Inês Margarida Silva Teixeira

Generation of Mutation Tests for Optical Line Terminal

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Carlos Nuno Bizarro e Silva Laranjeiro and Prof. João Nuno Gonçalves Costa Cavaleiro Correia and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

1 2 9 0

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Inês Margarida Silva Teixeira

Generation of Mutation Tests for Optical Line Terminal

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Carlos Nuno Bizarro e Silva Laranjeiro and Prof. João Nuno Gonçalves Costa Cavaleiro Correia and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Acknowledgements

Initially, I would like to thank my advisor,Professor Nuno Laranjeiro, and my co-advisor, Professor João Correia, for welcoming me into this dissertation and guiding me throughout the entire process. I would also like to express my gratitude to Altice Labs, especially João Carreira, for always being available to answer my questions.

To my friends Rita Rodrigues and Filipa Capela, for have always been there to help and advise me throughout this dissertation. For all the calls focused on me and my dissertation. Thank you for being there in all the challenging and stressful moments. As well, thank you for the good times, like watching a sunrise after we pulled an all-night study session, taking a break to play foosball, and moments of non-stop laughter. Without both of you, this dissertation, and I, would not be the same. Thank you for being my pillar not only during this stage of my life but also in all others.

To my friend Marta Freitas, for helping me throughout all the phases of this dissertation. For always being just a phone call away and ready to support me. Thanks for your constant presence during this journey.

To my friend Domício Neto, for constantly helping me overcome all my difficulties in this dissertation. For answering all my questions with the extreme kindness. But not everything is work, so thank you for the conversations and shared laughter in the computer engineering department room. I'm glad this dissertation provided the opportunity for us to meet.

To my other friends who have also been by my side, always believing in me. For all the advice and moments of relaxation during this dissertation that you shared with me. Thank you for all the nights out, gatherings, and conversations that made my academic journey the best years of my life.

To my family, especially my mother, for always supporting me not only throughout this dissertation but also throughout my entire academic journey. For always caring about me, bringing me food during the toughest study days. Thank you, Mom, for all the love and care you give me. I am eternally grateful.

I would also like to thank my cat for never leaving me alone during all the days and nights of study for this dissertation, as well as throughout the rest of my academic journey.

A big thank you to everyone for helping me grow professionally.

Abstract

Software quality assurance, driven by rigorous software testing, is central to ensuring that a software application is correct and ready to be delivered to clients. Among testing methodologies, mutation testing is a highly effective approach to employ. When combined with traditional testing, it provides a strong phase for testing for identifying weaknesses in the test suite and, consequently, detecting defects in software to improve the overall quality of the system. Mutation testing involves introducing controlled changes, originating mutants, on one software to simulate fault scenarios and evaluate test suite efficacy.

This dissertation focuses on Optical Line Terminals (OLTs), which are critical network components. Positioned at the supplier's premises, they manage optical signal transmission via Passive Optical Network (PON) cables, serving Optical Network Units (ONUs) and Optical Network Terminals (ONTs). In Fiber to the X (FTTx) network architectures, they deliver data, voice, video, high-speed internet access, and television services cost-effectively.

Altice Labs, a distinguished company in the telecommunications industry, is involved in developing and testing OLTs. One of the primary objectives of this dissertation is to aid in testing the OLT equipment from Altice Labs. To achieve this, the dissertation aims to develop an approach that automates mutation testing. Typically, this approach involves source code, but challenges arise due to the absence of the OLT software's source code. Consequently, it results in the proposal of an innovative black box mutation testing approach. The approach outlined in this dissertation involves injecting faults into the OLT by altering its operational state to achieve our objectives.

Keywords

Mutation Testing, Mutant, Mutation Score, Black Box Testing, Equivalent Partition Technique, Boundary Values Analysis Technique, Steps, ".features" Files.

Resumo

A garantia de qualidade de software, impulsionada por testes rigorosos de software, é fundamental para assegurar que um software esteja correto e pronto para ser entregue aos clientes. Entre as metodologias de teste, o teste de mutação é uma abordagem altamente eficaz para empregar. Quando combinado com testes tradicionais, ele fornece uma fase robusta para identificar fragilidades no conjunto de testes e, consequentemente, detectar defeitos no software para melhorar a qualidade geral do sistema. O teste de mutação envolve a introdução de alterações controladas, originando mutantes, num software para simular cenários de falha e avaliar a eficácia do conjunto de testes.

Esta dissertação concentra-se em Terminais de Linha Óptica (*OLTs*), componentes críticos de redes. Localizados nas instalações do fornecedor, eles gerenciam a transmissão de sinais ópticos por meio de cabos de Rede Óptica Passiva (*PON*), atendendo Unidades de Rede Óptica (*ONUs*) e Terminais de Rede Óptica (*ONTs*). Nas arquiteturas de redes de Fibra até o X (*FTTx*), eles fornecem dados, voz, vídeo, acesso à Internet de alta velocidade e serviços de televisão de maneira económica.

A Altice Labs, uma empresa destacada na indústria de telecomunicações, está envolvida no desenvolvimento e teste de *OLTs*. Um dos principais objetivos desta dissertação é auxiliar nos testes do equipamento *OLT* da Altice Labs. Para alcançar isso, a dissertação visa desenvolver uma abordagem que automatize o teste de mutação. Normalmente, essa abordagem envolve código-fonte, mas desafios surgem devido à ausência do código-fonte do software *OLTs*. Consequentemente, resulta na proposta de uma abordagem inovadora de teste de mutação em *black box*. A abordagem delineada nesta dissertação envolve a introdução de falhas no OLT alterando seu estado operacional para atingir nossos objetivos.

Palavras-Chave

Teste de Mutação, Mutante, Pontuação de Mutação, Testes em *Black Box*, Técnica de Partição Equivalente, Técnica de Análise de Valores Limite, *Steps*, ficheiros ".features".

Contents

Acronyms

AAR Array Reference for Array Reference Replacement.

AES Advanced Encryption Standard.

ASR Array Reference for Scalar Variable Replacement.

cca Coincidental Correctness.

CLI Common Line Interface.

CSV Comma-Separated Values.

EJB Enterprise Java Bean.

FOM First Order Mutant.

FTTb Fiber-To-The-Business.

FTTB Fiber-To-The-Building.

FTTc Fiber-To-The-Cell.

FTTC Fiber-To-The-Curb.

FTTdp Fiber-To-The-Distribution-Point.

FTTH Fiber-To-The-Home.

FTTx Fiber-To-The-X.

GPON Gigabit-capable Passive Optical Network.

HOM Higher Order Mutants.

MIMD Multiple Instruction Multiple Data.

NMS Network Management System.

OLT Optical Line Terminal.

ONT Optical Network Terminal.

ONU Optical Network Unit.

pda Predicate Analysis.

PON	Passive Optical Network.
sal	Statement Analysis.
SIMD	Single Instruction Multiple Data.
SiMut	Similarity-based Mutation.
SVR	Scalar Variable Replacement.

List of Figures

List of Tables

Chapter 1

Introduction

In the scope of the Master's Degree in Computer Engineering with specialization in Software Engineering from the University of Coimbra in collaboration with Altice Labs, this dissertation was proposed.

Software testing is a vital component of the software development lifecycle, ensuring the quality of software applications. Mutation testing adds an extra layer of assurance by assessing the effectiveness of the test set and leading to improvements in the software. Using both traditional testing methods and mutation testing can help organizations deliver software that aligns with user expectations while minimizing risks and costs associated with defects.

This testing method involves introducing controlled changes into the software to create mutants. Then, the test set is run against a wide range of mutants. Fault introduction is usually done through the source code, but it can also be accomplished in other ways, as it will be applied in this dissertation. The present work aims to apply mutation testing to an OLT.

In light of this, mutation testing is a powerful evaluation technique capable of identifying areas of weakness that other testing methods cannot. However, it has its own limitations, including being time-consuming, having a high computational cost, and the complexity of handling mutants.

In this dissertation, mutation testing is applied to an OLT. This network equipment is situated at the supplier's office and serves the purpose of managing the distribution of optical signals through Passive Optical Network (PON) through optical fibers cables between the supplier and the multiple customer, specifically an Optical Network Unit (ONU) or Optical Network Terminal (ONT). The OLT can be implemented in FTTx architectures for data, Wi-Fi, TV and others service distribution. Consequently, the OLT plays a central role in telecommunications networks.

1.1 Problem and Motivation

Altice Labs[1] is an organization offering a wide array of innovative telecommunications and technology solutions. They are dedicated to product development, encompassing both software and hardware. Altice Labs develops OLTs and tests them for quality assurance when delivered to their clients. This dissertation aims to assist in improving the testing process.

The Quality Assurance team at Altice Labs performs manual Mutation Testing on the OLT, needing to inject faults leading to the creation of mutants by hand. Doing this way has proven to be time-consuming. In this way, this dissertation aims to study, propose and develop a framework capable of automating OLT Mutation Testing, thus improving efficiency within the given time constraints.

As the most recent OLT developed by Altice Labs, the OLT2T4 is the primary focus of this dissertation. With the collaboration of Altice, the goal of this dissertation is to implement mutation testing for this equipment while ensuring that the approach can be adapted to other OLTs.

In this dissertation, since the OLT source code is not accessible, the aim is to inject faults using a black-box methodology. This presents the challenge of conducting mutation testing applied in a black-box approach.

1.2 Goals

As mentioned earlier, the main objective of this dissertation is to study, propose, and develop an automatic OLT mutation testing approach. To accomplish this, a series of smaller goals must be achieved, which are listed below:

- Perform a deeper analysis of the concepts and functionalities in the OLTs developed by Altice Labs. Additionally, understand how tests are conducted and executed within the OLT.
- Analyze and study state-of-the-art techniques for applying mutation testing in a black-box context, including recent mutation testing approaches.
- Study how to modify the operational state of an OLT and define the faults to be injected.
- Comprehend the process of automatically generating and executing mutants in conjunction with the test set.
- Implement the proposed approach, starting with the generation of mutants and subsequently running the test set. While doing this, ensure that the results of the mutation testing are carefully recorded for future analysis.

[1]https://www.alticelabs.com/

1.3 Planning

Due to the change in dissertation topics, the work plan only began in October 2022. Therefore, during the first semester, the majority of time and effort were dedicated to understanding OLT2T4 mutation testing. Altice Labs were consistently available to provide assistance for this process. Figure 1.1 illustrates the Gantt chart with the tasks for the first semester.

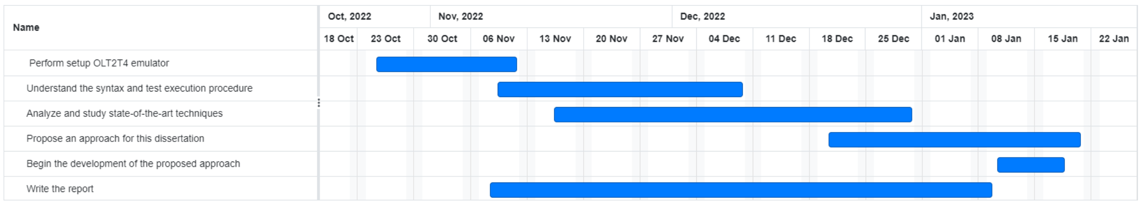


Figure 1.1: First Semester Gantt Chart.

As shown in figure 1.1, according to the Gantt chart, the first task was to set up the OLT2T4 emulator to make it possible to execute the tests and subsequently perform mutation testing. The second task involved understanding the syntax and procedures for executing tests in the OLT2T4 emulator. Subsequently, we moved on to studying the state-of-the-art techniques to gain insights from previous mutation testing approaches. The purpose of these last two tasks was to learn from them and gather information to help formulate our approach. Unfortunately, the task of initiating the development of the proposed approach was not completed as the previous tasks took more time than estimated. Throughout the completion of these tasks, document writing was in progress, with a significant focus on this effort in the final month.

In the second semester, the primary goals were to formulate a proposal, implement it, and execute it. Consequently, figure 1.2 displays the Gantt chart for the second semester of the dissertation.

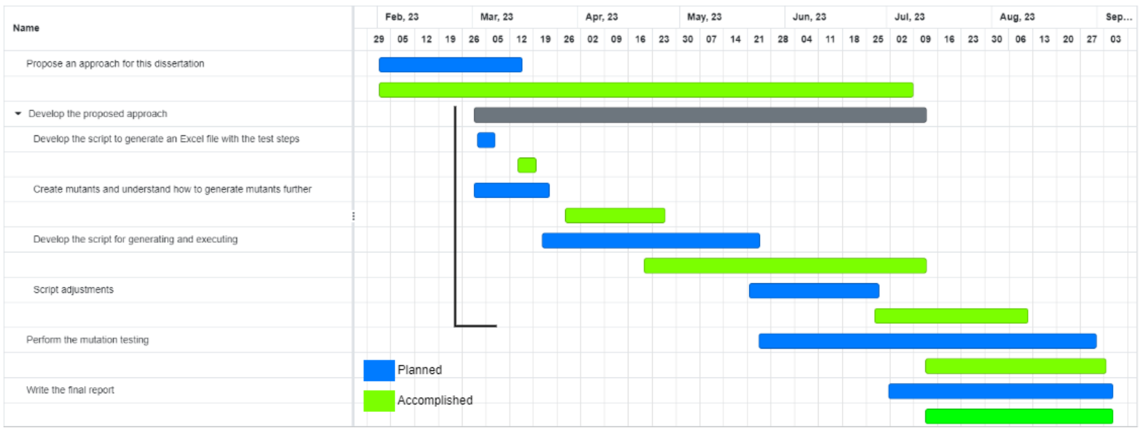


Figure 1.2: Second Semester Gantt Chart.

As illustrated in the Gantt chart in figure 1.2, during the second semester, the tasks faced delays due to the complexities involved in formulating a viable approach to address the dissertation's problem. Our OLT2T4 mutation testing approach had to be implemented within the context of black-box testing, and developing a solution for injecting faults into the OLT2T4 emulator took longer than initially projected. Additionally, initial testing revealed that the OLT2T4 software typically prevented any injected faults. Consequently, we had to reformulate our approach, which caused delays in subsequent tasks.

Given these challenges, the tasks for the second semester, as previously mentioned, involved proposing an approach within the scope of the dissertation. Subsequently, the development of this approach encompassed several tasks. This included creating a script file designed to generate the necessary spreadsheet file containing information required for the injection of faults into the OLT2T4 emulator. Additionally, the next task involved understanding how mutants could be created, initially starting manually and progressively advancing to automated methods. Therefore, the subsequent task involved the development of script files intended to generate the essential files required for fault injection and for executing Mutation Testing within the OLT2T4 emulator. To improve these scripts, we made adjustments to address errors identified during their execution.

The subsequent task was to perform OLT2T4 Mutation Testing by executing the scripts developed in the previously steps. Unfortunately, due to the delays encountered earlier in the semester, this task remained incomplete. Finally, as we approached the end of the second semester, it was written a comprehensive final report, encapsulating our academic journey and meticulously documenting our work.

1.4 Document Structure

This document is divided into six chapters, which are:

- The current chapter, chapter 1, introduces the dissertation, addressing its problems, motivation, goals, and planning.
- The second chapter, chapter 2, presents background information, including concepts about the OLT, the testing system applied in the OLT, sintaxe, mutation testing techniques, and strategies to reduce computational costs.
- The third chapter, chapter 3, discusses related work on recent applications of mutation testing and techniques for applying it in a black-box context.
- The fourth chapter, chapter 4, outlines the proposed approach for developing solutions and provides an explanation of how it is achieved.
- The fifth chapter, chapter 5, presents the development of the proposed approach and its results, followed by the analysis.

- The sixth chapter, chapter 6 offers the conclusion of this dissertation and suggests directions for future work.

Chapter 2

Background Concepts

This chapter offers an essential context concerning the topics relevant to Mutation Testing in an Optical Line Terminal (OLT), which can be explored in addressing the issue outlined in this dissertation. Firstly, the introductory chapter 2.1 provides a clear understanding of the concept of an OLT, an integral element of this dissertation. Furthermore, it delves into the characteristics and objectives of OLTs, encompassing different types employed at Altice Labs, along with key distinctions between them. Subsequently, the focus shifts to conducting a more comprehensive examination of the OLT2T4, the specific equipment being studied.

In the subsequent section (2.2), the testing procedures employed for the OLT2T4 at Altice Labs are outlined, encompassing the languages and frameworks chosen for testing. Moreover, the test composition process using the selected approach is elucidated.

The following sections are focused on Mutation Testing. The third section 2.3 thoroughly explains mutation testing, detailing its fundamental concepts and methodology. To enhance comprehension, an illustrative example of a mutant is also included.

Subsequently, the fourth section 2.5 addresses the high-cost nature of Mutation Testing and introduces techniques for cost reduction in this context. This section is subdivided into two subsections. In the first, 2.4.1, techniques to reduce the choice of mutants to be used for testing are presented. And in the second, 2.4.2, techniques to optimize the process of executing mutation tests are presented.

2.1 Optical Line Terminal

An OLT is a network device that serves as a terminal point in an optical fiber-based network. It is responsible for managing and controlling the communication between the end-user devices and the service provider's network. The OLT is located at the central office of providers, as illustrated in figure 2.1.

Altice Labs' OLT equipment portfolio offers the broadest and most scalable solution in today's market, providing Network Operators and Service Providers with a flexible and cost-effective approach to implement Passive Optical Networks xPON.

