FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Test Case Mutations to Improve Tests Quality

Carolina Vasconcelos Castro Azevedo

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Paiva

Co-Supervisor: André Restivo

April 1, 2021

Test Case Mutations to Improve Tests Quality

Carolina Vasconcelos Castro Azevedo

Mestrado Integrado em Engenharia Informática e Computação

April 1, 2021

Abstract

Testing is the process concerning all life cycle activities regarding planning, preparation and evaluation of software products. The purpose of tests is to verify if the software requirements were satisfied and demonstrate if they are suitable for their purpose, detecting the software's defects. Accordingly, testing is a crucial part of the development process concerning the software's verification and validation. Its importance comes from attributing confidence to the quality of the software it tests. Software has never been so important and mandatory as it is nowadays. Since testing is essential to ensure the software's quality, testing software has many limitations that become even more imperative to overcome. One of the most evident testing problems is the amount of time and effort developers have to invest in testing the software properly. Building test cases automatically could be a solution to this dilemma. However, the insurance of an adequate coverage level is still a challenge since all the methods developed in this matter still have a tremendous manual effort at some stage. Moreover, when this type of software services exist, they require a high maintenance level when changes occur. However, since there is no battery of test suits to be executed in these cases when the software suffers modifications, the software's quality can not be guaranteed. Mutation Testing is a white boxing testing technique and is considered one of the most effective methods in assessing the quality of input values and test cases. This technique resorts to the injection of mutants, which are modifications of the program's original component. These modifications intend to mimic real development mistakes, allowing the possibility to detect the most vulnerable parts of the test cases, exalting flaws where the code is shielded or usually improbable to reach. However, in this research we do not apply mutations to the code itself; we apply mutations to the generated test cases from execution traces. Accordingly, this research generates execution traces from the detection of the user's most frequent paths on a website. This retrieving is possible by using a previously developed tool called *MARTT*. The usage data is then stored and used to generate concrete test cases. These test cases were generated, extended, and reproduced to perform experiences with mutations operators resorting to other previously developed tool called *Mutations*. Lastly, experiences were conducted to reproduce the test suits generated and mutated to verify if there was an increase in the test cases' coverage. Specifically, to analyze if there were lines of the code that were not reached with the primary set of test cases generated by the most common executed traces that were reached after those tests suffer from the injection of mutants. Ultimately, verifying if the results mean an improvement of the software quality, without the effort and time usually required, and ensuring that future changes in the software will have a battery of test suits to execute.

Keywords: Software testing, software testing automation, mutation testing, software quality, software engineering

Resumo

Testar é o processo que consiste em todas as atividades do ciclo de vida que se relacionem com o planeamento, preparação e avaliação de produtos de *software*. Portanto, o objetivo de testar prende-se em determinar se os produtos satisfazem os requisitos especificados, de modo a demonstrar serem adequados tanto para a sua finalidade, como para a deteção de erros.

Sendo assim, testar é uma parte crucial para a verificação e validação de um produto durante o seu processo de desenvolvimento. Consequentemente, a importância de testar advém da sua atribuição de confiança de qualidade do *software* que está a ser testado.

Testar *software* nunca foi tão importante e crucial como hoje em dia. Visto que testar se torna essencial para assegurar a qualidade do *software*, as muitas limitações associadas a este tópico nunca foram tão imperativas de ultrapassar.

Um dos problemas mais evidentes de testar *software* é o investimento abundante de tempo e esforço requerido por quem testa, de forma a garantir que o *software* está testado corretamente. Uma possível solução para este problema é a construção de casos de teste automaticamente. No entanto, a garantia de um nível adequado de cobertura de código permanece desafiante, devido ao facto de que todos os métodos desenvolvidos para este propósito ainda dependerem de uma quantidade de trabalho manual considerável.

Do mesmo modo, quando estes serviços de *software* existem, normalmente requerem um elevado nível de manutenção sempre que há modificações. Sendo assim, não há nenhuma bateria de testes que possa ser executada nestas situações em que o *software* sofre alterações, de maneira a garantir a qualidade dos testes.

Mutation Testing é uma técnica de *white box testing* e é considerada, hoje em dia, como uma das técnicas mais eficazes na avaliação da qualidade dos valores de *input* e dos casos de teste. Devido à injeção de mutantes, ou seja, pequenas modificações nas componentes primárias de um programa, feitas com a intenção de imitar potenciais erros no desenvolvimento, é possível detetar as partes mais vulneráveis do *software*, enaltecendo falhas até onde o código está protegido.

Porém, neste trabalho não se aplicam mutações ao código em si, aplicam-se mutações a casos de teste gerados a partir de traços de execução efetuados por utilizadores. Sendo assim, as experiências conduzidas foram efetuadas de maneira a reproduzir os casos de teste gerados e extendidos com mutações, para verificar se ocorreu um aumento na cobertura de código do *software*. Mais concretamente, para analisar se houve linhas do código que não eram alcançadas pelos casos de teste gerados inicialmente, que posteriormente foram cobertas pelos casos de teste mutados.

Finalmente, os resultados destas experiências verificam se houve ou não um aumento na qualidade dos casos de teste, evitando o esforço e tempo requeridos normalmente, e garantindo que o *software* teria uma bateria de testes para executar caso ocorram mudanças.

Keywords: tests de *software* , automação de testes de*software*, testes de mutação, qualidade de testes, engenharia de software

Acknowledgements

To Professor Ana Paiva and Professor André Restivo, my sincere appreciation for turning this research into something that made me proud. Their patience, care and availability were crucial for the entire process.

To my mom, the most extraordinary woman I have ever met, that made sure that I had everything I possibly needed to get a degree. I cannot thank you enough because there are no words to describe the gratitude that I feel. Also, to my sister Marta, for always supporting me and accepting me with all my flaws.

To the friends that made all of this possible, that struggled with me during this years, they are the family I chose: Daniela, Nelson, Diogo, Gonçalo Bernardo, França, Daniel, Sofia, Leo, Gonçalo Moreno, João. Without them, these years would not be full of joy, adventures and victories, I am sure.

To my family at AEFEUP, that allow me to grow not only academically but also personally. Together we did several successful projects, through many sleepless nights and had more fun than anyone can image: Xavi, Xana, Bocage.

Last but not least, to Ricardo, for being everything that I needed, sometimes without even knowing it, every day. I am sure that without you, this journey would have been a lot more painful.

Carolina Vasconcelos Castro Azevedo

*"There is no royal flower strewn path to success.
And if there is, I have not found it, for whatever success I have attained
has been the result of much hard work and many sleepless nights."*

Madam C. J. Walker

Contents

List of Figures

Abbreviations

AUT	Application Under Test
GUI	Graphical User Interface
IQ	Intelligence Quotient
JSON	JavaScript Object Notation
MBMT	Model-Based Mutation Testing
MBT	Model-Based Testing
PHP	Hypertext Preprocessor
UI	User Interface
URL	Uniform Resource Locator
SAT	Scholastic Assessment Test
STLC	Software Testing Life-Cycle
SUT	System Under Test
XML	Extensible Markup Language
XPATH	XML Path Language

Chapter 1

Introduction

Nowadays it is practically impossible to not recognise software as one of our daily lives' vital constituents. From our work to our house, the software is among everything we see, and with it, there is also software engineering.

If we pay close attention to software development, we promptly realise that software testing is one of its core components. Hence, by definition of the International Software Testing Qualifications Board [1], software testing is the process regarding all activities (static and dynamic) concerned with planning, preparing, and evaluating a system. Furthermore, the related products determine if the system requirements are satisfied, demonstrating if they fit the purpose and detecting the defects in them [2].

Defining software testing is somewhat easy to do, due to the consensus of people that provided definitions. Although, comprehending the full importance of this matter, it is not trivial; in fact, many developers often neglect this subject, and their lack of attention has its damages on software quality [3].

Moreover, the software's quality is defined as the degree to which a system satisfies the stakeholders' (either stated or implied) needs [4]. Hence, the quality obtained by testing reveals to be extremely important for every involved part of the development. Additionally, as C.Kraner [5] explained, if the quality is value to the stakeholder, and if the value is driven by usability, security, performance, among other regards, then the testers should investigate and invest into these aspects.

Consequently, testing and quality are connected since the beginning, and these subjects motivated numerous investigators to discuss and write about them [2], [6], [7], [8], [9], [10], [11].

This research work intends to use one of the testing techniques, mutation testing, to improve tests quality and contribute to this important matter.

This chapter's purpose is to present the context of this work 1.1, followed by the motivation attained from it 1.2, accompanied by the problem statements that the research intends to reach 1.3. Finally, the goals for the work 1.4, and the document structure are presented 1.5.

1.1 Context

Testing is a crucial part of the validation and verification of software during its development stage since its importance arises from the attribution of quality to the software under testing [5]. However, an already existing system does not have necessarily a battery of tests to execute in order to maintain the system's quality.

Additionally, having a set of appropriate tests for the software, assuring its quality level, is a very demanding task, especially in terms of resources [7]. Hence, testing software is far from being perfect, several limitations have been challenging developers, and they prove to be mandatory to overcome [12, 13, 3, 14].

One approach commonly used lately to overcome the excessive time necessary for testing is to construct test cases automatically. Hence, it is a recent subject addressed in the recent literature to prevent excessive demand and time occupied on the testing process. Besides, it has shown to be an efficient way to improve the tests' quality, leading them to be as complete as possible [15].

Among the existing testing techniques, this work will focus on mutation testing since it is considered to be one of the most effective techniques in evaluating inputs and test cases. [16]

Mutation testing is a white box testing technique that evaluates tests since it might attribute code quality to them by injecting mutations. These mutants are a set of components emerging from modifying the program's original component, ideally mimicking real development mistakes [17]. Finally, the purpose of mutation testing is to verify if the test suit detects those mutants [3]. Such behaviour helps the developer detect the most vulnerable parts of the test cases and exalt flaws where the code is shielded or usually improbable to reach [17].

Finally, in the recent researches and advances in software, it is noticed that the resort to mutation testing has increased due to all of this essential aspects; chapter 2.1 refers more deeply this kind of literature.

1.2 Motivation

The motivation for this work resulted from the remaining hurdles and flaws that software systems developers currently face. Concretely, this work motivation emerged from the fact that testing software remains a demanding and exhausting task. [18, 19].

Furthermore, to increase software quality, we also should give attention to code coverage. Chapter 2.3 extensively refers to the literature regarding this topic. Specifically, code coverage can provide indications regarding the quality of the test cases, becoming one of the most indispensable parts of software maintenance. Therefore, the effectiveness of testing is evaluated through this information provided by the code coverage that the tests achieve. [20].

The work proposed is meant to improve the code coverage associated with the tests and increase these tests' quality, avoiding the disadvantages. These disadvantages related to software testing are more extensively explained in the following chapter 2, for instance, the time consumed on testing and resources needed, the difficulty of measuring code coverage accurately and extending tests with reasoning, among others.

Since the majority of software services are extensive, testing each and every component seems unrealistic. The expected accomplishment in this work is to generate test cases from typical user interactions with the system [21]. Accordingly, if the percentage of code covered increases, quality improves as well, reducing the time and effort required for building test cases [22], [7].

Automated testing may seem to solve all the disadvantages, such as situations where the software suffered changes that imply newly emerged problems that could be solved with this method, saving necessary time and resources. By consulting the literature regarding these methods, we realize that is not as simple as we would hope, as it is pointed more extensively in chapter 2. The methods that emerged from automated testing still involve a large amount of manual work, either from the models themselves or from documentation [23, 7]. Additionally, the majority of testing approaches do not reflect the software's real usage information, leading to test suits less efficient and useful.

It is important to refer that software is constantly changing, and these alterations demand readjustments of the test suits related to the software, implying more effort and more time, as is noted various times. Unfortunately in most cases, it does not exist in these software services a battery of tests that could be used when the changes occur. Therefore, the usage information contained on software services might be useful in an approach to rely on them to help to reduce the effort of maintaining the tests.

Finally, the hurdles and flaws remaining in software systems regarding testing, results in this thesis motivation, which emanates from the need to improve the quality of the test suits, resorting to mutation testing techniques, without the amount of effort and time usually required.

1.3 Problem Statement

Considering the context and motivation regarding this research work, this section disclosures the problem the work intends to study. The intention is to generate test cases using data from the usage of software services. Following, extend those test cases resorting to the application of mutation operators. Ultimately, to improve the tests' quality and hopefully guarantee that the quality of the software being tested remains.

To study the hypothesis of acquiring more quality to test cases without the usually demanded effort, we divided the problem into three crucial topics that need attention and planning for this work to happen:

1. **Collecting Data** It is essential to establish how to collect data from software usage in order to generate test cases from that information appropriately. If the gathered data, represents real usage of a software system, the test cases generated from that information would cover the most used parts of the software.

2. **Injecting Mutants** To extend the generated test cases using mutations, it is crucial to establish what types of mutation operators would be suited for injection.

   Furthermore, it is also essential that this injection is automated, considering our motivation for this thesis, if such injection requires manual work then it would be impracticable.

3. **Quality Improvement** Testing assigns confidence to software due to its quality, as referred in the previous section. Hence, competent testing means quality improvement, although it has numerous challenges [18]. In particular, complete testing is nearly impossible because it would imply testing every possible input for each variable and every path the user might execute.

   Code coverage is a metric used to provide an indication of the quality of the test cases. There are many criteria to calculate code coverage, but the most commonly used ones are branch coverage, condition coverage, statement coverage, edge coverage and function coverage [24]. Code coverage is a way to measure the completeness of testing, despite being challenging to achieve complete coverage [18].

   Consequently, generating test suits that return a considerate percentage of code coverage, even if it is an automatic generation, is a highly demanding task.

   Furthermore, even in the cases that the software services already exist, they required maintenance, and they do not have a collection of test suits to execute and prevent loss of software quality in case this suffers changes.

4. **Validation** The generation of the test cases, their extension due to mutation and the possible improvement of tests quality, can only be reliable if the data used for the generation is collected from real users, using real websites.

1.4 Goals

The problem statement lead to the following objectives that are intended to be achieved in this research work.

Firstly, one of the goals is to automatically generate test cases from usage data of software services. Accordingly, this work aims to use the information on the use of software services, which is similar to that collected by web analytics tools, and from there, generate test cases.

The following goal is to retrieve the most frequent paths taken by the users using these services. After detecting the user's most frequent paths, the intention is to generate test cases through them. After, the intention is to extend those test cases by injecting mutants that match the web testing context.

Finally, the ultimate goal is to perform experiences on selected subjects, resorting to the execution of the test cases generated and mutated. The purpose is to determine if the test cases' quality is improved, detecting if they reach different parts of the code that were not reached before. Hence, throughout the experiences, the aim is to resort to real users using the selected subjects to retrieve real usage data. Consequently, it will be possible to make assumptions based on the results of the experiences regarding the test cases' quality and the mutation operators that extended them.

1.5 Document Structure

The Introduction 1 is the first chapter of this dissertation, followed by other five subsequent chapters.

The second chapter, the state of the art 2 describes and reviews topics related to the advances and the definition of mutation testing, software quality, mutation-based test generation and code coverage.

Chapter three 3 regards the previous work that led to this research, consisting of two crucial tools that made part of this work. The tools are presented and followed by an explanation regarding their purpose in this research.

In chapter four 4 the experimental design is presented: the process of the experiments, its subjects and the technologies used on them.

Chapter five 5 presents the experiments themselves, focusing on the results and on the analysis of those values.

Lastly, chapter six 6 reveals the conclusions of the dissertation, including a summary of the document, a reinforcement of the limitations during the work, and prospects for future research.

Chapter 2

State of the Art

Software testing has become one of the most reliable techniques to guarantee software quality. Regarding the measurement of test suits' effectiveness to detect faults in the systems, mutation testing analysis has proven to be one of the best techniques to assess the quality of test cases [16]. Hence, the first section regards mutation testing itself and the developments in that subject 2.1.

