FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mutation-based Web Test Case Generation

Sérgio Miguel Almeida Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva

Second Supervisor: André Monteiro de Oliveira Restivo

July 22, 2019

Mutation-based Web Test Case Generation

Sérgio Miguel Almeida Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor João Pascoal Faria

External Examiner: Doctor João Saraiva

Supervisor: Doctor Ana Cristina Ramada Paiva

July 22, 2019

Abstract

One way to increase software quality is testing. Generating test cases with a high level of coverage may be neither a simple nor fast task and when the software suffers changes, it is even more difficult to guarantee this coverage. Mutation testing is a fault-based software testing technique that introduces small faults in the code - called mutations - and evaluates if the test suite is able to detect these faults, i.e., distinguish the results obtained in the original version of the program from the results obtained in the mutated version, based on a mutation score. It is measured by ‘killing’ or not the mutants when the tests are executed: if the mutant is killed, then the test case is able to detect this fault; otherwise, if the mutant survived, it is necessary to add more test cases to detect and cover this fault. In this project, this technique is applied in a different way: generate test cases.

This project aims to use information about the most frequent paths of the user interaction on a software service, collected by a web analytics tool in order to generate test cases and extend them with mutations (mutations applied over the tests) that can make sense in web testing. There are some approaches which generate automatically test cases. Nevertheless, there are several challenges to overcome in this area such as the complexity of the regression test, the necessity of knowing about this kind of test and how to implement them. It is also necessary to have documented all the input and output data and the steps to apply the test case and some tools actually cannot capture input values and it implies the needing of insert it manually by the tester.

This work includes the definition of a catalog of mutations such as, for example, change the order of execution of the tasks. The main objective is defining the appropriate mutations and implement them automatically in the test cases to check if they have different behaviour from the original test cases. This approach will allow to automatically generate test cases based on the logs information and web usage data of services and extend them with some possible mutations. The implementation of the generator of the test cases is also included. In this way, it is expected that the quality of the test suite will be improved once there will be test cases that simulate wrong user behaviors and aspects that are not tested yet. Consequently, the coverage will be increased with the new test cases. This tool is useful to detect automatically errors in web pages and have a better test suite based on the most frequent paths of the service.

This approach is validated with a simple developed scenario and with real usage information from the *Polytechnic Institute of Viana do Castelo* website. A considerable number of mutated test cases with different behaviour were generated in both situations.

Resumo

Uma forma de aumentar a qualidade do software é testando-o. A geração de casos de teste com um alto nível de cobertura pode não ser uma tarefa simples nem rápida e, quando o software sofre alterações, é ainda mais difícil garantir essa cobertura. O teste de mutação é uma técnica de teste de software baseada em falhas, que introduz pequenas falhas no código - chamadas mutações - e avalia se o conjunto de testes é capaz de detetar essas falhas, ou seja, distinguir os resultados obtidos na versão original do programa dos resultados obtidos na versão mutada, com base numa pontuação. Essa pontuação é calculada por "matar" ou não os mutantes quando os testes são executados: se o mutante é morto, então o caso de teste é capaz de detectar essa falha; caso contrário, se o mutante sobreviver, é necessário adicionar mais casos de teste para detectar e cobrir essa falha.

Este projeto visa utilizar informações sobre os caminhos mais frequentes da interação do utilizador num serviço de software, recolhidos por uma ferramenta de *web analytics* para gerar casos de teste e estendê-los com mutações (mutações aplicadas nos testes) que podem fazer sentido em testes web. Existem algumas abordagens que geram automaticamente casos de teste. No entanto, há vários desafios a serem superados nessa área, como a complexidade dos testes de regressão, a necessidade de conhecer esse tipo de teste e como implementá-lo. Também é necessário documentar todos os dados de *input* e *output*, os passos para aplicar o caso de teste, e algumas ferramentas não fornecem valores de *input* e isso implica a necessidade de inseri-los manualmente pelo *tester*.

Este trabalho inclui a definição de um catálogo de mutações como, por exemplo, alterar a ordem de execução das tarefas. O objetivo principal é definir as mutações apropriadas e implementá-las automaticamente nos casos de teste para verificar se eles têm um comportamento diferente dos casos de teste originais. Esta abordagem permitirá gerar automaticamente os casos de teste com base nas informações de logs e nos dados de uso da Web e estendê-los com algumas mutações. A implementação do gerador dos casos de teste também está incluída. Desta forma, espera-se que a qualidade do conjunto de testes seja melhorada, uma vez que haverá casos de teste que simulam comportamentos e aspectos errados do utilizador que ainda não foram testados. Consequentemente, a cobertura será aumentada com os novos casos de teste. Esta ferramenta é útil para detectar automaticamente erros em páginas Web e ter um conjunto de testes melhor baseado nos caminhos mais frequentes do *software*.

Esta abordagem é validada com um simples cenário desenvolvido e com informação de utilização real do site do *Instituto Politécnico de Viana do Castelo*. Um número considerável de casos de teste mutados com comportamento diferente foi gerado em ambas as situações.

Acknowledgements

First, to my parents João and Rosa. Thank you for all the effort and constant dedication. Thank you for believing in me and for encouraging me every day to be a good person and to fight against all the barriers of everyday life. This would not be possible without your full support.

To my godmother Ana for being the most incredible person. For unconditional support and for teaching me how to succeed in life without hurting anyone.

To my sister Silvina, grandmother Lena and aunt Carminha, for all the friendship and affection. Thank you for giving me love moments and for being part of my family.

To Professor Ana Paiva and Professor André Restivo, for all the guidance and patience throughout this project. Your advice and help were fundamental.

To Ritinha, for always being there and hearing my outbursts. Thank you for encouraging me and believing in me. This work is yours too.

To my friends Lopes, Charlotte, Hugo, Nuno, Grulha and Gonçalo, for all the moments of joy and happiness that you gave me in my breaks. For showing me life perspectives I had never known, even in the most strange ways.

To Santana, for being like a brother, and making me see that effort is rewarded. For your friendship.

To my lovely FEUP family: Carol, Paulo, Tiago, Rui, Marta, Chi, Ariana, Pingu, Beatriz, and Afonso for giving me these years of friendship and companionship. You have made this journey so much better and I would not be right here without your support!

And thank you for all the other people that I have met this season. It was a great experience!

Sérgio Almeida

“It always seems impossible until it is done.”

Nelson Mandela

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation and Goals	2
1.3	Problem	2
1.4	Solution	3
1.5	Contributions	4
1.6	Dissertation Structure	4
2	Background and State of the Art	5
2.1	Software Testing	5
2.1.1	Software Testing Levels	5
2.1.2	Techniques	6
2.1.3	Regression Testing	7
2.1.4	GUI Testing	8
2.2	Mutation Testing	9
2.2.1	Mutation Testing applied in models	11
2.3	Web Testing	13
2.4	Test Case Generation	15
2.4.1	Test case generation using Search-based algorithms	15
2.4.2	Test case generation using MBT	16
2.4.3	Test case generation using Adaptive Random testing algorithm	17
2.5	Web Analytics	17
2.6	Summary	18
3	Test Case Mutation	21
3.1	Introduction	21
3.2	Abstract Test Cases Extraction	21
3.3	Conversion into Concrete Test Cases	25
3.3.1	Data Generator	27
3.4	Mutations' Injection	28
3.4.1	Mutation Operators	28
3.4.2	Pre-processing	31
3.4.3	Injection process	33
3.5	Evaluation of the test cases generated	34
3.6	Execution	36
3.7	Summary	36

CONTENTS

4	Experiments	39
4.1	Simple Website	39
4.2	Real Website	43
4.2.1	Mutations' generation	45
5	Conclusions and Future Work	47
5.1	Conclusions	47
5.2	Future Work	48
	References	49
A	Experiment Simple Website	55
A.1	Abstract Test Case	55
A.2	Concrete test cases	58

List of Figures

2.1	Levels of Testing. Extracted from [CB02].	6
2.2	Test strategies. Extracted from [CB02]	7
2.3	Process of Mutation Testing. Extracted from [JH11].	10
2.4	Process of Model-based Testing. Extracted from [UPL12]	12
2.5	Process of Model-based mutation Testing. Extracted from [BBH ⁺ 16]	13
2.6	Process of Web Analytics. Extracted from [WK09]	17
3.1	Diagram of implementation.	22
3.2	Example of password verification.	27
3.3	Example of poorly implemented password verification.	29
3.4	Example of the state of the form before clicking on 'Sign Up' and perform a Back action (left) and after a Back action (right).	29
3.5	Example of Change Order Mutation Operator writing firstly the second field (left) and then the first field (right).	30
3.6	Example of a mandatory field.	31
4.1	First page of the simple website.	40
4.2	Second page of the simple website.	41
4.3	Relationship between the <i>Levenshtein Distance</i> and the mutated test cases of Experiment 1	42
4.4	Relationship between the number of interactions and the number of stored sessions	43
4.5	Relationship between the <i>Levenshtein Distance</i> and the mutated test cases of Experiment 2	44
4.6	Relationship between the size of the test suite and the number of generated test cases.	45

LIST OF FIGURES

List of Tables

2.1	Comparison between web testing techniques	15
3.1	Comparison between operators in injection process	33
3.2	<i>Levenshtein Distance</i> equals to 1	35
3.3	<i>Levenshtein Distance</i> higher than 1	35
4.1	Evaluation of the array of steps with <i>Levenshtein Distance</i> of Experiment 1 . . .	42
4.2	Evaluation of the array of steps with <i>Levenshtein Distance</i> of Experiment 2 . . .	44

LIST OF TABLES

Abbreviations

CLI	Command-line Interface
CPH	Competent Programmer Hypothesis
EFG	Event-flow Graph
FSMs	Finite State Machines
GUI	Graphical User Interface
GUIs	Graphical User Interfaces
LTS	Labeled Transition System
KPI	Key Performance Indicator
MBT	Model-based Testing
MBMT	Model-based Mutation Testing
SaaS	Software as a Service
SUT	System under testing
TCM	Test Case Mutation
UML	Unified Modeling Language

Chapter 1

Introduction

This dissertation is under the domain of Software Engineering, specifically in Software Validation and Verification. Nowadays, the software can be used as a service and some of these services must be safe and cannot allow errors. Software testing is a very important part of software development since it helps to ensure the quality of it.

In this chapter, it is demonstrated the context of the problem under this dissertation. The goals to be accomplished and the motivation are also described.

1.1 Context

Nowadays, the use of the internet to do tasks which were done physically is growing fast and the software can be used as a service. This type of services, Software as a Service (SaaS), must guarantee their quality in order to prevent fatal or dangerous failures. Software quality is a very important factor in these services. To ensure that the expected behavior of the system corresponds to the actions performed, the software should be tested.

Software testing is one of the most important parts of software development. It is estimated that fifty percent of the development time is spent on testing [[GFM15](#)]. Creating test cases with a high level of coverage may not be a simple task. As these services use Graphical User Interfaces (GUIs) to simplify the usability for the user, a minimal change in these may cause failures in the test cases when they are made through the GUI. This increases the costs and time of maintenance of software testing.

Automated testing can reduce significantly the time, cost and effort of testing. Although it costs more in the beginning, depending on the complexity of the software, it compensates when changes are made and in the future.

There are many techniques to perform software testing. Mutation testing is a testing technique which allows the tester to verify if the quality of the existing test suit is able to detect purposely

injected errors, and, on the other hand, extend the existing test cases. GUI testing is also a challenging problem once GUIs can be complex, which increases effort and time when performing software testing.

Automating these processes may be a good contribution to reduce the time and the effort spent by the testers and developers, increasing the software's quality.

1.2 Motivation and Goals

As the number of SaaS has been increasing as well as the need to ensure quality software when it is dealing with sensitive data, this kind of services must be exhaustively tested. Since it is through the GUIs which users interact with the service, this component becomes a very important focus of the testing task. Currently, it is important to make easy and *user-friendly* GUIs in order to keep the user interested in using the service. To do that, it is necessary to regularly update the GUI and this process may cause failures in the main functionalities of the service. To ensure that the requirements continue to be fulfilled, regression tests should be applied.

The most frequent interaction paths by the user are, probably, the ones which represent the main or the most important functionalities of the software. Save these interactions and replay them later is a way of generating test cases which represent very well real situations. In order to improve these generated test cases, some faults can be introduced and evaluate if the test cases are able to detect them. On the other hand, these changes may result in new test cases and improve the coverage and quality of the test suite.

