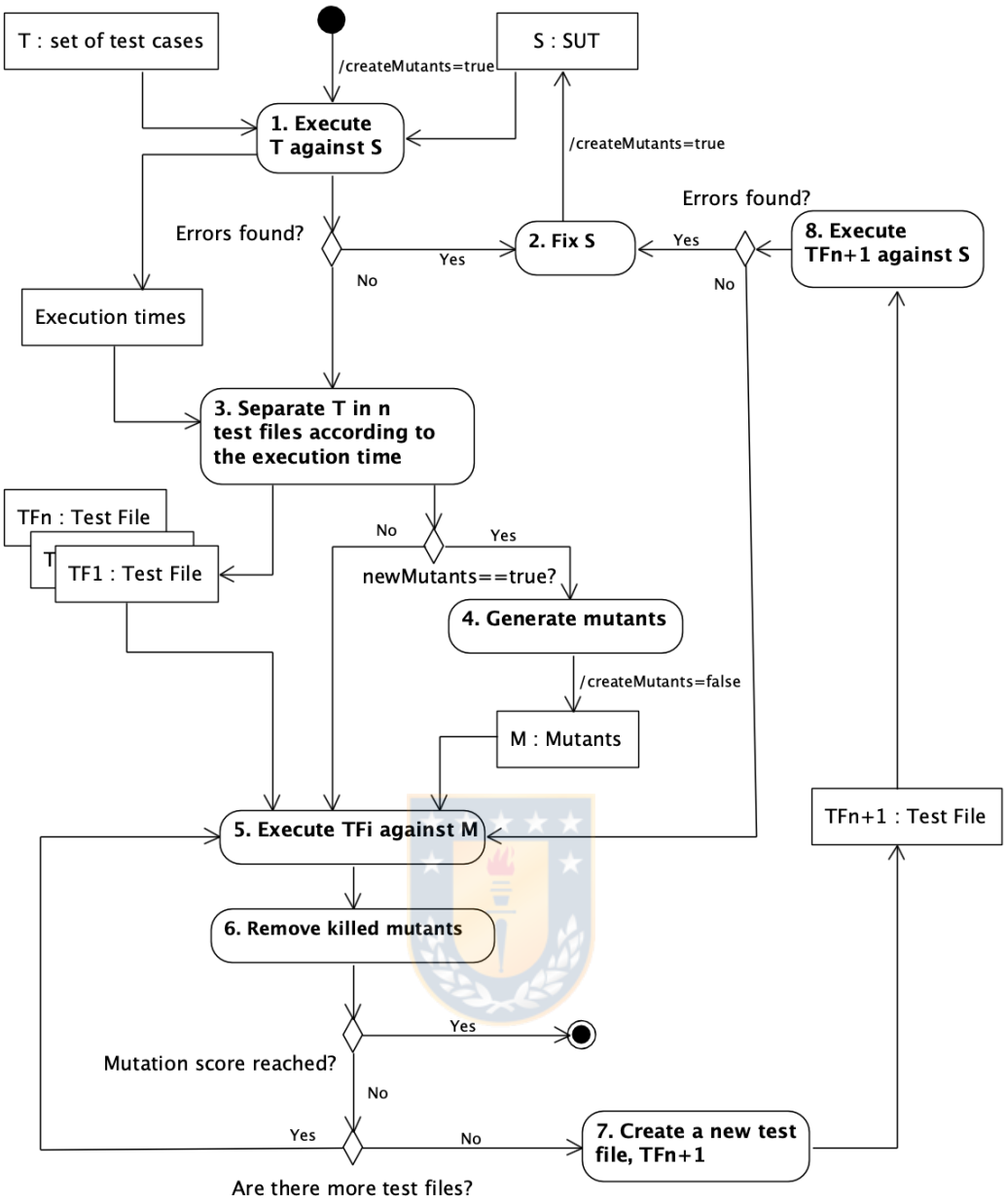


Figure 59. A mutation process, specifically adapted to mobile software

4.6.3 Structure of test cases

(1) Some mutants may lead the app to enter in an infinite loop (for example, if a counter variable is decreased inside a loop). Android test cases can be annotated with a timeout label: if the test case has not finished after this timeout, the mutant is considered killed. Some authors resolve this with weak mutation by the instrumentation of the code.