Mutation testing consists in inserting faults into the code, called mutations, producing mutants of the original code, and then judges the effectiveness of the test suit by its ability to detect those faults. Therefore, the first section of this chapter has a subsection which presents the resemblance of mutants to real mistakes 2.1.1, considering that the test suite ability to detect mutants is associated with the effectiveness in detecting real faults. This effectiveness of the test suits will lead to the software quality, so the second subsection presents the literature existent, which associates this technique with software quality itself 2.1.2.

Furthermore, the generation of test suits from mutations could reduce the time dispended by the developers to insert mutants manually, in order to reduce the cost of the attention demanded to analyse and evaluate the results. Hence the next section 2.2 intends to expose the recent work developed in the mutation-based test generation. Moreover, code coverage has a crucial role in this research due to the ability to determine how the tests cover the code being tested. Since it helps analyze and successfully validate a system under test, it is crucial to highlight its concepts, intends, and recent literature regarding mutation testing 2.3.

Lastly, the fourth section brings awarenesses to the remaining problems of this field of research and what still needs improvement 2.4.

2.1 Mutation Testing

Mutations testing can be defined as a fault-based technique that intends to slightly modify the program source code through the injections of mutations. Accordingly, mutations emerge from operators called mutations operators, which aim to mimic software's typical errors. These operators are rules that, applied to software, create mutants [17, 15].

To measure the quality of the generated test cases it is calculated a mutation score, obtained by dividing the number of mutants killed by the total number of non-equivalent mutants [25]. To better understand this concept, it is relevant to understand that a mutant is considered equivalent if there is no test case able to distinguish the mutated test cases' output from the original program's output. Therefore, a mutant is considered killed if the test cases' execution on that mutant returns a different value than the test cases executed on the original program [17, 26].

The mutation testing technique has gained relevance among other software testing techniques due to its effectiveness in developing test suites and improvement in detecting code weaknesses, even those less visible [16, 27].

Since the appearing of mutation testing in 1971 proposed by a student called Richard Lipton [28] and its first development by DeMillo, Lipton and Sayward [29], mutation testing has been continuously researched and applied due to its numerous advantages. As referred, one of the main advantages is the significant improvement of software quality, since it allows to detect faults in the source code, achieving a high level of code coverage which will be explained further on this chapter [25].

Since the beginning until nowadays, mutation testing has been continuously researched and explored by numerous researchers to improve its remaining hurdles.

Among the recent literature, approaches that intend to identify equivalent mutants were conducted. Hence, equivalent mutants are syntactically distinct yet functionally indistinguishable from the original program - meaning that it is impracticable to develop test data capable of making equivalent mutants behave differently from the original program under test. Several studies appeared recently regarding this matter. Namely, a study was conducted recently by Durelli, and Brito focused on how machine learning algorithms that generate predictive models can classify mutants as belonging to the minimal set or equivalent [30]. Additionally, Baer and Oster have created an approach to resolve this problem focused on the use of symbolic execution to diagnose identical mutants, remove them from further research, and produce test cases that eliminate the remaining mutants automatically [31]. In the same matter, a study using Predictive Mutation Testing (PTM) was recently developed by Torres and Ramle to predict the equivalent and killed mutants. This study used a classification model based on the mutated code's features and the test suit intended to predict the execution results of a mutant without actually executing it [32].

Besides the focus on discovering equivalent mutants, other recent studies focus on using mutation testing to reveal failures using the injection of flaws with the creation of new mutation operators, such as the study of Chekam and Sen. The authors discussed the issue of choosing mutants that expose the faults, namely the mutants that are more likely to kill and lead to test cases which reveal unknown mistakes on the software [33]. The study of Neto is also important to refer to in this matter since it focused on the failure information in a system under test to improve effectiveness. This study used the exercising of the test of similar parts of a system where mutants are likely to give the same results to measure and compare test suits, prioritizing them [34].

Another focus of research regards graphical user interfaces, where it is important to notice the work of Barbosa, Paiva and Campos. The authors describe an approach using model-based testing of graphical user interfaces from task models, where they start from a task model of a system under test, generating oracles to compare its behaviour to the running system's execution. Hence, the approach shows how task mutations can be generated automatically, allowing a broader range of user behaviours to be taken into consideration. This way, they provided a tool to classify users errors and to generate mutations [26].

Additionally, other work recently developed in the context of graphical user interfaces is the research of Oliveira, Freitas and Paiva. Their work presented a framework to generate software test cases automatically through user interaction data. Hence, it focused on an approach based on a data-driven software test generation, combining Markov chain modelling and the mining of use of frequent sequences. They also measured the events' distribution plausibility in the test suits generated using the Kullback-Leibler divergence [35].

Furthermore, late approaches using mutation testing appear in innovative areas that still do not dispose of tools and literature regarding testing. It concretely works for tools regarding *Android* mutation testing [36].

Regarding *Android* applications, the lack of testing methods, testing experts and time restrictions increased and caused concern, as mentioned. Saifan recently developed an approach initially beginning with defining a set of mutants operators according to the Android application features. Afterwards, an automatic generation of mutants from the operators retrieves them to determine the test cases' efficiency and allow new operators [37]. Additionally, Moran developed another new framework that uses mutation testing in android applications, which is *MDroid+* [38]. The framework aims to contribute to measuring and ensuring mobile testing practices' effectiveness, consisting on almost forty mutations operators, derived from android faults, and generating more than eight thousand mutants for more than fifty applications. Another work important to highlight in this matter is the work of Paiva and Gouveia. The authors present new mutation operators that aim to insert faults related to the non-preservation of users transient UI state when users send mobile applications to the background and then get them back to the foreground [35].

Recent literature also presents new work using mutation testing in network protocols. Its significance increased since mutation techniques can test whether the network protocol is robust enough to verify test cases that follow the protocol specifications and refute the test cases that do not show conformance. Among the recent studies, Zarrad developed a method to find faults that could not be discovered by classical network protocols' testing techniques [39]. Furthermore, another research focusing on real faults is from Neto. The study proposes a mutation testing approach for big data processing programs that support a data flow model. Initially, the method consists of gathering a set of mutation operators from programs characterised by the data flow, aiming to mimic real faults. Finally, they assess the proposed mutation operators and compare to the real faults measuring the difference in cost and effectiveness [40].

Furthermore, this testing technique has been applied to various programming languages as a fault-based, white-box testing technique, requiring a new set of studies for assessing the effectiveness of mutation testing in this context. Namely, languages such as *Java* [41, 42, 43, 44, 15], also *C* [45, 46, 47], *C#* [48, 49, 50], and *SQL* [51, 52, 53, 11].

The recent literature regarding mutation testing also showed improvements on matters such as Web Services, since their recent development faces new issues concerned with testing to ensure the delivery of software quality [54]. Mutation testing appears as a way to measure the adequacy of tests or to reveal errors. Hence, a lot of studies recently focused on using this testing technique to help ensure the software's quality when Web services need to integrate existing software applications or even work on different platforms, and to create new services [55, 14, 56, 57].

As it was perceptible during this first section, the topic of real faults in software is something crucial for mutation testing since an adequate test case is one that reveals real faults, leads to the exposing of the recent literature in the following section about the resemblance of mutants to real development faults.

2.1.1 The Resemblance between Mutations and Real Development Faults

The concept of a good test case revealing real mistakes is not useful for developers who build test suites, neither for researchers who create and analyze instruments that produce test suites, since the quantity of faults in software is usually unknowable [58].

Mutation testing is engaging in this matter since it is possible to randomly produce and use vast numbers of mutants to compensate for low concentrations or the lack of proven actual faults [58].

Since the mutation operators' purpose is to mimic the developers' typical mistakes, researchers have been trying to establish if the mutants resemble the real defects testing is trying to reveal.

Among the studies, Andrews and Briand [59] tried to parallel the ease of detecting manual and real faults, measured by calculating the facility of identifying faults through the percentage of killed mutations by test cases. The study concludes that the efficiency of mutants' discovery was, in fact, alike to real errors. However, their results rely on real faults from a single program, limiting the scope of inference. Additionally, Just and Jalali [60] studied the association between mutation score and test case effectiveness by applying more than 300 bugs from 5 different open source applications. Results showed that if the mutation score increases, test-suite effectiveness increases in 75% of the experiments; meanwhile, regarding code coverage, the raise was only 46% of the cases. Furthermore, conclusions lead to the assumption that the increase of coverage implies a raise in mutation coverage, although the opposite does not happen. Adding to the work recently developed motivated by the aspects that impact mutation score, Papadakis and Shin studied the correlation between mutant scores and real fault detection. Notably, the relationship between the two independent variables, mutation score and test suit size, detects real faults with one dependent variable. Moreover, their results suggest that mutants contribute to proper guidance toward improving test suites' fault detection, although their correlation with fault detection is poor [61].

Furthermore, Namin and Kakarla [62] explored the possibilities and precautions to take when using mutation testing as a proxy for real faults. The results showed the impact on the relationship between mutations and real faults, emerged from the programming language's choice, the type of mutants operators, and the test-suite size. Lastly, their research exposed a weak association between real flaws and mutations.

Luo's research also contributed to this matter, since it explored test case prioritization on mutants and real faults. The work aimed to achieve answers for this hurdle regarding the extension of test case prioritization performance on mutants being representative or not, of the performance attained on real faults. The method used was to conduct an analysis of the performance on test case prioritization applied to mutants and faults on real platforms [63].

Recent research has also focused on equivalent mutants' problematic, which appears due to the extension of mutants that the program could not correctly distinguish. Consequently, Marsit [64] studied the impact of these mutations operators in the generation of equivalent mutants. Additionally, work such as the research of Ayad tried to predict mutation equivalence based on the semantics used on the mutation operators [13].

Finally, as we come to understand reading this section, one of the main goals in the pursuit of analyzing, exploring and improving mutants is for them to reflect real development faults [60]. Consequently, by strengthening mutation operators, the mutation process is improved, leading to an improvement of software quality. Hence, in the next section, it is extensively explained this deep connection between the testing process of software and the quality attained from it, as well as the recent literature regarding this subject.

2.1.2 Mutation Testing and Software Quality

Throughout this dissertation, it is noticeable that testing and software quality are two concepts that appear synchronically. Hence, to fully comprehend all of these notions, it is convenient to define quality.

A product's quality is defined by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) as an "eighth characteristic composed model that relate to static properties of software and dynamic properties of the computer system" [65].

Specifically, these characteristics are **functional suitability** which is the degree to which, when used under defined circumstances, a device or system offers functions that satisfy stated and implied needs. **Performance efficiency** which illustrates the results compared to the number of assets used under specified conditions. **Compatibility** representing how a system can interact with other systems and perform its required functionalities while sharing the same environment. **Usability** represents the attained effectiveness, efficiency, and satisfaction used by costumers to reach their goals. **Reliability** meaning how a system can perform specific functionalities under a set of requirements for a period of time. **Security** which is the amount of information protected by a system while allowing users to access appropriate data related to their authorization levels. **Maintainability** representing how a system can be modified or improved while maintaining effectiveness and efficiency. Lastly, **portability** means transferring a system from one environment to another with effectiveness and efficiency [17].

Since mutation testing develops software tests, also serving as an indicator of the quality level of the tests, it is essential to recognize that one of its main intentions is to mimic typical programmers mistakes. Hence, it will be seeded intentionally, achieving the goal to build a set of mutants, each containing syntactic differences [7].

Therefore, to assess quality to test suites, the mutants are executed towards input test suites, which implies that the mutant is recognized if results are different for any test case.

Mutation score is an indicator of the test suit quality, being one consequence of mutation testing, it represents the ratio between the number of detected flaws over the total amount of manual input flaws. Hence, some recent research has been recognized due to the development of approaches that evaluate the impact of mutation operators on mutation score [64].

The work regarding the relationship between the improvement of quality of the software and mutation testing includes Trakhtenbrot [66]. This work intended mutations to assess the quality of tests for statecharts. Furthermore, other researchers investigate this matter, such as Fraser. The author presents a research about the minimization of the number of mutants resorting to special properties that help detecting equivalent mutants. The research intends to prevent creating test cases that lack quality, by avoiding the creation of redundant test-cases [67].

Additionally, some literature appeared regarding the search for improvement of the test data quality in mutation testing, as is the case of the work of Fleurey and Baudry [9]. They proposed an approach to improve the quality of test data using Mutation Testing with an algorithm regarding bacterias. Moreover, work to use mutation testing to improve the quality of the construction of unit test following parameters, was developed by Le Traon [68] and Xie [69].

Pérez also contributed to the seek for quality improvement, trying to reduce the costs of mutation simultaneously. His work emerged from recent advances regarding these concerns and proposed to apply them for addressing current drawbacks in Evolutionary Mutation Testing [70]. Addressing the same hurdles mentioned, Botaro's approach introduces a quality metric, extended to mutation operators, to reduce the number of mutants, particularly the equivalent ones [10]. Concerning this matter but regarding models, the recent literature presents the work of Vecarmmen, which intends not only to increase the quality of test cases for each method, instead of verifying each test suit quality. The results of his research showed a speed-up of 574 times for the mutants with a quality score of 80% [71].

Lastly, one interesting approach that regards two crucial matters to this work is quality evaluation and improvement, but also the injection of faults from mutation operators, is from Ma and Zhang. Their proposal presents a mutation testing framework specialized for deep learning systems to measure the test data quality. The method intends to inject faults to the system's source and then evaluate the data set's quality by analyzing what injected faults were detected after the process [72].

Concluding, the recent work has developed towards providing solutions to assess the quality of the test suites. Due to this matter's relevance, it is expected that this subject is continually improving and seeking mutants with higher quality to also generate higher quality to the test suites.

2.2 Mutation-Based Test Generation

The previous section described and explained the meaning and recent researchers of the importance of software quality in systems through the use of mutation testing. Nevertheless, mutation testing is not all about assessing quality to a test suit; it also supports testing activities such as test generation. Hence, test data generation has the intention to generate test suites with enough effectiveness to kill mutations successfully.

Many researchers struggled with the hurdle of methods intended to generate data automatically, not knowing the program's structure. Consequently, this data is not well assured to deal with bugs when faced with changes and improvements to the program. Recently, the work of Wang aimed to reform requirements by a genetic algorithm to generate inputs capable of killing as many mutants as possible of the target program. Their results showed a contribution of software maintenance with an efficient generation of test that could kill mutants in the program.

Naturally, recent work has been developed to explore the automatic generation of test suits from mutants. Baudry's [73] research showed results that demonstrated that 75% of mutants could be killed using his test generation technique, which consisted of an approach that considered the generated tests as predators used to verify the program.

In mutation testing, by reaching a higher level of mutation score, it is possible to reduce the manual effort involved due to test data generation. Therefore, some studies appeared trying to propose automated test generation approaches that could lead to efficient results. Among them, there is the research of Souza and Papadakis [74], which presents an approach applying the hill-climbing algorithm to mutation. The aim is to reach a strong mutation by forcefully and incrementally killing mutants, which is accomplished by focusing on mutants' propagation. Namely, the problem is how to kill mutants that are easily but not strongly killed. Furthermore, regarding the recent work that aims to reduce costs through test data generation, Qin proposed an approach implying a new mutant generation algorithm [75]. The paper is based on a primary path coverage that can help reduce mutants, and therefore, reduce manual effort as well as costs.

Rani and Dhawan recently researched the minimization of the cost incited by mutation testing, using a selective mutation technique to generate a lesser number of mutants by applying delete mutation operators instead of the commonly used mutation operators [76].

Additionally, some approaches also appeared with the intent of generating test cases from user execution traces, to battle the absence of models, such as the strategy of Paiva, Restivo and Almeida who developed a web testing approach based on user execution traces that generate test cases [25].