Automation testing is an important approach once it reduces the cost and time of the test suite's maintenance. Besides that, this technique improves the effectiveness and efficiency of the software. Applying regression testing automatically increases the confidence of the test suite and decreases the effort by the testers.

With this study, it is intended to automatically generate test cases using the most frequent paths gathered by a web analytics tool. From a catalog, the defined mutation operators will be injected automatically in the test cases. Additionally, the generated test cases will be evaluated with a developed metric about their relevance to the test suite.

1.3 Problem

Generating good test cases may be neither a simple nor fast task. When the software suffers changes, the test suite may need some adjustments. Even when a test suite covers all the code, that is, 100% of coverage, it does not mean that the quality of test cases is the best once it may not detect some errors. There are some approaches which are able to assess the quality of the test suite.

There is some research work around web usage data. It is known that this data is very important to understand user behavior on the website. With this information, the developers are able to make changes on the website according to what is retrieved in this data. For example, the number of

clicks, the average of traffic on certain pages, etc may be useful information in order to improve the usability of the website and keep the user interested in.

The usage of this information to generate test cases is not very explored. These test cases are very important once they offer a good consistency of the system and allow preventing and detect new faults in the system. In order to improve them, it can be applied a technique to extend them and increase the coverage reached by the generated test cases and to assess the quality of them. In this research work, it will be explored if mutation testing is a technique which is able to do that. Adding some faults in the code, it is possible to check if the existing test cases are able to detect these faults and, on the other hand, if it is possible to exercise different aspects of the software by adding these mutations. Once there is no oracle, because it is developed in a web environment, the goal of the use of mutation testing is to exercise more aspects and improve the test suite. The main purpose of this research work is to generate test cases based on web usage information and extend them with mutations.

1.4 Solution

Once the use of web analytics allows knowledge about the behavior of the user using the website, it is possible to get the most frequent paths of the user. The interaction between the user and the service is made by a GUI, so a good approach to generate test cases is replaying the most frequent paths since they represent the important functionalities of the service.

With a web analytics tool, web usage data was previously gathered and the most frequent paths were extracted and saved in a database. This information contains the session id, which represents a path of execution and then a test case. Each session contains information about what the user made on the website: it identifies the HTML element by its XPath and the action performed (click, text input, drag and drop). The input data is not recorded for privacy reasons.

A script is developed to get each session of a user and save them as an abstract test case in an appropriate format, such as JSON. Each test case is an individual file which contains the filename to identify the test case. After this step, the test cases are converted into concrete test cases by adding input data, and a structure is created: there an initial step and the result of the test case. Inside of each test case, there are steps which correspond to each individual action performed on an element. It contains the fields which identify the element and the action performed. Once they do not contain input data, this data is generated by an input random data generator. It is also saved the result of the test case.

In order to strengthen the test suite, mutation testing techniques will be applied. From a mutation catalog, which will be defined, some mutations will be applied to each script test. In this case, a mutation will be a transformation in the test case in order to exercise more aspects of the software. Adding or remove some steps, change the order, etc are examples of mutation operators which will be injected.

After that, the test suite will be replayed with a Capture & Replay tool, such as Selenium. A script is developed to read JSON files with the test case information. This script allows replaying the entire test case automatically and the results are saved on each test case.

In the end, the test cases which have different behaviour from the entire test suite are kept. The other ones are removed because they are not relevant. An adaptation of *Levenshtein algorithm* is used to compare the test cases and define what is a relevant test case.

1.5 Contributions

During the development of this dissertation, it was explored the use of mutation testing applied directly in test cases in web context. A tool was developed in order to automate this process by applying mutation operators in the test cases in an automatic way.

It was also possible to write a short-paper “Mutation-based Web Test Case Generation”, which was submitted and accepted in the 12th International Conference on the Quality of Information and Communications Technology, QUATIC 2019, and will be presented between September 10 and 13. This paper presents this approach in an initial phase, once the evaluation with the *Levenshtein algorithm* was not implemented.

1.6 Dissertation Structure

This dissertation, besides this introduction, is organized as follows: Chapter 2 introduces the main concepts of this work as well as the previous research in the main areas under the domain. In Chapter 3 it is explained, in detail and formally, each step of the implementation of this dissertation. The results of the development, applied to different scenarios, are described in Chapter 4. Finally, in Chapter 5, are presented the conclusions which result of the development of this work. The future work related to this project is also included in this chapter.

Chapter 2

Background and State of the Art

In this chapter, it is provided the background behind this dissertation. The concepts, methods, and techniques are also provided as well as related research work. In Section 2.1 it is introduced the concept, levels, phases and an overview of the techniques about Software Testing. In Section 2.2, it is explained how mutation testing works and how can be applied. An overview of Web Testing techniques is stated in Section 2.3. Test case generation techniques are described in Section 2.4. Once this approach uses website usage information, Web Analytics is introduced in Section 2.5. Finally, a summary of this research is provided in Section 2.6.

2.1 Software Testing

Nowadays, software systems are part of our life. They are used from social networks to bank or financial services. It is very important to ensure the quality of these services once they contain sensitive and personal data. Once these systems are developed by humans, they may have some defects which cannot produce the expected behavior of the service. To reduce this kind of situations and improve software quality, the software should be tested. In Subsection 2.1.1 the levels of the software testing are described and in Subsection 2.1.2 explains the main techniques to perform software testing. Regression Testing, which is part of the Software Testing process, is also introduced in Subsection 2.1.3, once this dissertation is under this domain as well as GUI Testing, in Subsection 2.1.4.

2.1.1 Software Testing Levels

During the software development, there are some phases such as requirement's elicitation, software design, implementation (coding), testing and deployment. For each one of these phases, different types of testing should be applied which leads to different test levels [Sil17]. As shown in Figure 2.1, there are some levels in software testing.

Figure 2.1: Levels of Testing. Extracted from [CB02].

Unit testing: Testing of individual components of the software. A unit is the smallest possible testable software component. Normally, it is a function or procedure. This level of testing is performed by the developers.

Integration testing: This phase is to test two or more components when they are assembled and detect defects on their interfaces.

System testing: When integration tests are completed, a system has been assembled. System testing test the software as a whole. This level is performed by a test team and it ensures that the system meets the initial specifications.

Acceptance testing: These tests are performed by the final customer and it allows determining if the customer initial requirements and expectations are met.

2.1.2 Techniques

The main goal when designing test cases is to develop an effective test case [CB02]. An effective test case is a test case which has a greater probability of detecting defects, a more efficient use of organizational resources, a higher probability for test reuse, closer adherence to testing and project schedules and budgets and the possibility for delivery of a higher-quality software product. To ensure that, there are two main strategies that can be used: black-box and white-box techniques.

Black-box techniques are the ones which the tester does not have access to the code. It is focused on the functionality of the system according to the specifications, so the tester only checks if the system does what is supposed to do. The main advantage of black-box testing is that testers do not need to have knowledge of a specific programming language. These techniques help to expose any ambiguities or inconsistencies in the requirements specifications [Nid12].

White-box techniques focus on the inner structure of the software. To use this strategy, the tester must be aware of that structure and the code must be also available. This technique is used for detecting logical errors in the program code and it can be applied at all levels of system development.

These approaches are summarized in Figure 2.2.

Figure 2.2: Test strategies. Extracted from [CB02]

2.1.3 Regression Testing

Regression Testing, according to ISTQB, is, by definition, “Testing of a previously tested component or system following modification to ensure that defects have not been introduced or have been uncovered in unchanged areas of the software, as a result of the changes made”¹. This activity is an important part of software development once it helps to ensure the quality when changes occur. Once it implies re-running the test cases every time a modification is made, it requires a lot of cost and time [KMB]. There are some techniques to execute this activity [DS08]:

1. **Retest all** This technique is composed by re-running all the test suite which is expensive but is the easier and the conventional method.
2. **Test selection** This technique selects part of the test suite if the cost of running this part is less than running all the tests. There are also many approaches to select the test cases shown in [DS08].
3. **Test prioritization** This technique prioritize the test cases which increase a test suite’s rate of fault detection. The many approaches are described also in [DS08].
4. **Hybrid approaches** These are the techniques which combine test selection with test prioritization.

Besides the techniques described above and detailed in [DS08, KMB, SAH], there is some research work in order to automate this process. There are two main impediments in regression test automation: test inputs and oracle. The approach proposed by Gao et al. in [GFM15] uses the GUI state to generate oracles and generate test cases from a formal model called event-flow graph (EFG), all of this automatically. Some other approaches like [BA], [ASH] and [USVH] contribute to perform this activity in an automatic way.

¹ ISTQB Glossary - <https://glossary.istqb.org/en/search/regression>

2.1.4 GUI Testing

GUIs are the most common way of interaction between the user and the system. As expected, GUIs have become very important and significant in the software engineering discipline [QN]. More complex systems probably have complex GUIs and the effort of testing increases.

GUIs are a better interface style based on Command-line Interface (CLI) [Pai06]. This interface supports other kinds of interaction styles like forms, menus and direct manipulation. Thus, a GUI offers a more pleasant environment of interaction between the software and the user.

Besides manual testing, which costs time and effort, there are three main approaches to automate GUI testing: Random testing, Model-Based approach, as previously explained in Subsection 2.2.1, and Capture and Replay.

2.1.4.1 Random testing

Random testing is an easy and cheap technique to test the software. Also known as *Stochastic testing* or *Monkey testing*, this technique is focused on having a “monkey” exercising randomly the software. These “monkeys” have no idea of the state of the software and the goal is to crash the SUT with the random interaction. This approach of GUI testing is able to detect 10% to 20% of bugs, reported by Microsoft [MPNM17]. There are some improvements around this technique, such as Hofer et al. in [HPW], which have implemented smart “monkeys” to detect the behaviour of the system while exercising it.

2.1.4.2 Model-based approach

Model-based testing has been increasing its use in GUI testing since it has achieved good results. This approach is able to generate test cases from a model constructed from the GUI. It allows checking the conformity between the implementation and the model of the SUT, introducing more systematization and automation into the testing process. The test generation phase is based on algorithms that traverse the model and produce tests as desired.

In the field of model-based approach applied in GUIs, there are many proposed techniques to generate test cases. The use of event-flow graphs is a popular approach [MSP01] proposed by A. Memon. It represents the flow of the events of the GUI with nodes and edges. A. Memon also proposed an approach which consolidates different models into one scalable event-flow model and outlines algorithms to semi-automatically reverse-engineer the model from an implementation [Mem07].

Another approach uses Labeled Transition System (LTS) where the transitions correspond to action words [MPNM17]. An LTS contains states and transitions from those states. The main purpose of this approach is to design test cases with action words before the implementation. It can be applied some variations, like mutations, by varying the order of the events, becoming possible to find undetected events.

Finite State Machines (FSMs) are also a very popular approach to model GUIs, as cited before. Miao et al., in [MY10], compare the efficiency of FSM against EFG, and they conclude that FSM

are able to model some situations which EFG can not. For example, GUI objects which are modified dynamically can not be represented by a EFG.

Although the above approaches are the most common, Petri nets can also be used to model a GUI. Reza et al. proposed an approach using Petri nets, specified in high class of Petri nets known as hierarchical predicate transitions nets (HPrTNs) [REG07].

2.1.4.3 Capture and Replay

This technique allows recording user interactions through GUIs and replays them later, as a test case. Capture & Replay captures all the interactions between the user and the application, including mouse movements. It was developed to allow testing web applications and regression testing.

The main advantages of Capture and Replay are that this technique saves time creating test cases once they are recorded instead of writing scripts and it is easy for programmers who have less knowledge in test automation [NB13]. The cost of maintenance of this approach is big since if there are changes in the GUI, the recorded test scripts which are involved in these changes, will be obsolete. Another drawback of this method is that some scripts also need human intervention.

Selenium is the most used tool which allow the use of this technique². It is a very powerful tool developed for automating web testing. Selenium allows using an extension to use in Chrome and Firefox which allow the record of all the interactions and save them as a test case. It can be replayed later and used as regression testing.

Selenium provides also a WebDriver component. It allows the creation of scripts which can reproduce automatically in the browser the test cases by identifying the target objects and the respective actions. It is a powerful tool when the specific target objects and actions are previously defined. Thus, the test cases can be automatically generated from the data and then run.