Passive Optical Network (PON) is an optical fiber cable that delivers broadband network access to homes, businesses and others. Given that PON is a network technology specific to fiber technology, it offers advantages such as a point-to-multipoint architecture, high-quality service capabilities for data, voice, and video, high-speed internet access, and TV services in a cost-effective manner. [Abbas and Gregory, 2016-05] Since the OLT uses PON cables, they are designed to address all of these services as well.

Another characteristic of the OLT is that they also support Active Ethernet services, which is a point-to-point fiber access technology used for delivering Internet services to residential and business subscribers through point-to-point links and Ethernet interfaces. Security is a crucial consideration, and therefore, the Advanced Encryption Standard (AES) is available for all OLT models from Altice Labs to protect sensitive data.

The OLT from Altice Labs can be deployed for all Fiber-To-The-X (FTTx) Point-to-Multipoint scenarios, including the architecture Fiber-To-The-Home (FTTH), Fiber-To-The-Distribution-Point (FTTdp), Fiber-To-The-Building (FTTB), Fiber-To-The-Curb (FTTC), Fiber-To-The-Cell (FTTc), and Fiber-To-The-Business (FTTb).

A conventional FTTx network consists of active equipment, which means it is powered by electricity located at the ends of the network, along with passive accessories, meaning that do not require electricity. This leads to the term PON networks, referring to all the passive components within an FTTx network [Keiser, 2006-02-06].

In the following figure, figure 2.1, some examples of FTTx architectures are illustrated and explained.

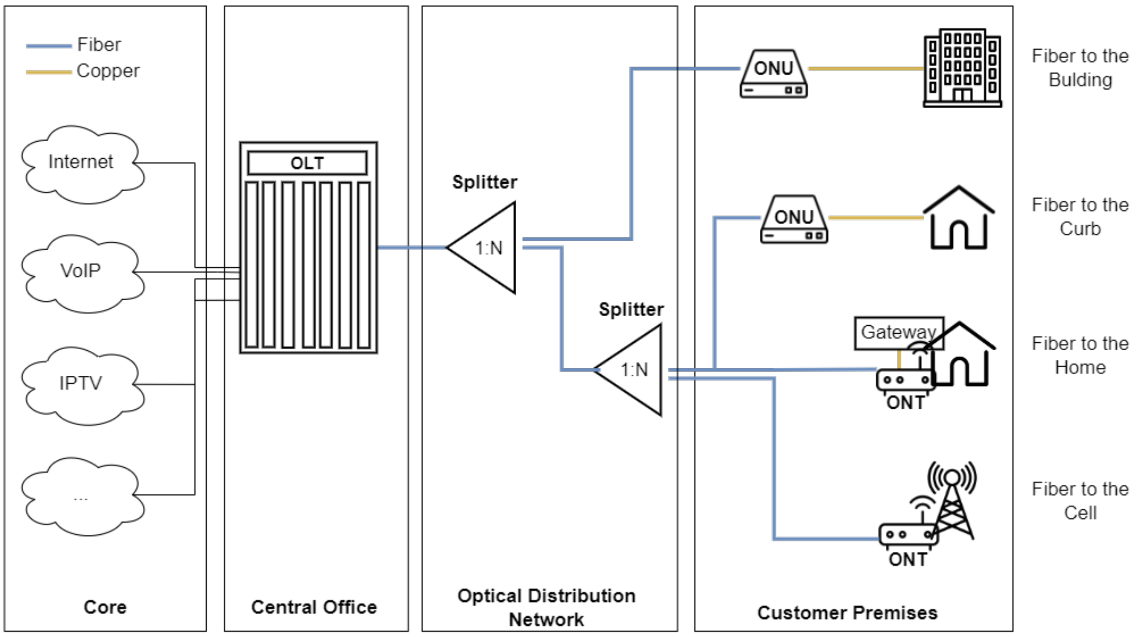


Figure 2.1: Some FTTx Architechtures. Adapted from [Marchetti, 2015-03-01] and [Keiser, 2006-02-06].

As shown in figure 2.1, the OLT and Optical Network Terminal (ONT) are the active elements of an FTTx network. They include transmitters and receivers and operate in two modes: upstream (sending information from the Optical Network Unit (ONU)/ONT to the OLT) and downstream (sending information from the OLT to ONU/ONT). The rest of the network comprises passive elements, such as the Splitter and the ONU. The splitter is responsible for distributing the optical signal from one fiber to several others, thus increasing the branching of the PON network. And the ONU, converts optical signals transmitted via fibers into electrical signals transmitted over copper wires and sends them to individual subscribers. The ONU can be located in different spaces, such as a garage (FTTB) or street cabinet or street pole (FTTC). [Keiser, 2006-02-06]

The ONT is essentially the same as the ONU, but its location changes, they are placed in the subscriber's house. The ONT can be a separate equipment, but it is usually built into the router. Therefore, the ONT has two inputs: one to receive fiber optics and the other to connect to the router.

So far, Altice Labs has developed two versions of this equipment: OLT1TX and OLT2TX. The most recent version is OLT2TX, and in this dissertation, we will exclusively focus on this one.

Within the OLT2TX family, developed by Altice Labs, there are 3 main devices: OLT2T4, OLT2T2 and OLT2T0 (figure 2.2). In this dissertation, the primary focus is on the OLT2T4, since it is the one being used.

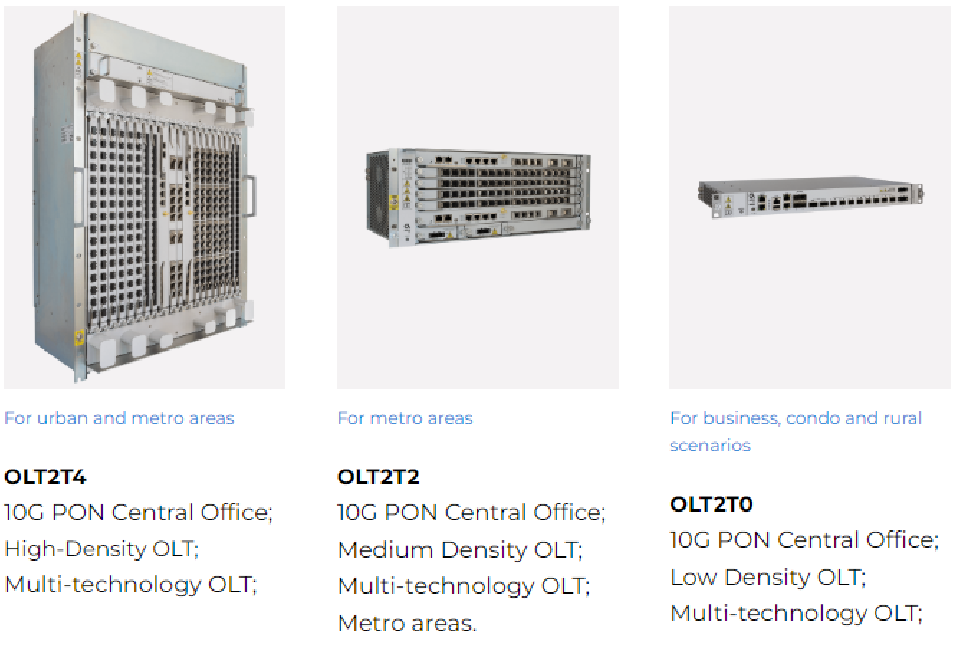


Figure 2.2: OLT2TX Family. [rita-c-felix]

There are quite a few differences between these devices, but one thing this product family has in common is the management level. It can be managed locally through the Common Line Interface (CLI), a text-based interface for direct device interaction. And remotely managed via the Network Management System (NMS), which offers centralized control and device monitoring.

Now to understand the difference between them, it is necessary to first understand how they are constituted. An OLT2TX consists of the Chassis, which houses the OLT2TX Software and supports N slots, where "N" represents the number of slots. These slots are filled with different types of cards that provide various functionalities to the OLT2TX, along with a ventilation unit.

There are three types of functionality provided by the cards, as follows:

- **Tax card/linecard:**Used to establish connections to customers.
- **Uplink card:** Utilized to establish connections to the operator's network.
- **Matrix/switch-fabric card:** Responsible for controlling the OLT and routing traffic between tributary and uplink cards. It also provides management interfaces and alarms.

Some cards may have more than one functionality, as will be further discussed ahead.

Knowing now how an OLT2TX is constituted, it is possible to understand the differences between the OLT2TX models. This is mostly related to the number of available entries provided by the cards.

The OLT2T0 serves as a central office FTTx equipment, specifically designed to cover low-density areas in urban, condominium, and rural environments. This equipment is equipped to endure an extended temperature range, making it suitable for installation within outdoor cabinets or even under more demanding atmospheric conditions. Regarding capacity specifications, the OLT2T0 has no slots, which means it does not support any card.

The OLT2T2 functions as a central office FTTx equipment, specifically designed to cover mid-density areas in urban and metro environments. Similar to the OLT2T1, this equipment is designed to support extended temperature ranges, making it suitable for installation within outdoor cabinets or even in locations exposed to more severe atmospheric conditions. Regarding capacity specifications, the OLT2T2, since is a medium-sized device, utilizes cards that combine the matrix and uplink functionalities into a single card. For this type of card the OLT2T2 have two slots which are both active. In addition to that, it also has four slots for the tributary cards.

The OLT2T4 is an FTTx central office equipment, designed to cater to high-density areas in urban and metro environments. Being the largest equipment in the family, it has more cards, resulting in more inputs and interfaces, as each input serves as an interface. Regarding capacity specifications, the OLT2T4 is the only model that supports up to two matrix cards, one of which is redundant, serving as an inactive matrix to provide protection in case the active matrix card fails. It also supports two uplink cards and sixteen tributary cards.

In the OLT2T4, the uplink cards are placed side by side in the center (slot ten and eleven), and next to each of these are the matrix cards (slot nine and twelve), with all the remaining slots reserved for the tributary cards (slot one to eight and thirteen to twenty). For this OLT, Altice Labs has designed one card for each type, the "CXO2T4A" for the matrix, and the "UL200G" for the uplink. The "UL200G" provides up to 4 interfaces of 10G plus up to 2 interfaces of 100G. For tributary cards, there are four types available: "AG16G", which offers 16 Gigabit-capable Passive Optical Network (GPON) interfaces; "AC16SXG", providing 16 GPON or 10 Gigabit Symmetrical PON (XGSPON) interfaces; "AE48GE", with up to 48 Active Ethernet (AE) interfaces or up to 24 interfaces of 1G and up to 16 interfaces of 10G; and0 "AT08SXG", offering 8 interfaces of XGSPON or Time Wavelength Division Multiplexing PON (TWDM-PON). All the xPON acronyms referred to earlier are variations of the PON technology. [Abbas and Gregory, 2016-05]

Since the OLT2T4 has individual matrix and uplink cards, these cards cannot be used in the OLT2T2. However, the tributary cards used in the OLT2T4 are also compatible with the OLT2T2.

Another difference is evident in figure 2.2, where the slots in the OLT2T2 are horizontally oriented, unlike the vertically oriented slots in the OLT2T4.

2.2 OLT2T4 Testing: Execution and Code Syntax

As previously mentioned, in this dissertation, mutation tests will be automatically generated for the OLT2T4 tests. Thus, in a first phase, it is necessary to understand the test system in order to later generate mutants based on this knowledge.

To run the tests, Altice Labs uses Jenkins tool, which triggers the Cucumber tool. Cucumber interprets the tests written in Ruby, which are represented by the ".feature" files, and executes the designated block of code in the sequence defined within the test. However, the tests are scripted in the Gherkin language and are subsequently mapped to Ruby.

While not obligatory, Jenkins simplifies the test execution process for Altice Labs, as it automates the execution of test sets and generates reports for each run. This automation significantly reduces the time and effort required by Altice, which could otherwise perform these tasks manually. Thus, all the results can be observed in Jira platform, the tool from which Jenkins retrieves the tests and displays the results. Jira is used because it helps in project management and tracking. Therefore, it is employed to have everything integrated into a single location, facilitating the visualization of test results.

As mentioned previously, the tests developed by Altice Labs are written using Gherkin since it is a language that follows a structure defined by keywords and is written in natural language. This framework aids in describing test scenarios in a clear and comprehensible manner for both developers and non-technical individuals. A Gherkin test usually follows the subsequent structure.

It starts with the keyword "Feature", in which a general description of the feature being tested is provided, usually consisting of a brief name and a more detailed description. Following this, an example of usage is provided. [Wynne et al., 2017-02-17]

```
Feature: Feature Name
   Detailed description of the feature being tested
```

Then, it may or may not have a "Background" keyword, in which a common background scenario is defined for several subsequent scenarios. Subsequently, an example of usage is shown. [Wynne et al., 2017-02-17]

```
Background:
   Given Some background condition
```

Then, it can include one or several "Scenario" keywords, each describing a specific situation to be tested. Afterward, an example of usage is demonstrated. [Wynne et al., 2017-02-17]

```
Scenario: Scenario name
   Given some initial condition
   When some action is performed
   Then some result is expected
```

The necessary Step(s) for conducting the test are positioned within each "Scenario", following the desired sequence. When mapped to Ruby, these steps encompass the Step definition along with the Step arguments. Step consistently begin with prepositions or adverbs, these being "Given", "When", "Then", "And", and "But".These keywords, upon translation to Ruby, also serve as matching steps in the step definition. The principal keywords are "Given", "When" and "Then", which structure the phases of the scenario. [Wynne et al., 2017-02-17]

"Given" ensures the scenario's settings by establishing the initial state, context, or situation in which the scenario initiates. "When" configures the test for the scenario, describing the action executed, typically an interaction with the system. Finally, "Then" verifies the test for this scenario, specifying the result after the action.[Wynne et al., 2017-02-17]

The "And" and "But" keywords enhance clarity and comprehensibility within the scenario's stages. The following script example demonstrates four steps. [Wynne et al., 2017-02-17]

```
Given some initial condition
And Other initial condition
When some action is performed
Then some result is expected
```

The step arguments are a component of Gherkin steps. As the name suggests, this is where the parameters and their corresponding values are inserted for each defined step, constituting the final parameter in a step definition when mapped to Ruby. The step argument may or may not include values, depending on whether arguments are needed for the given step definition. Typically, these are formatted using Data Tables, where the parameter descriptors are listed in the first row, separated by the pipe character, and the subsequent rows within the same block contain the values for each corresponding parameter, equally divided by the same character. The number of parameters must match the number of values, otherwise it will result in an error. Next, an example of usage is presented. [Wynne et al., 2017-02-17]

```
Given initial condition with <variable>:
   | variable1 | variable2 |
   | value1    | value4    |
   | value2    | value5    |
   | value3    | value6    |
```

In addition to these keywords, a Gherkin test can also have Tags, Hooks, and Environment Variables. Tags are markers that can be applied to scenarios or features. For those cases, the tag must be positioned above the "Scenario" or "Feature" keyword, and multiple tags can be placed for each, separated by a space. In the following, an example of usage is illustrated. [Wynne et al., 2017-02-17]

```
@featuretag
Feature: Feature Name

   @scenariotag1 @scenariotag2
    Scenario: Scenario name
           Given some initial condition
           When some action is performed
           Then some result is expected
```

The tags help in running specific subsets of scenarios or applying hooks to a exclusive subset of scenarios. This can be achieved using the following commands to execute or skip features or scenarios, respectively. [noa, c]

```
cucumber --tags "@scenariotag1 @scenariotag2"
cucumber --tags "not @scenariotag1"
```

Tags exhibit inheritance, meaning that when a tag is assigned to a feature, it extends to the scenarios within it. Additionally, tags can be employed to establish links to other documents. [noa, c]

Regarding Hooks, they are blocks are blocks of code that can be executed at different points during a test, although they are generally used before or after scenarios. Typically, they serve the purpose of setting up and tearing down the test environment before and after each scenario. In Gherkin, hooks are commonly defined within a support file as part of the test framework. [noa, a]

In this context, there are Scenario hooks, which are executed for each scenario, and they are three types: Before, After, and Around. As the names imply, the Before hook are executed before the initial step of each scenario, the After hook are executed after the last step of each scenario, and the Around hook are executed around a scenario. Additionally, there is also the Step hook, known as AfterStep. [noa, a]

Hooks can also be conditionally selected for execution based on scenario tags. This feature enables the linking of a Before, After, Around, or AfterStep hook with a specific tag expression, resulting in the execution of the corresponding hook only for particular scenarios. [noa, a]

Additional hook types include Global hooks, which are executed before any scenario or after all scenarios have run, referred to as the BeforeAll and AfterAll hooks, respectively. Furthermore, there is the InstallPlugin hook, which becomes active after Cucumber's configuration, and the AfterConfiguration hook. [noa, a]

Moving on to environment variables, Cucumber employs these variables to enable specific functionalities. They are utilized to establish configuration settings that influence the behavior of tests. This approach provides the advantage of segregating configuration considerations from the test logic, thus improving the manageability and flexibility of tests. Some examples of their usage include supplying specialized values to the step definitions or even facilitating the publication of Cucumber Reports while executing Cucumber. Environment variables can be configured through various methods, including the command line, configuration files, or directly within the testing framework. It is important to note that, due to security considerations, defining globally accessible environment variables containing sensitive information should be avoided. [noa, c] [noa, d]

When Cucumber executes a test, it identifies each Step definition and runs the corresponding Ruby code linked to it, along with the supporting Cucumber code. After the test is executed using Cucumber, various possible results are returned, such as Passed, Failed, Undefined, Pending, and Skipped. These outcomes depend on whether the step definitions successfully perform their intended actions. [Wynne et al., 2017-02-17] [noa, b]

During scenario execution, Cucumber processes one step at a time. If any step fails, all subsequent steps and scenarios are bypassed, resulting in either a skipped or undefined step and scenarios, depending on the situation. When a step passes, Cucumber proceeds to the next step, as well as in the scenarios. In this context, an undefined result occurs when Cucumber cannot find a step definition that matches the step. Furthermore, a pending result emerges when Cucumber encounters a step definition that is only partially implemented. Lastly, failing result when the code within a step definition raises an exception. [Wynne et al., 2017-02-17] [noa, b]

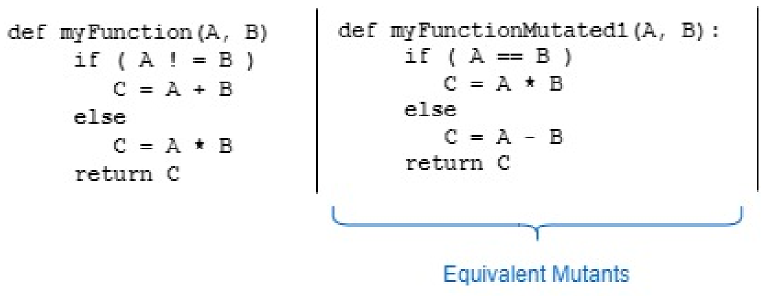
2.3 Mutation Testing

As mentioned earlier, the aim of this dissertation is to apply Mutation Testing. Therefore, to gain a better understanding of mutation testing, it is essential to start by clarifying several fundamental concepts.