Furthermore, research such as Mishra's work becomes relevant to highlight, focused on coverage-based test data that is generated and further exercised to cover all mutants present in the specific program under test [77]. Deepti and Mishra [78] also developed an approach focused on using real encoding for automatic test data generation, accompanying a typical test suite to achieve 100% of path coverage, as an optimal result.

High-quality test generation has always played a key role in mutation testing, although nowadays researchers gave more attention to this matter due to its direct relation to improving the software's quality. Consequently, different techniques arise for test-case generation, being evolutionary algorithms part of the most prevalent ones. Atani presented a paper where the main goal was to comprehend and analyze, for each evolutionary algorithm, which fitness function generates better test cases, since each algorithm needs an appropriate fitness function [79]. Moreover, regarding the work recently developed that intended to increase software quality, it is worth mentioning Almeida's research. Almeida's main objective was to collect information from services of logs and web usage data to generate test cases and extend them with mutations [80].

Lastly, it is perceptible that the quality associated with the test cases is of tremendous importance. Consequently, determining a method to find faults in a system is crucial for these techniques. The next section will explain and expose the literature regarding code coverage and its role in testing, particularly in mutation testing.

2.3 Code Coverage

At this point, it is well known that software testing is considered to be indispensable in software maintenance. Consequently, it is also crucial for testing to obtain code coverage: "Coverage is the extent that a structure has been exercised as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed and therefore, increase coverage" [1].

The levels of code coverage can help control the tests' quality and manage the generation of tests to create test cases that cover areas that have not been tested before [81]. Hence, measuring code coverage can provide information about the verification process, finding gaps in the test suits, namely parts of unshielded code.

Code coverage is not an unknown concept in the computer science community; their test suit adequacy measurements are continuously applied, specifically statement coverage, branch coverage, modified condition, and decision coverage.

The practice of code coverage has been acknowledged for decades now. Namely, Piwowarski stated that IBM performed code coverage measurements already in the '60s [82].

Companies such as Google spent more than a decade improving and developing their code coverage methodology. Ivanković wrote a paper specifying a study aiming to comprehend how developers perceive code coverage. Analyzing adoption and error rates, as well as code coverage ratios, the result is a paper proving concrete ways to implement and use code coverage in a business environment [21].

Recently, the pursuit of assessing effectiveness for test suits based on fault-revelling criteria also lead researches such as the work of Aghmohammadi to correlate such criteria with mutation score. Specifically, his work proposed a new code coverage criteria that performed different code coverage types associated with mutation score [83]. In the chase of improving mutants' efficiency based on their information, recent literature also uses code coverage. In Panichella and Zhu's work, the purpose was to test the reachability concerning mutants' impact in code, focusing their approach on mutation compression strategies [84]. Regarding this matter, Panichella and Zhu also explored the scenario where a developer faces low mutation score when testing the code. They state that code coverage is the most suitable technique to measure the test suit's effectiveness and provide developers with a perspective to understand mutation scores considering testability and observability. Therefore, they analyze the relation of observability and mutation score, measuring the extended code with code coverage [85].

Nowadays, code coverage has become a very prominent topic regarding web applications as well. Despite the numerous tools and approaches to test web applications, researchers such as Mallika and Lakshmi believe in a margin for improvement relying not only on code coverage by on the potential of mutation testing. Their investigation tried to analyze different mutation operators to expose web applications' vulnerabilities, verifying and improving their code coverage [86].

Finally, in the process of testing software, code coverage presents itself as a crucial indicator of quality and an indispensable software maintenance component [20]. Code coverage facilitates the evaluation of testing software's effectiveness due to the data provided of the different coverage in its components, proving to be an essential matter to this paper.

2.4 Current Issues

Despite all research and brand new methodologies developed towards the improvement of software using mutation testing, either related to mutants, software quality, code coverage or the automated process of mutation testing itself, there are still various concerns that demand solutions to be defeated. Consequently, it is essential to take a closer look at the mutation process in order to comprehend the remaining issues.

Briefly explaining, everything starts with a code component, which changed on a simple way, will originate the mutants. The original test cases are run with each mutant, creating in some cases live mutants which are undistinguished from the original program, but if they are distinguished, then they are killed. The hurdle is, if the mutants remain alive, meaning that they pass all test cases, then they are equivalent mutants and will be ignored, or in opposition, they are not equivalent. This concern demands the creation of more test cases in order to kill such mutants. Finally, this flow of killed mutants provides an overview of the rate of undetected flaws that remain in the original code [17, 25].

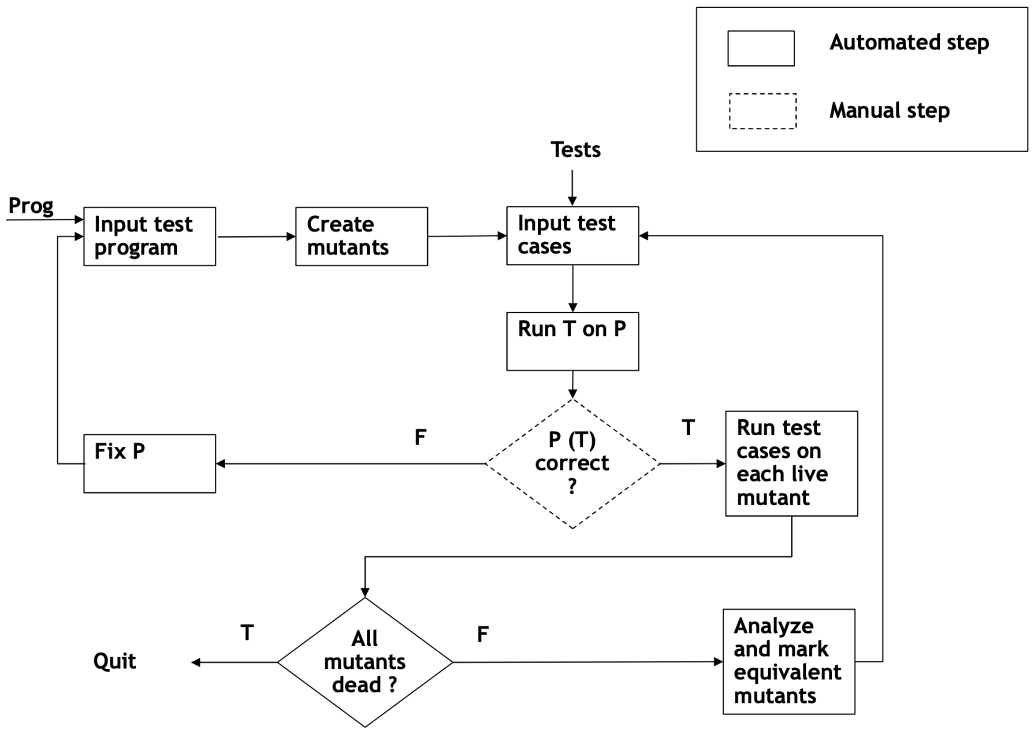


Figure 2.1: Mutation Testing from: [87]

Naturally, the first concern addressed, as well as the most investigated, is the **computational cost**. The amount of generated mutants is proportional to the data, which implies that even in a small software component, the number of mutants tends to be enormous [17]. The approaches developed to overcome this problem focus on at least one of three strategies: do fewer, do smarter, do faster [88, 89]. The first one explores methods to create fewer mutants without the loss of essential data. The second intends to distribute the computational cost over different machines, or factor the cost over various executions by preserving state information between runs, or attempt to avoid complete execution. The third one aims to generate and run each mutant program as quickly as possible. Although some of the most recent work involving the reduction of computational cost has been addressed previously in this section, it is important to refer some of the latest approaches [90, 91, 92, 93, 94, 95, 96].

Among the research which intended to reduce the computational cost, it is mentioned frequently the second biggest concern regarding mutation testing, which is **equivalent mutants**. It is one of the most significant obstacles to overcome due to the amount of effort needed to check if they are equivalent or not. Hence, the cost of mutation comes mainly from mutant generation, execution and most importantly, the assessment of equivalent mutants [67, 92]. Due to the effort needed not only from the developers to handle equivalent mutants, but also the increase of cost, many investigators focused on solutions and improvements for this matter [37, 92, 97, 94, 96, 13, 64].

Lastly, the third most prominent concern is human **oracle**. The oracle consists of examining the original program's output with each test case executed. Even though this hurdle is transversal to all types of testing, the downside of this problem increases regarding mutation testing [98]. Due to mutation testing being a demanding technique, it can lead to an increment of the test cases amount, ultimately resulting in an increase of the oracle cost. Naturally, recent work has been developed towards the improvement of this problem [2, 90, 99, 12, 100, 101, 102].

Concluding, the work intended to accomplish in this research has the aim to improve the quality of the test cases, using mutation testing, and will be an improvement of what researchers tried to resolve regarding some of the remaining problems. The approaches previously developed, despite presenting software services that require maintenance, do not yet present a battery of test suits that could be used when the software is changed, in a way that the software quality does not decrease, leading to the constant demand for effort and resources. In particular, this catalogue of test cases is meant to be based on real information from real software services. Therefore, the code coverage will be improved because of the expansion of the test cases using mutation, and the tests will attain more quality with less effort.

2.5 Summary

Mutation testing is a testing technique that gained relevance among other approaches, due to its effectiveness in developing test suites and improvement while detecting code weaknesses, even in parts of the software more shielded and hard to reach.

It is fundamental to notice all the work developed throughout the years that intended to clarify the results of this technique, as well as the benefits that bring to the testing process, and finally improve the weaknesses regarding mutation testing.

From mutation testing itself to the resemblance between mutants and real development faults, followed by the urge to automatically generate mutants, leading to the impact on software quality, code coverage, this paper gives a notion of the current state the art of all these significant subjects. However, it is essential to notice the remaining hurdles of such concepts and the need to find solutions for them, improve the testing process successfully, and improve the test cases' quality.

Mutation testing remains as a technique with numerous advantages that lead papers like this one, to continuously analyze and develop methodologies that bring benefits to the testing community.

Chapter 3

Previous Work

This chapter exposes the previous work that allowed this research to have the necessary tools for the experiments and analysis required. In the first place, we present the tools, then a clarification of the motivation to use this particular work, followed by a clear explanation of the work itself.

Furthermore, in the description of each tool is presented the contribution to this dissertation, followed by the tool's limitations. Finally, the incorporation of the tools to form a single methodology, the improvements made and the surpassed hurdles are presented.

3.1 MARTT

One of this thesis' purpose is to obtain test cases generated by using information concerning the utilization of software services.

To achieve this goal, a previously developed tool called *MARTT* is used for the retrieving and storage of this information.

MARTT introduces a solution to produce test cases from actual data from the user's interaction with a web application.

Firstly, the usage data is retrieved from the website using a *JavaScript* file. Then, this file makes API requests to store the information on a graph database, *Neo4j*.

The second stage is dedicated to the analysis of the data previously stored, according to the tool filters. These filters allow the user to select the execution traces by specific parameters:

- **Action type**: The user's interactions can be saved as click, double click, input or drag and drop. Hence, the user can filter the execution traces by the type of action or by the number of actions.

- **Element**: Providing a specific *XPATH* element, to test a specific component, the user can filter the traces by element.

- **Most Common**: Through the application of the *Levenshtein* algorithm, the tool can calculate the most common execution traces. The algorithm was implemented to perform as a metric to measure the difference between two arrays of integers, although the algorithm's initial purpose was for strings. The outcome of the algorithm is the smallest number of modifications to turn one array into another. This difference is called the *Levenshtein* distance, and its value is divided by the length of the longest sequence in order to mitigate the length effect on the value, getting a value between 0 to 1, meaning the percentage of how alike the sequences are. Therefore, two arrays with distance 1 mean they are entirely equal, while two arrays with distance 0 are entirely distinct. The algorithm compares all sessions getting the distances among all of them, and after all comparisons between each pair of sequences, the most common sequences are the ones with the smaller average of the distance between themselves and the remaining ones.

- **Length**: The user can also filter execution traces through their length. Consequently, it is possible to find the biggest and smallest sequence on data based on the number of interactions.

- **URL**: Execution traces can also be filtered by a particular URL since the URL of the user's interactions is also stored.

Finally, after applying the filters, it is possible to download the desired execution traces in *JSON* format that contains the data about the session, matching the selected criteria. This *JSON* file will later be used to generate test scripts.

Below it is possible to visualise the components diagram of the tool in order to understand the solution properly:

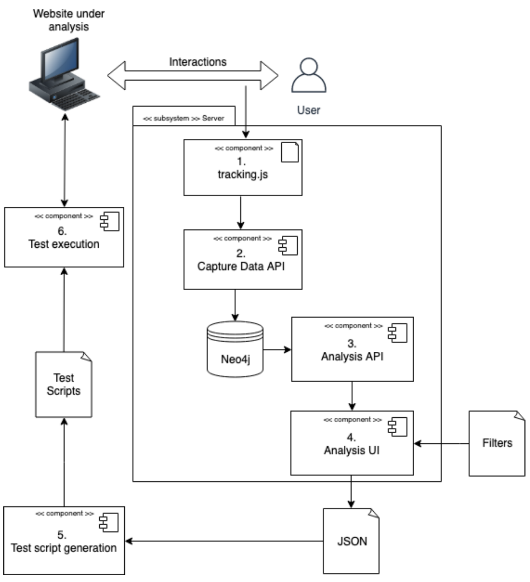

Website under analysis
Interactions
User
<< subsystem >> Server
<< component >> 1. tracking.js
<< component >> 2. Capture Data API
Neo4j
<< component >> 3. Analysis API
<< component >> 4. Analysis UI
Filters
JSON
<< component >> 5. Test script generation
Test Scripts
<< component >> 6. Test execution

Figure 3.1: Components diagram of *MARTT* from: [103]

Trough the use of *MARTT*, it is possible to collect real usage data, store it, and filter it in the most appropriate way. Therefore, it will be convenient to filter the most common execution traces to later generate abstract test cases to mutate and use on the experiments since each *JSON* file corresponds to an abstract test case.

3.1.1 Limitations

MARTT was initially installed and was working on a server located at the Faculty of Engineering in the University of Porto that has now been disabled. Hence, even though the tool was supposedly prepared for a local installation, the setup and installation were difficult to achieve due to the numerous updates needed for everything to work correctly.

The updates before-mentioned happened due to the tool being ready to access a specific website used in *MARTT*'s development. Also, the management of retrieved data and storage was working with the server, and changing everything for a local environment was not as trivial as thought in the beginning.

Furthermore, the tool was tested and developed, resorting to a particular online website, while the purpose of the experiments in this research was to use software services not necessarily similar to the one selected in its initial development. Consequently, it was necessary to adapt the tool to use any software chosen for the experiments.

Eventually, *MARTT* demanded some allocated time not only to allow familiarization with its logic but also to update outdated nomenclature used on the development of the tool.

Finally, these limitations were all successfully surpassed, although they demanded extra invested time that was not accounted for in the beginning.

3.2 Mutations

The tool *Mutations* was used to continue the process initiated by the previous tool, to reach a complete procedure for the upcoming experiments.

After obtaining the test cases in *JSON* format, which are referred previously as abstract test cases, generated by the *MARTT* tool, the next step was to use another tool that allowed the generation of concrete test cases and mutate them to perform the experiments. These concrete test cases are abstract test cases converted through the process of adding input data and changing the structure of the abstract test cases. Hence, the motivation to choose this previously developed tool was to, through the combination with *MARTT*, obtain a complete procedure that started on collecting real usage data and ended in the generation of a mutation of a test suit. *Mutations* is a tool with the ability to retrieve the abstract test cases generated by *MARTT* stored in the *Neo4j* database. The purpose of the tool is to convert these test cases into concrete test cases. Therefore, to make this conversion, the abstract test cases suffer injections of input data and are later executed, allowing the comparison with the subsequently generated test cases.

