INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

Depuração: Localização Automática de Falhas

CARLA MANUELA FERREIRA SAMPAIO
julho de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Debugging: Automated Fault Localization

Carla Sampaio

A dissertation submitted in partial fulfillment of the requirements for the degree of Master in Informatics, Specialisation Area of Software Engineering

Supervisor: Dr. Alberto Sampaio
Co-Supervisor: Dra. Isabel Sampaio

Evaluation Committee:
President:

Members:

Porto, July 1, 2021

Dedicatory

The journey to write this thesis would have not been possible without the people that stood by me during all these years. First of all, I would like to thank my supervisors Alberto Sampaio and Isabel Sampaio for their help and for making me believe I was up to the challenge. Special thanks to all of my friends, for all the times I bored them with my problems and for all the help they provided me.

Thanks to my family, specially my parents, that gave me the chance to make them proud and to repay all they have done for me. Finally, thanks to my boyfriend, that stood behind me through out this process always being very calm and for his attempt to make me calm either and making me believe I could finish my thesis and work at the same time.

Abstract

Developing a bug free software its impossible without the task of debugging. Due to the increasing complexity and size of software, it gets harder to find and fix these errors which lead to higher development costs. Therefore, software engineers try to either prevent the errors or debug them as fast as possible. Debugging is not only inevitable, but so difficult that it often consumes more time than creating the bogus piece of software. For this reason, researchers invested a considerable amount of effort in developing fault localization techniques and tools for supporting various debugging tasks.

Fault localization has been an active area of research, leading to the creation of several tools, such as Aletheia. Spectrum-based Fault Localization (SFL), the technique behind the outlined tool, is a statistical debugging technique that relies on code coverage information. However, there is always a way to improve the effectiveness of this tool.

This thesis, proposes an approach to overcome some challenges presented in the fault localization tool Aletheia and its integration with the similarity coefficient. This approach, tries to overcome the null symmetry when comparing the statements covered in the passed tests to the uncovered statements in the failed tests.

Keywords: Automated Debugging, Spectrum-based Fault Localization, Fault Localization, Debugging, Bug

Resumo

Desenvolver um software livre de erros é impossível sem efetuar depuração. Devido ao crescimento da complexidade e ao crescimento do software, torna-se mais difícil encontrar e corrigir esses erros que levam a custos de desenvolvimento mais elevados. Portanto, os engenheiros de software tentam evitar ou depurá-los o mais rápidamente possível. A depuração não é só inevitável,mas tão difícil que geralmente consome mais tempo do que criar o erro em si. Por esta razão, pesquisadores investiram uma quantidade considerável de esforço no desenvolvimento de técnicas de localização de erros e ferramentas para suportar várias tarefas de depuração.

A localização de erros tem sido uma área ativa de pesquisa, levando à criação de várias ferramentas, como o Aletheia. A localização de falhas baseada em espectro (SFL), a técnica por trás da ferramenta descrita, é uma técnica de depuração estatística que se baseia em informações de cobertura de código. No entanto, sempre existe uma maneira de melhorar a eficácia desta ferramenta.

Esta tese, propõe uma abordagem para superar alguns desafios apresentados na ferramenta de localização de falhas Aletheia e sua integração com o coeficiente de similaridade. Essa abordagem tenta superar a simetria nula ao comparar as instruções cobertas nos testes aprovados com as instruções descobertas nos testes que falharam.

Acknowledgement

I'd also like to thank my friends that are always online to check my spelling errors and giving me the strength while playing League of Legends to finish my thesis.