2.2 Mutation Testing

Mutation testing is a fault-based technique which allows measuring the quality of software tests. This technique can be used both in the evaluation of the actual test suite and in guidance to create new test cases. Mutation testing aims to locate and expose the weakness in test suites, introducing faults which generally represent the mistakes that programmers often make [JH11]. Each fault introduced is called mutant. Once the mutants are deliberately applied in the source code, they are executed against the actual test suite. After the execution of the test suite, they may have three types of mutants: killed mutants - which were the mutants detected by the test case; alive mutants - the ones which were not detected; equivalent mutants - mutants which have the same behaviour as the original program. If the result of a test case is different from the result with the non-mutated program, the test suite was able to detect that mutant. Otherwise, the mutant is alive

²Selenium Documentation - https://www.seleniumhq.org/docs/03_webdriver.jsp#selenium-webdriver-api-commands-and-operations

or it is equivalent. To these equivalent mutants, there are no possible test cases able to kill them, so they will require human intervention to distinguish them.

Figure 2.3: Process of Mutation Testing. Extracted from [JH11].

In Figure 2.3, it is described the process of mutation testing for one mutation operator. Mutation testing provides a testing criterion which is called mutation score and it is the ratio between the killed mutants over the non-equivalent mutants:

$$\text{Mutation score} = \text{Number of killed mutants} / \text{Number of non-equivalent mutants} \quad (2.1)$$

The range of the result of this equation is between 0 and 1. A result of 0 means that none of the mutants were killed. If the value is 1, all the mutants were killed by the test suite, which is the best case.

As the number of potential faults for a given problem is enormous and it is impossible to generate mutants representing all of them, mutation testing focus only on a subset of these faults [JH11], those which are close to the original program, with the hope that these faults will be sufficient to simulate all faults. This theory is based on two hypotheses:

- Competent Programmer Hypothesis (CPH)
- Coupling Effect

The CPH was first introduced by DeMillo et al. in 1978. This hypothesis says that the programmers are competent, so they tend to develop programs close to the correct version. Then, the faults which may exist are a few simple faults. Once mutation testing applies only faults based on small syntactical changes, these should represent the faults made by the competent programmers.

The Coupling Effect was also proposed by Demillo et al. in 1978 and says that the test data or the test case which is able to detect simple type of bugs are good enough to detect complex bugs [Dan16]. According to this hypotheses, mutants with more than one change are called *higher-order mutants*. These mutants are likely to be detected by test cases which detect simple mutants.

There are many approaches using Mutation Testing to assess the quality of the initial test suite. Papadakis and Nicos Malevris used a path selection strategy for selecting test cases able to effectively kill mutants [PM12]. They conclude that this strategy can play an important role when applying mutation testing techniques.

Although this technique is originally applied in the source code, Koroglu and Sen proposed Test Case Mutation (TCM) which mutates existing test cases in order to enrich them [KS18]. This research work introduces mutation directly in test cases instead of in the source code, as in this dissertation. It revealed a good performance in detecting failures in an Android environment. Paiva et al. [PEG19] also proposed an approach which introduce mutations in test cases generated by a mobile testing tool, in this case iMPAcT tool. This approach verifies if the application has the expected behaviour when it goes to background and comes back to foreground after the mutations' injection.

Xuan et al. use this technique, TCM, to try reproducing crashes via test case mutation [XXM15]. In this approach, the goal is to trigger crashes by increasing the chance of executing the specific path via test case mutation. Instead of creating new test cases, this tool leverages the existing ones in order to have a better test suite.

Also in Android context but with the original purpose of this technique, Deng et al. also use mutation testing in Android apps [DOAM17]. They define some mutation operators specific to Android apps and inject them in the source code. The evaluation of this approach is made through an empirical study on real-world apps.

2.2.1 Mutation Testing applied in models

Model-based testing (MBT) is a testing technique which the test cases are derived from a model that describes some (usually functional) aspects of the system under test (SUT) [Hig17]. This software testing technique can be formal and informal. The difference between them is that formal MBT uses formal test models that comply with certain standard modeling rules while informal MBT does not use formal test models. Models are abstract representations of systems [Sil17].

The process of model-based testing is described in Figure 2.4.

Step 1: From the software specifications, a model of SUT is built. Since this model, also known as test model, is an abstract representation of the system, implies that it is simpler and easier to maintain than the actual SUT.

Step 2: Definition of test selection criteria and metrics to guide test case generation so that it produces a good test suite.

Step 3: Test selection criteria are transformed into test specifications which formalize the notion of test selection criteria. A test case specification is a high-level description of a desired test case.

Step 4: Test case generation, satisfying all the test case specifications. Some test case generators reduce the number of test cases once each test case may cover many test case specifications.

Step 5: Test execution by running the test cases generated. This process can be manual or automated.

Figure 2.4: Process of Model-based Testing. Extracted from [UPL12]

Mutation testing can also be applied to models. Belli et al. introduces the concept of model-based mutation testing (MBMT) [BBH⁺16]. Instead of inject mutation operators directly in the source code, it is possible to use mutation testing as a black-box technique. To do this, a set of mutation operators is applied to a model as shown in figure 2.5. This turns MBMT a very powerful and versatile test case generation approach [KST⁺15].

Krenn et al. presented an approach which generate test cases from UML state machines [ABJ⁺15]. The highlights of this research works are the automated fault-based test case generation technique. Mutation operators are employed on the level of the specification to insert faults and generate test cases that will reveal the faults inserted.

Barbosa et al. described an approach which generates test cases from mutated task models [BPC11]. This research uses task models to generate oracles that simulate the behavior of execution of the running system. To counter this, mutations are applied based on a classification of user errors, enabling a broader range of user behaviors to be considered.

As the specifications can be written as a Finite State Machines, mutation testing can be applied into them. Hierons and Merayo describe several ways of mutating a probabilistic finite state machines in [HM07]. They apply each sequence test several times, comparing the result with statistical sampling theory.

Figure 2.5: Process of Model-based mutation Testing. Extracted from [BBH⁺16]

2.3 Web Testing

As the number of web applications has been increasing, it becomes necessary to study these applications and how to test them. A web page may be subject to testing in different aspects, which has been the major challenge when testing these applications. Usually, they are composed by different components implemented in different programming languages, frameworks, and encoding standards.

With the purpose of testing these applications, there are many testing techniques able to do it. In [LDD14], Li et al. provide a survey of this techniques and categorize them into based on graphs and models, scanning and crawling, search-based, mutation testing, *concolic* testing, user session-based, and random testing. Each of them is briefly described and some research works on the area are also included.

Graph and model-based testing basically creates a model of the application, as explained in Subsection 2.2.1. The test cases are derived from this model, according to the coverage criterion (all-path, all-branches, etc). FSMs and probable FSMs are also included in this web testing technique. Ricca and Tonella proposed an approach in [RT05] which use a UML model of Web applications for their higher level representation. It is helpful to define white box testing criteria and semi-automatically generate test cases, achieving a high level of automation. About FSMs, Andrews et al. proposed an approach that builds hierarchies of FSMs that model subsystems of the Web applications and then generates test requirements as subsequences of states in the FSMs [AOA05]. These subsequences are then combined and refined to form complete executable tests. The constraints are used to select a reduced set of inputs with the goal of reducing the state space explosion otherwise inherent in using FSMs. Besides simple FSM, Qian and Miao proposed an approach with probable FSMs in [QM11]. The testing process is based on that different parts of the website have different execution frequency.

Testing web applications using mutation testing technique is the main goal of this dissertation. As detailed in Section 2.2, it is usually applied directly into the source code. Applying

this technique in this way, some changes are made in the front-end part, as well as in the server-side. There is an approach which Praphamontripong and Offutt proposed in [PO10] which injects automatically defined mutation operators applied in the source code with a tool (webMuJava).

Search-based testing focus on covering as many branches as possible in a web application. This is usually made through defined heuristics to ensure that a large number of branches are tested. Alshahwan and Harman introduce three algorithms which significantly enhances the efficiency and effectiveness of traditional search-based techniques with a 30% reduction in test effort [AH11].

Scanning and crawling technique aims to test the security of web applications. In this technique, unsanitized input is injected, which may result in malicious modifications in the database if not detected. As security is a very important part of a web application, perform this technique promotes improving the overall security of the website. A state of the art of the automated black-box web application vulnerability testing is provided in [BBGM10].

Random testing goal is to test a web application with random inputs to check whether the web application works as expected and handle these inputs. Frantzen et al. proposed a tool, called *Jambition* in [FdINHkW08], which is a Java application to choose randomly inputs automatically from a set of inputs. This tool is to test Web Services based on functional specifications. Also in this field, Artzi et al. proposed in 2011 a testing framework called Artemis which consists in modeling the browser and the server, generating input data and a guidance to explore the state space of JavaScript applications [ADJ⁺11].

The aim of *concolic* testing, which means concrete and symbolic testing, is also to cover as many branches as possible. In this technique, random inputs are passed into the application to verify if new additional paths are taken as a result of these inputs. These paths are stored in a queue in the form of constraints which are then symbolically solved by a constraint solver until the desired branch coverage is achieved. Artzi et al. developed a tool about concolic testing, for PHP Web applications [AKD⁺08]. Symbolic execution generates path constraints, and they are stored in a queue and the constraint solver finds a concrete input which satisfies the condition.

In user session-based testing, a list of interactions performed by the user is collected in the form of URLs and name-value pairs of different attributes. Once there are a huge number of sessions resulted from the users' interactions, there are several techniques to choose what sessions are relevant for perform testing. Elbaum et al. proposed an approach in [EKR03] which user session data to generate test cases, and they conclude that they are as effective as the ones generated by white box testing techniques but less expensive. Sampath also analyses user session data and an application of this data to test case generation for web applications in [Sam04].

In the Table 2.1 is provided a brief comparison between the purposes of each technique described above.

Technique	Main purpose
Graph and model-based testing	Create a model of the application to test
Mutation testing	Find out rare and most common errors by changing the lines in the source code
Search-based testing	To test as many branches as possible in an application via the use of heuristics to guide the search
Scanning and crawling	faults in Web applications via injection of unsanitised inputs and invalid SQL injections in user forms, and browsing through a Web application systematically and automatically
Random testing	Detect errors using a combination of random input values and assertions
Concolic testing	To test as many branches as possible by venturing down different branches through the combination of concrete and symbolic execution
User session-based testing	Test the Web application by collecting a list of user sessions and replaying them

Table 2.1: Comparison between web testing techniques

2.4 Test Case Generation

Software testing may be a hard and slow task if it has low degree of automation. Test case generation is a very important part of testing. To generate test cases automatically, it is necessary to understand the requirements and specifications of the software. When the test cases are generated early, the developers can find inconsistencies in the requirements and specifications of the software.

Nowadays, there are some techniques which allow test case generation [KK13]. Besides the techniques explained in the next sections, there are some approaches which were proposed by some researchers.

2.4.1 Test case generation using Search-based algorithms

Search-based software testing uses heuristic search techniques to develop algorithms to generate test cases automatically [VG16]. These algorithms reduce the cost of the testing process while they maximize the acquirement of test goals.

There are two approaches with search-based algorithms: genetic algorithm and combinations of different optimization algorithms, and dynamic execution of symbolic inputs. The Genetic algorithm tries to find the best feasible solution that meets all the constraints. Alsmadi proposed an approach using genetic algorithms to optimize the generation test cases from GUIs [Als10]. Dynamic Symbolic Execution is a technique to manage data structures dynamically. It collects path constraint on input from predicates encountered in branch instructions. A survey of the use of this technique is provided in [CZG⁺13].

2.4.2 Test case generation using MBT

As described in Section 2.2.1, MBT is a black-box technique which describes the formal aspects of the software in a model and allows test case generation.

The specification of the software precisely describes what the system is to do without describing how to do it. Thus, the software test engineer has important information about the software's functionality without having to extract it from unnecessary details. After constructing the model, it can be used as input to a test case generator [Pai06]. There are several criteria which can be used in order to generate test cases because it is needed to know when to stop the generation and how to evaluate the generated test suite. Coverage criteria is a good approach because can solve the described problems.

Maciel et al. proposed an approach in [MPR18] which aligns the tests' specification, in a model-driven way, with the requirements specification. This approach includes model-to-model transformation techniques, such as test cases into test scripts transformations and also execute the test scripts with the Robot test automation framework.

Grilo et al. describe an approach using reverse engineering which diminishes the effort of constructing a model from a GUI [GPF10]. This model needs to be completed by manual exploration and then it can be used as input for automatic test generation and execution.

Paiva et al. developed an extension to Spec Explorer, a tool for model-based testing, which automates software testing through their GUIs based on a formal specification in Spec# [PFTV05]. This extension gathers information about the GUI elements that are the target of the user actions and generate .NET methods to allow Spec Explorer (which requires .NET methods) simulating those actions.