(2) It is also interesting to include frequent assertions in test cases (i.e., not only an oracle at the end of the test case): when we started to test *Kuar*, every test case reproduced a complete match. This required performing many movements to drive the board to its final state (from the left side of Figure 60 to the right side) and took a long time because initially there was only one oracle instruction (*assertX)* at the end of each test case to check the final result. To detect killed mutants as soon as possible, we introduced frequent assertions (one *assert* after each movement). This considerably accelerates test execution.

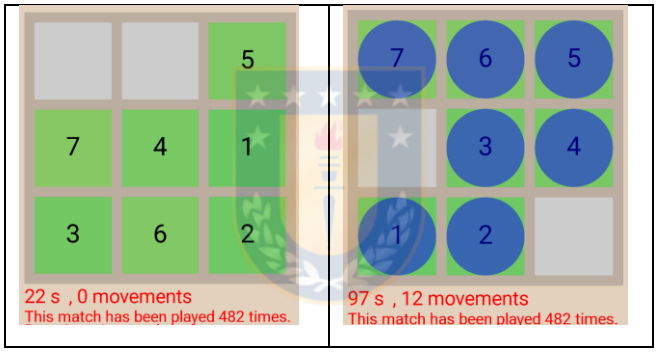


Figure 60. Kuar's board

4.7 THREATS TO VALIDITY

The nature of the experiments introduces some threats to validity, which must be considered in order to evaluate the conclusions.

Construct validity is the degree to which independent and dependent variables are accurately measured [17]. All our independent variables are nominal (*presence* or *absence of: Mutant Schema*, *Parallel Execution* and *Only Alive*), and the dependent variable (*time*) is measured objectively by the mutation tool. To alleviate bias, we performed several repetitions of each execution so as to reduce the threat.

Internal validity is the degree of confidence in a cause-effect relationship between the factor of interest and the observed results [17]. All the variables have been controlled during the experiments in order to minimize threats to internal validity.

External validity is the extent to which the research results can be generalized to the population under study and other research settings [17]. Obviously, it is quite risky to affirm that our models are valid and applicable to any other application and environment. In order to alleviate this threat, we have used a diverse set of apps with different characteristics and a variety of mutation operators.

With respect to the execution algorithms (*All against all* and *Only against alive*), it is important to note that, in our case, they are completely deterministic and do not implement any technique to prioritize the execution order of test cases. However, we consider that the similarity of the compilation, push and installation times, regardless of the use or not of Mutant Schema will allow the generalization of the mathematical model to other applications, contexts and environments (not only mobile mutation tests) and to realize other theoretical models that should be experimentally validated.

FUTURE WORK

Besides the convenience of applying cost-reduction techniques, mutation testing for mobile software is quite costly and it requires researching on new techniques that, furthermore, could be extended to other kinds of systems. Below we describe some lines of work we consider quite interesting and that could drive future research.

5.1 SPECIFIC OPERATORS FOR MOBILE SOFTWARE AND OPERATORS SUBSUMPTION

In this research we have applied classical and Android-specific mutation operators. Some of these have been reproduced from the descriptions given in the literature [15], [14] and we have proposed some others. The idea of mutation operators specifically built for a concrete technology is to introduce typical errors of such technology.

Many of the Android-specific operators introduce errors that may be also inserted by classic operators. Consider for example the MDL operator that deletes a lifecycle method of an activity. Deng et al. [15] propose this operator but they warn that it is "similar to the Overriding Method Deletion in muJava" [26]. Thus, if the tool offers the tester both operators, two redundant mutants will be generated.