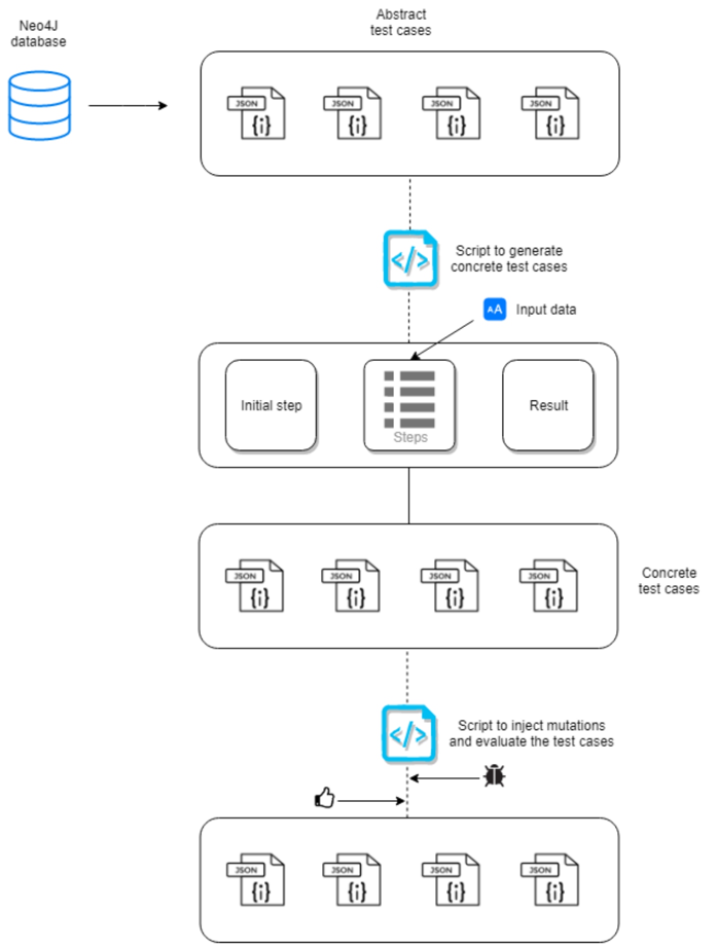

Neo4J database
Abstract test cases
Script to generate concrete test cases
Input data
Initial step
Steps
Result
Concrete test cases
Script to inject mutations and evaluate the test cases

Figure 3.2: Diagram Implementation of *Mutations* from: [104]

The conversion before-mentioned is possible through the use of the information held onto the nodes in the database. Each part of the information in the nodes is used to generate test cases using the concept of steps to differentiate the data. The nodes' information is passed to the test cases and contains it a path, a session, an action, an element position, a value, a URL and a boolean value mutated to verify if the test case was already mutated or not.

The tool then allows the user to resort to mutation operators from an established catalogue that inject defects, resulting in several varieties of test cases. The mutations operators available include:

- Add Step Mutation Operator: This operator intends to add new input actions in different points of the sequence of user actions recorded.
- Add Back Mutation Operator: This operator checks if the buttons are still working after changing to the preceding page on the website.
- Change Input Data Mutation Operator: This operator's goal is to verify the system's behaviour with different input data.

- Change Order Mutation Operator: The purpose of this operator is to swap two consecutive user actions to verify if they are dependent.
- Remove Step Mutation Operator: This operator skips actions to check if they are mandatory or not to check the program's behaviour when those actions are skipped.

Additionally, the tool development resorted to the programming language Java and the online testing framework Selenium.

Using *Mutations*, it is possible to convert the abstract test cases to concrete test cases. Furthermore, it is possible to inject mutations into the generated test cases to create new test suits that will be executed into the system, allowing it to reach different parts of the same code.

3.2.1 Limitations

Mutations demonstrated significant limitations originated from its establishment and utilization on this procedure that demanded to be surpassed.

The first limitation appeared in the installation stage when some drivers and packages used on the development of the tool were outdated and automatically updated when the tool was installed. Additionally, there were also hurdles regarding outdated nomenclature on the code of the tool that demanded an actualization according to the used terms in the corresponding languages.

The following interference for the correct function of the tool was regarding automatization. For this research experimental process, the tool needed to be fully automated to be possible to execute several test cases without constant manual effort. Hence, this obstacle demanded various code changes, requiring a first familiarization with the code previously developed, to finally allow the process to run automatically.

Additionally, the tool was designed with several interface menus to select options regarding the browser, the mutation operator and the number of mutants desired per test. This was a problem because it involved constantly choosing options, making the process too manual. Consequently, to tackle this limitation and get a direct output without having to select options in menus repeatedly, we restructured the tool to use a specific browser, a number of mutations and internal selection of the mutation operators allowing the test cases to be the only manual input.

The limitations mentioned where successfully surpassed, although requiring additional time that was not initially accounted for in the research.

The final hurdle that was faced was that it was expected the tool to inject different types of mutations into one single test cases, this way generating test cases that were constituted by multiple mutations. However, the tool was only designed to inject one type of mutations into a test, even though it was able to inject a number of mutations of that selected type by choice. Consequently, this limitation had repercussions on the way the experiments were conducted, since at the beginning of the research work, it was intended to use different combinations of mutation operators as well. Hence, it was not possible to modify this issue to match this research's initial intention due to time constraints, resulting in different choices regarding the experiments, as it is explained in the experimental design in the following chapter 5.

Finally, as already referred, due to these limitations, which demanded time invested on surpassing them and even choosing different paths to conduct the experiment, the time initially allocated for extending the mutations catalogue was invested on improving and adapting the structural part of the tool.

Chapter 4

Experimental Design

The purpose of the experiments is to use test cases generated from execution traces gathered from real usage data, followed by the extension of these test cases resorting to injection of mutants, to ultimately analyse the tests' quality and increase code coverage.

This chapter exposes the process of these experiments, as well as the subjects of the experiments, followed by the technologies used to analyse the outcomes, and finally, the way the validation of the experiments occurred.

4.1 Process of the Experiments

The first stage of this experiment was to select the subjects for this experiment, leading to the decision to gather a set of websites with different purposes for a more diverse number of functionalities. The subjects of the experiment, in particular the websites, were used for gathering data of users utilizing them and performing actions. In the subsequent section, there is a detailed explanation of each website for better perception.

Since the data for the experiments is intended to be as realistic as possible, where real users explore several web services' functionalities, we gathered a group of participants for this experiment. It was asked from the participants to fill in a form where they inserted their academic degree, age, gender, and if they were students or not to perceive the different variations of people who participate. The gathered data of this form is in the Appendix A. This information was useful to comprehend if the participants had identical characteristics, since it is convenient for them to be as diversified as possible to represent a more realistic group of users. The data show that the participants included not only students in college, but also in high school, as well as working people of different ages and genders.

Furthermore, the next stage was to ask this group of participants to use the websites for several days, knowing only their potential functionalities, to allow them to explore the website freely and without specific tasks. The information provided to the participants was a list of possible interactions with the websites as authenticated and non-authenticated users, i.e. edit profile, add items, remove items, check other users profiles, among many other possible actions. The goal was for them to make different choices, options, and decisions, to obtain various types of paths, in order to truly recreate what usually happens on a web service with several users every day. Ultimately, the purpose is to verify the users' most common paths to generate the test cases.

The following step was to retrieve the execution traces of the user's interactions, caught by a running script from *MARTT*, which stored the data in a *Neo4j* database, as explained in the previous chapter 3.2. More specifically, as the users where exploring and exercising the website's functionalities, a path of their actions was created and stored in the database. This resulted in the execution traces, meaning a set of nodes representing the users' interactions, all connected through relations reflecting and differentiated by sessions.

Since the purpose of the experiments is to conclude if there was improvement of tests quality through the mutation of test cases generated by the most common user interactions in a software service, *MARTT* was used again to filter the most common execution traces. Once these traces were obtained, they were downloaded in a *JSON* format as abstract test cases.

Next, the abstract test cases were uploaded to the tool *Mutations*, where it was possible to generate concrete test cases from them 3.2. These test cases were executed automatically on the website they corresponded to, resorting to *Selenium*, which allows the execution of test cases in the browser, to ultimately to verify their code coverage. Throughout the test cases' execution, we documented the levels of code coverage attained once the execution was finished. Specifically, the number of lines covered proportionally to the total number of lines in the code. The purpose of documenting was to keep track of each percentage of code coverage attained on each iteration. Ultimately, to evaluate the variations of code coverage levels attained by the tests.

The following stage was to apply mutation operators to the initial test cases and extend them. In each iteration, we used one mutation operator at a time, and extended the test cases with it, and then reproduced mutated test cases in the website. This process resulted on 60 different test cases per mutation operator and it was repeated for all of them : Add Step Mutation Operator, Add Back Mutation Operator, Change Input Data Operator, Change Order Mutation Operator and Remove Step Mutation Operator. Afterwards, as a final iteration, we generated another set of mutated test cases for each operator, resulting on a total of 300 extended test cases, and reproduce them on the website, to ultimately try to obtain reports of code coverage that had all types of mutation operators.

The reproduction of the test cases in the websites, after each iteration was possible resorting to Selenium, to gather the information about the code coverage attained. At the end of each iteration, the results of code coverage were documented regarding the execution of each set of mutated test cases.

Lastly, the final stage was to analyze the different values from each iteration and comprehend the difference between the code coverage of the initial test cases and the extended test cases generated through mutations. Hence, this analysis is fully detailed in the succeeding chapter about the results of the experiments 5.

4.2 Subjects of the Experiment

As explained in the previous sections, to gather the most frequent paths from users and to generate the test cases, it was required to select some websites to serve as subjects of the experiment.

These websites served as the experiment's subjects to analyze and verify if there was an increase of lines covered attained from the execution of the mutated test cases, thus achieving a higher level of quality for the test cases.

Since the purpose of our research is to verify that the code coverage improvement aligns with the generation of test suits based on mutations, it is essential to use a website suited for the analysis. Therefore, the websites must be complex enough to do all sorts of interactions, although not complex enough that the code coverage analysis on the server-side is too demanding to visualize.

Therefore, we selected four different previously developed websites by students in a module of Web Technologies. These websites had the advantage of their code being available, allowing us to tailor them to be used in the experiments. This is important since *MARTT* needs to run a script on the server-side and on the client-side to retrieve information, so we needed to be able to access the code to run the scripts properly.

Accordingly, the selected websites for the experiment were the following:

- **Host**: A website for renting properties, allowing the users not only to rent houses but also to place properties to rent. The users could also check the availability of the properties, profiles of other users, including their respective houses.

- **GET**: A channel that allowed users to initiate discussion topics, where other users could comment and give their personal thoughts about the discussed matter. The users can also comment on each other discussions and up-vote or down-vote other user's comments.

- **Eatr**: A platform to check available restaurants, their ratings from other costumers and the associated reviews, and obtain information about the establishment such as contacts, photos and location. The user could also add new establishments and view the profile of other users and their restaurants if they had any.

- **MANA'O**: A social media with a meditation area and also a space to share thoughts with other users. Each user can reply to these comments, like or unlike them, explore the remaining users' profiles and view their comments.

To complement this information, samples of images of these websites are available in the Appendix B.

It is necessary to highlight that the websites were launched online only to the audience that participated in the experiments, not being reachable to the public, so there is no link to access each website.

4.3 Technologies Used

In the first part of this process, it was used *Neo4j* [1] : a graph database that allows to query data relationships in real-time using the *Neo4j* query language denominated by *Chyper*. This database provides a visual browser that permits the visualization of the sessions and connections in interactions due to every record being saved as a node [105].

The script used to retrieve the user interactions data was part of development of the tool beforementioned *MARTT* 3.1. This script was created with the programming language *JavaScript* [2] to capture the user's interactions with the web page. This script also aims to control and change actions in the web browser.

The following process, already described in the previous section, resorted to the mentioned tool *Mutations* 3.2, developed in *Java* [3], and we had to alter the code, it is also considered a technology. The process of generating the concrete test cases and the mutated test cases was possible due to the assistance of *Selenium* [4] . This test automation tool permits the creation of scripts used to reproduce the test cases in the browser automatically, also being able to identify selected objects and their actions due to its *WebDriver* component.

4.3.1 Code Coverage

Regarding the technologies used in the research work, it was necessary to find a solution to detect and analyze the code coverage behaviour during the experiments.

In this research, it is crucial to visualize the test cases' different behaviour once they were mutated. Due to this, the tool selected should allow us to see clearly which lines of the code the tests were able to reach and which lines were not.

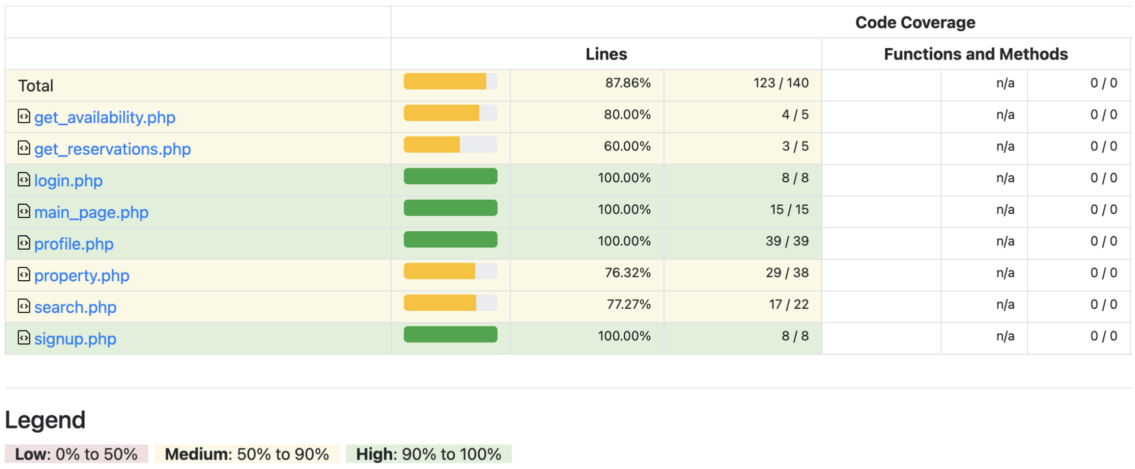
Consequently, we considered various tools; however, they were all suited for one particular programming language. Considering this information, the decision of the tool also involved the websites we had as possibilities for the experiment, and most of them were developed using several programming languages, or in other cases, *PHP* mainly.

The tool selected is *PHP Unit Testing* framework which supports *Xdebug*, *PCOV*, and *phpdbg* for code coverage reports, the library developed by Sebastian Bergmann named *php-code-coverage* [106] is what allows this functionality. The reason why this framework is so great for this research work is due to the use of *Xdebug* [5], which is an actively developed extension of *PHP*. Thus, it is possible to obtain detailed reports of which code lines were reached by tests, which ones were not, and even dead code.

Another motivation for using this solution was that it enabled the use of automation testing frameworks such as *Selenium* – which is one of the parts of the tool Mutations used in the experiments 3.2.

[1] https://neo4j.com/
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[3] https://www.java.com/
[4] https://www.selenium.dev/documentation/en/
[5] https://xdebug.org/

The following image represents a sample of how a report generated by this tool, regarding the code coverage of pages in a website, appears to be:



	Code Coverage					
	Lines			Functions and Methods		
Total		87.86%	123 / 140		n/a	0 / 0
get_availability.php		80.00%	4 / 5		n/a	0 / 0
get_reservations.php		60.00%	3 / 5		n/a	0 / 0
login.php		100.00%	8 / 8		n/a	0 / 0
main_page.php		100.00%	15 / 15		n/a	0 / 0
profile.php		100.00%	39 / 39		n/a	0 / 0
property.php		76.32%	29 / 38		n/a	0 / 0
search.php		77.27%	17 / 22		n/a	0 / 0
signup.php		100.00%	8 / 8		n/a	0 / 0

Legend

Low: 0% to 50% **Medium:** 50% to 90% **High:** 90% to 100%

Figure 4.1: PHP Code Coverage, report sample

It is possible to verify the number of lines covered by the tests in the total number of lines. The reports also classify the percentages in three levels: High percentage (90% to 100%), Medium percentage (50% to 90%) and Low percentage (0% to 50%). Ultimately, it is possible to visualize and comprehend the different percentages of code coverage on the website through these reports.