- **Fault:** Defect in a software product or application that causes it not to work properly.
- **Mutant:** Version of a system with fault injection(s).
- **Test:** Procedure used to evaluate and verify if a software application functions as it is supposed to.
- **Mutation operators:** Changes made to the original code involve modifying expressions by altering, adding, or removing operators and/or statements to generate mutants. These operators can include arithmetic, relational, logical, among others.
- **Equivalent mutant:** A mutant in which changes have been introduced, but there are no differences in the outputs compared to the original system, figure 2.3.

- **Non-Equivalent mutant:** The opposite of the equivalent mutant.

Mutation testing is a testing technique used to measure the effectiveness of tests in terms of their ability to detect faults in a software product or application. We are looking to see if the tests sufficiently cover the defined requirements and/or check the correctness of the implementation of a given system. Because the possible faults induced can be enormous, this technique of testing uses two principles to restrict the classes of mutations that are created: the competent programmer hypothesis, which states that a programmer can make mistakes while writing code, even if they are a great programmer; and the coupling effect hypothesis, which states that a set of tests that detects small errors is so sensitive that it will also detect more complex errors in the program. This allows us to focus on simple faults because even tests that detect basic faults can also detect more complicated faults. However, the number of potential mutations is still significant. Therefore, mutation sets have been defined, along with corresponding rules, such as Mothra's mutation operators, which will be introduced later.[Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]



```
def myFunction(A, B)
    if ( A ! = B )
       C = A + B
    else
       C = A * B
    return C
```

```
def myFunctionMutated1(A, B):
    if ( A == B )
       C = A * B
    else
       C = A - B
    return C
```

Figure 2.3: Original Code and Equivalent Mutant. Adapted from [Ma and Kim, 2016-05-01].

In the graphic below is shown how the Mutation Testing method works:

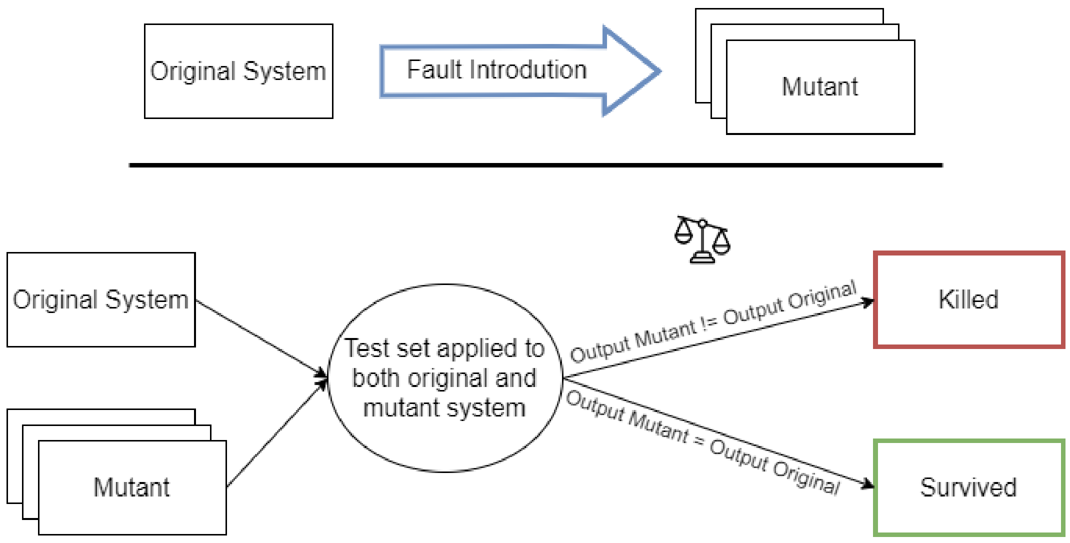

Original System
Fault Introdution
Mutant
Original System
Mutant
Test set applied to both original and mutant system
Output Mutant != Output Original
Output Mutant = Output Original
Killed
Survived

Figure 2.4: Description of the Mutation Testing method. Adapted from [Jia and Harman, 2011-09].

Analyzing the figure 2.4, we see that the "Mutation testing" method creates mutants in the first phase by introducing different syntactic faults in the original system. Then, to evaluate and ensure that the original system is flawless for those flaws introduced, we run the test set in the original system. To get the final result, we use the metric of comparing the output of the original system and the mutant. If the output obtained in the mutant it's different from the original that mutant is **"killed"**, otherwise it is said to have **"survived"**. [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

If a mutant is not "killed" by the test set, it means that the test set was unable to detect the faults represented by the mutant, or that the changed code was never executed and is therefore dead code. [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

Thus, to perform a mutation test, the following events must occur sequentially: first, a program is submitted for testing; second, a set of test cases is generated; third, all test cases are run through the original program; fourth, the mutation operators are selected; fifth, the mutants are generated; sixth, the mutants are run for each test case; seventh, the outputs are compared; and finally, eighth, the results are analyzed. If a mutant has not been "killed", it is necessary to improve the test cases and restart the process. [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

To evaluate the quality of the tests used in Mutation Testing, the mutation score is calculated, representing the percentage of "killed" mutants. This calculation is performed using the following mathematical expression 2.1. [Jia and Harman, 2011-09]

$$MutationScore = \frac{Mutants''killed''}{TotalNumberofMutants} \times 100 \tag{2.1}$$

In this testing method, the more mutants a test set can "kill", the more faults it detects in the program and the higher the mutation score. The best possible case is to get a mutation score of 100, which means that no mutants "survived" and therefore the test set is good enough to detect all the faults signaled by the mutants. In other cases, the test set needs to be improved by considering the surviving mutants in such a way that they can be "killed". [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

Mutation Testing has several advantages, such as maximum coverage of the code which allows finding hidden defects, identifying parts of the code that are not tested by the original test cases, and finding bugs usually not detected by normal testing. It also allows for the evaluation of the quality of the tests used to test the system and can give better feedback to the testers about it. However, one problem that prevents Mutation Testing from becoming a practical testing technique is the high computational cost of executing the enormous number of mutants against a test set. To combat this limitation, Cost Reduction Techniques are used. This technique consists of reducing the generated mutants ("do fewer") and reducing the execution cost (combining "do faster" and "do smarter"). [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

2.4 Cost Reduction Techniques in Mutation Testing

In the figure 2.5 is shown some of the existed techniques to combat the high cost of Mutation Testing. In this document we will only looking, within these, to the techniques more used. Two approaches to reduce the cost are the Mutant Reduction Technique and the Execution Cost Reduction Technique.

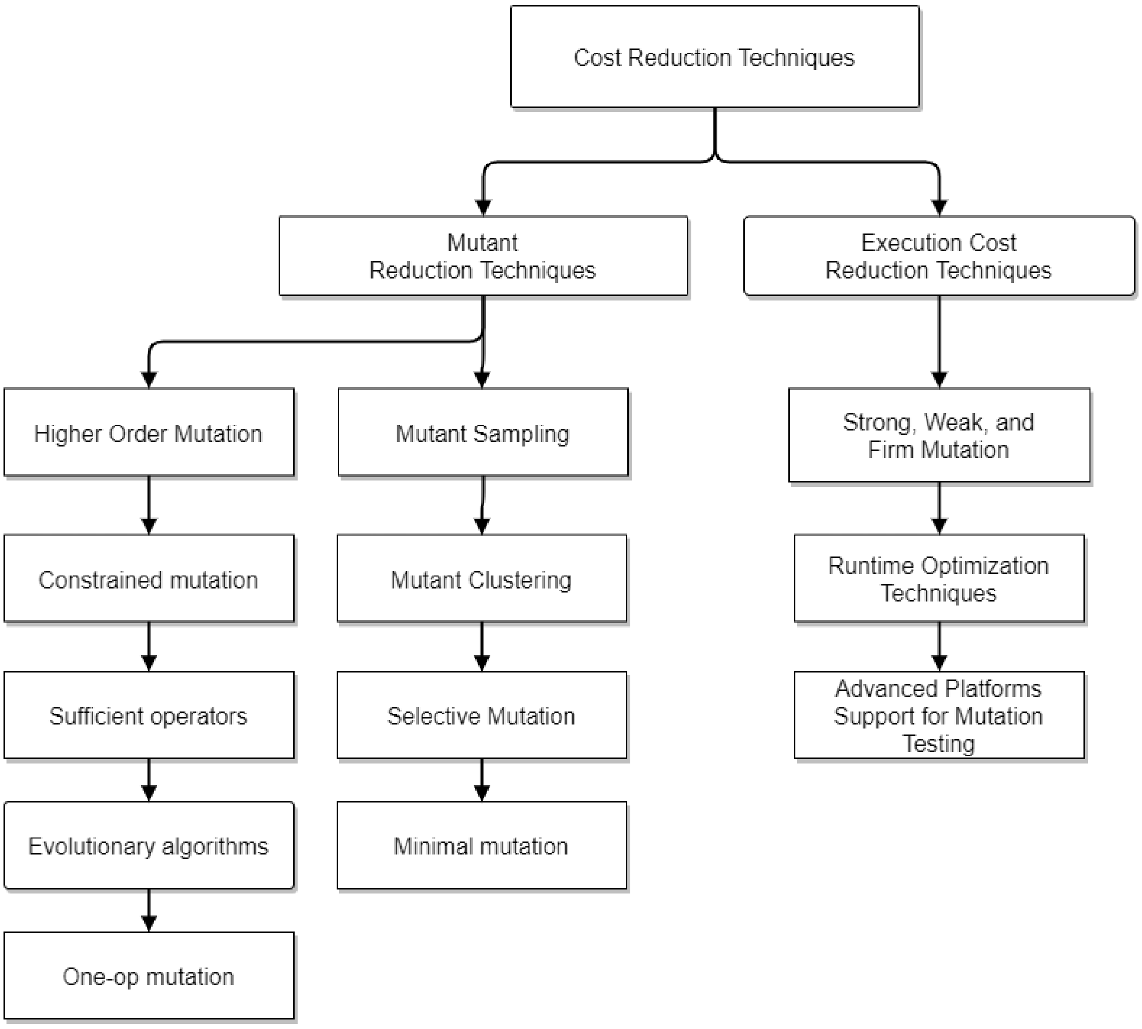

Cost Reduction Techniques
Mutant Reduction Techniques
Execution Cost Reduction Techniques
Higher Order Mutation
Constrained mutation
Sufficient operators
Evolutionary algorithms
One-op mutation
Mutant Sampling
Mutant Clustering
Selective Mutation
Minimal mutation
Strong, Weak, and Firm Mutation
Runtime Optimization Techniques
Advanced Platforms Support for Mutation Testing

Figure 2.5: Cost Reduction Techniques. Adapted from [Jia and Harman, 2011-09] and [Pizzoleto et al., 2019-11-01].

2.4.1 Mutant Reduction Techniques

Mutant reduction techniques aim to reduce the number of mutants used from the generated mutants for testing without a significant loss of test effectiveness. That is, for a given set of mutants M and a set of test data T, MS denotes the mutation score of the test set T applied to mutants M. Mutant reduction consists of finding a subset of mutants M' from M, where $MS(M') \approx MS(M)$. [Jia and Harman, 2011-09]

As illustrated in the figure 2.5 there are several possible techniques used to reduce the number of mutants in the generated set used for testing, but in this document only the most used ones will be presented.

The first approach is Mutant Sampling, in which all possible mutants are generated first, as in traditional Mutation Testing. Then, a percentage of these mutants are selected randomly for mutation analysis, and the remaining mutants are discarded. [Jia and Harman, 2011-09]

The second approach is Mutant Clustering, which is a combination of data clustering and mutation analysis, designed to dramatically reduce both the number of mutants and test data. To do this, a subset of mutants is chosen by applying clustering algorithms, which reduce the volume of data by segregating similar data. The sets originated are called clusters, and each has its own clustering center. This way, Mutant Clustering consists of classifying the mutants into different clusters based on the number of sets of test cases able to kill them, with the clustering centers being dynamically selected when grouping the classified mutants. In the end, different clusters are obtained, and it is guaranteed that each mutant in the same cluster is "killed" by a similar set of test cases (to better understand see [Dang et al., 2022-06]). This way, only a small number of mutants are selected from each cluster to be used in Mutation Testing, and the rest are discarded. Furthermore, a major advantage of this is that clustering algorithms are flexible, i.e. one can choose the method that fits best according to the application domain. [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

The third approach is Selective Mutation. To better understand this approach, it is important to first understand the methodology used to generate mutants. Initially, examples of operators are conditions such as "ifs" and "whiles"; arithmetic operators such as addition and division; logical operators such as "and" and "or"; and jump instructions such as "goto" and "return". Moving on to the methodology, the syntactic modifications that generate mutant programs are determined by a set of mutant operators. This set changes depending on the language of the program being tested and the mutation system used for testing. Mutant operators are created with one of two goals: to induce simple syntax changes based on errors that programmers typically make (such as using the wrong variable name), or to force common testing goals (such as executing each branch). As an example, we have Mothra's mutation operators, separated into three categories: Statement Analysis (sal), Predicate Analysis (pda), and Coincidental Correctness (cca). [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09]

The first, "sal", includes the following actions: replacing each statement with a "continue" or "trap", where "trap" stops the program and causes the mutant to die; replacing each statement of a subprogram with a "return"; and replacing each "goto" or "do" declaration with the label called. The second, "pda", comprises the following actions: replacing arithmetic (mathematical operations between two operators) or relational (Boolean values) or logical operators with all the others that exist; adding unary operators (operations with a single operand) before expressions; changing the values of constant variables; changing the value of expressions to their absolute value; and changing the values in data statements (values used as inputs in the program). Finally, the third, "cca", encompasses the following actions: replacing scalar variables (for example fixed-size data objects), arrays with pre-instantiated arrays, and constants with others existing in the code respectively. In the following table (Table 2.1), the existing Mothra mutation operators are classified and identified. [Pizzoleto et al., 2019-11-01] [Jia and Harman, 2011-09] [King and Offutt, 1991]

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

Table 2.1: Mothra Mutant Operators. Adapted from [King and Offutt, 1991].

The Mothra mutant operators represent more than ten years of refinement through several mutation systems. All of them are detailed described in [King and Offutt, 1991]. In this case, it will only described the "Array Reference for Array Reference Replacement (AAR)" mutant, which replaces each array reference in a program with another distinct array reference in the program. Each mutation operator generates different numbers of mutants, and some of them generate far more mutants than others, many of which may turn out to be redundant.