Another example is IPR, which replaces the second parameter of the *putExtra* method by a default value (zero if it is primitive, *null* and the empty string if it is a String, etc.). MJP is like IPR, but changes the values put in JSON objects. Actually, the same mutants can be generated with other classical operators, such as the Scalar Variable Replacement operator of the classic Mothra system [99].

Therefore, it is likely required to carry out an extensive study of operators' subsumption, to avoid the generation of duplicate mutants.

5.2 MUTANT GENERATION GUIDED BY METRICS

There are many studies that correlate software metrics with the fault-proneness of the system's modules [100]–[102]. With a previous static analysis of the system, the tester could focus mutant generation on those classes that have more coupling, which is the best predictor according to those studies.

As an example, the *Figures* project (that was specifically developed for testing some of the *BacterioWeb v.2* characteristics) has the *LocalCalculus* and *RemoteCalculus* classes, which are used to determine the type and perimeter of the figure in the self app or via a query to a web server. Since almost any test scenario runs one of these two classes, it can be promising to determine "execution clusters" to concentrate the mutant generation on them.

5.3 MUTANT EXECUTION GUIDED BY STATIC ANALYSIS

Aforementioned "execution clusters" would avoid the execution of test cases against mutants that they will not kill: for example, it probably makes no sense to execute a test case that determines the type of a *Triangle* against a mutant of *Quadrilateral*. Before the execution of the tests, a static analysis of the code could help to relate test cases with classes of the system under test, therefore producing a more fine-grained set of clusters, composed now by tests and classes of the SUT. The result of this analysis would guide the execution of each test only against the mutants from the classes it will visit.

Some authors have researched on mutant clustering to reduce the execution time, but applying other strategies: Ma and Kim [57] and Yu and

Ma [103] built the clusters based on the mutants that are expected to produce the same result with test cases. Ji et al. [104] proposed to cluster mutants based on their Hamming distance.

5.4 ALGORITHMS FOR PARALLEL EXECUTION

At first glance, putting more devices to execute tests and mutants is a brute force mechanism that, undoubtedly, improve the overall execution time.

As a mean, we can guess that using 2 devices will require half time than using 1, and that using *n* will take *1/n*. This premise is generally true, but during the experimentation we have observed situations where that assumption is false.

Figure 61 shows an example of *BacterioWeb v.2* executing the *Mangosta* test cases against a small sample (only 2%, since the figure is only for illustrating purposes) of mutants. As it can be seen, it is applying the *Only against alive* algorithm (see in the figure the selected "Matrix mode" radio button) and *With Mutant Schema*. There are four devices (Samsung tablets, model SM-T590, Android 8.1.0, 3 Gb RAM) that have received 17 mutants each. The 2nd, 3rd and 4th devices have finished the execution, whilst the first one still has 4 mutants left.