4.4 Summary

The experimental design was constituted by several stages to ultimately serve as a set of experiments to verify if the test cases, initially generated by the most common execution traces of real users using websites, followed by the injection of mutants with different mutation operators, could demonstrate an increment of code coverage and consequently improve the quality of those tests.

Concretely, the first stage of this experimental design constituted a selection of subjects for the experiment and participants. The following step is where the gathering of data and storage append, followed by the retrieval of this information based on the most common execution traces. Succeeding, the generation of the test cases from the data and their extent with the injection of mutants with five different mutation operators.

Lastly, all the test cases generated with mutants and without mutants were reproduced to verify the code coverage levels in each situation and compare them, as will be presented in the next chapter 5.

Chapter 5

Experiments and Results

As explained in the previous chapter, the experiments in this research work intended to ultimately verify if it was possible to improve the quality of the test cases by increasing code coverage through mutation injection in the most common execution traces of a web service.

Hence, during the experiments, it was expected that, in the end, it would be possible to verify different levels of code coverage depending on the use of the mutation operators.

In this stage, as the experiments' process is already determined and executed with real users as participants, one of the most important aspects is to compare the code coverage levels before the injection of mutations and after the injection of mutations.

Therefore this chapter's purpose is to exhibit the different levels of code coverage attained, with the generated test cases, with and without mutations. It will be exposed the behaviour of the of the levels of code coverage in each website, weather is increasing, decreasing or even maintained throughout the experiences.

As is continuously referred in this research, software quality, code coverage, and mutation testing are three crucial concepts that cooperate with each other. Consequently, resorting to mutation testing through the use of mutation operators, it is intended to increase the tests' coverage, improving the tests' quality as well, since they will be able to reach parts of the code that were not being tested before.

Accordingly, in this chapter, it is essential to verify if the test cases mutated achieved a more significant code coverage than the test cases without mutants. Additionally, it is also crucial to probe if this experimental design was relevant to improve the quality of the test cases using mutation testing, regarding not only the mutation operators used, but also the use of the most common execution traces of users as a starting point for this process.

Resorting to the *PHP* code coverage tool exposed in the preceding chapter, it is possible to obtain each code coverage values in each file of each website analyzed. The values obtained are displayed as percentages, meaning the ratio between the number of lines covered and the total number of lines in that particular file that is being analyzed.

It is possible to know accurately the amount of lines covered in each scenario, specifically when the tests reproduced were without mutants or with all types of mutants. The following graphic shows the percent values of code coverage attained for each website - subjects of the experiment - before using of the mutation operators and after.

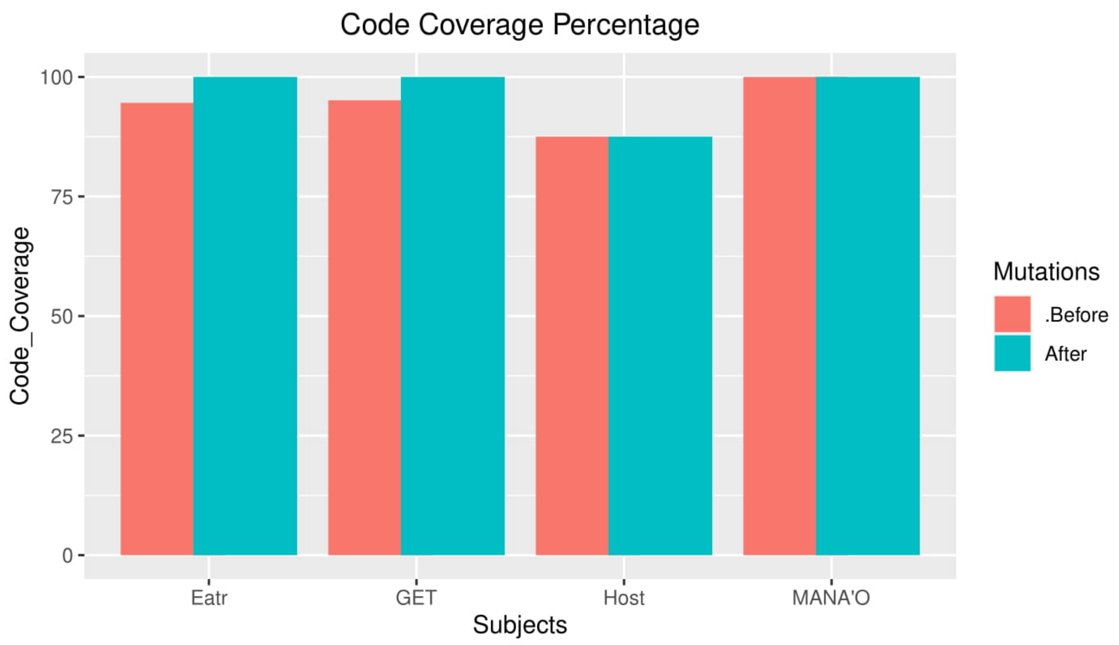


Figure 5.1: Code coverage levels in each website

The first website noticed in the graphic is Eatr, which exhibits adequate code coverage levels in both scenarios.

Accordingly it is noticeable that the code coverage level increased. Such means that some lines of the code that were not reached with the test cases without mutants, were reached by the test cases with mutants. The code coverage level attained with the tests with mutants represents total coverage, meaning that all lines of the code were executed by the tests.

The next subject of the experiment exhibited in the graphic is GET. Similar to the first one, this website also showed an increase in lines covered by the mutated tests, reaching a total coverage of the lines considered by the code coverage tool.

Therefore, both websites have shown an increase in the code coverage level. This means that the tests generated from the execution traces failed to reach lines of the code that were later reached by those same tests mutated through the application of mutation operators.

It is convenient to calculate the growth rate in each situation regarding the code coverage with or without mutants, to understand the growth between the scenarios. The formula used is as follows:

$$\frac{LevelWithMutants - LevelWithoutMutants}{LevelWithoutMutants} \times 100$$

This formula is an application of the general formula for growth rate, which is:

$$\frac{CurrentObservation - PreviousObservation}{PreviousObservation} \times 100$$

Hence, the growth rate of the first website, Eatr, is $\frac{100-94.57}{94.57} \times 100 = 5.74\%$. Accordingly, the growth rate of the second website, GET, is $\frac{100-95.11}{95.11} \times 100 = 5.14\%$. One of the reasons why the first website has a higher growth rate may be because the website is smaller, meaning it has fewer code lines, and in proportion, the improvement could be greater.

In the website Host, it is possible to verify that the test cases generated from the most common execution traces achieved the lowest code coverage value. Since this website, in particular, is the biggest one regarding the number of code lines, it is probably the reason why it achieves the lowest value of code coverage. After the mutation operators were applied, and the test cases were reproduced, it is possible to notice that an increase did not occur on the coverage levels. This means that the parts of the code reached by the initial test cases showed high coverage levels indeed, but the mutated tests could not reach the remaining code lines.

On the last website in the graphic MANA'O, it is possible to notice that code coverage levels are also the same for the two scenarios. Additionally, both levels represent a total coverage by the tests of all lines of the code. This means that by reproducing the first test cases, the ones that are generated through the execution traces and do not have mutations, all lines of code were reached. The reason why this happened is probably due to this website being the smallest and the least complex of all.

Therefore, in the last two subjects of experience referred, it is not reasonable to calculate the growth rate because there is no increase of lines covered from the test cases without mutants to the ones with mutants.

It is also convenient to demonstrate the level of code coverage attained from each mutation operator individually, ultimately to understand how each operator influenced the level of lines covered.

The levels of the code coverage of each website regarding the mutation operators individually, as well as the levels of code coverage before the use of these operators, are visible in the following graphic:

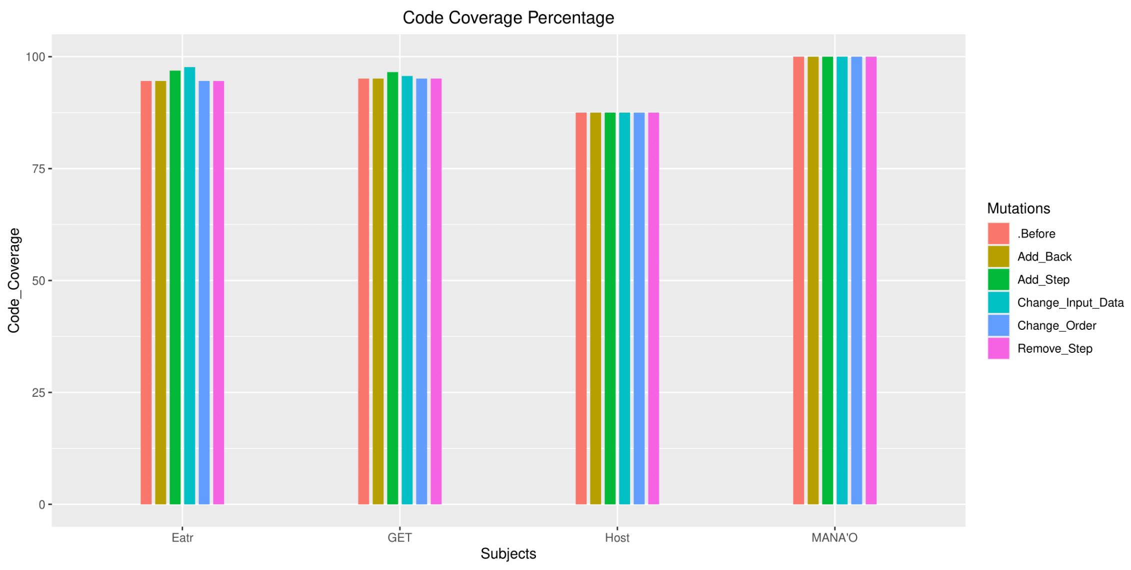


Figure 5.2: Code coverage levels of each mutation operator for each website

The websites Host and MANA'O don't show any improvement of the code coverage when we analyze each mutation operator separately. Such behaviour makes sense when we take into account the previous graphic - which exposes the code coverage levels before and after the application of all mutation operators. Since both websites do not show an improvement of the lines covered after mutations, it is also expected that each mutation operator individually does not demonstrate an increase of lines covered as well.

On the opposite, the experiences in the websites GET and Eatr showed different numbers of lines covered depending on the mutation operator used.

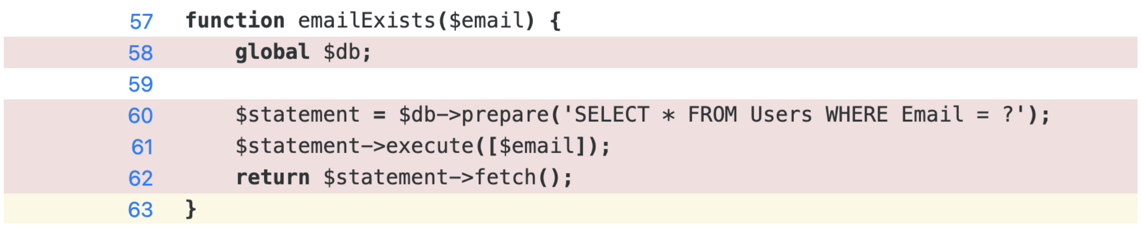
As it is visible through the graphic, the only mutation operators that showed improvements in the lines covered are the Add Step operator and Change Input Data operator.

It is essential to notice that these two operators reach different code lines from the ones achieved through the tests generated from the most common execution traces.

The Add Step mutation operator adds new input actions in different points of the sequence of user actions recorded, leading to the coverage of functions that verify fields.

The Change Input Data operator verifies the website's behaviour with different input data, leading to the change of coverage in functions that have the purpose of verifying different fields of input.

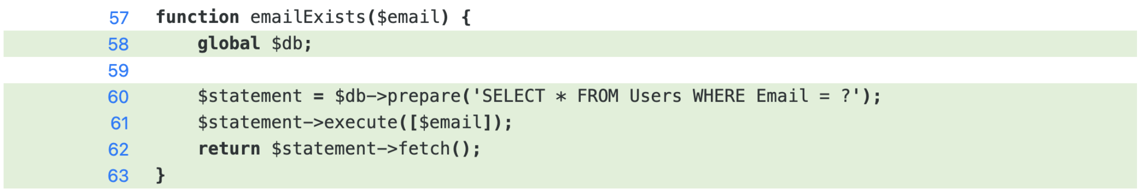
To allow a better understanding, below there is an example of one of the functions which the tests generated from the execution traces failed to cover, but was then reached by the tests mutated:



```
function emailExists($email) {
    global $db;

    $statement = $db->prepare('SELECT * FROM Users WHERE Email = ?');
    $statement->execute([$email]);
    return $statement->fetch();
}
```

Figure 5.3: Coverage of a function before mutations



```
function emailExists($email) {
    global $db;

    $statement = $db->prepare('SELECT * FROM Users WHERE Email = ?');
    $statement->execute([$email]);
    return $statement->fetch();
}
```

Figure 5.4: Coverage of a function after mutations

It is also convenient to calculate the growth rate of these two operators in the websites where an increase of lines covered is verified - GET and Eatr. To calculate this, we will resort to the same formula used previously in this chapter.

The value of the growth rate of the Add Step mutation operator in the website GET is $\frac{96.55-95.11}{95.11} \times 100 = 1.5\%$. The value of the growth rate of the Change Input Data operator in the same website is $\frac{95.69-95.11}{95.11} \times 100 = 0.6\%$.

On the other hand, the value of the growth rate of the Add Step mutation operator in the website Eatr is $\frac{96.89-94.57}{94.57} \times 100 = 2.45\%$. Also, the value of the growth rate of the Change Input Data operator in the same website is $\frac{97.67-94.57}{94.57} \times 100 = 3.3\%$.

As referred above, the website Eatr is the one that showed the biggest increase of code lines when comparing the tests with mutations to the tests without mutations; it is logical that the mutations achieve a higher growth rate in this website when compared to GET.

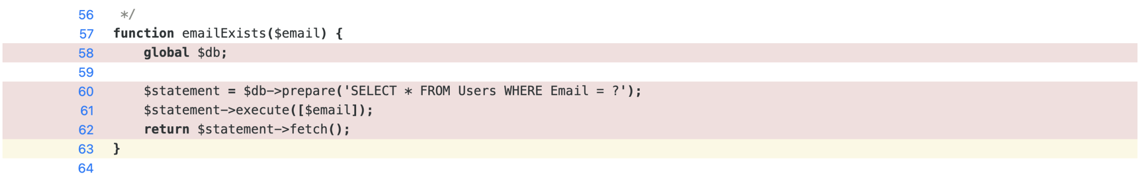
Concluding, there are several reasons that help us understand the results of the experiences. The most obvious reason for the websites to achieve such high code coverage levels is due to the lack of complexity in all of them. Accordingly, these websites were small in terms o number of code lines, allowing it to be easy for the tests to cover all lines.

Furthermore, one of the websites, Host, did not increase the number of covered lines after executing the mutated test cases. Also, the results of the same website showed that the test cases did not reach all lines. This could have happened because the mutation operators may not have been the best ones to extend the test cases, preventing them from reaching the more code lines. Also, the generation of test cases may have influenced this behaviour since it did not fully contain the users' inputs. For instance, it is possible that they do not reproduce the login in the websites properly, mostly due to the generation of data not having the initial logins of the users due to privacy reasons.

5.1 Conclusions of Results

After concluding the experiments, the results have shown that, in general, all code coverage levels regarding the subjects of the experiment were high. However, in two websites, there was no increase of lines covered by the tests in the two different scenarios: tests without mutants and tests with mutants. Such means that probably the selected mutant operators were not the most adequate and that the data generation for the test cases was not complete enough for these experiences.