Zhou et al. proposed an approach which uses Markov usage model to generate test cases [ZWH⁺12]. First, software usage is described using a Markov process. Once it uses probabilities, the test case generation is made by generating a random number and use it to choose the next action to perform. They conclude that the use of this approach against random generation is high-efficient and promising.

Another important issue related to the generation of test cases is the input data that also need to be generated. There are some methods which allow the test data generator how it should choose the input values. For example, when the SUT has several parameters and many configuration parameters, there is a test case explosion which is a big problem.

Random method. The data is randomly generated. This is probably the simplest method but it probably does not generate good quality test data as it does not perform well in terms of coverage.

Goal-oriented method. In this method, there are two approaches: chaining approach and assertion-oriented approach. The first one tries to identify the nodes of the control flow graph that are vital to the execution of the goal node. The second one tries to find any path to an assertion that does not hold.

Pathwise method. This approach does not give the generator the choice of selecting between multiple paths but just gives it one specific path for it to work on. It has two inputs: the program

to be tested and the testing criterion (path coverage, statement coverage, etc).

Intelligent method. This method is dependent on a good analysis of the code to guide the search of test data. It essentially uses a test data generator method plus an analysis of the code.

2.4.3 Test case generation using Adaptive Random testing algorithm

This technique is commonly used and the test data is randomly generated, considering the SUT's preconditions. It is a very simple method to implement and can be used as a component while testing the whole system. In Random testing, the number of test cases for SUT is developed in advance by the developer. In Adaptive Random testing, no such requirement of test case development is required.

2.5 Web Analytics

Web Analytics is the science of improving websites to increase their profitability by improving the user's website experience [WK09]. It deals with the methods for measurement, data collection, data analysis and providing related feedback on the internet for understanding behavior of the customer using website [KSK12]. The main objective of Web Analytics is to understand and improve the experience of online users while increasing revenues for online business. The process is shown in figure 2.6.

Figure 2.6: Process of Web Analytics. Extracted from [WK09]

Defining goals. The goals help to understand the most important question that why should website exist? [KSK12]. The defined objectives are very important to recognize the metrics which will evaluate the success of the website.

Defining metrics. To achieve measuring goal it can be used a proper Key Performance Indicator (KPI) that tell whether the website is getting closer to its objectives or not. Web metrics depend on the context of the website. The most common and general Web metrics are [Fag14]:

- **Visits.** In web analytics, a visit is a device which interacts with the website, during a specific time frame. For example, if the same person visits the same website with different devices, it counts as two visits.
- **Unique visitors.** This metric attempts to count the number of individual people, during a specific time frame. Unique visitors are tracked through authentication and cookies.

- **Page views.** The metric page views counts the number of times a given page was used.

Collecting data. Collecting data is very important to the analysis result. The data should be saved on a local or external database for further analysis. There are four main ways of capturing behaviour data from websites:

- **Web Logs.** Every time a visitor requests some information such as a link to another web page, the server registers this request in a log file.
- **JavaScript Tagging.** This technology consists of injecting JavaScript code in every page of a website and when the visitor opens the page, the script is activated and the visitor information is saved in a separated file.
- **Web Beacons.** This technology is used to measure banner impressions and click throughs. It is commonly used in tracking customer behavior across different websites.

There are several web analytics tool which provides this information. The most used tool is Google Analytics, it tracks traffic for more than 50% websites [CF17]. This tool is focused on the evaluation in terms of sales and marketing and provides statistical information about their users like current and historic traffic, user behavior and their properties, sales conversions, etc.

Besides Google Analytics, which is the most common tool, there are several other tools: Mouseflow, SessionCam, UsabilityTools, CrazyEgg, Hotjar, and OWA.

Čegan and Filip proposed an Open Web Analytics Platform called Webalyt which provides advanced functionality for processing real-time data which is needed for recommendation systems [CF17]. The platform is open source and has a scanning robot to search information about users of third-party applications and marketplace for data interchange between trading partners.

Čegan and Filip also implemented a new solution to capture mouse movements of web users, to identify their area of interest [CF18]. This solution is based on real-time data transformation, which converts discrete position data with high sample period to predefined functions.

Garcia and Paiva presented a tool named REQAnalytics, a recommendation system that collects web usage information about a SaaS. The tool analyses the data and generates recommendations in a more readable format than the web analytics tools' report [GP18]. In [GP16], it is shown an example how to use web usage data to change the requirements' priority, identify new requirements and functionalities that may be removed.

Silva et al. proposed an extension of REQAnalytics [SPGR18]. The web usage data is gathered by OWA, a web analytics tool. The main goal of the developed extension is to generate test cases from this data and diminish the effort in regression testing as in this dissertation.

2.6 Summary

This research about the state of the art addresses several themes under the Software Testing domain. An introduction about Software Testing, including its levels and types of techniques, is

provided as well as an overview of GUI Testing and Regression Testing. As this dissertation concerns in Mutation Testing, a section is dedicated to detail this technique and once it is directed to test cases in a web context, a section about the techniques to perform Web Testing is also provided. An overview of the test case generation techniques is also provided once this dissertation is under this topic. Finally, it is presented the concept of Web Analytics and some research works in this field, once the generated test cases of this dissertation derive from web usage information.

Regarding these topics, this dissertation combines these areas in order to use the web usage information to generate test cases once they represent the most common interactions of the user with the service. To improve these existing test cases, it is intended to add mutations to them and compare the results with the previous ones. Perform this task automatically is a good contribution once it can help to reduce time and effort on performing web testing and ensure the quality of the service.

Background and State of the Art

Chapter 3

Test Case Mutation

This chapter will be focused on the detailed description of the solution explained in Chapter 1. The mutation operators will be also described as well as the tool's functionalities.

3.1 Introduction

In this dissertation, it was developed a tool with the purpose of injecting automatically mutation operators into test cases. The tool has different functionalities, and they will be explained in the next sections.

As detailed in Figure 3.1, the tool is able to extract abstract test cases from a Neo4J database. In order to have test cases with a structure and some input data, the tool converts the gathered abstract test cases into concrete test cases. In this step, the abstract test cases are injected with input data and are executed to have a basis to compare with the subsequently generated test cases. After this, the tool injects automatically mutation operators, from a catalog, and generate different test cases. These are evaluated with a metric to decide if they are relevant to the test suite.

To develop this tool, it was used Java with the help of Selenium framework. This one provides a set of functions which allow find the elements and reproduce the desired actions.

3.2 Abstract Test Cases Extraction

The tool is designed to receive abstract test cases (without input data) from a graph database Neo4j. The gathered information in this database is extracted by each session and saved separately in different JSON files. Each file corresponds to one abstract test case.

The information provided by this database is a set of grouped nodes, which each node contains the properties represented by the Listing 3.1.

Test Case Mutation

Figure 3.1: Diagram of implementation.

```

1 "properties": {
2   "path": "id(\"search\")/input[@class=\"text\"]",
3   "session": "dfdf5a5d-98a4-d90d-334d-094fb7180d80",
4   "actionId": 3,
5   "action": "input",
6   "pathId": 5,
7   "elementPos": 2,
8   "value": [
9     "char",
10    "char",
11    "char",
12    "Enter"
13  ],
14   "url": "http://www.ipvc.pt/"
15 }

```

Listing 3.1: JSON structure of a node's properties.

From the information of each session with the ordered nodes, the tool creates an abstract test case. Each abstract test case is a sequence of ordered steps which each one contains the following information:

- **Path** This field is a string which contains the XPath of the HTML element. It is used to identify the element by the Selenium framework.
- **Session** It is a unique identifier with 32 random characters. It is used to group the interactions.
- **Action** Represents the action performed by the user.
- **Element Position** It is a number used to order the steps to make sure that will be reproduced in the same order as the user did.
- **Value** This field is an array of strings containing the type of each key pressed (char, space, backspace, etc). If the action performed is a Drag and Drop, this field represents the XPath of the HTML element where the element referred in the field Path will be dropped. This field is not filled when the action is Click.
- **URL** Represents the URL where the action was performed.
- **Mutated** This field is a boolean which indicates if it is a mutated step or not. By default, it is false.

An example of an abstract test case is shown in Listing 3.2. It is represented by a JSON Array, whose JSON Objects are the ordered steps.

Test Case Mutation

```
1  [
2    {
3      "path": "/html[1]/body[1]/div[@class=\"thisIsADiv\"]/form[1]/input[1]",
4      "session": "37ea86be-delf-29a7-792e-22ba2a1090ff",
5      "action": "click",
6      "elementPos": 1,
7      "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
8    },
9    {
10     "path": "/html[1]/body[1]/div[@class=\"thisIsADiv\"]/form[1]/input[1]",
11     "session": "37ea86be-delf-29a7-792e-22ba2a1090ff",
12     "action": "input",
13     "elementPos": 2,
14     "value": [
15       "char",
16       "char",
17       "char",
18       "char",
19       "char",
20       "char"
21     ],
22     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
23   },
24   {
25     "path": "/html[1]/body[1]/div[@class=\"thisIsADiv\"]/form[1]/input[2]",
26     "session": "37ea86be-delf-29a7-792e-22ba2a1090ff",
27     "action": "click",
28     "elementPos": 3,
29     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
30   }
31 ]
```

Listing 3.2: Example of abstract test case.

3.3 Conversion into Concrete Test Cases

Once it is needed to have a basis to compare each test case and there is no oracle, the abstract ones need to be converted into concrete test cases by adding some input data and having a result. In this step, the abstract test cases change their structure. Besides the insertion of input data into input fields, the test is run with this data and the result is saved. The result is composed by the result of the Selenium framework, if it fails or not and the URL where the test stopped. A test fails if the Selenium throws an exception.

In this process, it is generated one concrete test by each abstract test case. In case of having more than one field of type *password*, it is generated a different test with the same and different input data in order to generate tests which pass and fail. In Listings 3.3, 3.4, and 3.5, it is shown the result of the new concrete tests cases derived from the abstract one: one with the same data in the *password* fields and another one with different data. The structure of the test also includes the initial step which is the first element of the array of steps.

```

1  {
2      "path": "id(\"password\")",
3      "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
4      "actionId": 3,
5      "action": "input",
6      "pathId": 8,
7      "elementPos": 8,
8      "value": [
9          "password"
10     ],
11     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
12 },
13 {
14     "path": "id(\"confirm\")",
15     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
16     "actionId": 3,
17     "action": "input",
18     "pathId": 10,
19     "elementPos": 10,
20     "value": [
21         "password"
22     ],
23     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
24 },

```

Listing 3.3: Example of an abstract test case with *password* fields.

Test Case Mutation

```
1 {
2   "path": "id(\"password\")",
3   "session": "f48453c0-0eaa-bec9
      -989b-ac10e9493c9f",
4   "action": "input",
5   "elementPos": 8.0,
6   "value": [
7     "Password123!"
8   ],
9   "url": "file:///C:/xampp/htdocs/
      diss/extractApp/index.html",
10  "mutated": false,
11  "inputType": "password"
12 },
13 {
14   "path": "id(\" confirm\")",
15   "session": "f48453c0-0eaa-bec9
      -989b-ac10e9493c9f",
16   "action": "input",
17   "elementPos": 10.0,
18   "value": [
19     "Password123!"
20   ],
21   "url": "file:///C:/xampp/htdocs/
      diss/extractApp/index.html",
22   "mutated": false,
23   "inputType": "password"
24 },
```

Listing 3.4: Example of two steps of a concrete test case with equal passwords.

```
1 {
2   "path": "id(\"password\")",
3   "session": "f48453c0-0eaa-bec9
      -989b-ac10e9493c9f",
4   "action": "input",
5   "elementPos": 8.0,
6   "value": [
7     "Password123!"
8   ],
9   "url": "file:///C:/xampp/htdocs/
      diss/extractApp/index.html",
10  "mutated": false,
11  "inputType": "password"
12 },
13 {
14   "path": "id(\" confirm\")",
15   "session": "f48453c0-0eaa-bec9
      -989b-ac10e9493c9f",
16   "action": "input",
17   "elementPos": 10.0,
18   "value": [
19     "soothe"
20   ],
21   "url": "file:///C:/xampp/htdocs/
      diss/extractApp/index.html",
22   "mutated": false,
23   "inputType": "password"
24 },
```

Listing 3.5: Example of two steps of a concrete test case with different passwords.