Contents

List of Figures **xv**

List of Tables **xvii**

List of Algorithms **xvii**

List of Source Code **xix**

List of Acronyms **xxi**

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem . . . 1
1.3 Objectives . . . 2
1.4 Expected Results . . . 2
1.5 Value Analysis . . . 2
1.6 Document Structure . . . 2

2 Background **5**
2.1 Debugging . . . 5
2.1.1 Definition . . . 5
2.1.2 Debugging Tools . . . 6
2.2 Definitions . . . 6
2.2.1 Error, Fault and Failure . . . 7
2.2.2 Dependability Means . . . 7

3 State Of Art **9**
3.1 Fault Localization . . . 9
3.1.1 Spectrum-based Fault Localization (SFL) . . . 9
Program Spectra . . . 10
Metric-based Techniques . . . 12
Statistics-based Techniques . . . 13
Artificial Intelligence-based Techniques . . . 14
Program Dependence-based Techniques . . . 15
Execution Models . . . 15
3.1.2 Model-Based Diagnosis . . . 16
3.1.3 Other References . . . 16
3.2 Delta Debugging . . . 17
3.3 Program Slicing . . . 17
3.3.1 Static Slicing . . . 17
3.3.2 Dynamic Slicing . . . 18

3.4 Tools . . . 19
3.4.1 Aletheia . . . 20
3.4.2 AutoFLox . . . 21
3.4.3 FLAVS . . . 21
3.4.4 GZoltar . . . 21
3.4.5 MZoltar . . . 22
3.4.6 Tarantula . . . 22
3.4.7 UnitFL . . . 22
3.5 Conclusion . . . 23

4 Value Analysis **25**
4.1 Business and Innovation Process . . . 25
4.1.1 Opportunity Identification . . . 27
4.1.2 Opportunity Analysis . . . 27
4.1.3 Idea Generation and Enrichment . . . 27
4.1.4 Concept Definition . . . 27
4.2 Value Analysis . . . 27
4.2.1 Customer Value . . . 28
4.2.2 Perceived Value . . . 28
4.2.3 Value Proposal . . . 30
4.3 Analytic Hierarchy Process . . . 30
4.3.1 Pairwise Comparisons . . . 31

5 Design and implementation **35**
5.1 Aletheia . . . 35
5.1.1 Data Generation . . . 35
5.1.2 Fault Localization . . . 37
5.1.3 Exam score calculation . . . 38
5.2 Symmetry Coefficient . . . 39
5.3 Integration of Aletheia with Symmetry Coefficient . . . 40
5.4 Improvements To Aletheia . . . 41

6 Evaluation **43**
6.1 Methodology . . . 43
6.2 Metrics . . . 43
6.3 Investigation Hypotheses . . . 44
6.4 Evaluation Indicators and Information Sources . . . 44
6.5 Evaluation Results . . . 44
6.5.1 Aletheia . . . 44
6.5.2 Example Test Software . . . 45
6.5.3 Fault Localization Techniques Applied . . . 45
6.5.4 EXAM score Results . . . 46
6.5.5 Open Source Results . . . 46

7 Conclusion **47**
7.1 Achieved Requirements . . . 47
7.2 Future Work . . . 48

Bibliography **49**

A AHP Javascript Code **55**

B Hit spectra generated for example Application **57**

C Method to calculate the suspiciousness ranking **59**

D Code extract from Aletheia to calculate the rank of each statement(C#) **61**

E Code extract from Aletheia to perform the Exam score calculation in all the fault localization files (C#) **63**

List of Figures

2.1 Debugging Processes 6
2.2 Cause And Effect Relationship 7

3.1 SFL Techniques 9
3.2 SFL Input 10
3.3 Aletheia Components 20

4.1 The Innovation Process 25
4.2 The New Concept Development 26
4.3 Hierarchical Decision Tree 31

5.1 Aletheia - Steps to Generate Hit Spectra 36
5.2 Hit Spectra Output CMD 37
5.3 Aletheia - Steps to Calculate the Suspiciousness Ranking 38

List of Tables

3.1 Coverage matrix for GreatestNumber.cs (3.2) 11
3.2 Ranking metrics for fault localization. Based on (J. A. Jones, M. J. Harrold, and John Stasko 2002) 13
3.3 The Performance of Standalone Techniques on All 357 Faults (based on (Zou et al. 2021)) 23
3.4 Comparative Analysis of Automated Fault Localization Tools 23