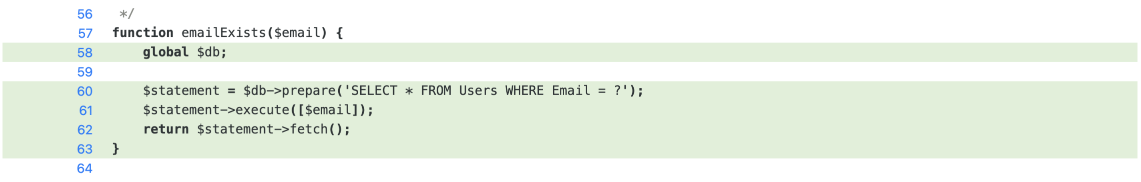
The remaining websites have shown an increase of the lines covered, leading to levels of coverage of 100%, meaning that all lines of the website were reached after reproducing the tests with an application of mutation operators. The achievement of total coverage means that the websites chosen were not complex or big enough to verify significant variations of code coverage levels. Regarding the increase of lines covered, to show how this was visible to us, the following images represent two print screens of an example of the reports provided by the *PHP* code coverage tool where the coverage changed after executing mutated test cases:



```
 */
function emailExists($email) {
    global $db;

    $statement = $db->prepare('SELECT * FROM Users WHERE Email = ?');
    $statement->execute([$email]);
    return $statement->fetch();
}
```

Figure 5.5: Example of a report showing a function which lines were not covered by the tests before mutations, website Eatr



```
 */
function emailExists($email) {
    global $db;

    $statement = $db->prepare('SELECT * FROM Users WHERE Email = ?');
    $statement->execute([$email]);
    return $statement->fetch();
}
```

Figure 5.6: Example of a report showing the same function after the execution of the mutated tests where the lines were covered, website Eatr

Another reason why the results of the experiences showed such high levels of code coverage is that in some cases, the real total of lines of the website's code was higher than the considered. That happens because there are always different programming languages on a website, and all the tools we considered using in this experiment to detect code coverage worked exclusively with one programming language. Therefore, we chose a tool that worked exclusively to *PHP* 4.3.1 because we believed that it was the most advantageous tool to detect lines covered on a website with the tests we were executing. So obviously, we used websites developed almost entirely on *PHP*. However, inevitably, some files could contain other programming languages, such as *HTML*. Therefore, the total of lines detected by the tool is in some cases smaller than the real total of lines of the websites, allowing it to be easier to reach high percentages of lines covered.

Finally, the experiments show that it is possible to reach lines of the code that were not reached by test cases through the extension of those test cases resorting to mutation operators. Nevertheless, the results are not explicit enough to make sure this is true for all situations; due to the lack of complexity of the websites chosen, the use of only one tool for code coverage, and a different choice of the mutation operators may lead to different results in the cases where the websites did not show an increase of lines covered.

Chapter 6

Conclusions

6.1 Summary

Nowadays, software is among everything we see and everything we need. A successfully developed software demands testing since it is one of the most essential stages for verifying and validating the quality of a product [6].

In software applications, mutation testing is one of the most influential and valuable white box testing techniques. The reason for its popularity comes from the possibility to continually discover more errors in the system under testing. Moreover, mutation testing leads to an extension of code coverage, ultimately resulting in an increase of the software's quality [14].

Having a set of appropriate software tests and assuring its quality level, is a very demanding task, especially in terms of resources [7]. Hence, testing software still has several limitations have been challenging developers, and they prove to be mandatory to overcome [12, 13, 3, 14].

Consequently, the remaining flaws in mutation testing, and even in testing in general, served as motivation for this research, which tried to improve tests quality of a software by resorting to mutation testing, avoiding the consequent amount of effort and time usually required. The problem is that the test suit of a software does not stays the same forever. Software often requires maintenance to properly function; either this maintenance results from improvements or problems. This way, maintenance requires time and effort, demanding a readjustment of the initial test set, implying investment of more resources and time [64, 107].

This thesis idea was that the software's usage information contained data regarding the changes needed, which might be useful for reducing the effort of maintaining the tests. Therefore, we collect the usage information using *MARTT* and generated test cases from that data, posteriorly mutating those test cases using *Mutations*. The purpose was to extend the test cases and reproduce them on the software to verify if code coverage has increased, improving the tests quality as well.

Hence, it was determined an experimental design to conduct experiences on four different websites, with thirty-one participants that explored and used the websites for several days. After collecting the usage information, test cases were generated with the most common execution traces, followed by the injection of mutants resorting to five different mutation operators. Finally, all the test cases were reproduced on the websites to retrieve information about the code coverage levels on each execution of the different types of tests.

Once the experiments were conducted, the results have shown that, in general, all code coverage levels regarding the subjects of the experiment were high. Specifically, three out of four websites reached 100% coverage after the mutated test cases being reproduced. However, the tests in one of the websites achieve all lines of the code even before the mutations being injected. This could be related to the lack of complexity and the total number of lines of the websites. The remaining website, which tests could not reach all lines of the code, also did not show an increase of the lines covered after mutations. Some reasons for such a result could be the way the generation of test cases is made, since it uses incomplete data from inputs due to privacy reasons, or the selected mutation operators for these experiences.

Finally, it is not possible to conclude with certainty that the experiments achieved their purpose to verify if the mutation of test cases, initially generated by the most common paths made by the users, could imply an increment of the code coverage in software services, improving the tests quality.

6.2 Challenges

The challenges of this research work began with the tools used that were previously developed. Therefore, additional work was necessary not only to update the tool's nomenclature and drivers but also to set them up to work as it was intended. Additionally, it was also required to automate one of the tools to allow the experiences to perform as expected, as mentioned in chapter 3.

Furthermore, due to the need to verify the code coverage, it was needed to select a tool that allowed tracking that could be visualized. Specifically, a tool that generated reports after each test execution that allow the developer to verify which lines were reached by the tests and which were not. Although the chosen tool fulfils every requirement, it only works with a particular language, *PHP*, in this case, which limited the number of code lines analyzed, resulting in fewer lines of code in total. Such was an inevitable challenge since all the tools considered for the experiments also worked with only one language, and despite that, the tool selected had everything we hoped for to allow a proper view of the lines covered. All the websites were mainly developed on *PHP*, but is almost inevitable for them to have some parts of other programming languages, e.g. *HTML*.

Another challenge related to the choice of websites was due to the retrieving of usage data, demanding scripts to run on the client-side and user-side. Therefore, the experiment subjects selected should also be suited for changing the code to use the scripts, which also limited the options.

Additionally, one of the significant limitations of this research work was time since multiple changes and adjustments that were not accounted for initially demanded time allocated to them, as referred to in the previous chapters.

Concluding, the limitations were part of this work, shaping it and leading it towards different decisions than the initial proposal. However, the experiences were successfully executed and lead to exciting results for the problem stated in the first chapter 1.

6.3 Future Work

This research work faced many hurdles, limitations and choices of procedures that allow various matters to be discussed in this section regarding future work.

One of the main aspects that is interesting for future work is the use of multiple tools for code coverage analysis. This way, it will be possible to detect the lines covered on various programming languages, allowing to use websites more complex and diversified for the experiments, attaining more exciting results. Therefore, it would be interesting to perform such experiences on websites with a higher level of complexity and thousands of daily users.

Furthermore, another interesting matter is regarding the procedure to generate tests cases in MARTT 3.1. This tool did not capture complete information about each element due to privacy reasons, for example usernames and passwords. Hence, if the data gathered had more accuracy, the generation of test cases by the tool Mutations 3.2 could be improved, allowing a more precise data to generate test cases and prevent issues such as the difficulty to login correctly. Future work could involve getting data licenses for allowing to extract the real inputs to use on this procedure.

Concluding, another intriguing aspect that deserves further work is the mutation operators used in the experiments. In the future, it would be interesting to do these experiments with different mutation operators related to more situations that may occur in a web context, extending the present mutation operators to detect them.

Appendix A

Form of the experience's participants

Several images of the forms' results were filled by the experiments' participants, where they provided information regarding their age, gender, academic degree and if they are still studying.

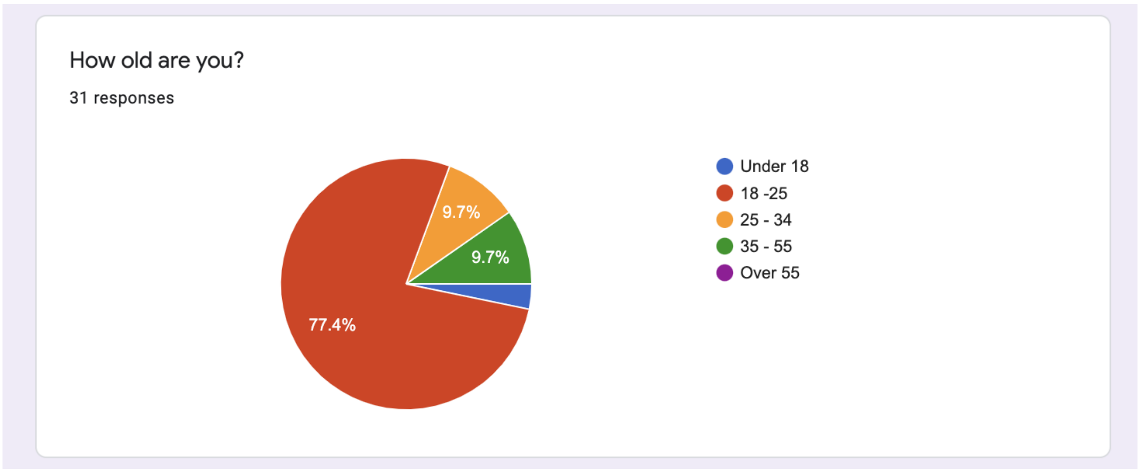


Figure A.1: Responses of the form filled by the participants about their age.

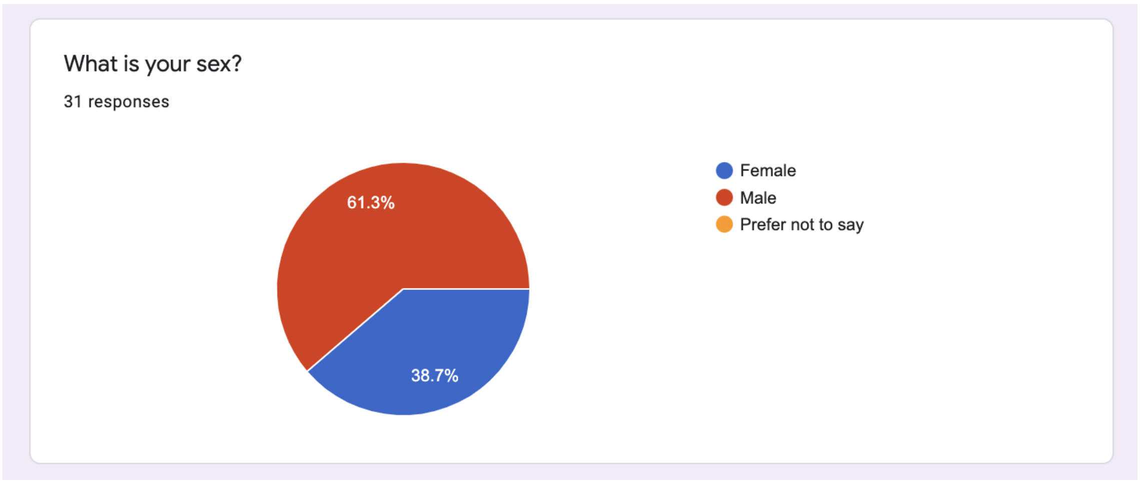


Figure A.2: Responses of the form filled by the participants about their gender.

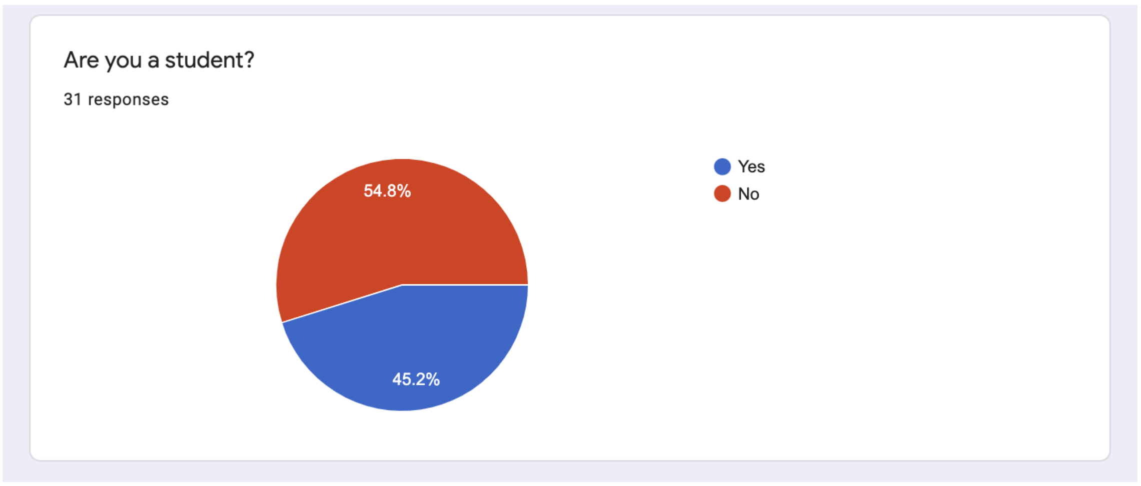
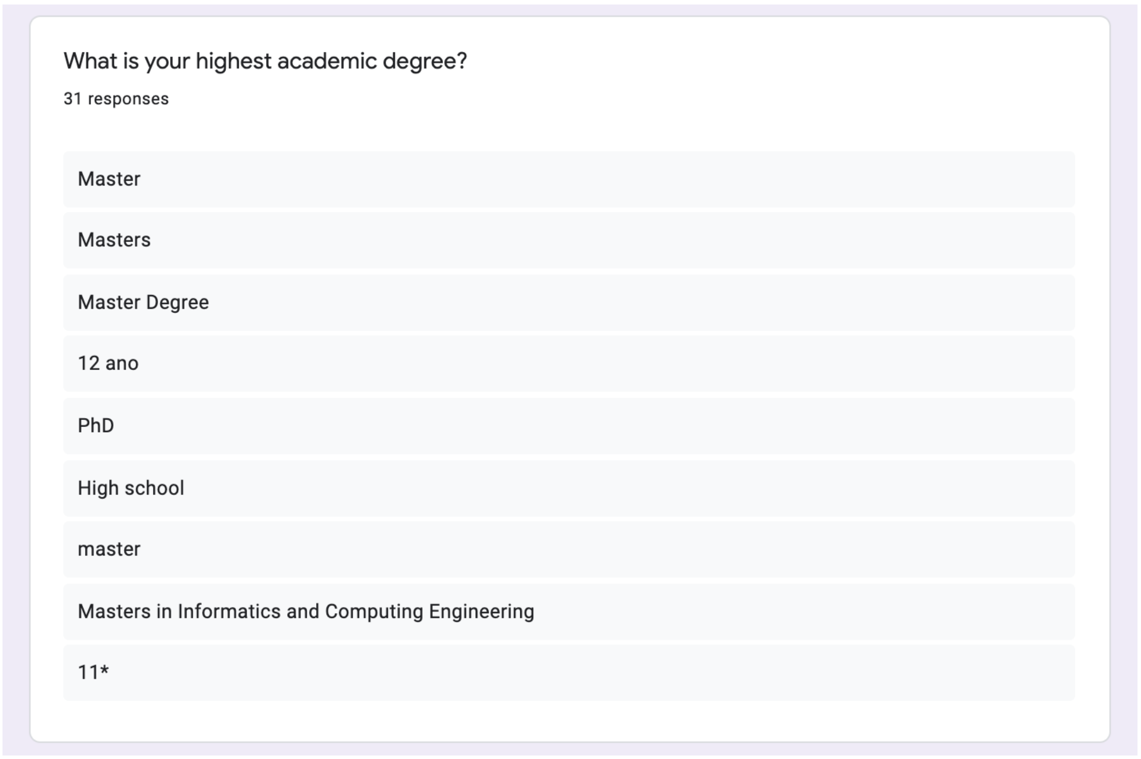


Figure A.3: Responses of the form filled by the participants regarding if they were students.