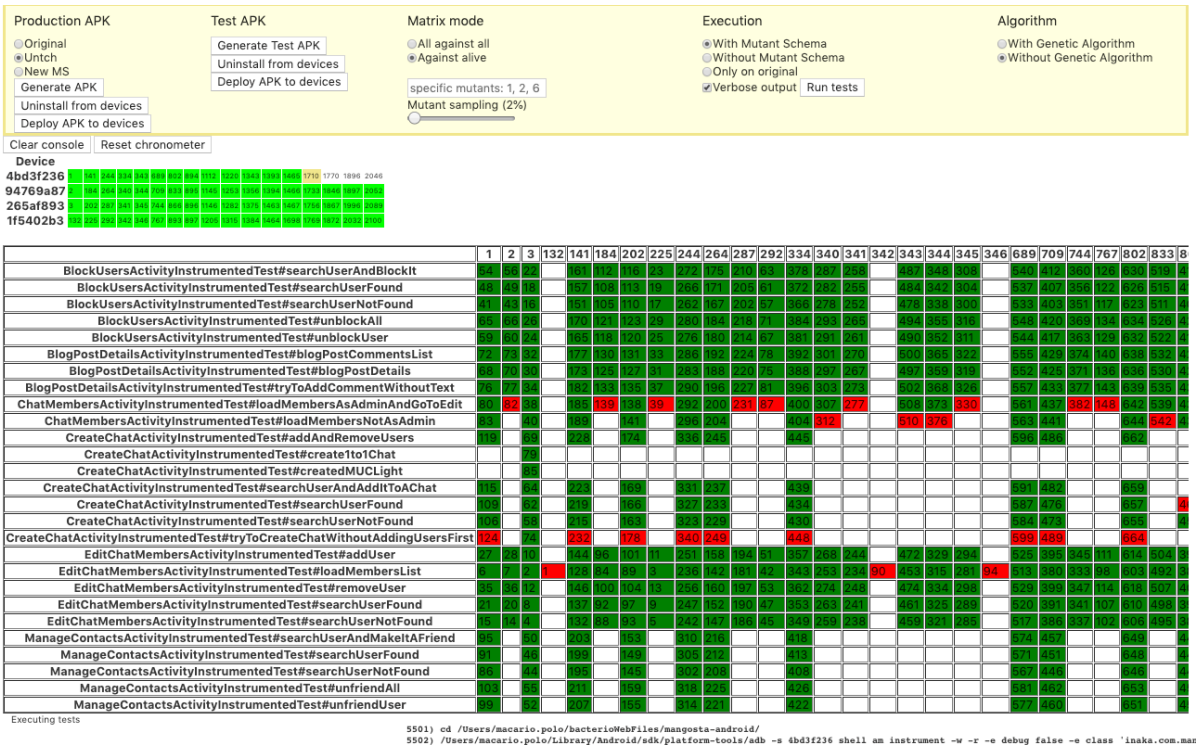


Figure 61. Killing matrix during one execution

A more equitable time distribution could be achieved by distributing the mutants with other parallel execution algorithms. For example, when a device finishes the execution of the test suite (either with *Only against alive* or with *All against all*), it could ask for a new mutant to be executed.

Polo et al. [84] analyzed five different algorithms for parallel execution in mutation testing, which can improve the performance of test execution: *Distribute mutants between operators*, *Distribute test cases*, *Give mutants on demand*, *Give test cases on demand* and *Parallel execution with dynamic ranking and order*.

It is possible to build new mathematical models or to extend those in Section 3.2.4, with the inclusion of these or of other algorithms.

CONCLUSIONS

The main contributions of this thesis are related to how several well-known cost reduction techniques help to the effective improvement of testing time. The techniques we have used are Mutant Schema, Only Alive and Parallel Execution, as well as several combinations of them. The baseline for the comparisons is a classic model of mutation testing, where there is a complete cycle of compilation, deployment, and test execution per mutant.

The first technique considered is Mutant Schema, which requires to generate the schema and just one deployment onto the running device. Being the deployment a very significant task in mobile testing, the obtained results always show meaningful cost savings when Mutant Schema is applied. However, an interesting finding is that its improvement factor tends to be asymptotic with respect to the number of mutants: in fact, as more mutants there are and greater is the execution time, less significant is the influence of the deployment time on the total cost. In addition, we give technical details about the implementation of the Mutant Schema technique for its reproducibility and improvement.

The second technique used compares the execution of the test cases against all the mutants versus only the mutants remaining alive. Here, the improvement in the testing time depends on the quality of the test cases: the earlier the mutants are killed, the less test cases will have to be executed.

The third technique is Parallel execution, which has evidenced to be the most influencing cost-reduction factor. The experiments have shown that the reduction in time is roughly proportional to the number of devices. The total execution time is the time required by the device that needs more time for executing its set of mutants. In this factor influence both the characteristics of the device (memory, processor...) as the nature of every

mutant. When the number of mutants is very high, they are fairly distributed on the devices (both the "quick" and the "slow" mutants). Thus, the reduction is not strictly 1/n, but it is very approximate. Even though, the reduction may be improved with other parallel execution algorithms (see future work 5.4).