4.1 Value for the costumer based on (Woodall 2003) 29
4.2 The fundamental scale for pairwise comparisons (Saaty 2008) 32
4.3 AHP criteria comparison 32
4.4 AHP comparison for accuracy 33
4.5 AHP comparison for reliability 33
4.6 AHP comparison for performance 33
4.7 Ranking matrix for every criterion 33
4.8 Ranking matrix 34

5.1 Fault Localization Techniques with Symmetry Coefficient 40

6.1 Fault Localization Techniques Added to Aletheia 45
6.2 Result EXAM score 46

List of Source Code

3.1 Greatest Number (C#). 11
3.2 Static Slicing - Greatest Number (C#). 17
3.3 Dynamic Slicing - Greatest Number (C#). 18
5.1 Code extract from Aletheia with the Tarantula fault localization technique with Symmetry Coefficient (C#). 40

List of Acronyms

AHP	Analytic Hierarchy Process.
AI	Artificial Intelligence.
COTS	Commercial Off-the-shelf.
DD	Delta Debugging.
FEI	Front End of Innovation.
FFE	Fuzzy Front End.
GCC	GNU Compiler Collection.
HSFL	Historical Spectrum Based Fault Localization.
IDE	Integrated Development Environment.
MBD	Bodel-based Diagnosis.
NCD	New Concept Development.
NPD	New Product Development.
PCEG	Probabilistic Cause-effect Graphs.
PDG	Procedure Dependence Graph.
PMD	Potential Memory-address Dependence.
SFL	Spectrum-based Fault Localization.

Chapter 1

Introduction

This chapter presents the context, problem and objectives throughout this dissertation's work. It also details, briefly, the expected results, the value analysis and the approach taken. Finally, the document structure is also presented, where the contents of each chapter are summarized.

1.1 Context

Debugging helps developers to find and fix bugs in the program, although it can be a particularly time consuming task requiring intensive human activity. As program complexity grows, it becomes increasingly more difficult for the developer to perceive and keep track of all possible errors, failures and outcomes of the program. Even with the help of breakpoints and tracing, there is still a lot of time being spent finding and fixing the bugs. It is estimated that the attempts to reduce the number of faults in software consume up to 50% of the development and maintenance effort (M. Wen et al. 2019).

Due to the complexity and increasing scale of software today, manually locating faults when failures occur is rapidly becoming impossible, and consequently, there is a strong request for techniques that can guide software developers to the locations of faults in a program with minimal human intervention. Automated debugging aims to decrease the time spent and complexity in the debugging process. Its main objective is to find where the bug is without any human effort whatsoever.

In order to improve debugging process, accurate and efficient fault localization techniques have the potential to greatly reduce the overall effort of software development.

1.2 Problem

Whenever an error is detected, it is necessary to start a debugging process. One of the objectives of research in the area of software engineering has been to automate the debugging process. The automatic repair process includes the task of locating the error. This is considered a task of great difficulty in the debugging process (Adragna 2008).

There is a need for tools which narrow down the most likely fault locations. There tools must find fault locations with as little user interaction as possible and the results computed must be beneficial so that such tools appeal to developers. Currently, there are several techniques and tools for automated debugging, but all of them can be improved in order to find a wider use in the software industry (Parnin and Alessandro Orso 2011).

In this dissertation, the main problem at hand is to improve an automated debugging technique in order to be more reliable.

1.3 Objectives

The objectives of this dissertation are:

1. Identify and analyze the existing automated debugging techniques and tools they use.
2. Analyze the pros and cons of the tools and techniques already in use, including as success rates of localization of faults of each one, with the intention of maximizing a possible final solution, extracting, in this way, the positive aspects and avoiding the mistakes that were once made.
3. Improve an automated debugging tool in order to provide more accurate results
4. Implement the symmetry coefficient to Spectrum-based Fault Localization (SFL) automated debugging techniques and compare results between them

1.4 Expected Results

The expected result developing this dissertation is to improve a reliable and efficient automated debugging technique. To analyze the method reliability, the implemented solution will be evaluated through comparisons with the original technique and/or tool according to several criteria, such as, for example, time to find errors and efficiency in finding defects.