Therefore, this approach seeks to reduce the number of mutation operators applied by not using the operators that create the most mutants, and finding a small set of mutation operators that generate a subset of all possible mutants without significant loss of test quality. To do this, the *theory of N-selective mutation* is used, which consists of removing N mutant operators. For example, the two mutation operators of the 22 Mothra operators that generated the most mutants are "Array Reference for Scalar Variable Replacement (ASR)" and "Scalar Variable Replacement (SVR)", so it is implemented the “2-selective mutation”. The "4-selective" and the "6-selective" are also available, by omitting four and six mutation operators respectively. [Jia and Harman, 2011-09]

Moving on to the fourth and final approach in traditional Mutation Testing, mutants can be classified into First Order Mutant (FOM)s and Higher Order Mutants (HOM)s. FOMs are created by applying a mutation operator only once, and HOMs are generated by applying mutation operators more than once. For example, looking at the figure 2.6, second order mutants arise by combining first order mutants, as an example, by applying two mutation operators. [Jia and Harman, 2011-09] [Polo et al., 2009]

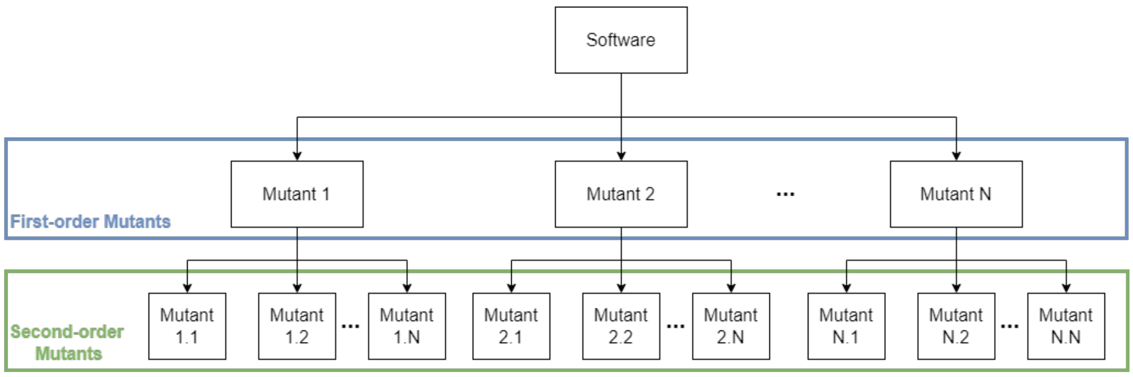


Figure 2.6: Exponential growth of the number of second-order mutants. Adapted from [Polo et al., 2009].

Also, when a FOM is combined with another mutant, the second-order mutant may or may not be equivalent, depending on the equivalence of the FOMs used, as we can see in table 2.2.

M_i	M_j	M_i, j
Equivalent	Equivalent	Equivalent
Equivalent	Non-Equivalent	Non-Equivalent
Non-Equivalent	Equivalent	Non-Equivalent
Non- Equivalent	Non-Equivalent	Non-Equivalent (very probably)

Table 2.2: Equivalence of a second-order mutant. Adapted from [Polo et al., 2009].

The generation of second-order mutants can be made of various combinations among the FOMs; however, three possible combinations will be presented: LastToFirst, DifferentOperators, and RandomMix. The LastToFirst consists in combining the FOMs with the last ones, i.e. the first with the last, the second with the penultimate, and so on. The DifferentOperators algorithm consists of combining FOMs originated by different mutation operators. Finally, RandomMix consists of randomly combining two FOMs only once, except when there is an odd number of FOMs, in which case a mutant can be used twice. With this algorithm, there is a 50% reduction since the number of second-order mutants will be half of the first-order ones. [Polo et al., 2009]

2.4.2 Execution Cost Reduction Techniques

An alternative way to reduce the computational cost is to reduce the number of generated mutants and optimize the mutant execution process. In this document, it will be present three types of techniques, shown in figure 2.5, used to optimize the execution process.

The first testing technique is Strong, Weak, and Firm Mutation, where the difference between these depends on the decision to analyze or not if a mutant is "killed" during the execution process. Strong Mutation is considered the traditional mutation type, where for a given software S, a mutant M of software S is considered "killed" only if mutant M gives a different output from the original software S. Given this context, the mentioned approach evaluates the test results exclusively after executing the complete test set on all mutants. [Jia and Harman, 2011-09] Furthermore, within this methodology, when dealing with a program of reasonable size, the potential number of conceivable faults becomes practically boundless. To narrow down the scope of analysis, two fundamental principles are predominantly employed. Firstly, the "Competent programmer" hypothesis involves injecting errors into the code that closely resembles the original code. Secondly, the "Coupling effect" hypothesis suggests that a single dataset might be adequate to identify both simple and complex faults. This process makes it a resource-intensive technique in terms of computational resources. [Jia and Harman, 2011-09]

Conversely, weak mutation examines intermediate results between the original code and the mutated code following the execution of an individual mutated code. [Offutt and Lee, 1994-05]

Weak Mutation appears in order to optimize the execution of the Strong Mutation. This technique is based on the assumption that S represents a software, C = C_1, ..., C_n denotes a set of components of S, and a set of mutants C' = C'_1, ..., C'_n is formed by altering component c_n. The mutant C' is considered "killed" if any execution of component C'_n differs from the original component C_n. [Jia and Harman, 2011-09]

So, in Weak Mutation, instead of checking the output of the mutants after the execution of the entire program, the mutants can be "killed" before executing all the program since after one mutant component be "killed" the rest of the mutants components are not run. Thus, although this technique clearly requires significantly less program execution, as disadvantage it clear too that this is less effectiveness that Strong Mutation.[Jia and Harman, 2011-09]

As previously mentioned, in order to implement weak mutation, it is necessary to identify components within the original programs where errors could potentially reside. This process involves examining the fundamental elements of the program. These components, referred to as "elementary computational structures", encompass various aspects such as variable references, assignments, arithmetic operations, as well as relational and Boolean expressions. [Singh and Srivastava, 2017-12]

This components can be defined in four types: components after the first execution of an expression (Ex-Weak/1), the first execution of a statement (St-Weak/1), the first execution of a basic block (BB-Weak/1), and after N iterations of a basic block in a loop (BB-Weak=N). [Offutt and Lee, 1994-05]

Finally, Firm Mutation falls between Strong Mutation and Weak Mutation in terms of efficiency and effectiveness. Unlike Strong Mutation, which assesses mutants only after their generation, and Weak Mutation, stops when a mutant is "killed", Firm Mutation introduces an intermediate approach. This method aims to address the limitations of both previously mentioned mutation types. Consequently, Firm Mutation testing operates with the combined characteristics of Weak and Strong Mutation, as it concludes execution and analyse the results after mutated code runs and the final output. Commonly adopted in highly interactive development environments, Firm Mutation testing encompasses the partial execution of program segments. This strategic approach proves advantageous by efficiently allocating time and resources for the detection of errors. [Singh and Srivastava, 2017-12]

The second testing technique is using Runtime Optimization Techniques that focus on reducing the compilation cost. Following this, the most used and relevant techniques are addressed. The Interpreter-based technique is one of the optimization techniques used in the first generation of mutation testing tools. This technique implements an interpreter to interpret the result of a mutant from its source code directly, so the run time cost is $it \times n \times m$ where it is the interpreting time, m the number of mutants and n the number of test cases. The main cost of this technique is determined by the cost of interpretation. This technique is efficient for small programs, however, due to the nature of interpretation, it becomes slower as the scale of programs under test increases. [Jia and Harman, 2008-08-01]

Compiler-Based Technique was aimed at reducing the cost of interpretation. In this technique, each mutant is first compiled into an executable program, then the compiled mutant is executed by a number of test cases. Compared to the source code interpretation techniques, this approach is much faster since the total run time cost in this case is $m(ct + rt \times n)$, where *ct* is the compilation time and *rt* is the run time for each test set. This technique is faster mainly for large programs, since the run time (*rt*) for an executable program is faster than the interpretation time. However, this technique also introduces an extra overhead cost *ct*. [Jia and Harman, 2008-08-01]

Now, seeking to reduce the overhead cost of the previous technique, the Mutant schemata approach was created. This, instead of compiling each mutant separately, generates a metaprogram by the mutant schemata technique, where the metaprogram is like a "super mutant" that can be used to represent all possible mutants. Thus, to run each mutant against the test set, only the metaprogram needs to be compiled. Since the metaprogram is a compiled program, its execution speed is faster than that of a technical interpreter, so this technique has a total execution time of $ct + rt \times n \times m$ where *ct* and *rt* are the compilation and execution time of the metaprogram, respectively. Since this metaprogram is a compiled program, its running speed is faster than the interpreter-based technique. [Jia and Harman, 2008-08-01]

The last testing technique consists of Advanced Platforms Support for Mutation Testing, where the computational cost of mutation testing is sought to be distributed among many processors. Single Instruction Multiple Data (SIMD) consists of executing mutants simultaneously, in other words, one instruction is applied to a bunch of information or distinct data at constant time. On the other hand, Multiple Instruction Multiple Data (MIMD) looks to improve the parallel execution of mutants. MIMD architecture includes a set of N-individual, tightly-coupled processors. Overall, MIMD is more efficient in terms of performance than SIMD. [Jia and Harman, 2011-09]

2.5 Summary

OLT is a network device that serves as a bridge between providers and customers. It manages and distributes data traffic between them using optical signals transmitted over PON fiber cables. This network device can also be applied in various FTTx architectures to deliver downstream information to clients and receive upstream information from clients. To ensure data security, this equipment uses a specific protocol.

Altice Labs has developed two versions of the OLT: OLT1Tx and OLT2Tx, each with its own family of OLTs. This dissertation focuses on the OLT2T4 equipment, which belongs to the OLT2Tx family. The main difference among the OLT2Tx family members lies in the number of slots they provide. These slots are filled with cards, which are hardware boards that provide interfaces for connecting optical fibers. This distinction leads to different processing and traffic-switching capabilities for each OLT2Tx device.

The tests conducted by Altice Labs for OLT2Tx devices are written in Gherkin and subsequently interpreted by the Cucumber framework, where they are mapped to the Ruby language for execution. Creating a Gherkin language file involves using specific syntax and keywords. These keywords allow the mapping of Gherkin tests to Ruby. These Gherkin test files typically have the ".feature" extension.

Regarding mutation testing concepts, it is essential to note that this approach aims to enhance software quality by evaluating the effectiveness of a test set for that software. In mutation testing, mutants are generated by injecting faults into the software. The test set is then executed for each mutant to detect faults and "kill" the mutant. If a fault is not found, the mutant is considered to have "survived". As a result, the mutation score is calculated at the end of executing all mutants.

This approach involves both spatial and temporal costs due to the creation of numerous mutants and the execution of tests for each mutant. Two techniques, Mutant Reduction Technique and Execution Cost Reduction Technique, seek to mitigate these costs.

Chapter 3

State of the art

In this chapter, recent mutation testing works using the technique of testing are presented, what are their approaches and results. This section aims to learn from these methodologies and, if possible, apply them to adjust the concepts within the scope of this dissertation.

Initially, we begin explore Mutation testing methodologies that have been previously employed in black box testing scenarios. To begin, let's delve thoroughly examine the distinctions that exist between White Box and Black Box testing methodologies.

In White Box testing, also known as Structural testing, test cases are constructed based on a comprehensive understanding of the software's internal implementation, treating the software as a "white box" entity. Within this framework, the code's internal structure is analyzed to formulate test cases that trigger the execution of specific code segments, such as statements, program branches, or paths. Considering this, the primary objective is to analyze if the software follows the expected path for each test case. [Nidhra, 2012-06-30]

On the contrary, Black Box testing, also known as Functional testing, concentrates only on the input and output interactions of the software. The software itself is treated as a black box, implying that its internal mechanics are considered opaque. In this methodology, test case selection relies on the software's predefined requirements or specifications. Test cases are consequently constructed by inputting particular values into the software. The results are then evaluated based solely on the obtained outputs. Essentially, these tests verify whether the expected output corresponds to the output received for a given input, regardless of how the software operates. [Nidhra, 2012-06-30]

Equivalence partitioning, also referred to as equivalence class testing, stands as a significant software testing methodology that involves categorizing input values into distinct partitions based on their behaviors or functionalities. The primary goal of equivalence partitioning is to treat all values within a particular partition as equivalent, expecting them to exhibit the same behavior within the software. [Nidhra, 2012-06-30]

Subsequently, test cases are strategically designed to use only one test case as an representative value from each partition. Through the utilization of equivalence partitioning, testers can adeptly formulate test cases that effectively represent entire sets of inputs, eliminating the necessity to individually test each input. This practice not only facilitates achieving comprehensive test coverage but also minimizes the number of required test cases. This simplifies the testing procedure, leading to increased efficiency in the testing process. [Nidhra, 2012-06-30]

Given the advantage of this approach, limitations also arise, such as assuming that all values within a partition will be processed in the same way by the software. Furthermore, this approach is not used in isolation. It should be complemented with the Boundary Value Analysis technique. [Nidhra, 2012-06-30]

Boundary Value Analysis is a software testing technique that focuses on testing values at the boundaries or edges of input domains. This approach aims to identify and test values that are at the "boundaries" of input ranges, as these frequently tend to exhibit unique behavior or are more likely to lead to errors. [Nidhra, 2012-06-30]

The main goal of Boundary Value Analysis is to ensure the software's robustness and effectiveness.This is achieved by identifying any issues related to boundary conditions and ensuring that the software behaves as expected even in these critical scenarios. This brings a distinctive perspective that can make a significant difference, since testers frequently uncover defects that might not be found through typical test cases. [Nidhra, 2012-06-30]

Boundary Value Analysis also has limitations as it cannot be applied to Boolean and logical values, as well as other input types such as names. [Nidhra, 2012-06-30]

Transitioning to the study of approaches that apply mutation testing in the context of black box. Murnane and Reed [2001-08] explore how mutation testing, typically employed for White Box testing (analyzing internal code structures), can be effectively employed as a black box testing technique. Additionally, the article includes a comparison of the efficiency between the Equivalent Class Testing, Boundary Value Analysis and Mutation testing within the domain of black box tests. It is important to note that the first two methodologies are categorized as Black Box testing techniques. [Murnane and Reed, 2001-08]

For this purpose, Murnane and Reed [2001-08] adopts the approach of treating specifications as a linguistic framework, facilitating the mutation of terminal sets. A specification is characterized as a compilation of language elements that collectively outline a program's input and output behavior. Similar to how programming language syntax and semantics determine valid program forms, each data item in a specification can be regarded as a language element or a "terminal" element. The combination of these terminal elements is governed by production rules. Given this, the process of mutation involves the substitution of one terminal element with another, resulting in the creation of mutant specifications. These mutants can be categorized as "single-defect" when one element is substituted, and "double-defect" when two elements are substituted. As an example, given the terminal set <terminal1><terminal2><terminal3>, the authors create the mutant <terminal2><terminal2><terminal3> by replacing the second terminal element with the first. For the results, Murnane and Reed [2001-08] considers that if the program is capable of detecting the invalid element and returns the correct message, then it is considered as passed. [Murnane and Reed, 2001-08]

In this article, Murnane and Reed [2001-08] introduces new concepts related to the generation of inputs through the utilization of mutation operators. These inputs can syntactically valid or invalid. Syntactically valid inputs prompt the program to function as expected, where the replaced terminal element is considered syntactically equivalent to the one it substitutes. Syntactically invalid inputs uncover program inadequacies or faults and can be categorized as either correct or incorrect. A syntactically invalid correct input is one that the program acknowledges as having a syntactic error. The authors also suggest that generating double-defect mutants or utilizing production rule mutation might amplify testing, potentially leading to an excessive number of test cases and complicating the testing process. [Murnane and Reed, 2001-08]

Furthermore, the article highlights the significance of well-structured specifications in mutation testing. Additionally, the authors also suggest that integrating automatic test case generation could help alleviate some of the cost implications associated with mutation testing. [Murnane and Reed, 2001-08]

In conclusion, this article demonstrates that both mutation testing and boundary value and equivalence class testing produce similar test cases. However, when applied in a black box approach, mutation testing can identify distinct types of errors compared to equivalent class and boundary value testing. Given this, mutation testing can provide valuable insights into the correctness of program behavior, but its applicability may be constrained to particular types of specifications. [Murnane and Reed, 2001-08]

Moving to another approach, Jiang et al. [2008-02] introduces a contract-based mutation technique designed for testing black box components. This method involves employing Mutation Testing on the contracts that are provide alongside whit these components. Notably, this approach allows for the independent mutation of contracts without requiring access to the components' source code. [Jiang et al., 2008-02]

A mutation score is then employed to assess the effectiveness of the test set in eliminating these mutants. Similar to traditional Mutation Testing, Jiang et al. [2008-02] quantifies the contract mutation score of a test set by calculating the ratio of "killed" mutants to the total count of non-equivalent mutants. [Jiang et al., 2008-02]

To generate contract mutations, thus simulating potential misunderstandings of requirements or errors, Jiang et al. [2008-02] outlines contracts as extensions of the interfaces of the target component. To achieve this, the language utilized for writing these contracts is Enterprise Java Bean (EJB), which corresponds with the specification language of the targeted component. To elaborate further, Jiang et al. [2008-02] explains that within the EJB component context, "preconditions" outline the conditions necessary for proper operation of the component interface. Similarly, "postconditions" express characteristics of the outcomes resulting from the successful execution of the component interface. [Jiang et al., 2008-02]

Jiang et al. [2008-02] establishes the Contract Mutation Operators based on the "preconditions" and "postconditions" specified within component contracts. Jiang et al. [2008-02] approach centers on identifying errors arising from discrepancies between component contracts and the formal specification. For this purpose, Jiang et al. [2008-02] categorizes potential errors and define corresponding mutation operators for each type. [Jiang et al., 2008-02]

Given this, the mutated contracts originate from changes made to the original contracts. With this intention, Jiang et al. [2008-02] have defined which type of changes can be applied to the original contract. These changes primarily involve altering the relational and logical operators by substituting them with their opposite values.