In our opinion, this research complements other research works on mutation testing applied to mobile software. One additional contribution of this research is the suitability of the proposed mathematical models to estimate a priori the time required to perform a mutation test on a mobile app. This result is interesting to build prediction models before implementing tools and techniques for mutation testing, what can shorten the research times. On the other hand, we present the design and architecture of *BacterioWeb v.2*, a web tool for the mutation testing of mobile applications in a distributed environment; with the goal of enabling mutation testing for testers teams and contribute to the transition of mutation testing from academic to industrial application.

BIBLIOGRAPHY

[1] M. P. Usaola and P. R. Mateo, "Mutation Testing Cost Reduction Techniques: A survey," *IEEE Softw.*, vol. 27, no. 3, pp. 80–86, 2010.

[2] A. J. Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer," in *Proceedings., International Test Conference*, 1994, pp. 824–830.

[3] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," *Tech. Rep. ISSE-TR-96-100*, Dept. of Information and Software Systems Eng., George Mason Univ., pp. 1–14, 1996.

[4] R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer.*, vol. 11, no. 4, pp. 34–41, 1978.

[5] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[6] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards Mutation Analysis of Android Apps," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[7] L. Deng, J. Offutt, and D. Samudio, "Is Mutation Analysis Effective at Testing Android Apps?," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 86–93.

[8] K. Moran *et al.*, "MDroid + : A Mutation Testing Framework for Android," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 33–36.

[9] A. Abuljadayel and F. Wedyan, “An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms,” *Int. J. Intell. Syst. Appl.*, vol. 10, no. 1, pp. 34–45, 2018.

[10] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino, “Why does the orientation change mess up my Android application ? From GUI failures to code faults,” *Softw. Testing, Verif. Reliab.*, vol. 28, no. 1, p. e 1654, 2018.

[11] B. Kushigian, A. Rawat, and R. Just, “Medusa: Mutant equivalence detection using satisfiability analysis,” in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 77–82.

[12] A. C. R. Paiva, J. M. E. P. Gouveia, J. D. Elizabeth, and M. E. Delamaro, “Testing when mobile apps go to background and come back to foreground,” in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 102–111.

[13] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, “Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 114–124.

[14] C. Escobar-Velasquez *et al.*, “Enabling Mutant Generation for Open- and Closed-Source Android Apps,” *IEEE Trans. Softw. Eng.*, pp. 1–1, Apr. 2020.

[15] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, “Mutation operators for testing Android apps,” *Inf. Softw. Technol.*, vol. 81, pp. 154–168, 2017.

[16] M. G. P. V. Marcela Genero, Marcela Genero Bocco, José A. Cruz Lemus, *Métodos de investigación en ingeniería del software*. Spain:

Ra-Ma, 2014.

[17] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, 1st ed. Springer, Berlin, Heidelberg, 2012.

[18] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

[19] M. Polo Usaola, B. Perez Lamancha, and P. Reales Mateo, *Técnicas combinatorias y de mutación para testing de sistemas software*, 1st ed. España: Ra-Ma, 2012.

[20] P. R. Mateo and M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Softw. Testing, Verif. Reliab.*, vol. 25, no. 5–7, pp. 464–489, 2015.

[21] "Robolectric." [Online]. Available: http://robolectric.org. [Accessed: 10-May-2020].

[22] A. J. Offutt, "Investigation of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 3–18, 1992.

[23] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation Based Software Testing," *Softw. Pract. Exp.*, vol. 21, no. 7, pp. 685–718, 1991.

[24] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 3, pp. 228–247, 2001.

[25] M. E. Delamaro, "Interface Mutation Test Adequacy Criterion : An Empirical Evaluation," *Empir. Softw. Eng.*, vol. 6, no. 2, pp. 111–142, 2001.

[26] Y. S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[27] P. Reales Mateo, M. Polo Usaola, and J. Offutt, “Mutation at the multi-class and system levels,” *Sci. Comput. Program.*, vol. 78, no. 4, pp. 364–387, 2013.