In order to do the process described above, it was developed a script in Java with the Selenium framework allowing reproduce the test cases. Through the console interface, the user just needs to introduce the directory or file path of the abstract test case(s). The user may also choose between two browsers to reproduce the script: *Google Chrome* and *Mozilla Firefox*. With the information described in Section 3.2 the script is able to reproduce each step. There are 5 types of actions which the script is able to simulate:

1. **Click** This action simulates a simple click on the element.
2. **Input** This action checks the input type and use it to write in *Value* attribute a value returned by the data generator and insert it in the input element with the XPath indicated in the *Path* attribute.
3. **Drag and Drop** This action picks the element with the XPath indicated in the *Path* attribute and drags it into the element with the XPath indicated in the *Value* attribute.

4. **Back** This action navigates to the previous page.

5. **Forward** This action navigates to the next page.

The action Drag and Drop is performed by a helper JavaScript file once the Selenium framework does not support this action in HTML5.

After the execution of all the steps, the test is created with the following structure:

$$T = \langle N, I, S[], R \rangle$$

where N is the filename of the test saved on the process described in Section 3.2, I the initial step, S[] is the array of steps, and R the result. R has the following structure:

$$R = \langle SR, E, U \rangle$$

where SR is the Selenium Result, which is an exception message or a message of success, E is a boolean showing if occurred an error or not and U is the URL captured just before the window of the browser is closed.

After the execution of this process, a different JSON file is created for each concrete test case. It is also possible to change it manually to refine the input data.

3.3.1 Data Generator

Once there is no concrete data input for privacy reasons, it is necessary to generate data. The only information provided is the number of characters inserted by the size of the *Value* field and it is not enough to provide accurate data. Although the abstract test cases do not contain the type of the input field, it is saved when they are converted into concrete tests, once they are being run among this process. With this information, it is possible, at least, to have data of the type of the input field.

In order to perform this generation, in the abstract test cases, the *Value* field may be manually changed into a *tag* which corresponds to the type of generator. Then, if the *tag* does not correspond to any generator, when converting into concrete test cases, the generated input data will be of the type specified by the type of the input field, i.e, text, number, email, etc. For example, when forms have *password* fields, like in Figure 3.2, there is a generator which returns the same password in order to be possible to have the same value in both fields.

Figure 3.2: Example of password verification.

3.4 Mutations' Injection

The main purpose of this dissertation is to inject mutations directly in the test cases in order to increase the number of exercised aspects and the number of test cases of the test suite. To do that, mutation operators which make sense in web context were created to make some changes in the test cases.

The inputs of this process are the generated concrete test cases by the process described in the previous section. Thus, the mutation operators are applied in these test cases.

3.4.1 Mutation Operators

In the next subsections, the mutation operators will be formally detailed and explained. In order to better understand how these mutation operators are applied and how the transformation in the test cases are made, the tests cases will be formally defined. A test case can be expressed by a tuple, as referred in Section 3.3:

$$T = \langle N, I, S[], R \rangle$$

Once the mutations are injected into the steps, it is better to detail it. The array of steps is defined as:

$$S[] = \{s_0, s_1, s_2, \dots, s_n\}$$

where n is the length of $S[]$, i.e., the number of steps which the test case contains and s_i is the step in the i position.

Each new mutated test case corresponds to the application of only one mutation operator. There is none generated test case which derives from another mutated test case.

These operators are applied to steps which have *Input* or *Click* actions. In each of them, it is detailed which of them are applied to each action. For the explanation of each operator, the basis of each test case is the one detailed above, which the steps ordered from 1 to n , without any change. The representation of the array of steps, $S[]$, is also provided after the injection of the mutation operator.

3.4.1.1 Add Step Mutation Operator

The main purpose of this operator is to check if the verification of some fields is done correctly. For example, on a registration form, there is usually one field to insert the password and another field to confirm the password. If the password check is made only on the second step, another input in the first field will not be validated and the program will not be able to catch this error. To detect this problem, it is possible to inject a mutation by adding an input value in the first field, after the second one. As can be seen in Figure 3.3, the password matches even after the insert of some text in the *password* field. The Add Step mutation operator receives two input fields as parameter. It adds another input step in the first input field after the execution of the two inputs.

$$S[] = \{s_0, s_1, s_{1,1}, \dots, s_n\}$$

Test Case Mutation

Password

.....

Repeat Password

.....

matching

Figure 3.3: Example of poorly implemented password verification.

The step $s_{1,1}$ is executed after the step s_1 , by adding an input value in the step s_0 .

3.4.1.2 Add Back Mutation Operator

This mutation operator has a goal to check if buttons are still working or if sensitive data is not available after a page change. After a click, the browser's 'back' button is pressed to navigate to the previous page. After this, the click is performed again and a check is done to see if we achieved the same result. It also checks if the *password* fields are not filled: if the form contains passwords and, after a click, the form is displayed again, the password should not be there. In Figure 3.4 is possible to see the form before clicking in 'Sign Up' button. On the right side, it is shown the form after performing the Back action and the *password* fields are empty. The Add Back mutation is the operator which applies a back after a click and tries to click the same element.

Sign Up
Please fill in this form to create an account.

Your Full Name
Sérgio

Address
Almeida

Email
sergioalmeida@email.com

Password
.....

Repeat Password
.....

matching
☒ Remember me

By creating an account you agree to our [Terms & Privacy](#).

Cancel Sign up

Sign Up
Please fill in this form to create an account.

Your Full Name
Sérgio

Address
Almeida

Email
sergioalmeida@email.com

Password
Enter Password

Repeat Password
Repeat Password

☒ Remember me

By creating an account you agree to our [Terms & Privacy](#).

Cancel Sign up

Figure 3.4: Example of the state of the form before clicking on 'Sign Up' and perform a Back action (left) and after a Back action (right).

$$S[] = \{s_0, s_1, s_{1,1}, s_{1,2}, \dots, s_n\}$$

The step $s_{1,1}$ corresponds to the *Back* action and the step $s_{1,2}$ is equal to the step s_1 , simulating the *Click* in the same element as s_1 .

3.4.1.3 Change Input Data Operator

The tool is also capable of checking the behaviour of the system with different input data. The tool is prepared to add data generators as described in Subsection 3.3.1. For example, if it is pretended to test the application with different data, it is possible to define the data generators, as explained before, with name generators, address generators, etc, that can be used to test the software with different data. It is only necessary to define the appropriate *tag* for each type of the required data.

This operator changes the *Value* field of the step by calling a random generator. This may be useful to test situations which an input field should only accept data from a specific type.

Since the change is made internally in step, there is no structural change in the array. The variation is only made in the *Value* of the selected step (which action must be of type *Input*).

$$S[] = \{s_0, s_1, s_2, \dots, s_n\}$$

3.4.1.4 Change Order Mutation Operator

Another way to detect the problem of password verification is to swap the execution order of these steps. This operator is important if there are *dropdowns* which depend on each other. Change Order mutation is the operator which changes the order of two actions.

Figure 3.5 shows an example when the *Repeat Password* field is filled first and then the *Password* field. It can be seen that the validation is not matching.

Figure 3.5: Example of Change Order Mutation Operator writing firstly the second field (left) and then the first field (right).

The array of the steps will have the following structure:

$$S[] = \{s_0, s_2, s_1, \dots, s_n\}$$

3.4.1.5 Remove Step Mutation Operator

This operator was created in order to detect problems if an action is removed. The purpose is exercise situations when the user skips interaction with mandatory fields and it is expected that the generated test case has different behaviour.

Figure 3.6 represents a mandatory field and the error which is shown in the case of this field is not filled.

Figure 3.6: Example of a mandatory field.

Considering the initial array of steps $S[]$, if s_1 is removed, the new test case will have the following array of steps:

$$S[] = \{s_0, s_2, \dots, s_n\}$$

3.4.2 Pre-processing

The process of injecting the mutations in the test cases must have a pre-processing part. Since this is an automatic process, it is important to decide where the mutation operators will be applied. Each mutation operator, by its type, has a different pre-processing.

It is important to avoid injecting mutation operators in actions which are performed at different parts of the test case. To do this, it is relevant to capture the actions which occur in a sequence. For each mutation operator will be explained the process considering the following array of steps $S[]$:

$$S[] = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}\}$$

First of all, depending on the mutation operator, the indexes of the *Input* which occur in a sequence or the *Click* actions are stored. This is made by saving the *Input* actions which occur immediately after another *Input* action or with a different action between them. Then, if needed, it is performed the process described below, depending on the operator.

Considering that $s_0, s_1, s_2, s_7, s_8, s_{10}$ are *Input* actions and that s_4, s_5, s_6 are *Click* actions this is an example of the result of the pre-processing described above:

$$inputIndexes[] = \{\{0, 1, 2\}, \{7, 8, 10\}\}$$

$$clickIndexes[] = \{4, 5, 6\}$$

3.4.2.1 Add Step Mutation Operator

As explained above, this mutation operator is only applied in *Input* actions. To this mutation operator, all the possibilities are calculated. For each group of *Input* actions resulted from the initial processing, the combination each two steps are calculated. For example, in the group $\{s_0, s_1, s_2\}$, the possibilities to inject this mutation operator are:

$$possibilities[] = \{\{0, 1\}, \{0, 2\}, \{1, 2\}\}$$

3.4.2.2 Add Back Mutation Operator

This mutation operator is only applied to *Click* actions. As explained in the definition of this operator, it only adds a *Back* action after a *Click* action. It does not require more pre-processing.

3.4.2.3 Change Input Data Mutation Operator

As the name describes this operator, it is only applied in *Input* actions. This mutation operator requires calculating all the possible combinations between the number of available inputs returned by the pre-processing. The array of possibilities, in this operator, is the set of all the possibilities from sets of size 1 until the number of available inputs. Considering $\{s_0, s_1, s_2\}$, the array of possibilities will be:

$$possibilities[] = \{\{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$$

3.4.2.4 Change Order Mutation Operator

This operator is applied in both types of actions. Starting with *Input* actions, this mutation suffers the same pre-processing as the Add Step operator. It only changes the order between actions occurred in a row, but it can change all of them inside each group. The array of possibilities is also equal:

$$possibilities[] = \{\{0, 1\}, \{0, 2\}, \{1, 2\}\}$$

In *Click* actions, it is performed a simple process: from the available clicks, the possibilities are only clicks which occur strictly in a sequence. Thus, the array of possibilities contains sets of size 2. Considering $\{s_4, s_5, s_6\}$, the array of possibilities is:

$$possibilities[] = \{\{4, 5\}, \{5, 6\}\}$$

3.4.2.5 Remove Step Mutation Operator

This operator is also applied in both actions. There is no pre-processing for this mutation operator once it only removes a step from the array and it does not need a special treatment for its nature. So, the possibilities are:

$$possibilities(input) = \{\{0, 1, 2\}, \{7, 8, 10\}\}$$

$$possibilities(click) = \{4, 5, 6\}$$

3.4.2.6 Comparison of the operators in the process

In Table 3.1, it is provided a brief summary of the differences between the operators in the process of injecting mutations. For each developed operator, the actions where they apply, the existence of pre-processing and the array structure are compared.

	Input action	Click action	Pre-processing input	Pre-processing click	Change array structure
Add Back		X	N/A		X
Add Step	X		X	N/A	X
Change Input Data	X		X	N/A	
Change Order	X	X	X	X	X
Remove Step	X	X			X

Table 3.1: Comparison between operators in injection process

All of these mutation operators derive from a general class Mutation. Each of them receives, as parameter, the position of the step where will occur the transformation. For the cases where there are changes in more than one step, the respective operator receives different additional parameters, as explained below:

- **Add Step Mutation Operator** This operator receives the position of the step where will be introduced the new value, after the execution of the step defined.
- **Change Input Data Mutation Operator** This operator receives an array of positions where the changes will occur. This operator ignores the common parameter since this operator can be applied in more than one step.
- **Change Order Step Mutation Operator** This operator receives the position of the step which will be changed with the defined step.

3.4.3 Injection process

In order to apply these mutation operators automatically, the tool, given a file or directory of concrete test cases, allows the user to apply which of these operators wants to apply and the desired number of them. The number of mutations is per type of mutation, and, in case of the operators are applied in both actions, is also per action. For example, if the desired number is 1 and the selected operator is Change Order, 2 new test cases will be generated per test case inside the directory path.