1.5 Value Analysis

Developers often find debugging process very overwhelming and exhausting. Despite the creation of integrated or unit tests, there is still an generous amount of bugs in the software. Automated debugging is being improving over the years to help reduce debugging time and improve software quality (Parnin and Alessandro Orso 2011).

The technique improved from the project of this dissertation could be used to help developers localize more errors in software and reduce debugging time. There is a more in-depth approach to value analysis in section 4.

1.6 Document Structure

This document has the following structure:

1. **Introduction** - Chapter 1: in this chapter, the context of this dissertation's work is presented, in addition to the problem at hand and its objectives. Further, the expected results are presented, followed by a small value analysis. Finally, a recommended approach is presented.
2. **Background** - Chapter 2: in this chapter, there are described the concepts used in this work.
3. **State of Art** - Chapter 3: all the automated debugging techniques are analyzed and described in order to explore the existent related work.

4. **Value Analysis** - Chapter 4: in this chapter, there is evaluated how this work can bring value to the customer. This include items such as **value for the customer** and **perceived value**. The New Product and Process Development is presented here in addition to the analytic hierarchy process.
5. **Design and Implementation** - Chapter 5: this chapter presents the improvements made to an automated debugging tool and the Implementation of SFL automated debugging techniques with the symmetry coefficient
6. **Evaluation** - Chapter 6: this chapter is dedicated to the evaluation of the solution.

Chapter 2

Background

This chapter provides context about important definitions and concepts within the topic of debugging.

2.1 Debugging

The computing pioneer Maurice Wilkes described his 1949 encounter with debugging like: "As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs" (Moore, Benson, and Budd 2007). This section presents the definition of debugging followed by the concept of debugger.

2.1.1 Definition

Debugging is the process of finding and fixing errors in the software (Beller et al. 2018). These errors or defects are referred to as "bugs", hence the term "debugging". Bugs are very common to find, and in severe cases it may cause the program to stop. When a bug is found, the developer must debug the program in order to fix the problem.

Traditionally debugging a program consists primarily of stopping the program under certain conditions and then examining the state of the program stack and the values stored in program variables (Wambugu and Njeru 2017). The developer can stop the execution of the program by setting breakpoints. Through breakpoints, the developer can inspect the code to find out whether the program is functioning as expected. It allows the developer to inspect variables, memory and logs.

The debugging process can be designed by 4 stages (Hernández 2002): reproduce, diagnose, fix and reflect (see Figure 2.1). First, the developer starts by trying to reproduce the bug. Secondly, the developer makes a diagnose of the bug analyzing its gravity, priority and impacts. After fixing the bug, the developer applies new measures that guarantee that the problem doesn't happen again, like tests and creating more resilient code.

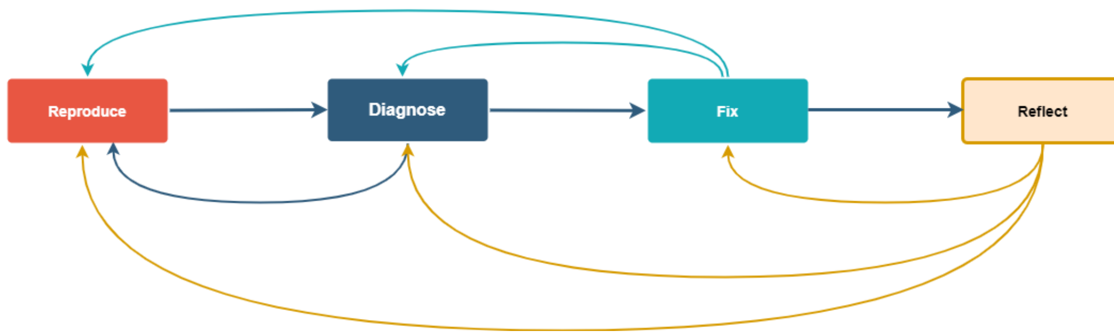

Figure 2.1: Debugging Processes

There are several techniques applied to the debugging:

- Reverse Engineering: Analysis of logs, outputs, stack traces, among others.
- Try/Catch: When the contours of the problem are not sure and several hypotheses are posed, followed by its progressive elimination.
- Revert the code: The code is reverted gradually until the problem is found.
- Isolate the problem: A sample project can be created only with relevant code.
- Tracing: The most used technique, printing variables and/or actions in several moments.

2.1.2 Debugging Tools

Debugging Tools, also known as debugger, consists in a program used to test and find errors in the developed software. Its a tool that is typically used to allow the developer to view the execution state and data of another application as it is running (Beller et al. 2018).

When the program stops, the debugger shows the position on the code where the bug occurred. Typically, debuggers offer a query processor, a symbol resolver, an expression interpreter and a debug support interface at its top level. Debuggers also offer more sophisticated functions such as running a program step by step (single-stepping or program animation), stopping (breaking) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of variables. Some debuggers have the ability to modify program state while it is running.

It may also be possible to continue execution at a different location in the program to bypass a crash or logical error. A debugger can be offered in an isolated way (e.g. GNU Compiler Collection (GCC)), merged in the Integrated Development Environment (IDE) (e.g. XCode, Visual Studio, Eclipse) or integrated in another tool (e.g. Chrome Developer Tools).

2.2 Definitions

The term "bug" were first mentioned in Ada Lovelace's notes on Charles Babbage's analytical engine in 1840. Through out the time, there where present several uses of this term to indicate errors in computers and software. In this section there are present de concepts used to describe faults, errors and failures and dependability means.

2.2.1 Error, Fault and Failure

Many terms are used in the software engineering literature to describe a malfunction: notably **fault**, **error**, and **failure**, among others. The following figure 2.2 illustrates the flow from error to failure.

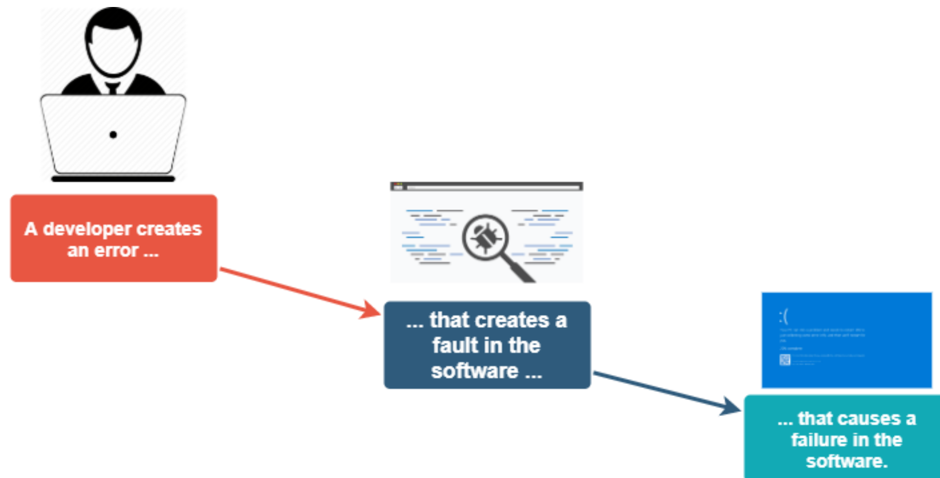

Figure 2.2: Error, Fault and Failure

An **error** is an human action that produces an incorrect result ("ISO/IEC/IEEE International Standard" 2017).

Fault (bug or defect) consists in a flaw in a component or in any part of the software that can cause the system to behave incorrectly (e.g., an incorrect statement) (Bourque and Fairley 2014). A **fault** only becomes a **failure** when it reaches the final user and it causes the system to behave incorrectly.