[28] H. Shahriar and M. Zulkernine, “Mutation-based Testing of Format String Bugs School of Computing,” in *2008 11th IEEE High Assurance Systems Engineering Symposium*, 2008, pp. 229–238.

[29] A. Derezińska, “Advanced mutation operators applicable in C# programs,” in *Software Engineering Techniques: Design for Quality*, 2006, vol. 227, pp. 283–288.

[30] A. Derezi and A. Szustek, “Tool-Supported Advanced Mutation Approach for Verification of C# programs,” in *2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, 2008, pp. 261–268.

[31] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano, “Class mutation operators for C++ object-oriented systems,” *Ann. Telecommun.*, vol. 70, no. 3–4, pp. 137–148, 2015.

[32] H. Shahriar and M. Zulkernine, “MUSIC : Mutation-based SQL Injection Vulnerability Checking School of Computing,” in *2008 The Eighth International Conference on Quality Software*, 2008, pp. 77–86.

[33] J. Tuya, M. J. Suárez-cabal, and C. De Riva, “SQLMutation : A tool to generate mutants of SQL database queries,” in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, 2006, pp. 1–1.

[34] P. Anbalagan and T. Xie, “Automated generation of pointcut mutants for testing pointcuts in aspectj programs,” in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[35] P. Anbalagan and T. Xie, “Efficient mutant generation for mutation

testing of pointcuts in aspect-oriented programs," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, 2006.

[36] F. C. Ferrari, S. Paulo, and C. Sp, "Mutation Testing for Aspect-Oriented Programs," in *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 52–61.

[37] A. Derezińska and K. Hałas, "Analysis of mutation operators for the Python language," in *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, Brunów, Poland. Advances in Intelligent Systems and Computing*, 2014, vol. 286, pp. 155–164.

[38] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Inf. Softw. Technol.*, vol. 49, no. 4, pp. 398–417, 2007.

[39] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer, "Towards systematic mutations for and with ATL model transformations," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[40] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for WS-BPEL compositions," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 142–150.

[41] S. S. Batth, E. R. Vieira, A. Cavalli, and M. Ü. Uyar, "Specification of Timed EFSM Fault Models in SDL," in *International Conference on Formal Techniques for Networked and Distributed Systems. Springer*, 2007, pp. 50–65.

[42] N. Bombieri, F. Fummi, and G. Pravadelli, "A Mutation Model for the SystemC TLM 2.0 Communication Interfaces," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, pp. 396–401.

[43] G. Vigna, W. Robertson, and D. Balzarotti, "Testing Network-based

Intrusion Detection Signatures Using Mutant Exploits," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004, pp. 21–30.

[44] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu, "Mutation Testing of Protocol Messages Based on Extended TTCN-3," in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, 2008, pp. 667–674.

[45] S. Lee, X. Bai, and Y. Chen, "Automatic Mutation Testing and Simulation on OWL-S Specified Web Services," in *41st Annual Simulation Symposium (anss-41 2008)*, 2008, pp. 149–156.

[46] J. XU, Wuzhi; OFFUTT, Jeff; LUO, "Testing Web services by XML perturbation," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005, pp. 10–266.

[47] Y. L. Traon, T. Mouelhi, and B. Baudry, "Testing Security Policies: Going Beyond Functional Testing," in *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, 2007, pp. 93–102.

[48] T. Mouelhiv, F. Fleurey, and B. Baudry, "A Generic Metamodel For Security Policies Mutation," in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 278–286.

[49] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 23–42, 2014.

[50] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 192–199.

[51] T. A. Budd, "Mutation Analysis of Program Test Data," ProQuest Dissertations Publishing, Ann Arbor, United States, 1980.

[52] A. P. Mathur and W. E. Wong, “An empirical comparison of data flow and mutation-based test adequacy criteria,” *Softw. Testing, Verif. Reliab.*, vol. 4, no. 1, pp. 9–31, 1994.