Test Case Mutation

```
1 Mutation Remove Step successfully injected on test testSession-f48453c0-0eaa-bec9
  -989b-ac10e9493c9f.json on step 5!;
2   New filename: 337100e8-266d-4430-abf3-e241277d0b03
3 Mutation Change Order successfully injected on test testSession-f48453c0-0eaa-bec9
  -989b-ac10e9493c9f.json on step 2!;
4   New filename: e38b301f-cd47-47ce-ae32-52683540baa7
5 Mutation Add Step successfully injected on test testSession-f48453c0-0eaa-bec9-989b
  -ac10e9493c9f.json on step 3!;
6   New filename: f2a66240-b23b-4539-a284-c0e2b1137e8a
7 Mutation Change Input Data successfully injected on test testSession-f48453c0-0eaa-
  bec9-989b-ac10e9493c9f.json on step 1 3 9 !;
8   New filename: 8a9c13c7-d6b3-44e7-9bb0-0b419575578c
9 Mutation Add Back successfully injected on test testSession-f48453c0-0eaa-bec9-989b
  -ac10e9493c9f.json on step 6!;
10  New filename: 8e4a5f0e-afad-48e4-81a5-085113ca3be7
```

Listing 3.6: Example of a mutation process's report.

After the selection of the operators, the tool uses the number desired or, if it is larger than the size of the array of possibilities, uses this number to create the mutated test cases. For each one, one mutation is applied. There are no overlapping of mutations.

To choose where the mutations will be applied, the tool chooses randomly one of the possibilities returned by the pre-processing and inject the mutation there. When a step is derived from a mutation, the *Mutated* field becomes true. After the injection, the possibility is erased from the array. The new test case is stored in a different JSON file.

For each injection process, a report is created to easily understand if there was any problem with the process or identify which mutated file corresponds to each mutation operator. It is a text file containing, for each mutation operator, its type, the original filename, the step where occurred the mutation and the new filename. An example of this report is shown in Listing 3.6.

3.5 Evaluation of the test cases generated

From the generated test cases, it is necessary to evaluate the tests which are relevant to include in the original test suite. It is important to decide which of them are relevant or not. Once this is an automatic approach, it is not very easy without human intervention to decide if each test case is important to keep in the test suite.

In this approach, it is used the initial step, the array of steps, and the result of the test to compare two test cases. Each of these structures is compared one by one.

To compare the initial step between two test cases, the following attributes must be equal in a step:

- **Path** To check if the action is performed in the same element.
- **Action** To check if the action is the same.

- **Value** If the value is different, then the step is different.
- **URL** To make sure that the action is performed in same URL.
- **Element Position** If the order is different, then the step is different.

The result of the test case is also compared by its attributes and all of them must be equal:

- **Selenium Result** To check if Selenium returns the same result.
- **Error** To check if there was an error.
- **URL** To check if the test ended in the same URL.

The array of steps is compared with an adaptation of *Levenshtein distance* algorithm. The *Levenshtein distance* is a string metric which measures the difference between two strings. It represents the minimum number of single-character edits (insertions, deletions or substitutions) to transform one word into the other ¹.

In this context, it is used as an adaptation of this algorithm by comparing the steps instead of strings. It compares step by step, by their attributes as in the initial step, and count the minimum number of differences between the two array of steps. This number is used, as a parameter, to decide if the new test case will be stored in the test suite or not as well as the other conditions.

Defining the new mutated test case as TM , it will be compared with each one of the existing test cases in the actual test suite and, for each one, defined as T_i , will be tested against three conditions:

$$\begin{aligned}
 p &= TM.initialStep == T_i.initialStep \\
 q &= levenshteinDistance(TM.steps, T_i.steps) \leq desiredDistance \\
 r &= TM.result == T_i.result
 \end{aligned}$$

The Table 3.2 shows, for *Levenshtein Distance* equals 1, when a test case is inserted in the test suite and the Table 3.3 for *Levenshtein Distance* higher than 1.

p	q	r	$p \wedge q \wedge r$	Insert
T	T	T	T	F
T	T	F	F	T
T	F	T	F	T
T	F	F	F	T
F	T	T	F	Impossible
F	T	F	F	Impossible
F	F	F	F	T

Table 3.2: *Levenshtein Distance* equals to 1

p	q	r	$p \wedge q \wedge r$	Insert
T	T	T	T	F
T	T	F	F	T
T	F	T	F	T
T	F	F	F	T
F	T	T	F	T
F	T	F	F	T
F	F	F	F	T

Table 3.3: *Levenshtein Distance* higher than 1

¹The Levenshtein Algorithm - <https://www.cuelogic.com/blog/the-levenshtein-algorithm>

If exists a T_i which holds these 3 conditions for T_m , T_m is not inserted in the test suite once it is considered “similar” through this metric. The main purpose of this evaluation is to identify which similar tests have different behaviour from the original ones. Two different test cases are considered similar if they start and finish at the same point and the steps are “more or less” alike.

Comparing an array of steps, the Levenshtein Distance is able to count the removes, changes or additions of any step. Thus, this is used to evaluate the path traveled by the test case. Depending on the objective of the testing activity, this parameter may be changed in order to consider a test “more similar” or not.

3.6 Execution

Besides the functionalities stated before, the tool is also able to simply execute a simple or a set of test cases. The process is similar to the one explained in Section 3.3. The input is also a JSON file of a concrete test case. The main difference of the process is that, in this process, the tool reproduces exactly what is indicated by the information contained in the test case, without changes.

As each test case is stored in a file, they can be changed manually. As this is an automatic process and the data was randomly generated, there may be a need for refining the input values manually. This is possible once each test case is a different JSON file.

A report is also created when executing test cases. It is shown if the test passes or fails and the number of these cases. An example is shown in Listing 3.7.

```

1 Test testSession-f48453c0-0eaa-bec9-989b-ac10e9493c9f.json was successfully
  executed.
2
3
4 Number of passed 1
5 Number of failed 0

```

Listing 3.7: Example of an execution process’s report.

3.7 Summary

The developed tool assemble some functionalities detailed in the previous sections. The main purpose of this tool is to help reproduce and improve a test suite based on captured interactions between the user and the service. The functionalities are briefly described above:

- **Get abstract test cases** Starting from gathering test cases, the tool is able to get abstract test cases from a graph Neo4J database and save them into different JSON files.
- **Conversion into concrete test cases** From abstract test cases, the tool adds input data from a generator which can be modified to generate data according to the user preferences.

Test Case Mutation

- **Execution** The simple execution of a test case is possible and it saves a report with the result.
- **Injection of mutations** The focus of this tool is to apply automatically mutation on the test cases in order to improve the existing test suite.
- **Evaluation** Finally, the tool also provides a metric based on *Levenshtein algorithm* to evaluate the mutated test cases, in order to check if they are relevant or not to the test suite.

Test Case Mutation

Chapter 4

Experiments

In this chapter will be shown the application of this tool in two contexts. The first application is a simple website site developed to test if the tool is able to inject the mutation with the defined problems. The second experiment is applied to test cases gathered from a database which represent the most frequent paths of execution of a real website. These two scenarios were used to verify if the tool was able to generate mutation from the previous test cases and if they are relevant, through the defined metric.

4.1 Simple Website

In order to detect the problems and test the developed mutation operators, it was built a simple website. It consists of a registration form with some input fields to be filled with personal information.

In the Figure 4.1, it is shown the registration form. It is composed by 5 input fields: Name, Address, Email, Password e Repeat Password. The form has some validation features:

- All of the fields are mandatory.
- The verification if the two passwords are equal is only made in the Repeat Password field whenever a key is pressed.
- The ‘Sign Up’ button only works if the passwords are equal and all the fields are filled. It redirects the first page into the second one.

Figure 4.2 represents the second page to where the ‘Sign Up’ button redirects when the form is correctly filled. It is also a simple website just to have a second link.

In order to validate the tool’s functionalities, a test case was written simulating form filling correctly. The test case was recorded with random information but with the same password in the fields Password and Repeat Password. The following steps were recorded:

Sign Up

Please fill in this form to create an account.

Your Full Name

Enter your full name

Address

Enter your address

Email

Enter Email

Password

Enter Password

Repeat Password

Repeat Password

☒ Remember me

By creating an account you agree to our [Terms & Privacy](#).

Cancel

Sign up

Figure 4.1: First page of the simple website.

Figure 4.2: Second page of the simple website.

1. Click on input name field
2. Write the name
3. Click on input address field
4. Write the address
5. Click on input email field
6. Write the email
7. Click on input password field
8. Write the password
9. Click on input repeat password field
10. Write again the password
11. Click on 'Sign Up' button
12. Click on Link (page 2)
13. Click on Link (page 2)

This test case is available in Appendix A.1. As this abstract test case contains password fields, two concrete test cases were created: one with equal input data on password fields and another with different data. They are also available in Appendix A.2.

From these two generated concrete test cases, the mutation operators were applied. In this experiment, all of the mutation operators were applied with the maximum possible number. The tool was able to generate 160 mutated test cases. In order to evaluate them, it was used the method explained in Section 3.5.

Observing the Table 4.1 and the Figure 4.3 it is possible to see the number of relevant test cases depending on *Levenshtein Distance*. When the *Levenshtein Distance* is equal to 1, 42 of the 160 mutated tests were considered irrelevant. This value corresponds to the test cases which are considered equal: same initial step, *Levenshtein Distance* is 0, and the same result. It is also

Experiments

<i>Levenshtein Distance</i>	Number of mutated test cases	Percentage of relevance
1	118	73.8%
2	103	64.4%
3	99	61.9%
4	90	56.3%
5	28	17.5%
6	20	12.5%
7	20	12.5%

Table 4.1: Evaluation of the array of steps with *Levenshtein Distance* of Experiment 1

Figure 4.3: Relationship between the *Levenshtein Distance* and the mutated test cases of Experiment 1

possible to see that the larger difference is between the values 4 and 5, which already are tests with a considerable difference from the original test suite.

As the *Levenshtein Distance* increases, the differences between the mutated test cases and each test of the initial test suite are higher. When the distance is 6 and higher, the number of test cases stabilizes in 20. It means that the initial step or the result of the test case is different regardless of this distance.

It is important to do not include the test cases which are strictly equal from the ones which are in the initial test suite. The base point is the *Levenshtein Distance* equals to 1 because it rejects these test cases. However, it may be infeasible because it is adding a higher number of test cases with a “little difference”.

With this experience, it was possible to generate automatically from a single test case, 113 test cases with different behaviour from the initial one. This means that the tool was able to generate 73.8% of relevant test cases. Taking into account efficiency and time variables, the tool allows adjusting this distance to balance these parameters.

4.2 Real Website

In order to validate this approach in a real scenario, it was applied to captured test cases from a real website using a web analytics tool. The approach has been applied on the website of the *Polytechnic Institute of Viana do Castelo* (<http://www.ipvc.pt/>). The user's interaction across the public parts of the website have been recorded and stored into a graph database Neo4J.

The set of records corresponds to 18124 interactions captured during 11 days. These interactions are grouped by session resulting in 4899 sessions. A session corresponds to an abstract test case which will be got by the tool.

As the number of recorded sessions is 4899, it is infeasible to test all the possibilities. Once the goal is to the most frequent paths, it is possible to get the sessions ordered by frequency. However, there is a high number of sessions which are composed by just 1 single action, e.g. just a click on the homepage or a click into a page outside the range of the collection script. In the Figure 4.4, it is possible to see a graph with the number of test cases with a greater number of interactions and see that there are a lot of sessions with just a few actions.

Figure 4.4: Relationship between the number of interactions and the number of stored sessions

To this experience, the minimum length chosen was 10 in order to have more reliable test cases and avoid the ones which are just simple interactions. So, the number of gathered test cases was 321. Once each test case has, at least, length equal to 11, were used the first 10 most frequent ones for efficiency reasons.

The 10 abstract test cases result in 10 concrete test cases. The number is equal because on the website of the *Polytechnic Institute of Viana do Castelo* there is no password fields and just one input field, which corresponds to a search box. From this point, the mutation operators which are applied only in input fields are applied in a fewer number.

From these 10 concrete test cases were generated 431 mutated test cases. This is the result of applying all the mutation operators. To this experiment, the configuration was equal to the

Experiments

previous experience: all the mutation operators were applied and the maximum number was also injected.

<i>Levenshtein Distance</i>	Number of mutated test cases	Percentage of relevance
1	285	66.1%
2	28	6.5%
3	28	6.5%

Table 4.2: Evaluation of the array of steps with *Levenshtein Distance* of Experiment 2

Figure 4.5: Relationship between the *Levenshtein Distance* and the mutated test cases of Experiment 2

From the 431 mutated test cases, it is possible to see through the Table 4.2 and Figure 4.5 that 146 are irrelevant to the test suite. When the *Levenshtein Distance* is 2 or 3, the number of relevant test cases decreases significantly and there is no difference between them. It may be due to the possibility of the most frequent sessions are different from each other and the generated mutated test cases are similar to those already in the test suite because each mutation operator injects at most 2 differences. Thus, the tool is able to generate 66.1% of relevant mutated test cases taking into account the *Levenshtein Distance* equal to 1.