What is your highest academic degree?

31 responses

Master

Masters

Master Degree

12 ano

PhD

High school

master

Masters in Informatics and Computing Engineering

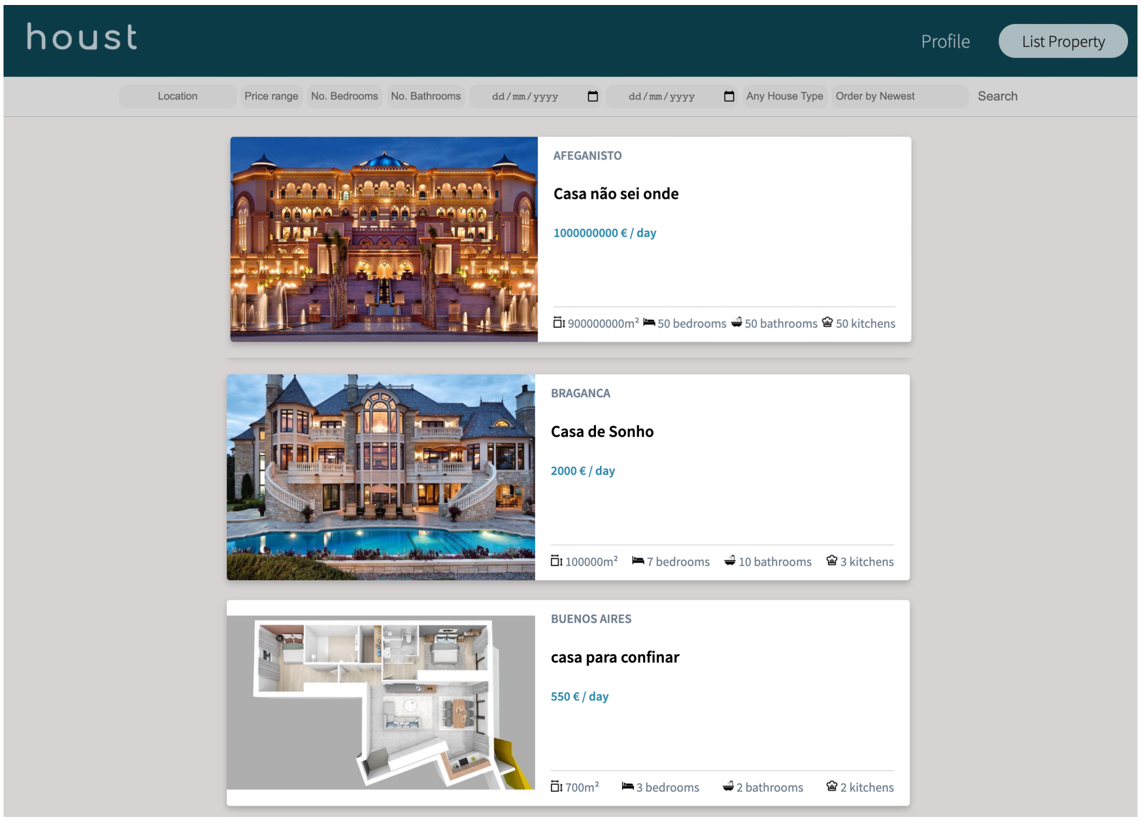
11*

Figure A.4: Responses of the form filled by the participants about their academic degree.

Appendix B

Subjects of the Experiment: Websites

In this Appendix is displayed some screenshots of the four websites used as subjects of the experiments.



AFEGANISTO

Casa não sei onde

1000000000 € / day

900000000m² 50 bedrooms 50 bathrooms 50 kitchens

BRAGANCA

Casa de Sonho

2000 € / day

100000m² 7 bedrooms 10 bathrooms 3 kitchens

BUENOS AIRES

casa para confinar

550 € / day

700m² 3 bedrooms 2 bathrooms 2 kitchens

Figure B.1: Website Host - Renting houses - Image of one of the webistes used on the experiments.

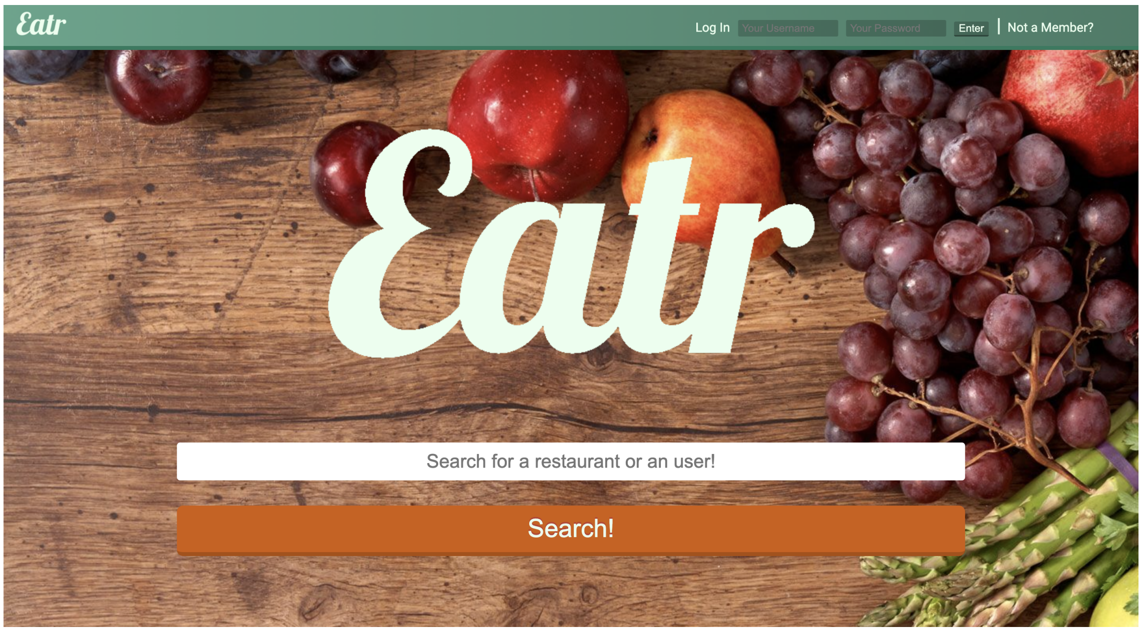
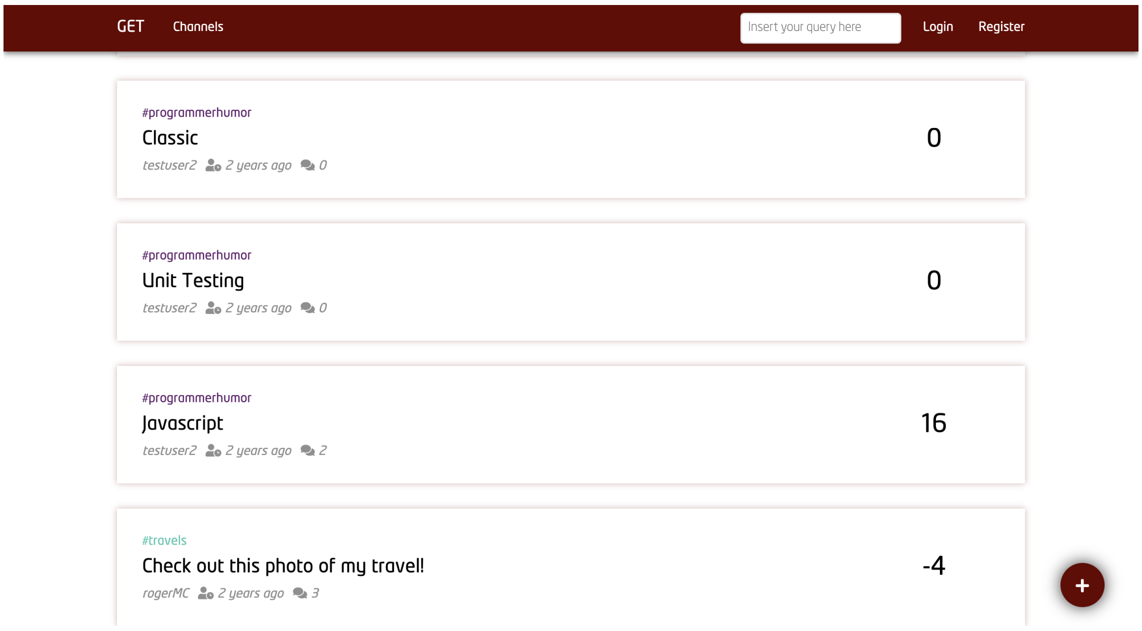


Figure B.2: Website Eart - Restaurants - Image of one of the webistes used on the experiments.



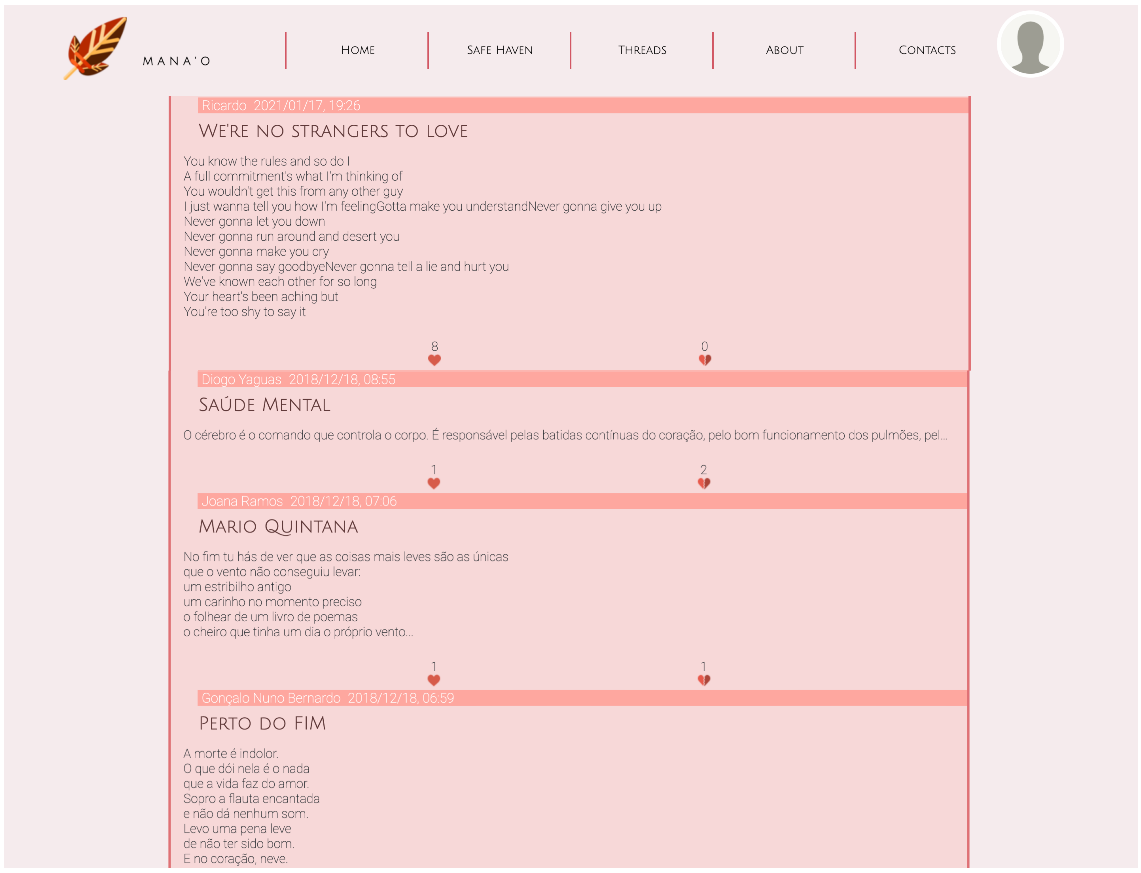
#programmerhumor
Classic
testuser2 2 years ago 0
0

#programmerhumor
Unit Testing
testuser2 2 years ago 0
0

#programmerhumor
Javascript
testuser2 2 years ago 2
16

#travels
Check out this photo of my travel!
rogerMC 2 years ago 3
-4

Figure B.3: Website GET - Channel - Image of one of the webistes used on the experiments.



Ricardo 2021/01/17, 19:26

WE'RE NO STRANGERS TO LOVE

You know the rules and so do I
A full commitment's what I'm thinking of
You wouldn't get this from any other guy
I just wanna tell you how I'm feelingGotta make you understandNever gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbyeNever gonna tell a lie and hurt you
We've known each other for so long
Your heart's been aching but
You're too shy to say it

8 0

Diogo Yaguas 2018/12/18, 08:55

SAÚDE MENTAL

O cérebro é o comando que controla o corpo. É responsável pelas batidas contínuas do coração, pelo bom funcionamento dos pulmões, pel...

1 2

Joana Ramos 2018/12/18, 07:06

MARIO QUINTANA

No fim tu hás de ver que as coisas mais leves são as únicas
que o vento não conseguiu levar:
um estribilho antigo
um carinho no momento preciso
o folhear de um livro de poemas
o cheiro que tinha um dia o próprio vento...

1 1

Gonçalo Nuno Bernardo 2018/12/18, 06:59

PERTO DO FIM

A morte é indolor.
O que dói nela é o nada
que a vida faz do amor.
Sopro a flauta encantada
e não dá nenhum som.
Levo uma pena leve
de não ter sido bom.
E no coração, neve.

Figure B.4: Website MANA'O - Social Media - Image of one of the webistes used on the experiments.

References

[1] International Software Testing Qualifications Board, "ISTQB Glossary."

[2] A. L. Smith, "Chasing Mutants," in *The Future of Software Quality Assurance*, pp. 147–159, Cham: Springer International Publishing, 2020.

[3] Y. Jia, S. Member, and M. H. Member, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, pp. 1–27, 2017.

[4] International Software Testing Qualifications Board, "ISTQB Glossary."

[5] P. C. Jorgensen and C. Kaner, "Exploratory Testing," *Software Testing*, no. c, pp. 369–374, 2006.

[6] G. Grano, "A new dimension of test quality: Assessing and generating higher quality unit test cases," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 419–423, Association for Computing Machinery, 2019.

[7] M. Alenezi, M. Akour, A. Hussien, and M. Z. Al-Saad, "Test suite effectiveness: An indicator for open source software quality," *2016 2nd International Conference on Open Source Software Computing, OSSCOM 2016*, no. December, 2017.

[8] R. M. L. M. Moreira, A. C. R. Paiva, M. Nabuco, and A. Memon, "Pattern-based GUI testing: Bridging the Gap Between Design and Quality Assurance," *Software Testing Verification and Reliability*, 2017.

[9] B. Baudry, F. Fleurey, J. M. Jezequel, and Y. Le Traon, "Genes and bacteria for automatic test cases optimization in the.NET environment," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. January, no. November, pp. 195–206, 2019.

[10] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, and A. García-Domínguez, "Quality metrics for mutation testing with applications to WS-BPEL compositions," *Software Testing, Verification and Reliability*, vol. 25, pp. 536–571, aug 2015.

[11] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," in *2008 The Eighth International Conference on Quality Software*, pp. 77–86, IEEE, aug 2008.

[12] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 112–122, Association for Computing Machinery, 2019.