[53] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, “An Extended Overview of the Mothra Testing Environment,” in *Workshop on Software Testing, Verification, and Analysis*, 1988, pp. 142–151.

[54] A. Derezinska and M. Rudnik, “Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing,” in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2017, pp. 1315–1324.

[55] S. Hussain, “Mutation Clustering,” Master’s thesis., Kings College, London, 2008.

[56] A. Derezińska, “A quality estimation of mutation clustering in c# programs,” in *New Results in Dependability and Computer Systems*, 2013, pp. 119–129.

[57] Y. S. Ma and S. W. Kim, “Mutation testing cost reduction by clustering overlapped mutants,” *J. Syst. Softw.*, vol. 115, pp. 18–30, 2016.

[58] Y. Jia and M. Harman, “Constructing subtle faults using Higher Order mutation testing,” in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 249–258.

[59] M. Polo, M. Piattini, and I. Garía-Rodríguez, “Decreasing the cost of mutation testing with second-order mutants,” *Softw. Test. Verif. Reliab.*, vol. 19, no. 2, pp. 111–131, 2009.

[60] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo, “Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation,” in *Proceedings of the 29th ACM/IEEE international*

conference on Automated software engineering - ASE '14, 2014, pp. 397–408.

[61] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *J. Syst. Softw.*, vol. 83, no. 12, pp. 2416–2430, 2010.

[62] A. P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," in *1991 The Fifteenth Annual International Computer Software & Applications Conference*, 1991, vol. 1, pp. 604–605.

[63] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of 1993 15th International Conference on Software Engineering*, 1993, pp. 100–107.

[64] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 571–582.

[65] P. Delgado-Perez, I. Medina-Bulo, and M. Nuñez, "Using Evolutionary Mutation Testing to Improve the Quality of Test Suites," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, 2017, pp. 596–603.

[66] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Trans. Reliab.*, vol. 66, no. 3, pp. 854–874, 2017.

[67] C. A. Sun, L. Pan, Q. Wang, H. Liu, and X. Zhang, "An empirical study on mutation testing of WS-BPEL programs," *Comput. J.*, vol. 60, no. 1, pp. 143–158, 2017.

[68] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proceedings of the 7th Joint Meeting of the European*

Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2009, pp. 297–298.

[69] P. R. Mateo and M. P. Usaola, "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 646–649.

[70] B. Bogacki and B. Walter, "Evaluation of Test Code Quality with Aspect-Oriented Mutations," in *Extreme Programming and Agile Processes in Software Engineering*, 2006, pp. 202–204.

[71] B. Bogacki and B. Walter, "Aspect-oriented response injection: An alternative to classical mutation testing," in *Sacha K. (eds) Software Engineering Techniques: Design for Quality. IFIP International Federation for Information Processing*, 2006, vol. 227, pp. 273–282.

[72] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *SIGSOFT Softw. Eng. Notes*, vol. 18, no. 3, pp. 139–148, 1993.

[73] S. W. Kim, Y. S. Ma, and Y. R. Kwon, "Combining weak and strong mutation for a noninterpretive Java mutation system," *Softw. Test. Verif. Reliab.*, vol. 23, no. 8, pp. 647–668, 2013.

[74] M. Papadakis and N. Malevris, "Automatic Mutation Test Case Generation via Dynamic Symbolic Execution," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 121–130.

[75] P. Reales-Mateo and M. Polo-Usaola, "Mutant execution cost reduction: Through MUSIC (Mutant Schema Improved with Extra Code)," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 664–672.

[76] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[77] D. Jeffrey and N. Gupta, “Test suite reduction with selective redundancy,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, 2005, pp. 549–558.

[78] A. Marshall, D. Hedley, I. Riddell, and M. Hennell, “Static dataflow-aided weak mutation analysis (SDAWM),” *Inf. Softw. Technol.*, vol. 23, no. 1, pp. 99–104, 1990.