Besides these numbers explained above, when based on 10 test cases, the tool was able, to generate at least 28 test cases. It almost increases the size of the test suite in three times, which means 280%.

Depending on the goal of the test activity, the tool is able to adapt the size of the test suite to fit the efficiency in terms of time by adapting the *Levenshtein Distance* to define what is a relevant test case. If the goal is to exhaustively test the software, it is possible to have a big increase in the number of test cases by setting it to 1. Otherwise, it is possible to adjust it to reduce the number the test cases and get a test suite with a balance of performance in execution and exercised aspects.

4.2.1 Mutations' generation

However the evaluation process of the generated test cases is a very important part because it helps to reduce the time of execution of the test suite, it was performed an evaluation of how many mutated test cases the tool is able to generate based on the size of the existing test suite of real test cases.

To evaluate how many mutated tests based on the number of original concrete test cases, an experience was made with 80 of the most frequent test cases with a length greater than 10. Figure 4.6 shows a comparison between this aspect.

Figure 4.6: Relationship between the size of the test suite and the number of generated test cases.

As expected, this function approximates a linear function. The number of generated test cases increases almost linearly according to the size of the test suite. The minimal variations due to the length of the test case which varies between 11 and 82, in this case.

Experiments

Chapter 5

Conclusions and Future Work

In this chapter, the main conclusions of this research work are presented. There were also errors and some details which may be improved and can be pointed as future work.

5.1 Conclusions

In this dissertation, it was made a research in order to evaluate the existing testing techniques using web usage information to generate test cases and how mutation testing can be applied directly into them. The main objective of this work was to develop a tool which could be able to extend automatically these test cases with some mutations. The result of this dissertation is to contribute to reducing the effort increasing the level of automation when testing web applications.

It was developed a tool which is able to inject mutations directly in test cases. In order to perform this activity, the process starts by gathering the information stored in a database and save the information into separated test cases. Then, the tool structures these files and add them input data, since it is not captured by privacy reasons. These tests are also executed to be possible to have a result to compare later.

With this tool, it is possible to add the mutations operators detailed in the Section 3: Add Back, Add Step, Change Order, Change Input Data, and Remove Step. These operators are automatically injected in the test cases. Besides this, the tool also evaluates them in order to add to the existing test suite, the ones which are relevant. It means, the ones which have different behaviour from the previous one. It is used the *Levenshtein Distance* algorithm to perform this evaluation by counting the minimum number of differences between the steps.

In order to validate this approach, two different applications were used. Firstly, a simple website was constructed to detect the problems related to each operator. It is composed by a simple form asking for some personal information. It was able to perform the main objective of the tool: from one test case, which represents the correct fulfillment of the form, were generated a considerable number of relevant test cases. On the other hand, the tool was also tested in a

Conclusions and Future Work

real scenario. It was used data from a real web site (*Polytechnic Institute of Viana do Castelo*) to generate test cases representing the most frequent paths of the users' interaction. The tool converted these abstract test cases into concrete test cases and inject the mutations. Also a good number of test cases were generated automatically.

In short, this tool is a contribute to automating the testing of web applications. It is able to extend the existing test suite automatically by applying some changes directly in the test cases.

5.2 Future Work

During the development of this dissertation, there were some obstacles which would be interesting to improve in future works. This research work points to automate Web Testing based on the most frequent paths performed by the user and extend them with mutations and there are some points which may be improved.

The information gathered in order to generate the test cases should be improved. For example, the information about each element should be more complete: in input fields, the type should be saved, avoiding the need for previously executing the abstract test case to detect this. It would be important to have more precise data in an efficient way. The data generators may also be improved to have data with more accuracy. The data generation is always a challenging field in testing activities once it can not be captured for privacy reasons.

The identification of the element, by its XPath, ensures a good identification but also have barriers. The dynamic behaviour of the website may cause an XPath different from every different session. This may cause failures in the captured test cases, and they cannot be replayed again without errors.

In future works, some new mutation operators may be created. It implies more research work on problems which may occur in a web context and develop more mutation operators able to detect them.

References

- [ABJ⁺15] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing Strategies for Model-based Mutation Testing. *Softw. Test. Verif. Reliab.*, 25(8):716–748, 2015.
- [ADJ⁺11] S Artzi, J Dolby, S H Jensen, A Moller, and F Tip. A framework for automated testing of javascript web applications. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 571–580, 5 2011.
- [AH11] N Alshahwan and M Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12, 11 2011.
- [AKD⁺08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. page 261, 2008.
- [Als10] Izzat Alsmadi. Using Genetic Algorithms for test case generation and selection optimization. *Canadian Conference on Electrical and Computer Engineering*, pages 1–4, 2010.
- [AOA05] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing Web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, 2005.
- [ASH] H Al Shaar and R Haraty. Modeling and automated blackbox regression testing of Web applications. *J. Appl. Inf. Technol. (Austria)*, 4(12):1182 – 98.
- [BA] A Bhattacharyya and C Amza. PReT: A Tool for Automatic Phase-Based Regression Testing. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). Proceedings*, pages 284 – 9, Los Alamitos, CA, USA.
- [BBGM10] J Bau, E Bursztein, D Gupta, and J Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, 2010.
- [BBH⁺16] Fevzi Belli, Christof J. Budnik, Axel Hollmann, Tugkan Tuglular, and W. Eric Wong. Model-based mutation testing - Approach and case studies. *Science of Computer Programming*, 120:25–48, 2016.
- [BPC11] Ana Barbosa, Ana C. R. Paiva, and José Creissac Campos. Test case generation from mutated task models. *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)*, pages 175–184, 2011.

REFERENCES

- [CB02] Jean-Francois Collard and Ilene Burnstein. *Practical Software Testing*. Springer-Verlag, Berlin, Heidelberg, collard, j edition, 2002.
- [CF17] Lukas Cegan and Petr Filip. Webalyt: Open web analytics platform. *2017 27th International Conference Radioelektronika, RADIOELEKTRONIKA 2017*, pages 1–5, 2017.
- [CF18] Lukas Cegan and Petr Filip. Advanced web analytics tool for mouse tracking and real-time data processing. *2017 IEEE 14th International Scientific Conference on Informatics, INFORMATICS 2017 - Proceedings*, 2018-Janua:431–435, 2018.
- [CZG⁺13] Ting Chen, Xiao Song Zhang, Shi Ze Guo, Hong Yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773, 9 2013.
- [Dan16] Jyoti J Danawale. Survey on Different Approaches for Mutation Testing. *IJCA Proceedings on National Conference on Advancements in Computer & Information Technology*, pages 25–28, 2016.
- [DOAM17] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation Operators for Testing Android Apps. *Inf. Softw. Technol.*, 81(C):154–168, 1 2017.
- [DS08] Gaurav Duggal and Bharti Suri. Understanding Regression Testing Techniques. 2008.
- [EKR03] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web Application Testing with User Session Data. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.
- [Fag14] Jody Condit Fagan. The suitability of web analytics key performance indicators in the academic library environment. *Journal of Academic Librarianship*, 40(1):25–34, 2014.
- [FdlNHKW08] Lars Frantzen, Maria de las Nieves Huerta, Zsolt Gere Kiss, and Thomas Wallet. On-The-Fly Model-Based Testing of Web Services with Jambition. In *WS-FM*, 2008.
- [GFM15] Z Gao, C Fang, and A M Memon. Pushing the limits on automation in GUI regression testing. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 565–575, 11 2015.
- [GP16] J Esparteiro Garcia and Ana C R Paiva. An Automated Approach for Requirements Specification Maintenance. In Álvaro Rocha, Ana Maria Correia, Hojjat Adeli, Luis Paulo Reis, and Marcelo Mendonça Teixeira, editors, *New Advances in Information Systems and Technologies*, pages 827–833, Cham, 2016. Springer International Publishing.
- [GP18] Jorge Esparteiro Garcia and Ana C R Paiva. Manage Software Requirements Specification Using Web Analytics Data. In Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, and Sandra Costanzo, editors, *Trends and Advances in Information Systems and Technologies*, pages 257–266, Cham, 2018. Springer International Publishing.

REFERENCES

- [GPF10] A M P Grilo, A C R Paiva, and J P Faria. Reverse engineering of GUI models for testing. In *5th Iberian Conference on Information Systems and Technologies*, pages 1–6, 6 2010.
- [Hig17] Banur-ambala Highway. Model base testing : A review. *International Journal of Advanced Science and Research*, 2(3):44–51, 2017.
- [HM07] R M Hierons and M G Merayo. Mutation Testing from Probabilistic Finite State Machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 141–150, 2007.
- [HPW] B Hofer, B Peischl, and F Wotawa. GUI savvy end-to-end testing with smart monkeys. In *2009 ICSE Workshop on Automation of Software Test (AST 2009)*, pages 130 – 7, Piscataway, NJ, USA.
- [JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing, 2011.
- [KK13] K Karambir and Kuldeep Kaur. Survey of Software Test Case Generation Techniques. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(6):937–942, 2013.
- [KMB] P Kandil, S Moussa, and N Badr. A study for regression testing techniques and tools. *Int. J. Soft Comput. Softw. Eng. (USA)*, 5(4):64–84.
- [KS18] Yavuz Koroglu and Alper Sen. TCM: Test Case Mutation to Improve Crash Detection in Android. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 264–280, Cham, 2018. Springer International Publishing.
- [KSK12] Lakhwinder Kumar, Hardeep Singh, and Ramandeep Kaur. Web analytics and metrics. *Proceedings of the International Conference on Advances in Computing, Communications and Informatics - ICACCI '12*, page 966, 2012.
- [KST⁺15] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard Aichernig, Elisabeth Jöbstl, and Harald Brandl. MoMut::UML model-based mutation testing for UML. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015.
- [LDD14] Yuan Fang Li, Paramjit K. Das, and David L. Dowe. Two decades of Web application testing - A survey of recent advances. *Information Systems*, 43:20–54, 2014.
- [Mem07] Atif M Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17:137–157, 2007.
- [MPNM17] Rodrigo M L M Moreira, Ana Cristina Paiva, Miguel Nabuco, and Atif Memon. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629, 2017.
- [MPR18] Daniel Maciel, Ana C R Paiva, and Alberto Rodrigues. From Requirements to Automated Acceptance Tests of Interactive Apps : An Integrated Model-based Testing Approach. *Proceedings of ENASE'2019*, 2018.

REFERENCES

- [MSP01] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. Coverage Criteria for GUI Testing. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 256–267, New York, NY, USA, 2001. ACM.
- [MY10] Y Miao and X Yang. An FSM based GUI test automation model. In *2010 11th International Conference on Control Automation Robotics Vision*, pages 120–126, 12 2010.
- [NB13] Stanislava Nedyalkova and Jorge Bernardino. Open Source Capture and Replay Tools Comparison. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E ’13, pages 117–119, New York, NY, USA, 2013. ACM.
- [Nid12] Srinivas Nidhra. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications*, 2(2):29–50, 2012.
- [Pai06] Ana C. R. Paiva. *Automated specification-based testing of graphical user interfaces*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2006.
- [PEG19] Ana C R Paiva, Jean-david Elizabeth, and M E P Gouveia. Testing When Mobile Apps Go to Background and Come Back to Foreground. *ICST*, 2019.
- [PFTV05] Ana C R Paiva, João C P Faria, Nikolai Tillmann, and Raul A M Vidal. A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, pages 450–464, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [PM12] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Information and Software Technology*, 54(9):915–932, 2012.
- [PO10] Upsorn Praphamontriping and Jeff Offutt. Applying Mutation Testing to Web Applications. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW ’10, pages 132–141, Washington, DC, USA, 2010. IEEE Computer Society.
- [QM11] Zhong Sheng Qian and Huai Kou Miao. Towards Testing Web Applications: A PFSM-Based Approach. In *Advanced Research on Industry, Information System and Material Engineering, IISME2011*, volume 204 of *Advanced Materials Research*, pages 220–224. Trans Tech Publications Ltd, 2011.
- [QN] I A Qureshi and A Nadeem. GUI testing techniques: a survey. *Int. J. Future Comput. Commun. (Singapore)*, 2(2):142 – 6.
- [REG07] Hassan Reza, Sandeep Endapally, and Emanuel Grant. A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. In *Proceedings - International Conference on Information Technology-New Generations, ITNG 2007*, pages 366–370, 2007.