To provide a clearer understanding of the concept, Jiang et al. [2008-02] presents the an instance of a contract definition within the context of an Automated Teller Machine, also known as an ATM. In this scenario, a specific component is requested and the relevant interfaces are "ValidatePin" and "Withdraw". Drawing from this context, e article subsequently proceeds to exemplify errors that may arise during the writing of contracts. For instance, within the contract of the ATM component, a potential error could be writing "@pre inputAmount < 2000" when it should be "@pre inputAmount <= 2000", unintentionally restricting withdrawals of exactly 2000 each time. [Jiang et al., 2008-02]

Based on the experimental results, a notable advantage highlighted by Jiang et al. [2008-02] is that this approach has the potential to generate fewer equivalent mutants compared to traditional mutation methods. Considering that these equivalent mutants require manual detection, the contract-based mutation approach can be recognized as a cost-effective strategy for mutation testing. Notably, the implementation of contract mutation operators in this approach demonstrates almost equivalent effectiveness when compared to traditional mutation operators. Additionally, this technique emerges as an effective solution for creating suitable test sets, aiming to minimize the costs linked to the testing of Black Box components. [Jiang et al., 2008-02]

In a different context, Lefticaru et al. [2011-03-01] proposes three techniques on how to test through a set of configurations, for example how to select which configurations to use. It also shows the respective results and conclusions. [Lefticaru et al., 2011-03-01]

The proposed techniques were the following: Use only the latest settings; Use the union of settings; Use the sequence of settings, for example perform multiset concatenation between the settings. [Lefticaru et al., 2011-03-01]

Thus, Lefticaru et al. [2011-03-01] introduces a configuration-based testing methodology to address the first two proposals. Consider the expected set of configurations $\mathrm{Conf} = \{C_1, \ldots, C_m\}$ and a constant k, representing the maximum number of steps needed to obtain all configurations $C_1, \ldots, C_m$ from the initial multiset. This approach consists of checking whether the system P can reach the given configuration C_i after at most k steps for each configuration $C_i \in \mathrm{Conf}$, $1 \leq i \leq m$. [Lefticaru et al., 2011-03-01]

As a response to the third proposal Lefticaru et al. [2011-03-01] applies configuration-based testing methodology sequence, which consists in joining the configurations of the same sequence from each multiset. Because this method considers combinations and not just single configurations it becomes a stronger test, as later confirmed by the author in his results. [Lefticaru et al., 2011-03-01]

Mutation tests will be performed on equipment employed in a network system for this dissertation. Therefore, it is beneficial to study previous works applying this technique to network systems. In accordance with this, Laurent et al. [2022-09-01] proposes a methodology to analyze the performance results from queueing network models, and the associated system costs required to meet them. Additionally, a set of appropriate mutation operators is proposed. [Laurent et al., 2022-09-01]

The mutation-based approach proposed by Laurent et al. [2022-09-01] is based on the original approach, but it introduces modifications, as larger or smaller queue size, different queueing strategy and others, in a Queueing Network model to better understand the system's performance. For the selection of operators for the Queueing networks, the author proposes, according to what is intended to be investigated in the article, the following configurations: Queue size, Queueing strategy and Queue parallelism. [Laurent et al., 2022-09-01]

Proceeding with the examination of recent applications of mutation testing approaches, Kontar et al. [2019-11] several techniques to use during mutation testing are analyzed. Firstly three techniques to generate higher order mutants were considered, Kontar et al. [2019-11] concluded that HOMs can find subtle errors that cannot be pointed out by FOMs and that HOMs are necessary because a high percentage of FOMs are equivalent, leading to wasted computing power and time. [Kontar et al., 2019-11]

Another point covered by Kontar et al. [2019-11] was the Mutant Clustering approach. According to the testing results, Kontar et al. [2019-11] affirmed that the Mutation Clustering method proves advantageous when applied to a test set that will be reused multiple times. It also points out cases, based on available resources, when the four approaches of mutation testing studied should be used. [Kontar et al., 2019-11]

The next approaches studied are related to techniques used to reduce the cost of mutation testing. In light of this, table 3.1 provides an overview of techniques used in 2018 for mutation testing. Additionally, it includes references to the respective articles associated with each technique.

Cost Reduction Technique	Primary goals and respective studies
Higher order mutation	Reducing the number of mutants: Abuljadayel and Wedyan
	Reducing the number of test cases or the number of executions: Gopinath et al. [2018-04]
Weak mutation	Reducing the number of mutants and reducing the number of test cases or the number of executions: Zhu et al. [2018-04]
Control-flow analysis	Reducing the number of mutants and automatically detecting equivalent mutants: McMinn et al. [2019-05] and Marcozzi et al. [2018-05]
	Executing faster: Zhu et al. [2018-04]
Evolutionary algorithms	Reducing the number of mutants: Abuljadayel and Wedyan

Table 3.1: Mutation testing techniques used in 2018. Adapted from [Pizzoleto et al., 2019-11-01].

In the context of approaches to reduce mutation testing costs, Petrović et al. [2022-10] introduced three essential ideas for conducting mutation tests that are computationally infeasible. Their proposals also apply in cases where existing solutions are used to accelerate mutation analysis. These strategies are: Mutation testing performed incrementally, focusing solely on changed code during the code review for mutant generation, rather than the entire codebase; Transitive mutation suppression, utilizing heuristics grounded in developer feedback. This approach filters mutants in two ways: by removing those considered irrelevant to developers and by restricting the number of mutants per line and per code review process; Mutants selected based on the historical performance of mutation operators. This selection process includes a basic random selection strategy that generates one mutant per line, considering information about unproductive nodes. Additionally, they implemented a targeted selection strategy that considers the past performance of mutation operators in similar contexts. The goal of these strategies is to create mutants with a higher chance of survival by eliminating irrelevant mutants and improving their quality [Petrović et al., 2022-10].

To validate the approach, the paper empirically validates the proposed approach by analyzing its effectiveness in a code review-based scenario used by over 24,000 programmers in over 1,000 projects. The results show that the proposed approach produces orders of magnitude fewer mutants and that context-based mutant filtering and selection improve mutant quality and actionability. Overall, the proposed approach represents a mutation testing framework that integrates seamlessly into the software development workflow and is applicable to industrial environments of any size [Petrović et al., 2022-10].

Another approach is provided by Pizzoleto et al. [2020-10], which introduces Similarity-based Mutation (SiMut). This technique aims to reduce the cost of mutation testing by reusing the results of previously performed mutation testing on similar programs. SiMut consists of six steps: Create a group of similar programs (G) to the untested program, select a cost reduction technique (C) for G, apply technique C to G, and obtain cost reduction measures and parameters (S). Then, retrieve parameters resulting from cost reduction S for group G. Next, apply technique C to the untested program based on S. And finally, evaluate the results, such as the mutation score, for experimental purposes [Pizzoleto et al., 2020-10].

SiMut aims to reduce the cost of mutation testing by making use of the similarities between programs. It achieves this by decreasing the number of mutants that need to be created and evaluated. In the SiMut approach, a test set from one program is employed to test a mutant of another program. The results of this testing are then used to identify potentially equivalent mutants in the other program that can be skipped [Pizzoleto et al., 2020-10].

However, SiMut's effectiveness can be influenced by various factors, including the characteristics of the programs being tested and the level of similarity between them. Consequently, in some cases, SiMut may not be as effective in reducing the cost of mutation testing. [Pizzoleto et al., 2020-10]

Still in the context of reducing the cost associated with mutation testing, [Mateo and Usaola, 2012-09] presents Bacterio as a useful tool to address this issue. As [Mateo and Usaola, 2012-09] explains, Bacterio is a tool designed to automate Java mutation testing tasks. To achieve this, Bacterio implements a set of mutation techniques, including Selective Mutation, Mutant Sampling, and HOM, aimed at reducing the overall cost of this technique. The author concludes that this tool is advantageous in helping testers perform mutation testing more efficiently, allowing for a more time-effective evaluation of the effectiveness of their tests. [Mateo and Usaola, 2012-09]

To understand if Mutation Testing improves testing practices, Petrović et al. [2021-05] addressed four critical questions. The first question pertains to the effects on testing quantity: How does continuous mutation testing affect the quantity of test code produced by developers? The second question relates to effects on testing quality: How does continuous mutation testing affect the survivability of mutants in a project? The third question concerns fault coupling: Are reported mutants associated with actual software failures? Can testing written based on mutants improve the effectiveness of testing for real software faults? And the fourth question is about mutant redundancy: Are mutants generated for a given line redundant? Is it sufficient in practice to select a single mutant per thread? [Petrović et al., 2021-05]

In their study, Petrović et al. [2021-05] utilized a vast dataset containing 15 million mutants. Their findings demonstrated that mutation testing is indeed associated with real failures that have practical significance. Additionally, presenting mutation testing as an approach to developers leads them to write more and higher-quality tests. [Petrović et al., 2021-05]

3.1 Summary

Black Box testing is an approach that treats the software as a Black Box as it lacks access to the source code. This technique tests software by injecting faults through manipulation of input values and verifies if the obtained output aligns with expectations. Two commonly used approaches for selecting input values are Equivalence Partitioning and Boundary Value Analysis, often applied in conjunction.

Mutation testing can be applied as a black box technique. One possible approach is treating specifications as a linguistic framework, which facilitates the mutation of terminal sets. This involves manipulating terminal elements, which represent data items in a specification. The concept includes valid and invalid inputs, where valid inputs lead the program to function as expected, and invalid inputs result in syntactic errors. Another approach is the contract-based mutation technique, which generates contract mutations to simulate faults.

Mutation testing designed for black box testing can offer valuable insights into the correctness of program behavior, which traditional mutation testing may not provide.

In recent applications of mutation testing approaches, efforts aim to reduce the cost through various strategies. Some of the approaches used include Mutation Clustering and HOM. Additionally, there are proposals involving Incremental Mutation Testing, Transitive Mutation Suppression, Mutants selected based on the historical performance of mutation operators, and SiMut.

In conclusion, after this study, some solutions for applying mutation testing in a black-box testing context were explored, but these were found to be inapplicable to our specific case. However, this exploration significantly contributed to our understanding of mutation testing. Additionally, no existing solutions for OLT mutation testing were identified, making the solution proposed in our dissertation unique.

Chapter 4

Proposed Approach

This chapter presents the proposed approach for solving the identified problem alongside with the applied methodologies.

4.1 Overview

In this section, we provide an overview of our proposal for automating OLT2T4 emulator mutation testing. The methodology employed to achieve our goal consists of creating mutants for Altice Labs' OLT2T4 and subsequently testing them using the existing test battery developed by Altice Labs for this equipment. As a result, the mutation score is obtained to evaluate the efficacy of the test set, and is provided with graphical illustrations, along with files that concatenate the survival status of the mutants, indicating whether they were "killed" or not.

Initially, since we do not have direct access to the source code, the OLT2T4's software is regarded as a black box. As a result, methodologies specifically designed for this type of testing will be employed. Additionally, for conducting mutations testing on the OLT2T4, Altice Labs has provided an OLT2T4 emulator, which functions as a virtual machine to simulate the OLT2T4 software.

In essence, our goal is to simulate fault scenarios, which leads to the modification of the OLT2T4's operational states, thereby generating the mutants. For that, ".feature" files are generated, written in Gherkin, similar to the test files created for OLT testing. They are executed using Cucumber in conjunction with Ruby. A more detailed explanation of the syntax of these files is provided in section 2.2.

The operational state of an OLT is determined by settings and parameters configured to ensure its correct functioning within a network. Using the steps outlined in the generated ".feature" files, we can modify these configurations, thereby changing its operational state. For instance, consider the step "configuring an interface's traffic shaper", which requires the following arguments: "slot", "port", and "max_bandwidth". These arguments can take on various values, including 1 and 2, "pon.", "eht.", and "lag.", and 0 and 1, respectively. By modifying these values, we can alter the configuration of the interface's traffic shaping and, consequently, the OLT's operational state.

The generation of these ".feature" files was supported by utilizing the existing test set at Altice Labs for the OLT2 family (OLT2Tx), combined with an spreadsheet file provided by Altice Labs. This spreadsheet file contain an extensive set of possible steps for the OLT, covering all types of keywords, including "Given," "When," "Then," "And," and "But". As shown in section 2.2, some steps include Data Tables where values for the respective step's arguments are introduced. In the provided spreadsheet file, the arguments for each step type are detailed, indicating their types and possible values.

For a clearer understanding, figure 4.1 illustrates an overview of the functioning of our system.

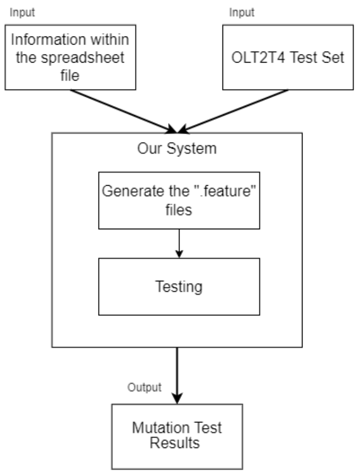


Figure 4.1: System Representation.

In summary, in our approach, the system receives two inputs provided by Altice Labs: the pre-established test set for the OLT2T4 created by Altice Labs itself and and the content within a spreadsheet file containing all the necessary details to generate ".feature" files for subsequent testing. As an output it returns the obtained results from the mutation testing. Further elucidation of the system will be provided as this chapter progresses.

4.2 Methodology

The process of our system to perform OLT2T4 emulator mutation testing is illustrated in figure 4.2.

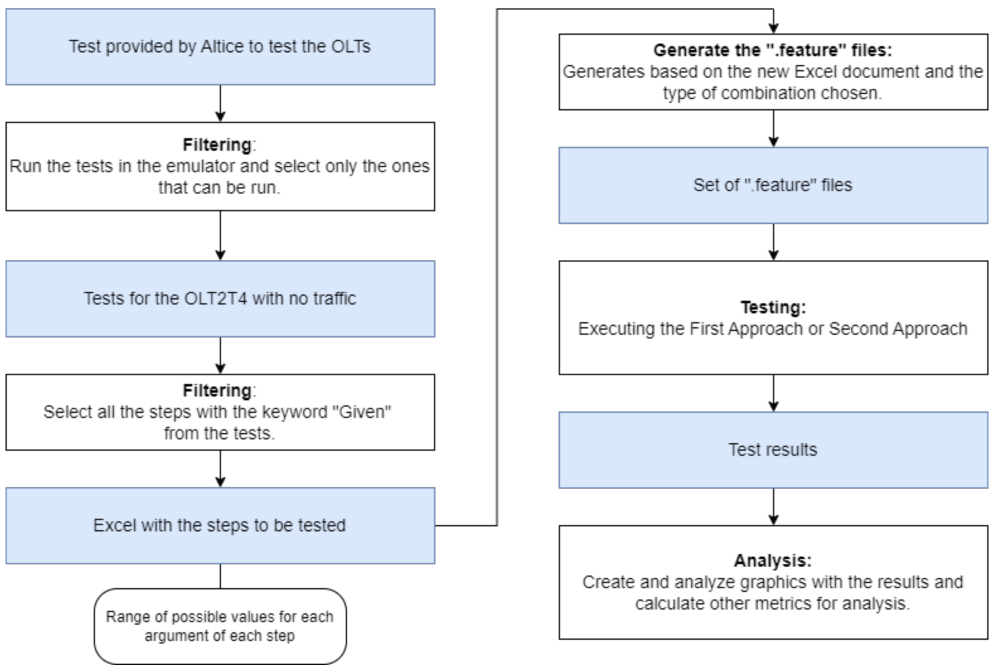

Test provided by Altice to test the OLTs
Filtering: Run the tests in the emulator and select only the ones that can be run.
Tests for the OLT2T4 with no traffic
Filtering: Select all the steps with the keyword "Given" from the tests.
Excel with the steps to be tested
Range of possible values for each argument of each step
Generate the ".feature" files: Generates based on the new Excel document and the type of combination chosen.
Set of ".feature" files
Testing: Executing the First Approach or Second Approach
Test results
Analysis: Create and analyze graphics with the results and calculate other metrics for analysis.

Figure 4.2: OLT2T4 emulator Mutation Testing Diagram.

As represented in figure 4.2 in order for our system obtain results, we begin by filtering the test set to include only those intended for the OLT2T4. Following this initial filtration, we proceed to apply a more specific filter, selecting the tests that can be executed in the OLT2T4 emulator. This is essential due to the absence of a traffic network and the exclusion of certain elements that are typically present in physical OLTs within the emulator environment. As a result of these considerations, the size of the test set is reduced to sixty-four tests.

Moving on to the next step of our approach is to define which step will be used to create the files ".feature". For that we focus on the steps with the keyword "Given". Since is in these steps where configuration setups are defined, as mentioned in section 2.2, only these are used for injecting faults into the OLT2T4. To determine which steps from the spreadsheet file are used for the keyword "Given" steps, our system identifies and collates all the steps with the "Given" keyword that existed in the test set previously filtered. For each step, the system queries the spreadsheet file provided by Altice Labs, extracting and incorporating the corresponding information into a new spreadsheet file. This new spreadsheet file compilation contains comprehensive details about each step, facilitating the testing process.

With the spreadsheet file generated, the system gains access to a comprehensive range of testing steps, each accompanied by their corresponding arguments, when applicable. For the purpose of fault injection, the system creates ".feature" files with only one step at a time, manipulating the values of arguments within steps that possess them. Derived from the spreadsheet file, the designated mutation operators have been established. These mutation operators, which are the values to be tested, cover hexadecimal, integer, and string data types of arguments. The selection process employed the Equivalent Class Partition approach and Boundary Values approach, as described by [Nidhra, 2012-06-30].