[13] A. Ayad, I. Marsit, N. Mohamed Omri, J. M. Loh, and A. Mili, “Using Semantic Metrics to Predict Mutation Equivalence,” *Communications in Computer and Information Science*, vol. 1077, no. October, pp. 3–27, 2019.

[14] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, “Performance mutation testing,” *Software Testing, Verification and Reliability*, jan 2020.

[15] S. Suguna Mallika and D. Rajya Lakshmi, “MUTWEB-A testing tool for performing mutation testing of java and servlet based web applications,” *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 12, pp. 5406–5413, 2019.

[16] S. Hamimoune and B. Falah, “Mutation testing techniques: A comparative study,” *Proceedings - 2016 International Conference on Engineering and MIS, ICEMIS 2016*, no. November 2016, 2016.

[17] A. Paiva, “TVVS – Software Testing , Verification and Validation Mutation Testing,” 2019.

[18] C. Kaner, “Fundamental Challenges in Software Testing,” no. April, pp. 1–37, 2003.

[19] M. Vasconcelos, “Mining Web Usage to Generate Regression GUI Tests Automatically,” 2019.

[20] M. Shahid and S. Ibrahim, “An Evaluation of Test Coverage Tools in Software Testing,” *ResearchGate*, vol. 5, no. January, pp. 216–222, 2011.

[21] M. Ivankovi, G. Petrovi, R. Just, and G. Fraser, “Code coverage at Google,” *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, no. August, pp. 955–963, 2019.

[22] N. Gupta, V. Yadav, and M. Singh, “Automated Regression Test Case Generation for Web Application,” *ACM Computing Surveys*, vol. 4, no. 1-25, p. 51, 2018.

[23] P. Haar, “Automated GUI Testing : A Comparison Study With A Maintenance Focus,” 2018.

[24] S. Reid, *The Art of Software Testing, Second edition. Glenford J. Myers. Revised and updated by Tom Badgett and Todd M. Thomas, with Corey Sandler. John Wiley and Sons, New Jersey, U.S.A., 2004. ISBN: 0-471-46912-2, pp 234*, vol. 15. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.

[25] A. C. Paiva, A. Restivo, and S. Almeida, “Test Case Generation Based on Mutations Over User Execution Traces,” *Software Quality Journal*, 2020.

[26] A. Barbosa, A. C. Paiva, and J. C. Campos, “Test case generation from mutated task models,” *Proceedings of the 2011 SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2011*, pp. 175–184, 2011.

[27] G. E. Murphy, “An abstract of the dissertation of,” *October*, no. October, p. 302, 2007.

[28] A. J. Offutt and R. H. Untch, “Mutation 2000: Uniting the Orthogonal,” *Mutation Testing for the New Century*, pp. 34–44, 2001.

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

[30] C. Brito, V. H. S. Durelli, R. S. Durelli, S. R. S. d. Souza, A. M. R. Vincenzi, and M. E. Delamaro, "A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 304–313, 2020.

[31] M. Baer, N. Oster, and M. Philippsen, "Mutantdistiller: Using symbolic execution for automatic detection of equivalent mutants and generation of mutant killing tests," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 294–303, 2020.

[32] A. Duque-Torres, N. Doliashvili, D. Pfahl, and R. Ramler, "Predicting survived and killed mutants," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 274–283, 2020.

[33] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, pp. 434–487, jan 2020.

[34] F. G. D. O. Neto, F. Dobslaw, and R. Feldt, "Using mutation testing to measure behavioural test diversity," *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, pp. 254–263, 2020.

[35] A. Oliveira, R. Freitas, A. Jorge, V. Amorim, N. Moniz, A. C. R. Paiva, and P. J. Azevedo, "Sequence mining for automatic generation of software tests from GUI event traces," in *Intelligent Data Engineering and Automated Learning - IDEAL 2020 - 21st International Conference, Guimaraes, Portugal, November 4-6, 2020, Proceedings, Part II* (C. Analide, P. Novais, D. Camacho, and H. Yin, eds.), vol. 12490 of *Lecture Notes in Computer Science*, pp. 516–523, Springer, 2020.

[36] H. N. da Silva, P. R. Farah, W. D. F. Mendonça, and S. R. Vergilio, "Assessing android test data generation tools via mutation testing," in *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing*, SAST 2019, (New York, NY, USA), p. 32–41, Association for Computing Machinery, 2019.

[37] A. A. Saifan and A. A. Alzyoud, "Mutation Testing to Evaluate Android Applications," *International Journal of Open Source Software and Processes*, vol. 11, pp. 23–40, jan 2020.

[38] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 33–36, 2018.

[39] A. Zarrad, I. Alsmadi, and A. Yassine, "Mutation testing framework for ad-hoc networks protocols," in *2020 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–8, 2020.

[40] J. B. de Souza Neto, A. Martins Moreira, G. Vargas-Solar, and M. A. Musicante, "Mutation Operators for Large Scale Data Processing Programs in Spark," in *International Conference on Advanced Information Systems Engineering*, pp. 482–497, Springer, Cham, 2020.

[41] M. B. Bashir and A. Nadeem, "An Evolutionary Mutation Testing System for Java Programs: eMuJava," in *Intelligent Computing - Proceedings of the Computing Conference*, pp. 847–865, Springer, Cham, 2019.

[42] P. Pinheiro, J. C. Viana, L. Fernandes, M. Ribeiro, F. Ferrari, B. Fonseca, and R. Gheyi, "Mutation Operators for Code Annotations," in *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing - SAST '18*, (New York, New York, USA), pp. 77–86, ACM Press, 2018.

[43] Y.-S. Ma, J. Offutt, and Y. Kwon, "MuJava: an automated class mutation system: Research Articles," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[44] K. Kumar, P. K. Gupta, and R. Parjapat, "New Mutants Generation for Testing Java Programs," in *International Conference on Advances in Communication, Network, and Computing*, pp. 290–294, Springer, Berlin, Heidelberg, 2011.

[45] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Trans. Softw. Eng.*, vol. 27, p. 228–247, Mar. 2001.

[46] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," *Proceedings - International Computer Software and Applications Conference*, no. September, pp. 979–984, 2008.

[47] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *J. Syst. Softw.*, vol. 115, p. 18–30, May 2016.

[48] P. Pinheiro, J. C. Viana, M. Ribeiro, L. Fernandes, F. Ferrari, R. Gheyi, and B. Fonseca, "Mutating code annotations: An empirical evaluation on Java and C# programs," *Science of Computer Programming*, vol. 191, p. 102418, jun 2020.

[49] S. Uzunbayir and K. Kurtel, "An Analysis on Mutation Testing Tools For C# Programming Language," in *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pp. 439–443, IEEE, sep 2019.

[50] A. Derezinska 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*, pp. 261–268, IEEE, 2008.

[51] A. A. Saifan and M. B. Ata, "Mutation Testing for Evaluating PHP Web Applications," *International Journal of Software Innovation*, vol. 7, pp. 25–50, oct 2019.

[52] R. Jahanshahi, A. Doupé, and M. Egele, "Mitigating SQL Injection Attacks on Legacy Web Applications," *Pro- ceedings ofthe 15th ACMAsia Conference on Computer and Communications Security (ASIA CCS '20)*, no. July, pp. 1–13, 2020.

[53] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "SQLMutation: A tool to generate mutants of SQL database queries," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 1–1, IEEE, nov 2006.

[54] I. Bluemke and A. Sawicki, "Tool for Mutation Testing of Web Services," in *International Conference on Dependability and Complex Systems*, pp. 46–55, Springer, Cham, 2019.

[55] C.-A. Sun, H. Dai, G. Wang, D. Towey, T. Y. Chen, and K.-Y. Cai, "Dynamic Random Testing of Web Services: A Methodology and Evaluation," *IEEE Transactions on Services Computing*, pp. 1–1, 2020.

[56] Y. Li, Z.-G. Sun, and T.-T. Jiang, “An Automated Test Suite Generating Approach for Stateful Web Services,” in *International Conference on Software Analysis, Testing, and Evolution*, pp. 185–201, Springer, Cham, 2018.

[57] 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)*, pp. 149–156, IEEE, apr 2008.

[58] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 654–665, Association for Computing Machinery, 2014.

[59] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proceedings of the 27th International Conference on Software Engineering*, ICSE ’05, (New York, NY, USA), p. 402–411, Association for Computing Machinery, 2005.

[60] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 654–665, Association for Computing Machinery, 2014.

[61] M. Papadakis, D. Shin, S. Yoo, and D. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 537–548, 2018.

[62] A. S. Namin and S. Kakarla, “The use of mutation in testing experiments and its sensitivity to external threats,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA ’11, (New York, NY, USA), p. 342–352, Association for Computing Machinery, 2011.

[63] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, “Assessing test case prioritization on real faults and mutants,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 240–251, 2018.

[64] I. Marsit, M. N. Omri, J. M. Loh, and A. Mili, “Impact of mutation operators on the ratio of equivalent mutants,” *Frontiers in Artificial Intelligence and Applications*, vol. 303, no. September, pp. 664–677, 2018.

[65] International Organization for Standardization, “ISO/IEC 25010.”

[66] M. Trakhtenbrot, “New Mutations for Evaluation of Specification and Implementation Levels of Adequacy in Testing of Statecharts Models,” no. October, pp. 151–160, 2008.

[67] G. Fraser and F. Wotawa, “Mutant Minimization for Model-Checker Based Test-Case Generation,” no. May 2014, pp. 161–168, 2008.

[68] Y. Le Traon, B. Baudry, and J. M. Jézéquel, “Design by contract to improve software vigilance,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 571–586, 2006.

[69] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mutation analysis of parameterized unit tests," *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, no. May 2014, pp. 177–181, 2009.

[70] P. Delgado-Pérez, I. Medina-Bulo, and M. G. Merayo, "Using Evolutionary Computation to Improve Mutation Testing," in *International Work-Conference on Artificial Neural Networks*, pp. 381–391, Springer, Cham, 2017.

[71] S. Vercammen, M. Ghafari, S. Demeyer, and M. Borg, "Goal-oriented mutation testing with focal methods," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, (New York, NY, USA), p. 23–30, Association for Computing Machinery, 2018.

[72] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 100–111, 2018.

[73] Y. L. T. B. Baudry, V. Le Hanh, *Trustable Components: Yet Another Mutation-Based Approach.* Springer, Boston, MA, proceeding ed.

[74] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, (New York, NY, USA), p. 45–54, Association for Computing Machinery, 2016.

[75] X. Qin, S. Liu, and Z. Tao, "A New Mutant Generation Algorithm Based on Basic Path Coverage for Mutant Reduction," pp. 167–186, 2020.

[76] S. Rani, H. Dhawan, G. Nagpal, and B. Suri, "Implementing Time-Bounded Automatic Test Data Generation Approach Based on Search-Based Mutation Testing," in *Progress in Advanced Computing and Intelligent Engineering*, pp. 113–122, Springer, Singapore, 2019.

[77] D. B. Mishra, R. Mishra, A. A. Acharya, and K. N. Das, "Test Data Generation for Mutation Testing Using Genetic Algorithm," in *Soft Computing for Problem Solving*, pp. 857–867, Springer, Singapore, 2019.

[78] D. B. Mishra, R. Mishra, K. N. Das, and A. A. Acharya, "Test Case Generation and Optimization for Critical Path Testing Using Genetic Algorithm," in *Soft Computing for Problem Solving*, pp. 67–80, Springer, Singapore, 2019.

[79] R. E. Atani, H. Farzaneh, and S. Bakhshayeshi, "A Glance on Performance of Fitness Functions Toward Evolutionary Algorithms in Mutation Testing," in *The 7th International Conference on Contemporary Issues in Data Science*, pp. 59–75, Springer, Cham, 2020.

[80] S. Almeida, A. C. Paiva, and A. Restivo, "Mutation-Based Web Test Case Generation," *Communications in Computer and Information Science*, vol. 1010, pp. 339–346, 2019.

[81] D. M. Shahid and S. Ibrahim, "An evaluation of test coverage tools in software testing," 01 2011.

[82] M. Lormans and A. van Deursen, "Reconstructing requirements coverage views from design and test using traceability recovery via lsi," in *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, (New York, NY, USA), p. 37–42, Association for Computing Machinery, 2005.

[83] A. Aghamohammadi, S.-H. Mirian-Hosseinabadi, and S. Jalali, “Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness,” *Information and Software Technology*, vol. 129, p. 106426, 2021.

[84] Q. Zhu, A. Panichella, and A. Zaidman, “An investigation of compression techniques to speed up mutation testing,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 274–284, 2018.

[85] Q. Zhu, A. Zaidman, and A. Panichella, “How to kill them all: An exploratory study on the impact of code observability on mutation testing,” *Journal of Systems and Software*, vol. 173, p. 110864, mar 2021.

[86] Suguna Mallika S. and Rajya Lakshmi D., “Mutation Testing and Its Analysis on Web Applications for Defect Prevention and Performance Improvement,” *International Journal of e-Collaboration*, vol. 17, pp. 71–88, jan 2021.

[87] A. Paiva, “TVVS – Software Testing , Verification and Validation Mutation Testing,” p. 27, 2019.

[88] Q. V. Nguyen and L. Madeyski, “Problems of Mutation Testing and Higher Order Mutation Testing,” in *Advanced Computational Methods for Knowledge Engineering*, pp. 157–172, Springer, Cham, 2014.

[89] N. Jatana, B. Suri, P. Kumar, and B. Wadhwa, “Test suite reduction by mutation testing mapped to set cover problem,” in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ICTCS ’16, (New York, NY, USA), Association for Computing Machinery, 2016.

[90] N. Jatana and B. Suri, “An Improved Crow Search Algorithm for Test Data Generation Using Search-Based Mutation Testing,” *Neural Processing Letters*, jun 2020.

[91] N. Shomali and B. Arasteh, “Mutation reduction in software mutation testing using firefly optimization algorithm,” *Data Technologies and Applications*, vol. ahead-of-p, may 2020.

[92] G. Guizzo, F. Sarro, and M. Harman, “Cost Measures Matter for Mutation Testing Study Validity,” *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).*, no. Cost Measures Matter for Mutation Testing Study Validity, p. 13, 2020.

[93] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, “Predictive mutation testing,” *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.

[94] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, “A systematic literature review of techniques and metrics to reduce the cost of mutation testing,” *Journal of Systems and Software*, vol. 157, p. 110388, nov 2019.

[95] M. Singh and V. M. Srivastava, “Extended firm mutation testing: A cost reduction technique for mutation testing,” in *2017 Fourth International Conference on Image Information Processing (ICIIP)*, pp. 1–6, 2017.

[96] Z. X. Lu, S. Vercammen, and S. Demeyer, “Semi-automatic test case expansion for mutation testing,” in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pp. 1–7, 2020.

[97] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Employing dynamic symbolic execution for equivalent mutant detection," *IEEE Access*, vol. 7, pp. 163767–163777, 2019.

[98] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, pp. 275–378, Elsevier, 2019.

[99] S. Liu and S. Nakajima, "Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[100] "Oracle Assessment , Improvement and Placement Gunel Jahangirova A report submitted in fulfillment of the requirements for the degree of Doctor of Philosophy of University College London . Department of Computer Science University College London," 2019.

[101] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[102] E. Guerra, J. Sánchez Cuadrado, and J. de Lara, "Towards effective mutation testing for atl," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 78–88, 2019.

[103] M. Vasconcelos, "Mining Web Usage to Generate Regression GUI Tests Automatically," 2019.

[104] S. Almeida, A. C. Paiva, and A. Restivo, "Mutation-Based Web Test Case Generation," *Communications in Computer and Information Science*, vol. 1010, pp. 339–346, 2019.

[105] "The database model showdown: An rdbms vs. graph comparison,"

[106] S. Bergmann, "php-code-coverage,"

[107] Y. M. Choi and D. J. Lim, "Model-based test suite generation using mutation analysis for fault localization," *Applied Sciences (Switzerland)*, vol. 9, no. 17, 2019.