REFERENCES

- [RT05] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proceedings - International Conference on Software Engineering*, pages 25–34, 6 2005.
- [SAH] D Suleiman, M Alian, and A Hudaib. A survey on prioritization regression testing test case. In *2017 8th International Conference on Information Technology (ICIT). Proceedings*, pages 854 – 62, Piscataway, NJ, USA.
- [Sam04] S Sampath. Towards defining and exploiting similarities in Web application use cases through user session analysis. *IET Conference Proceedings*, pages 17–24, 2004.
- [Sil17] Valter Silva. *Model-Based Testing : From Requirements to Tests*. PhD thesis, FEUP, 2017.
- [SPGR18] Pedro Silva, Ana C. R. Paiva, Jorge Esparteiro Garcia, and André Restivo. Automatic Test Case Generation from Usage Information. *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018.
- [UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
- [USVH] S Ulewicz, D Schutz, and B Vogel-Heuser. Software changes in factory automation: towards automatic change based regression testing. In *IECON 2014. 40th Annual Conference of the IEEE Industrial Electronics Society. Proceedings*, pages 2617 – 23, Piscataway, NJ, USA.
- [VG16] Vishawjyoti and P Gandhi. A survey on prospects of automated software test case generation methods. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 3867–3871, 3 2016.
- [WK09] Daniel Waisberg and Avinash Kaushik. Web Analytics 2.0: empowering customer centricity. *The original Search Engine Marketing . . .*, 2(1):7, 2009.
- [XXM15] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help. 2015.
- [ZWH⁺12] Kuanjiu Zhou, Xiaolong Wang, Gang Hou, Jie Wang, and Shanbin Ai. Software reliability test based on markov usage model. *Journal of Software*, 7(9):2061–2068, 9 2012.

REFERENCES

Appendix A

Experiment Simple Website

A.1 Abstract Test Case

```
1  [
2    {
3      "path": "id(\"name\")",
4      "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
5      "action": "click",
6      "actionId": 1,
7      "pathId": 1,
8      "elementPos": 1,
9      "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
10   },
11   {
12     "path": "id(\"name\")",
13     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
14     "action": "input",
15     "actionId": 3,
16     "pathId": 2,
17     "elementPos": 2,
18     "value": [
19       "char",
20       "char",
21       "char"
22     ],
23     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
24   },
25   {
26     "path": "id(\"myform\").div[@class=\"container\"]/label[2]/input[1]",
27     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
28     "actionId": 1,
29     "action": "click",
30     "elementPos": 3,
31     "pathId": 3,
```

Experiment Simple Website

```
32     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
33 },
34 {
35     "path": "id(\"myform\")/div[@class=\"container\"]/label[2]/input[1]",
36     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
37     "actionId": 3,
38     "action": "input",
39     "elementPos": 4,
40     "pathId": 4,
41     "value": [
42         "char",
43         "char",
44         "char"
45     ],
46     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
47 },
48 {
49     "path": "id(\"email\")",
50     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
51     "action": "click",
52     "actionId": 1,
53     "pathId": 5,
54     "elementPos": 5,
55     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
56 },
57 {
58     "path": "id(\"email\")",
59     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
60     "action": "input",
61     "actionId": 3,
62     "pathId": 6,
63     "elementPos": 6,
64     "value": [
65         "char",
66         "char",
67         "char"
68     ],
69     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
70 },
71 {
72     "path": "id(\"password\")",
73     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
74     "actionId": 1,
75     "action": "click",
76     "pathId": 7,
77     "elementPos": 7,
78     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
79 },
80 {
```

Experiment Simple Website

```
81     "path": "id(\"password\")",
82     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
83     "actionId": 3,
84     "action": "input",
85     "pathId": 8,
86     "elementPos": 8,
87     "value": [
88         "password"
89     ],
90     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
91 },
92 {
93     "path": "id(\"confirm\")",
94     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
95     "actionId": 1,
96     "action": "click",
97     "pathId": 9,
98     "elementPos": 9,
99     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
100 },
101 {
102     "path": "id(\"confirm\")",
103     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
104     "actionId": 3,
105     "action": "input",
106     "pathId": 10,
107     "elementPos": 10,
108     "value": [
109         "password"
110     ],
111     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
112 },
113 {
114     "path": "id(\"submit\")",
115     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
116     "actionId": 1,
117     "action": "click",
118     "pathId": 14,
119     "elementPos": 11,
120     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html"
121 },
122 {
123     "path": "/html[1]/body[1]/div[@class=\"navbar\"]/a[2]",
124     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
125     "actionId": 1.0,
126     "action": "click",
127     "elementPos": 12,
128     "pathId": 12,
129     "url": "file:///C:/xampp/htdocs/diss/extractApp/page2.html#"
```

Experiment Simple Website

```
130 },
131 {
132     "path": "/html[1]/body[1]/div[@class=\"navbar\"] /a[3] ",
133     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
134     "actionId": 1.0,
135     "action": "click",
136     "elementPos": 13,
137     "pathId": 13,
138     "url": "file:///C:/xampp/htdocs/diss/extractApp/page2.html#"
139 }
140 ]
```

Listing A.1: Recorded abstract test case.

A.2 Concrete test cases

```
1 {
2     "start": {
3         "path": "id(\"name\")",
4         "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
5         "action": "click",
6         "elementPos": 1.0,
7         "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
8         "mutated": false
9     },
10    "steps": [
11        {
12            "path": "id(\"name\")",
13            "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
14            "action": "click",
15            "elementPos": 1.0,
16            "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
17            "mutated": false
18        },
19        {
20            "path": "id(\"name\")",
21            "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
22            "action": "input",
23            "elementPos": 2.0,
24            "value": [
25                "mate"
26            ],
27            "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
28            "mutated": false,
29            "inputType": "text"
30        },
31    ]
32 }
```

Experiment Simple Website

```
31  {
32    "path": "id(\\myform\\)/div[@class\\u003d\\\"container\\\"]/label[2]/input[1]",
33    "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
34    "action": "click",
35    "elementPos": 3.0,
36    "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
37    "mutated": false
38  },
39  {
40    "path": "id(\\myform\\)/div[@class\\u003d\\\"container\\\"]/label[2]/input[1]",
41    "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
42    "action": "input",
43    "elementPos": 4.0,
44    "value": [
45      "hate"
46    ],
47    "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
48    "mutated": false,
49    "inputType": "text"
50  },
51  {
52    "path": "id(\\email\\)",
53    "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
54    "action": "click",
55    "elementPos": 5.0,
56    "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
57    "mutated": false
58  },
59  {
60    "path": "id(\\email\\)",
61    "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
62    "action": "input",
63    "elementPos": 6.0,
64    "value": [
65      "spotty"
66    ],
67    "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
68    "mutated": false,
69    "inputType": "text"
70  },
71  {
72    "path": "id(\\password\\)",
73    "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
74    "action": "click",
75    "elementPos": 7.0,
76    "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
77    "mutated": false
78  },
79  {
```

Experiment Simple Website

```
80     "path": "id(\"password\")",
81     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
82     "action": "input",
83     "elementPos": 8.0,
84     "value": [
85         "Password123!"
86     ],
87     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
88     "mutated": false,
89     "inputType": "password"
90 },
91 {
92     "path": "id(\"confirm\")",
93     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
94     "action": "click",
95     "elementPos": 9.0,
96     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
97     "mutated": false
98 },
99 {
100     "path": "id(\" confirm\")",
101     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
102     "action": "input",
103     "elementPos": 10.0,
104     "value": [
105         "Password123!"
106     ],
107     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
108     "mutated": false,
109     "inputType": "password"
110 },
111 {
112     "path": "id(\"submit\")",
113     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
114     "action": "click",
115     "elementPos": 11.0,
116     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
117     "mutated": false
118 },
119 {
120     "path": "/html[1]/body[1]/div[@class\u003d\"navbar\"]/a[2]",
121     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
122     "action": "click",
123     "elementPos": 12.0,
124     "url": "file: ///C:/xampp/htdocs/diss/extractApp/page2.html#",
125     "mutated": false
126 },
127 {
128     "path": "/html[1]/body[1]/div[@class\u003d\"navbar\"]/a[3]",
```


Experiment Simple Website

```
129     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
130     "action": "click",
131     "elementPos": 13.0,
132     "url": "file:///C:/xampp/htdocs/diss/extractApp/page2.html#",
133     "mutated": false
134   }
135 ],
136 "filename": "testSession-f48453c0-0eaa-bec9-989b-ac10e9493c9f.json",
137 "testResult": {
138   "seleniumResult": "Test testSession-f48453c0-0eaa-bec9-989b-ac10e9493c9f.json
139     was successfully executed. ",
140   "finalURL": "file:///C:/xampp/htdocs/diss/extractApp/page2.html#",
141   "withError": false
142 }
143
144
145
146 {
147   "start": {
148     "path": "id(\"name\")",
149     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
150     "action": "click",
151     "elementPos": 1.0,
152     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
153     "mutated": false
154   },
155   "steps": [
156     {
157       "path": "id(\"name\")",
158       "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
159       "action": "click",
160       "elementPos": 1.0,
161       "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
162       "mutated": false
163     },
164     {
165       "path": "id(\"name\")",
166       "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
167       "action": "input",
168       "elementPos": 2.0,
169       "value": [
170         "mate"
171       ],
172       "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
173       "mutated": false,
174       "inputType": "text"
175     },
176     {
```

Experiment Simple Website

```
177     "path": "id(\"myform\")/div[@class\u003d\"container\"]/label[2]/input[1]",
178     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
179     "action": "click",
180     "elementPos": 3.0,
181     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
182     "mutated": false
183 },
184 {
185     "path": "id(\"myform\")/div[@class\u003d\"container\"]/label[2]/input[1]",
186     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
187     "action": "input",
188     "elementPos": 4.0,
189     "value": [
190         "hate"
191     ],
192     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
193     "mutated": false,
194     "inputType": "text"
195 },
196 {
197     "path": "id(\"email\")",
198     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
199     "action": "click",
200     "elementPos": 5.0,
201     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
202     "mutated": false
203 },
204 {
205     "path": "id(\"email\")",
206     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
207     "action": "input",
208     "elementPos": 6.0,
209     "value": [
210         "spotty"
211     ],
212     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
213     "mutated": false,
214     "inputType": "text"
215 },
216 {
217     "path": "id(\"password\")",
218     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
219     "action": "click",
220     "elementPos": 7.0,
221     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
222     "mutated": false
223 },
224 {
225     "path": "id(\"password\")",
```

Experiment Simple Website

```
226     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
227     "action": "input",
228     "elementPos": 8.0,
229     "value": [
230         "Password123!"
231     ],
232     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
233     "mutated": false,
234     "inputType": "password"
235 },
236 {
237     "path": "id(\"confirm\")",
238     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
239     "action": "click",
240     "elementPos": 9.0,
241     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
242     "mutated": false
243 },
244 {
245     "path": "id(\"confirm\")",
246     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
247     "action": "input",
248     "elementPos": 10.0,
249     "value": [
250         "warlike"
251     ],
252     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
253     "mutated": false,
254     "inputType": "password"
255 },
256 {
257     "path": "id(\"submit\")",
258     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
259     "action": "click",
260     "elementPos": 11.0,
261     "url": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
262     "mutated": false
263 },
264 {
265     "path": "/html[1]/body[1]/div[@class\u003d\"navbar\"] /a[2]",
266     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
267     "action": "click",
268     "elementPos": 12.0,
269     "url": "file:///C:/xampp/htdocs/diss/extractApp/page2.html#",
270     "mutated": false
271 },
272 {
273     "path": "/html[1]/body[1]/div[@class\u003d\"navbar\"] /a[3]",
274     "session": "f48453c0-0eaa-bec9-989b-ac10e9493c9f",
```

Experiment Simple Website

```
275     "action": "click",
276     "elementPos": 13.0,
277     "url": "file: ///C:/xampp/htdocs/diss/extractApp/page2.html#",
278     "mutated": false
279   }
280 ],
281 "filename": "testSession-f48453c0-0eaa-bec9-989b-ac10e9493c9f.json",
282 "testResult": {
283   "seleniumResult": "Test testSession-f48453c0-0eaa-bec9-989b-ac10e9493c9f.json
      wasn\u0027t successfully executed. \n  Error finding element /html[1]/body
      [1]/div[@class\u003d\"navbar\"]/a[2] on position 12.0",
284   "finalURL": "file:///C:/xampp/htdocs/diss/extractApp/index.html",
285   "withError": true
286 }
287 }
```

Listing A.2: Generated concrete test cases.