For the hexadecimal data type in the spreadsheet file, a maximum length (`max_len`) is determined. Consequently, the values selected for testing were established in the subsequent manner, as shown in Table 4.1.

Hexadecimal Test Cases
Testing the boundary values: random hexadecimal with lengths of `max_len - 1`, `max_len`, and `max_len + 1`
Testing a random hexadecimal with a length between 1 and `max_len`
Testing the largest hexadecimal with the maximum length: `F`'s multiplied by `max_len`
Testing the smallest hexadecimal: 0
Testing hexadecimals with upper and lower letters
Testing an empty value
Testing a null value

Table 4.1: Hexadecimal Test Cases.

For the integer data type in the spreadsheet file, a range of possible values [x, y] is given. Given this range, the values to be tested were determined as shown in table 4.2.

Integer Test Cases
Testing the boundary values: `x-1`, `x`, `x+1`, `y-1`, `y`, and `y+1`
Testing a random value between `x` and `y`
Testing the largest integer: 2147483647
Testing the smallest integer: -2147483648
Testing an empty value
Testing a null value

Table 4.2: Integer Test Cases.

We consider the value 2147483647 as the largest value for the integer data type, as it represents the maximum value that can be accommodated within 32 bits, including Python [Lemire, 2019-01-31]. Other considerations were taken to address the two cases of a lack of information in the values for integer data type arguments in the spreadsheet file. In one case, when the defined range appears as "N, ?", the "?" is changed to the value 2147483647. In the other case, when only one number "N" is given, it is defined that the range of values starts from 0 to "N".

For the string data type in the spreadsheet file, there are two ways of limiting values for the string argument. One way involves providing specific allowable words as potential values, and the other way establishes a maximum length (`max_len`) for the string. As a result, the values selected for testing were determined through the following process, shown in table 4.3.

Case	Strings Test Cases
1st Case	Testing the provided options
	Testing a string of length 10 with special characters (excluding pipe |)
	Testing numbers represented as strings with a length of 5
	Testing a string containing only a space
	Testing an empty value
	Testing a null value
2nd Case	Testing the boundary values: random strings with lengths 0, 1, 2, `max_len - 1`, `max_len`, and `max_len + 1`
	Testing a string of length 10 with special characters (excluding pipe |)
	Testing random numbers represented as strings with a length of 5
	Testing a string containing only a space
	Testing an empty value
	Testing a null value

Table 4.3: String Test Cases.

For two of the values to be tested, the chosen sizes were 10 and 5, as the objective is to test special characters and disguised numbers within strings, respectively, rather than focusing on the size.

Within the chosen values for each data type, we have both valid and invalid values. Valid values are those falling within the range of possibilities, while invalid values are those that fall outside this range. However, the valid value chosen for testing may not always be syntactically valid. Meaning that it might not align with the expected context or usage.

Based on this information, our proposed approaches consist of two instances for implementing mutation testing in the OLT2T4 emulator, with one instances complementing the other.

4.2.1 First Instance

With the spreadsheet file now containing all the essential information required to create ".feature" files, our initial instance involves the automatically generation of the desired quantity of these files. Subsequently, each of these files is executed, and for every execution, the system alters the state of the OLT2T4 emulator. Following this state manipulation, the complete set of tests provided by Altice Labs is executed, with the objective of determining whether these tests are capable of identifying any injected faults.

Throughout the execution of the ".feature" files and the corresponding tests, the system accumulates the resultant data. Then this various outcomes, are processed and presented in the form of graphical representations and statistics. This instance contributes to a comprehensive understanding of the achieved outcomes to further analysis.

To determine the combination of values for each argument within the same step, our initial instance involved generating all possible combinations. However, this instance resulted in an excessive number of mutants for testing, some of which could include multiple invalid value, that are values beyond the valid range or, in the case of strings, values not predefined. Consequently, during the result analysis phase, it would be challenging to discern which invalid value led to the detected fault during Mutation Testing.

In light of this, we decided to inject only one fault at a time across the ".feature" files. Therefore, for each argument of every step, all the invalid values are tested one at a time. However, when one argument contains invalid values, the others retain valid values. It is important to recall that the generated files test only one step at a time, resulting in multiple files with the same step if it includes arguments. Each of these files represents a different set of possible combinations of argument values.

After implementing this instance, it was observed that in most cases, if a ".feature" file contained an invalid value, the OLT emulator's software would detect it and subsequently prevent the alteration of the OLT emulator's state. As a result, we categorized this as the initial phase of Mutation Testing within the OLT emulator, with the goal of verifying whether the emulator's software could effectively detect all instances of invalid inputs. The second phase occurs when our system successfully modifies the OLT emulator's state, either by the ".feature" file containing exclusively valid values or due to the emulator's software not detecting an invalid value. In this scenario, our system executes the entire test set with the OLT emulator functioning in the modified state.

During the execution, as mentioned earlier, the system collects data. To facilitate this, the system generates an spreadsheet file, where each row corresponds to a tested step, representing an individual ".feature" file that has been tested. The information within each row includes the step being tested, the values inserted into its arguments (if applicable), which argument contained invalid values, and the outcomes of both the first and second phases of Mutation Testing.

In this way, during the initial phase, the system reports whether the OLT2T4 emulator software detected the invalid values within the ".feature" file or not. If the fault was caught, the system proceeds to the next ".feature" file. At the end of the Mutation Testing, this outcome can be further analyzed. On the other hand, if the file succeeded and the OLT2T4 emulator has transitioned to a new state, the second phase begins. In this phase, the report indicates whether the mutant was "killed" or not. Figure 4.3 demonstrates how the results are defined.

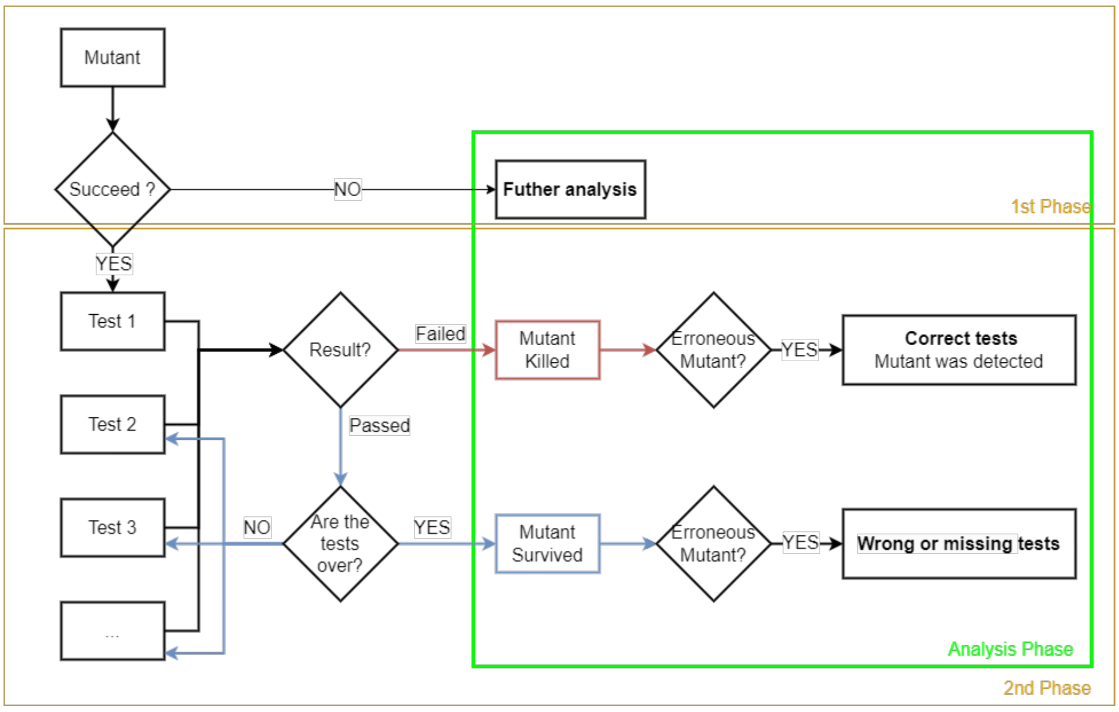


Figure 4.3: Results Definition Diagram - 1st Approach.

A mutant is considered "killed" if the test set was able to identify the injected fault by causing a failure during its execution. If the mutant was not "killed", it is considered to have "survived".

Based on the analysis, two possible conclusions can be drawn, depending on whether the mutant was "killed" or not, and if it was erroneous (meaning the values in the ".feature" file were invalid). If a mutant was "killed" and it is erroneous, then the tests have been correctly designed to detect that specific mutant. On the other hand, if the mutant "survived" and is erroneous, it indicates that there are issues with the test set, either due to incorrect or missing tests. Additionally, if the mutant is not erroneous, further analysis is required. Even though it might seem correct, it needs to be verified if the output is as expected, as the mutant might be created with valid values that are not semantically valid.

Given this, for the first instance of our approach, the following graphics are generated:

- **Total Results of the 1st Phase:** This graphic displays the outcomes of the first phase of the first instance, indicating how many ".feature" files from the previously generated set were successful in execution and how many failed.

- **Survival Status of Succeeded Mutants in the 2$^{\text{nd}}$ Phase:** This graphic illustrates the results of the second phase of the first instance of our approach. It shows, for the ".feature" files that succeeded in the first phase, how many mutants were "killed" and how many "survived".
- **Mutation Type of Killed Mutants in the 2$^{\text{nd}}$ Phase:** This graphic presents the types of mutants "killed" in the second phase, whether they are erroneous or not.
- **Mutation Type of Survived Mutants in the 2$^{\text{nd}}$ Phase:** This graphic indicates the types of mutants that "survived" in the second phase, whether they are erroneous or not.

As referred before, other results provided by our system are the statistics. Specifically, the mutation score, which represents the success rate of creating mutants, is displayed for the first phase of Mutation Testing. This is calculated using the following formula, where "mutants created" represents all the ".feature" files that were successfully executed in the first phase, progressing to the second phase of our approach.:

$$\textit{Success rate of creating mutants} = \frac{\textit{Mutants created}}{\textit{Total files ".feature"}} \times 100 \tag{4.1}$$

For the second phase, the success rate of killing mutants is shown, achieved through the following formula:

$$\textit{Success rate of killing mutants} = \frac{\textit{Mutants killed}}{\textit{Mutants created}} \times 100 \tag{4.2}$$

4.2.2 Second Instance

During the execution of the first instance, it was observed that the initial steps within the test files establish the proper state of the OLT2T4 emulator for test execution by utilizing the "Given" keyword. Consequently, in the first instance, when the state of the OLT2T4 emulator changes due to the introduction of a fault injection, it often returns to the correct emulator state. However, this must be tested in cases where the test does not adequately configure the OLT2T4 emulator to execute the test. In light of this, the second instance to OLT2T4 mutation testing gains significance as it aims to overcome this specific challenge.

In figure 4.4, the diagram illustrating the testing phases of both instances is shown.

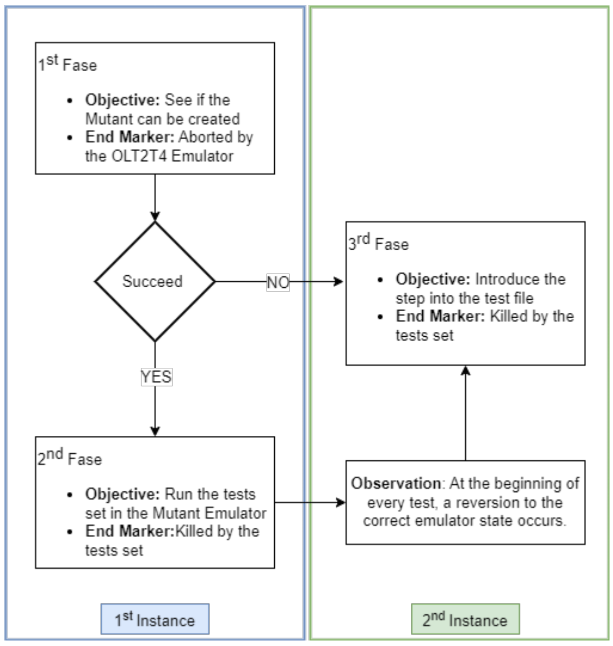

1st Fase
Objective: See if the Mutant can be created
End Marker: Aborted by the OLT2T4 Emulator
Succeed
NO
YES
2nd Fase
Objective: Run the tests set in the Mutant Emulator
End Marker:Killed by the tests set
3rd Fase
Objective: Introduce the step into the test file
End Marker: Killed by the tests set
Observation: At the beginning of every test, a reversion to the correct emulator state occurs.
1st Instance
2nd Instance

Figure 4.4: Testing Phases.

Within this context, the second instance involves the automatic generation of ".feature" files by our system. Similar to the first instance, these files contain the steps with the "Given" keyword that need to be tested, along with all the possible value combinations for the arguments, if applicable. However, instead of directly executing the ".feature" files, the faults generated for testing purposes are incorporated into each test file of the test set after all the existing steps with the "Given" keyword within the respective test file. A fault is a step with one of the combinations, if applicable, from various argument value combinations for different steps. If the step does not have arguments, the fault consists only of the step itself.

Regarding the generation of combinations in this instance, our initial efforts revolved around introducing invalid values as arguments for steps involving parameters. However, these attempts resulted in outcomes similar to those experienced in the first instance. The incorporation of invalid values led the OLT2T4 emulator's software to prevent the execution of most tests, making them incapable of proper execution. Recognizing this, the current instance exclusively focuses on testing valid values from the predefined set of test values, as demonstrated previously.

Given the nature of the generated test files, which center on individual steps, it is inevitable that for steps with arguments, multiple files will share the same step. Although, each file will incorporate distinct argument values, ensuring a comprehensive exploration of all established combinations.

In essence, this instance involves modifying and subsequently executing all the test files in the test set for each testing fault. During the execution of these new test files, the system records all the results. These results are then utilized to generate graphical representations and compile statistics for further analysis.

In light of this, the results provided by the system when applied in this instance determine whether the mutant was "killed" or not, along with the mutation score. For the initial outcome, figure 4.5 illustrates the procedure for defining the results.

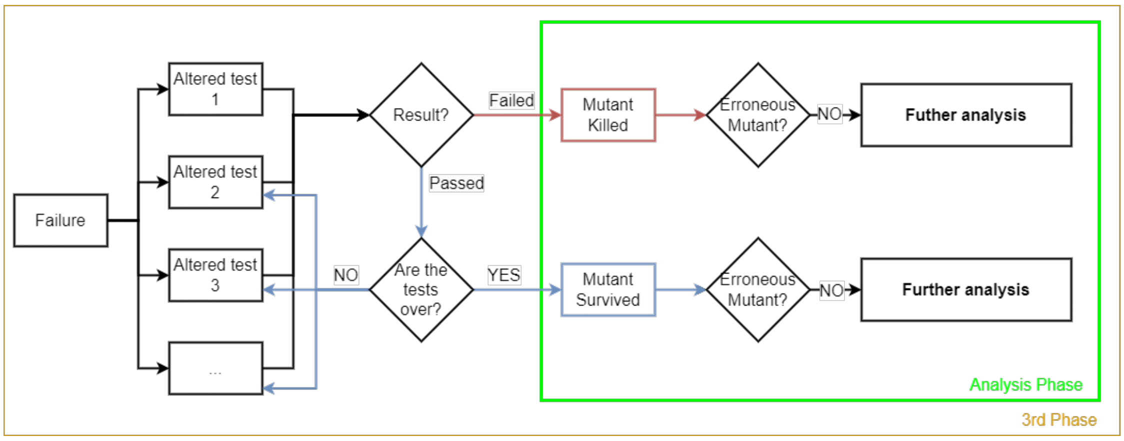


Figure 4.5: Results Definition Diagram - 2$^{\text{nd}}$ Approach.

Looking at figure 4.5, similar to the first instance, in this instance, the system identifies a mutant as "killed" when, during the execution of the test set for each tested step, one of the tests fails. Otherwise, if all tests within the test set execute without issues, the mutant is considered to have "survived". This procedure is performed for each tested failure.

Since only valid values are tested for the arguments in the steps, all the mutants generated within this instance are not erroneous. However, despite its apparent correctness, it is essential to verify whether the output matches expectations. Mutants might be created with valid values that are not semantically valid. further analysis is required to better comprehend the outcome, whether a mutant is "killed" or not.

Consequently, our system generates the following graphics:

- **Killed Mutants with Different Failures in the 3$^{\text{rd}}$ Phase:** This graphic illustrates how many mutants with different failures were "killed".
- **Total Killed Mutants in the 3$^{\text{rd}}$ Phase:** This graphic shows how many mutants were "killed" and how many "survived" from all the altered tests.
- **Killed Mutants for Each Failure in the 3$^{\text{rd}}$ Phase:** This graphic shows for each set of altered tests how many were "killed" and how many "survived".

In terms of the mutation score in this instance, the calculation involves three rates: the rate of total "killed" mutants, the rate of "killed" mutants with different failures and the success rate of "killed" mutants for each different failure.