[79] S. D. Lee, “An Empirical Evaluation of Weak Mutation,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 5, pp. 337–344, 1994.

[80] A. J. Offutt and S. D. Lee, “How Strong is Weak Mutation?,” in *Proceedings of the Symposium on Testing, Analysis, and Verification*, 1991, pp. 200–213.

[81] C. Byoungju and A. P. Mathur, “High-performance mutation testing,” *J. Syst. Softw.*, vol. 20, no. 2, pp. 135–152, 1993.

[82] A. J. Offutt, R. P. Pargas, S. V Fichter, and P. K. Khambekar, “Mutation Testing of Software Using a MIMD Computer,” in *1992 International Conference Parallel Processing*, 1992, vol. 2, pp. 257–266.

[83] C. J. Wright, G. M. Kapfhammer, and P. McMinn, “Efficient mutation analysis of relational database structure using mutant schemata and parallelisation,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 63–72.

[84] P. Reales-Mateo and M. Polo Usaola, “Parallel mutation testing,” *Softw. Testing, Verif. Reliab.*, vol. 23, no. 4, pp. 315–350, 2013.

[85] A. J. Offutt and J. Pan, “Automatically detecting equivalent mutants and infeasible paths,” *Softw. Testing, Verif. Reliab.*, vol. 7, no. 3, pp. 165–192, 1997.

[86] B. Kirubakaran and V. Karthikeyani, “Mobile application testing — Challenges and solution approach through automation,” in *2013*

International Conference on Pattern Recognition, Informatics and Mobile Engineering, 2013, pp. 79–84.

[87] M. Linares-Vásquez *et al.*, "Enabling Mutation Testing for Android Apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 233–244.

[88] M. P. Usaola, G. Rojas, I. Rodriguez, and S. Hernandez, "An Architecture for the Development of Mutation Operators," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 143–148.

[89] R. Jabbarvand and S. Malek, "μDroid: An Energy-aware Mutation Testing Framework for Android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 208–219.

[90] I. C. Morgado and A. C. R. Paiva, "Impact of Execution Modes on Finding Android Failures," *Procedia Comput. Sci.*, vol. 83, pp. 284–291, 2016.

[91] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," *Adapt. Extensible Compon. Syst.*, vol. 30, no. 19, 2002.

[92] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Education, 2009.

[93] N. Smith, D. Van Bruggen, and F. Tomassetti, "JavaParser-Home," *JavaParser: Visited Analyse, transform and generate your Java code base*, 2019. [Online]. Available: https://javaparser.org/. [Accessed: 11-May-2020].

[94] R. Just, F. Schweiggert, and G. 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)*, 2011, pp. 612–615.

[95] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble java byte code to measure the effectiveness of unit tests," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 169–175.

[96] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Softw. Testing, Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[97] G. Myers, *The Art of Software Testing, Second edition*. 2004.

[98] G. Fraser and A. Gargantini, "Experiments on the test case length in specification based test case generation," in *2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 18–26.

[99] B. J. Choi *et al.*, "The Mothra tool set (software testing)," in *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, 1989, vol. 2, pp. 275–284.

[100] Y. Luo, K. Ben, and L. Mi, "Software Metrics Reduction for Fault-Proneness Prediction of Software Modules," in *Network and Parallel Computing*, 2010, pp. 432–441.

[101] A. Boucher and M. Badri, "Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software," in *2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD)*, 2016, pp. 169–176.

[102] Ping Yu, T. Systa, and H. Muller, "Predicting fault-proneness using OO metrics. An industrial case study," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, 2002, pp. 99–107.

[103] M. Yu and Y. S. Ma, “Possibility of cost reduction by mutant clustering according to the clustering scope,” *Softw. Testing, Verif. Reliab.*, vol. 29, no. 1–2, p. e1692, 2019.

[104] C. Ji, Z. Chen, B. Xu, and Z. Zhao, “A Novel Method of Mutation Clustering Based on Domain Analysis,” in *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE 2009)*, 2009, pp. 422–425.