The success rate of total "killed" mutants encompasses all the results obtained from the altered tests executed for each tested failure. These rate are determined using the following expression:

$$\textit{Success Rate} = \frac{\textit{Sum of mutants killed for each failure}}{\textit{Sum of mutants created for each failure}} \times 100 \tag{4.3}$$

On the other hand, the success rate of "killed" mutants with different failures specifically focuses on the results obtained for each unique failure tested. This rate is calculated using the following formula:

$$\textit{Success Rate} = \frac{\textit{Mutants with different failures killed}}{\textit{Total different failures tested}} \times 100 \tag{4.4}$$

Finally, the success rate of "killed" mutants for each different failure encompasses the results of the altered test for each failure. This is achieved using the formula below, where *N* represents the identification for each different failure, *mutants killed* represent the mutants "killed" for each different failure, and *mutants survived* represent the mutants "survived" for each different failure:

$$\textit{Success Rate}_N = \frac{\textit{Mutants Killed}_N}{\textit{Mutants Killed}_N + \textit{Mutants Survived}\ _N} \tag{4.5}$$

Chapter 5

Implementation

In this section, we detail the implementation of the previously described proposed approach in chapter 4. Additionally, we present the results of the OLT2T4 emulator mutation testing, along with its analysis.

5.1 Proposed Approach Development

To initiate the implementation of our approach, the first step involved setting up the environment. This encompassed installing VirtualBox, the virtual machine software, and configuring the OLT2T4 emulator from Altice Labs as a virtual image. Subsequently, network system configurations were adjusted to enable SSH protocol to the OLT2T4 emulator. Once SSH connection was established, we proceeded with the installation of Cucumber and Ruby to be able to execute the ".features" files. Additionally, to generate and execute the ".features" files automatically, Python is utilized, requiring its installation as well. A step-by-step guide to achieve that is shown in A. This collective setup formed the architecture illustrated in figure 5.1. With these components in place, the environment was fully prepared for executing the tests.

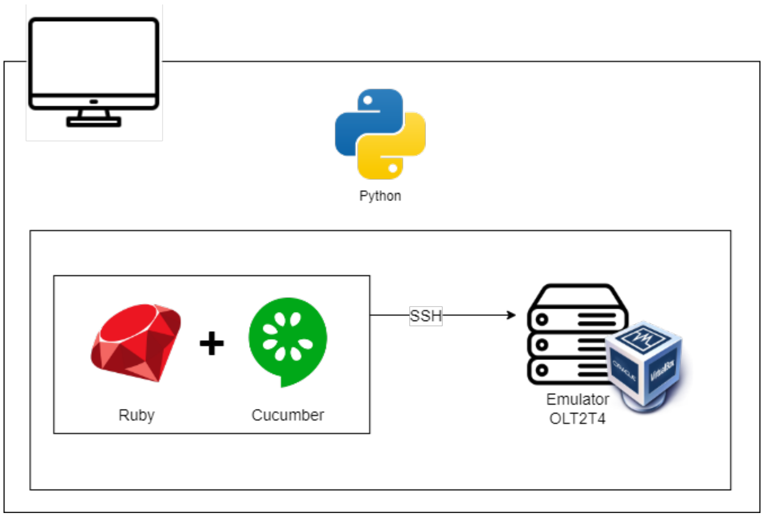
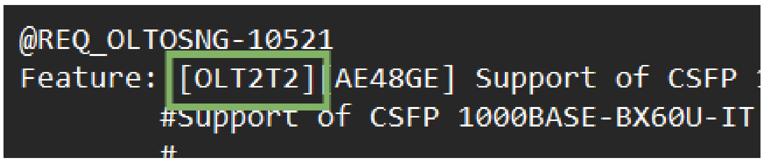


Figure 5.1: Test Environment Architecture.

As previously mentioned, our system utilizes a filtering process within the test set provided by Altice Labs. In order to specifically extract the tests for the OLT2T4, we rely on the fact that all the test files begin by specifying the equipment for which the test is intended. This characteristic is illustrated in the figure 5.2.



```
@REQ_OLTOSNG-10521
Feature: [OLT2T2][AE48GE] Support of CSFP 
        #Support of CSFP 1000BASE-BX60U-IT
        #
```

Figure 5.2: Equipment Identification in the Test File.

After collecting the tests for the OLT2T4, the second filtering step aims to retain only those tests compatible with the OLT2T4 emulator. Consequently, all these filtered tests are executed in the emulator, resulting in a refined test set comprising only executable tests.

To generate the spreadsheet file used in our system, a script is executed. This script collects all the steps that start with the keyword "Given" within the test files and compiles them into a new spreadsheet file. The spreadsheet file includes detailed information about the step tokens from the spreadsheet file provided by Altice Labs. Tables 5.1 and 5.2 illustrate the structure of the generated spreadsheet file, featuring two example steps used for testing, one with arguments and the other without, respectively.

Step	Parameter	Type	Is Range	Values	Max Length	Observation
an interface's MAC limit exists with	slot	integer	TRUE	1,2		
an interface's MAC limit exists with	port	string	FALSE	pon.,eth.		
an interface's MAC limit exists with	admin	string	FALSE	enable,disable		
an interface's MAC limit exists with	maximum	integer	TRUE	1,?		
an interface's MAC limit exists with	action	string	FALSE	none,limit		
an interface's MAC limit exists with	trap	string	FALSE	enable,disable		

Table 5.1: Structure of the Spreadsheet for a set with arguments.

Step	Parameter	Type	Is Range	Values	Max Length	Observation
A user connects to the OLT equipment using the CLI interface						

Table 5.2: Structure of the Spreadsheet for a set without arguments.

The following step in our system is to generate the ".feature" files. This is accomplished by executing a new script for both the first and second instances of our approach. Within the script, we determine whether we are running the first instance or the second instance, in order to select the appropriate code to execute. It is important to define the instance because they are different in terms of their methodologies and outcomes. Additionally, we organize the results into different folders for each instance to facilitate access to them.

When generating these files, it is essential to consider that after the execution of a ".feature" file, the state of the OLT2T4 emulator is normally reset. However, since this dissertation intends to keep the state of the OLT2T4 emulator altered, a new hook was added. Specifically, a "@not_hook" was introduced to ensure that ".feature" files with this hook do not restore the emulator's state. Consequently, this hook is added to all the generated ".feature" files. For a better understanding of what a hook is, please refer to section 2.2.

This script generates the ".feature" files for each step to be tested. Each generated file contains only one step along with its arguments and their respective values, if applicable. Given this, the script calculates all the different possibilities for the values of the arguments for the selected test steps when applicable. This possible combinations generation is different for each instance, as discussed in section 4.2. Subsequently, the generated ".feature" files are stored in a folder. An example of the generated file is shown below:

```
Feature: Mutant3

Background:
Given a user connects to the OLT equipment using the "CLI" interface

@not_hooks
Scenario:
    Given an interface's MAC limit exists with
    | slot | port | admin  | maximum | action | trap   |
    | 1    | pon. | enable | 1       | none   | enable |
```

This script also generates an spreadsheet file in Comma-Separated Values (CSV) format report where all the results of the OLT2T4 emulator mutations testing are filled. The script begins by filling in the CSV with information related to each ".feature" file. This CSV file is slightly different for each instance. Table 5.3 presents the structure of the resulting CSV for the first instance, including an example for one ".feature" file with arguments and another without:

Mutant	Parameters	Was a Fault	What's the Fault	Was Created	Was Killed
2	[]	[]	[]		
3	['slot', 'port', 'admin', 'maximum', 'action', 'trap']	[1, 0, 0, 0, 0, 0]	(0, 'pon.', 'enable', 1, 'none', 'enable')		

Table 5.3: Structure of the CSV file for the first instance.

Given this, the CSV for the first instance of our approach includes the identification of the ".feature" file by its number, the arguments of the step tested, which argument has an invalid value (represented by the number 1), the values for those arguments, and leaves two entries with empty values to be further filled with the results obtained from the execution of the corresponding ".feature" file. For the second instance, the CSV file has a similar structure to the CSV of the first instance. The difference is that in this one, the "was created" column is removed.

For the execution of the ".feature" files along with the test set, another script is employed. To be able to execute the files in the OLT, it is necessary to ensure that the SSH connection with the OLT has been established. To perform the execution, the following command is used in the appropriate location:

```
cucumber path_file_to_execute.feature > path_file_result.txt
```

Once again, in this file, it is necessary to identify which instance is being executed since the methodology is different, as explained in chapter 4.

For the first instance, as mentioned earlier, the OLT2T4 emulator's state is restored after running the tests. Consequently, for the first instance during the second phase, after executing each test from the set provided by Altice Labs, it is necessary to run the same ".feature" file generated once more to inject the fault into the OLT2T4 emulator again.

For each instance of our approach, this script generates result files in different folders, including the outcomes obtained by the Cucumber framework after the execution of each ".feature" file. From these result files, the script updates the CSV file with the Mutation Testing results, as defined in section 4.2.

In the first instance, during the second phase of our OLT2T4 emulator Mutation Testing approach, the test set is executed for each individual mutant created, and the results obtained for each test within the test set for each mutant are stored in separate CSV files.

In the second instance of our approach, a similar process is followed. As described in section 4.2, the test files within the test set are modified and executed sequentially. For each fault introduced, a CSV file is generated containing all the results obtained by the Cucumber framework for the tested alterations.

Therefore, for the first instance of our approach, we fill the CSV file, as illustrated in table 5.3, with "passed" or "failed" in the "was created" column and "YES" or "NO" in the "was killed" column. For the second instance of our approach, we fill the "was killed" column in the CSV file with "YES" or "NO".

Finally, the same script is executed to generate graphical representations, as explained in chapter 4, of the Mutation Testing results along with the statistics stored in a text file for each instance of our approach. In the case, the file related to the first instance, includes the Success Rate of Creating Mutants (first phase) and the success rate of killing mutants (second phase), along with the identification of the files that were "killed" and those that "survived", specifying whether each is an erroneous mutant or not (second phase). For the second instance, the file comprises the rate of "killed" mutants with different failures, the rate of total "killed" mutants and the success rate of "killed" mutants for each different failure (third phase).

5.2 Mutation Testing Results and Analysis

As mentioned earlier, while our system automatically conducts Mutation Testing, it keeps a record of all the necessary data. In this case, the data is stored in a text file, which contains the results obtained by the Cucumber framework for each ".feature" file, including those generated and the corresponding tests. From this file, the system updated the CSV file, which concatenates all the results for each instance of our approach (for the file structure, please refer to the section before 5.1).

Using the CSV file that compiles the results gathered from each ".feature" file, our system generates graphical illustrations of the Mutation Testing results obtained from each instance of our approach. Test results are based on what was defined in chapter 4.

As previously mentioned and explained in section 4.2, the first instance of our approach generates the following graphics: Total Results of the 1st Phase, Survival Status of Succeeded Mutants in the 2nd Phase, Mutation Type of Killed Mutants in the 2nd Phase and Mutation Type of Survived Mutants in the 2nd Phase.

Additionally, for the first instance of our approach, the success rate of creating mutants and the success rate of killing mutants are displayed, along with the identification and specification of the type of mutants, whether erroneous or not, "killed" or "survived".

For the second instance of our approach, it should be noted that for the same failure, which is associated a step with the keyword "given" containing a single possible value combination for the arguments, all the tests from the test set are altered. This results in the creation of several altered tests with the same failure. Consequently, our system generates the following graphics, as earlier discussed and explained in section 4.2: Killed Mutants with Different Failures in the 3rd Phase, Total Killed Mutants in the 3rd Phase and Killed Mutants for Each Failure in the 3rd Phase.

In addition to these graphics, the success rate of "killed" mutants with different failures, the success rate of total "killed" mutants, and the success rate of "killed" mutants for each different failure are presented.

With the aim of testing our system, we employed OLT2T4 emulator Mutation Testing in each instance of our proposed approach, with a stronger focus on the first instance. For the first instance, it was executed for 412008 previously generated ".feature" files. Within these files, seven different steps were used. However, since one of these steps has thirteen arguments, it generates many possible combinations of argument values. On the other hand, for the second instance, it was applied to 15 previously generated ".feature" files, meaning that it was tested with 15 different steps. This process resulted in the execution of 960 ".feature" files when applied to the test set. As a result, the graphics obtained are displayed below to provide a comprehensive understanding. For the first instance, the following graphics were obtained:

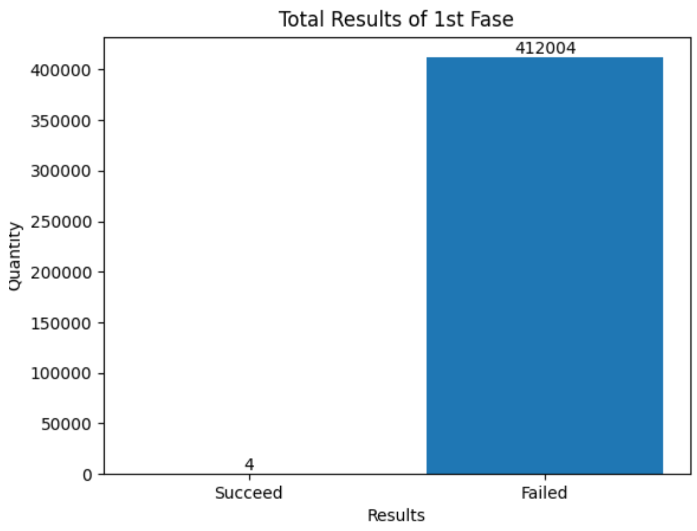


Figure 5.3: Total Results of the 1st Phase Graphic.

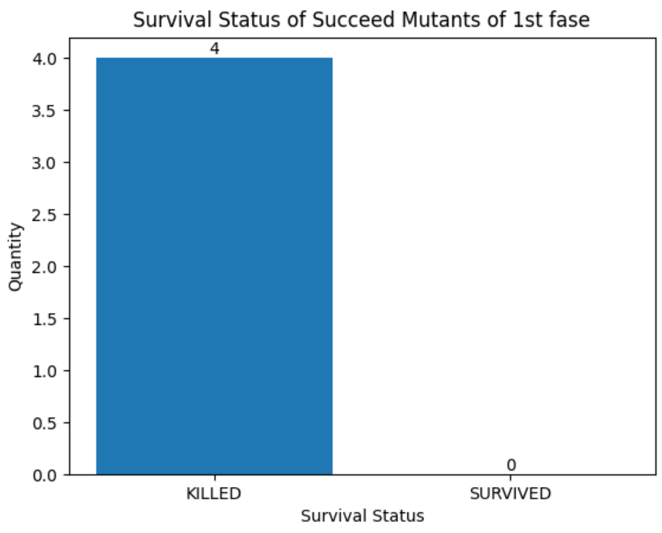


Figure 5.4: Survival Status of Succeeded Mutants in the 2nd Phase Graphic.

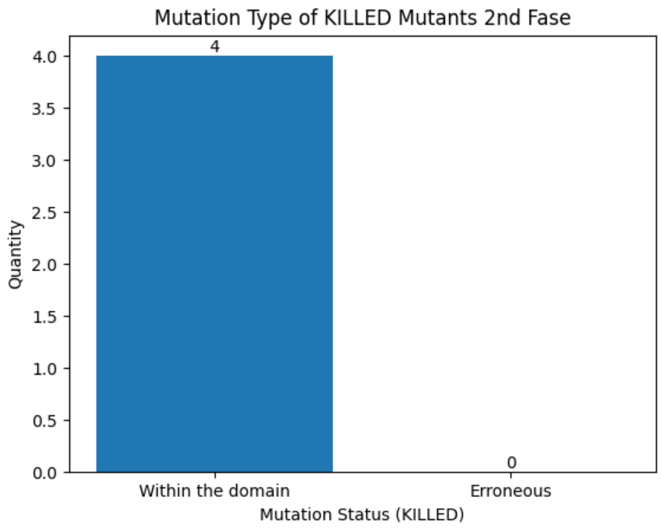


Figure 5.5: Mutation Type of Killed Mutants in the 2nd Phase Graphic.

Since no "survived" mutants were obtained, the graphic for "Mutation Type of Survived Mutants in the 2nd Phase" is not shown as it is empty.

The Success Rate of Creating Mutants obtained was 0.000971% in 412008 previously generated ".feature" files, as evident from the graphics. Additionally, the mutants "killed" were the ones created with the ".feature" files 1, 2, 1091, and 1184, all of which are non-erroneous mutants. All these files that advanced to the second phase of our proposed approach contain steps that do not have arguments.

From the results obtained, it is clear that the majority of the tests were caught by the OLT software due to invalid values in the arguments of the steps. However, for each of the existing ".features" files with steps that have no arguments, the test set was executed, and one of the tests failed during the execution. Since these steps have no arguments and, therefore, no invalid values, further analysis is required based on the results files that contain the output provided by the Cucumber framework, along with the CSV file that concatenates the results obtained for each ".feature" file for this particular instance.

Moving to the second instance, in this case, the graphics obtained were as follows:

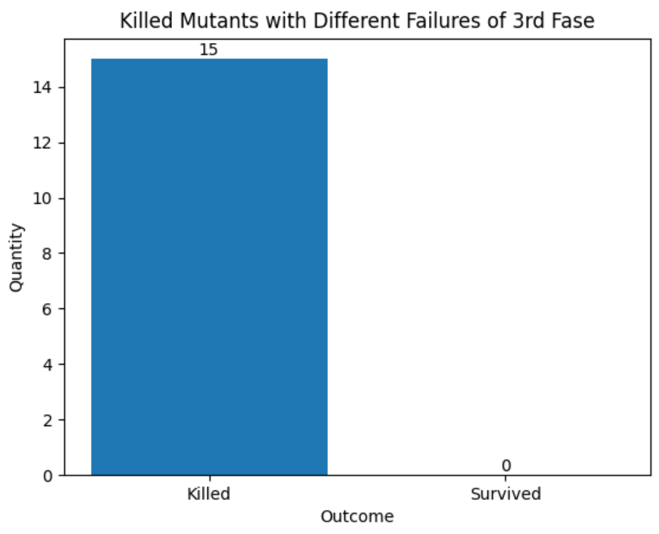


Figure 5.6: Killed Mutants with Different Failures in the 3rd Phase Graphic.

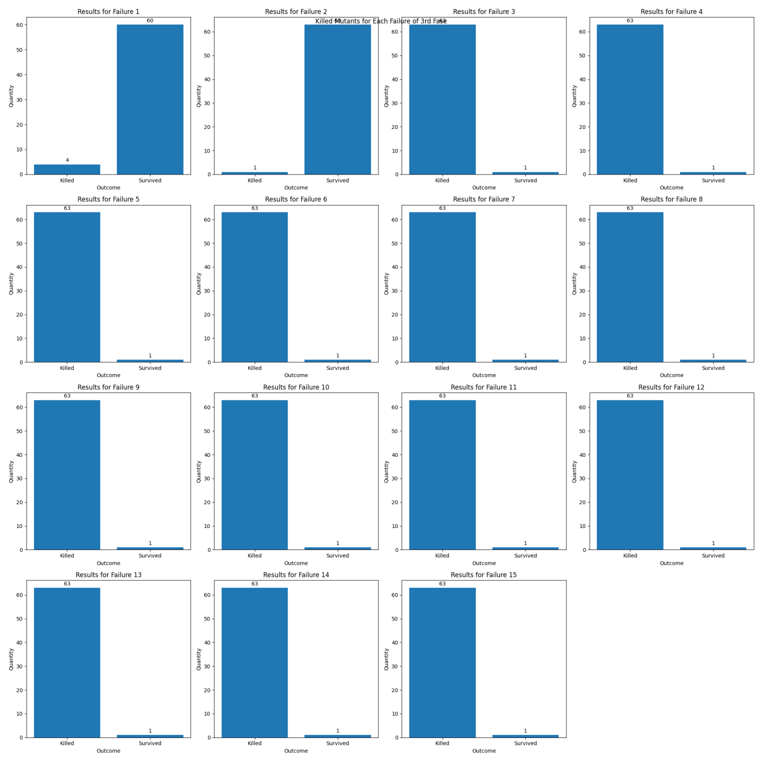


Figure 5.7: Killed Mutants for Each Failure in the 3rd Phase Graphic.

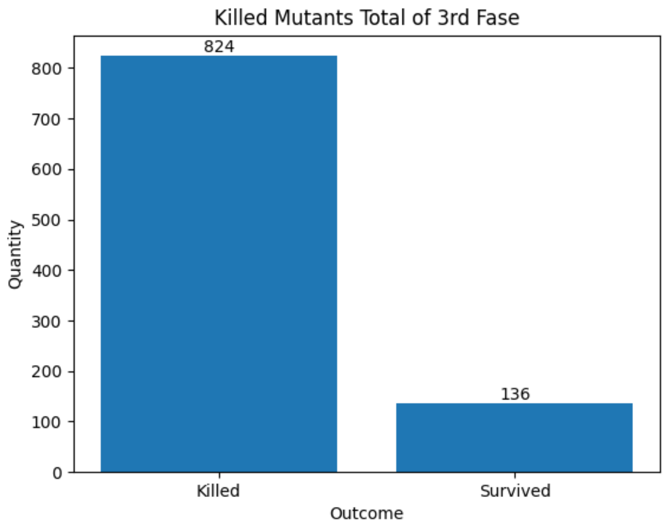


Figure 5.8: Total Killed Mutants in the 3rd Phase Graphic.

The success rate of "killed" mutants with different failures was 100.00% in 15 mutants, and the success rate of total "killed" mutants was 85.83% in 960 mutants, as illustrated in the graphics. Additionally, the success rate of "killed" mutants for each different failure is shown in table 5.4.

Failure	Success Rate of Killed Mutants
1	6.25% in 64 mutants
2	1.56% in 64 mutants
3	98.44% in 64 mutants
4	98.44% in 64 mutants
5	98.44% in 64 mutants
6	98.44% in 64 mutants
7	98.44% in 64 mutants
8	98.44% in 64 mutants
9	98.44% in 64 mutants
10	98.44% in 64 mutants
11	98.44% in 64 mutants
12	98.44% in 64 mutants
13	98.44% in 64 mutants
14	98.44% in 64 mutants
15	98.44% in 64 mutants

Table 5.4: Success Rate of Killed Mutants for Different Failures.

Analyzing the graphics, it is observed that all the failures were detected during the execution of the test files. This indicates that all the mutants created were "killed" by the test set provided by Altice Labs. After analyzing the CSV file generated for each failure, it becomes clear that, except for the first and second failures, the same test consistently succeeded. Consequently, all the other tests were able to "kill" the mutant. The first two failures are steps without arguments, while all the others have arguments. Therefore, the difference in the results is based on whether the step has arguments or not. However, in both cases, it is necessary to conduct further analysis. This analysis aims to determine whether the values introduced in the arguments are valid and if the behavior of the software aligns with its intended function.

Even though the test set caught almost all failures, the next step is to enhance the test set with the results we obtained to ensure better quality when testing the OLT.

In consideration of these results, it is possible to conclude that our system is executing mutation testing for the OLT2T4 emulator automatically and correctly. Even though it operates automatically, our system effectively achieves the desired results, making it advantageous for Altice Labs. This capability extends to future OLTs, enabling Altice Labs to enhance the testing phase in terms of time. Additionally, it helps in identifying weaknesses and errors in the software, leading to eventual improvements.

However, with all approaches, there comes a cost, and in our case, it is the time required for the automated execution of OLT mutation testing. Due to the numerous failures that need to be tested, this process takes a considerable amount of time, spanning several weeks, to complete.

Chapter 6

Final Considerations

In this chapter, the final considerations of this dissertation are presented. It includes a conclusion and suggests future work.

6.1 Conclusion

Software testing is a crucial step in the development of software to ensure its quality. One of the many approaches to performing software testing is Mutation Testing. This approach aims to evaluate the effectiveness of tests for a specific software. When applied in conjunction with traditional techniques, it enhances the testing phase by identifying vulnerabilities and weaknesses in the software, leading to a more thorough analysis of software quality.

The Mutation Testing approach involves injecting faults into the software by creating mutants using predefined mutation operators. For each mutant created, the existing tests for the specific software are executed to determine if they can detect the previously injected faults. If the tests detect a fault, the mutant is considered "killed"; otherwise, it is deemed to have "survived". This approach results in a mutation score, representing the test set's capability to detect faults. However, this technique is time-consuming, as it requires manual creation of all mutants.

Altice Labs is an international telecommunications company that focuses on developing and innovating telecommunications technologies and solutions. Altice Labs develops hardware and software, including OLTs, which is a modern PON technology. An OLT is a crucial network equipment for delivering services to clients.

Altice Labs tests the software of the OLT to deliver high-quality equipment to its clients. These tests are written in the Gherkin meta-language, using keywords like "Given," "When," and "Then" to describe the behavior of the software system under test. Test files have the extension ".feature". To execute these tests, the Cucumber testing framework is used, which interprets the ".feature" files and maps them into Ruby language to execute the code.

The challenge of this dissertation is to propose an approach, in partnership with

Altice, that applies mutation testing techniques to Altice Labs' OLT software automatically. This approach aims to reduce the time consumed by the testing process. In this case, the testing technique is applied in a black box manner since the source code of the OLT software is not provided. Therefore, our approach injects faults into the OLT software by altering its operational state.

Our proposed approach focuses on two instances. In each instance, ".feature" files are generated to alter the operational state of the OLT. In the first instance, the previously generated ".feature" files are executed, and whenever possible, the existing test set provided by Altice Labs is sequentially executed. In the second instance, for each ".feature" file, the contents inside it are integrated into all the tests within the test set and executed sequentially. During Mutation Testing, the results are stored for further analysis. In order to validate our approach, this document presents the tests that were conducted for this. With these results, it was possible to conclude the efficiency of our approach in improving software testing.

In conclusion, this proposed approach provides valuable insights for the testing phase and contributes to the fields of software testing and software engineering by offering a time-efficient approach for mutation testing.

6.2 Future Work

In our proposed approach, only combinations with one invalid value or with only valid values were considered for the first and second instances, respectively. However, as we started by creating all possible combinations of valid and invalid values, some code for this case has already been developed. As future work, we consider the possibility of continuing and improving the implementation of this aspect. On the other hand, another interesting future work would be to consider, in each instance, the execution of the combinations used in the other instance. With the existing code, as it is already possible to generate these combinations, only minor adjustments would be necessary.

Additionally, our approach only considers three data types, limiting the possible tests performed on the OLT. Although these data types are the most commonly used, others could be employed. Therefore, implementing this approach for other data types in future work would be interesting and would enhance our system.

References

Cucumber reference - cucumber documentation, a. URL `https://cucumber.io/docs/cucumber/api/?lang=ruby#hooks`.

Cucumber reference - cucumber documentation, b. URL `https://cucumber.io/docs/cucumber/api/?lang=ruby#steps`.

Cucumber reference - cucumber documentation, c. URL `https://cucumber.io/docs/cucumber/api/?lang=ruby#tags`.

Environment variables - cucumber documentation, d. URL `https://cucumber.io/docs/cucumber/environment-variables/`.

Huda Saleh Abbas and Mark A. Gregory. The next generation of passive optical networks: A review. *Journal of Network and Computer Applications*, 67: 53–74, 2016-05. ISSN 10848045. doi: 10.1016/j.jnca.2016.02.015. URL `https://linkinghub.elsevier.com/retrieve/pii/S1084804516000989`.

Anas Abuljadayel and Fadi Wedyan. International journal of intelligent systems and applications(IJISA). *International Journal of Intelligent Systems and Applications(IJISA)*, 10(1):34. URL `https://www.mecs-press.org/ijisa/ijisa-v10-n1/v10n1-5.html`.

Xiangying Dang, Dunwei Gong, Xiangjuan Yao, Tian Tian, and Huai Liu. Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm. *IEEE Transactions on Software Engineering*, 48(6):2141–2156, 2022-06. ISSN 1939-3520. doi: 10.1109/TSE.2021.3052987. Conference Name: IEEE Transactions on Software Engineering.

Rahul Gopinath, Björn Mathis, and Andreas Zeller. If you can't kill a supermutant, you have a problem. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 18–24, 2018-04. doi: 10.1109/ICSTW.2018.00023.

Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full c language. *Proc. Int. Conf. Testing: Academic and Industrial Conf. Practice and Research Techniques*, 2008-08-01. doi: 10.1109/TAIC-PART.2008.18.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011-09. ISSN 1939-3520. doi: 10.1109/TSE.2010.62. Conference Name: IEEE Transactions on Software Engineering.

Ying Jiang, Shan-Shan Hou, Jin-Hui Shan, Lu Zhang, and Bing Xie. AN APPROACH TO TESTING BLACK-BOX COMPONENTS USING CONTRACT-BASED MUTATION. *International Journal of Software Engineering and Knowledge Engineering*, 18(1):93–117, 2008-02. ISSN 0218-1940, 1793-6403. doi: 10.1142/S0218194008003556. URL `https://www.worldscientific.com/doi/abs/10.1142/S0218194008003556`.

Gerd Keiser. *FTTX Concepts and Applications*. John Wiley & Sons, 2006-02-06. ISBN 978-0-471-76909-5.

K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 1991.

Karam Al Kontar, Joumana Naji, Freddy Demiane, Salma Sobeh, and Ramzi Haraty. A survey on mutation testing approaches. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, pages 1–7, 2019-11. doi: 10.1109/CHILECON47746.2019.8987448.

Thomas Laurent, Paolo Arcaini, Catia Trubiani, and Anthony Ventresque. Mutation-based analysis of queueing network performance models. *Journal of Systems and Software*, 191:111385, 2022-09-01. ISSN 0164-1212. doi: 10.1016/j.jss.2022.111385. URL `https://www.sciencedirect.com/science/article/pii/S0164121222001078`.

Raluca Lefticaru, Marian Gheorghe, and Florentin Ipate. An empirical evaluation of p system testing techniques. *Natural Computing*, 10(1):151–165, 2011-03-01. ISSN 1572-9796. doi: 10.1007/s11047-010-9188-y. URL `https://doi.org/10.1007/s11047-010-9188-y`.

Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation*, 29(1):1–12, 2019-01-31. ISSN 1049-3301, 1558-1195. doi: 10.1145/3230636. URL `https://dl.acm.org/doi/10.1145/3230636`.

Yu-Seung Ma and Sang-Woon Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, 2016-05-01. ISSN 0164-1212. doi: 10.1016/j.jss.2016.01.007. URL `https://www.sciencedirect.com/science/article/pii/S0164121216000078`.

Nicola Marchetti. Towards 5th generation wireless communication systems. *ZTE Communications*, 13:11–19, 2015-03-01.

Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 456–467, 2018-05. doi: 10.1145/3180155.3180191. ISSN: 1558-1225.

Pedro Reales Mateo and Macario Polo Usaola. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 646–649, 2012-09. doi: 10.1109/ICSM.2012.6405344. ISSN: 1063-6773.

Phil McMinn, Chris J. Wright, Colton J. McCurdy, and Gregory M. Kapfhammer. Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas. *IEEE Transactions on Software Engineering*, 45(5):427–463, 2019-05. ISSN 1939-3520. doi: 10.1109/TSE.2017.2786286. Conference Name: IEEE Transactions on Software Engineering.

T. Murnane and K. Reed. On the effectiveness of mutation analysis as a black box testing technique. In *Proceedings 2001 Australian Software Engineering Conference*, pages 12–20, 2001-08. doi: 10.1109/ASWEC.2001.948492. ISSN: 1530-0803.

S. Nidhra. Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications*, 2:29–50, 2012-06-30. doi: 10.5121/ijesa.2012.2204.

A.J. Offutt and S.D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994-05. ISSN 00985589. doi: 10.1109/32.286422. URL `http://ieeexplore.ieee.org/document/286422/`.

Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 910–921, 2021-05. doi: 10.1109/ICSE43902.2021.00087. ISSN: 1558-1225.

Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2022-10. ISSN 1939-3520. doi: 10.1109/TSE.2021.3107634. Conference Name: IEEE Transactions on Software Engineering.

Alessandro V. Pizzoleto, Fabiano C. Ferrari, Lucas D. Dallilo, and Jeff Offutt. SiMut: Exploring program similarity to support the cost reduction of mutation testing. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 264–273, 2020-10. doi: 10.1109/ICSTW50294.2020.00052.

Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019-11-01. ISSN 0164-1212. doi: 10.1016/j.jss.2019.07.100. URL `https://www.sciencedirect.com/science/article/pii/S0164121219301554`.

Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, 2009. ISSN 1099-1689. doi: 10.1002/stvr.392. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.392`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.392.

rita-c-felix. Central office. URL `https://www.alticelabs.com/products/central-office/`.

Mayank Singh and Viranjay M. Srivastava. Extended firm mutation testing: A cost reduction technique for mutation testing. In *2017 Fourth International*

Conference on Image Information Processing (ICIIP), pages 1–6, 2017-12. doi: 10.1109/ICIIP.2017.8313788.

Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017-02-17. ISBN 978-1-68050-496-5. Google-Books-ID: fA9QDwAAQBAJ.

Qianqian Zhu, Annibale Panichella, and Andy Zaidman. An investigation of compression techniques to speed up mutation testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 274–284, 2018-04. doi: 10.1109/ICST.2018.00035.

Appendices

Appendix A

Setting Up the Environment: A Step-by-Step Guide

This appendix presents the steps to follow for setting up the environment.

1. Install VirtualBox and WinSCP.
2. Insert the OLT emulator image provided by Altice Labs into the VM.
3. Configure the VM's network.
   - 3.1. Configure the host-only network adapter in the VM by going to *File* > *Tools* > *Network Manager* and adding a new network with the address 192.168.56.1 and network mask 255.255.255.0.
   - 3.2. Enable one of the host-only network adapters by going to *Settings* > *Network* > *Adapter 2*, enable it, and associate it with the host-only adapter named after the network created in step 3.1.
   - 3.3. Configure one of the network interfaces in the OLT emulator with the address 192.168.56.101.
   - 3.4. Establish an SSH connection with the emulator.
4. Copy the workspace folder from the OLT emulator to the computer using WinSCP.
5. Install Ruby 2.4 language.
6. Install Cucumber framework by running the command "gem install cucumber -v 3.1.0."
7. Install Altice Labs' gems-swauto by running the command "gem install gem_name."
8. Install the necessary gems until it is possible to run the dummy test.
   - 8.1. Run the dummy test in the workspace folder by executing the command "cucumber -t @dummy_test."

8.2. Install the gem requested in the output of step 7.1 by running the command "gem install requested_gem." (Usually, the requested gems include rspec, json, json-schema, net-ssh, net-ssh-telnet, netsnmp, xml_simple, gyoku, and snmp).

8.3. Repeat Step 7.1 followed by Step 7.2 until it's possible to run the dummy test.