A **failure** occurs when a component or system behaves in a way that is not expected ("ISO/IEC/IEEE International Standard" 2017). For example, when the program is expected to show a screen with user information and instead it shows a blue screen with an error message.

Failures in the software are usually caused by **faults**. Thus testing can reveal **failures**, but it is the **faults** that can and must be removed.

2.2.2 Dependability Means

Systems that aim to avoid all types of failures by preventing faults and recovering from them, typically have dependability requirements. The definition of dependability is the ability of a system to avoid service failures that are more frequent and more severe than acceptable (Laprie 1995). In other words, dependability is the justifiable amount of trust, one can put in the system to deliver the correct service.

Dependability may be viewed according to different properties, which enable the attributes of dependability to be defined:

- **Availability** - The property that refers to readiness of the service
- **Reliability** - The property continuity of service delivery

- **Safety** - Non-occurrence of catastrophic consequences on the environment
- **Confidentiality** - Non-occurrence of unauthorized disclosure of information
- **Integrity** - Non-occurrence of improper alterations of information
- **Ability** - Aptitude to undergo repairs and evolution

When referring to security, an additional attribute has great importance **confidentiality**, that means the absence of unauthorized disclosure of information. Dependability Means is composite by a set of methods that can be classified into:

- **Fault Tolerance** - Ensures that the presence of faults does not lead to system failure (Dubrova 2013).
- **Fault Prevention** - Set of techniques attempting to prevent the introduction or occurrence of faults in the system in the first place. Its achieved by quality control techniques during the specification, implementation, and fabrication stages of the design process (Dubrova 2013).
- **Fault Removal** - Set of techniques that aims to detect the presence of faults, and then to locate them and remove them (Dubrova 2013).
- **Fault Forecasting** - Estimates how many faults are present in the system, possible future occurrences of faults and their consequences (Dubrova 2013).

Chapter 3

State Of Art

B. W. Kernighan and P. J. Plauger (1978) state that debugging is not only inevitable, but so difficult that it often consumes more time than creating the bogus piece of software in the first place. On that account, researchers invested a considerable amount of effort in developing automated techniques and tools for supporting various debugging tasks.

This chapter presents the techniques and tools under the context of automated debugging.

3.1 Fault Localization

Fault localization is the activity of identifying the exact locations of program faults (W. Wong and Vidroha Debroy 2009). Fault localization methods are used by developers in order to focus their debugging efforts on a subset of program entities (such as statements or predicates) that are most likely to be causes of the error (Landsberg et al. 2015).

Fault localization can be categorized into spectrum-based fault localization (SFL, correlating failures with abstractions of program traces), and model-based diagnosis (MBD, logic reasoning over a behavioral model) (R. Abreu, P. Zoeteweij, and A. J. C. van Gemund 2009).

3.1.1 SFL

SFL is a light-weight statistical debugging approach , that identifies program elements more likely to contain faults. This approach uses the results of test cases and their corresponding code coverage information to estimate the possibility of each program component (e.g., statements) of being faulty (Pearson et al. 2017). This section presents the program spectrum data used and details the SFL techniques (see Figure 3.1).

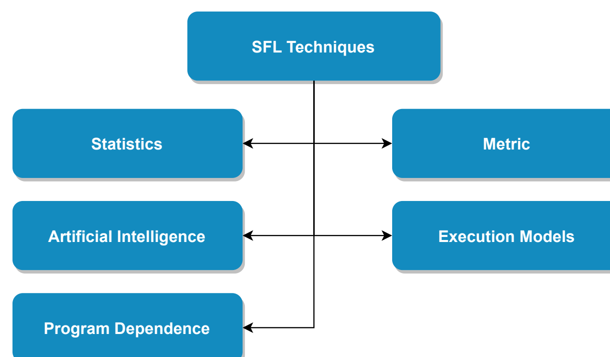

Figure 3.1: SFL Techniques

Program Spectra

SFL uses information from program entities executed by test cases to indicate entities more likely to be faulty. It relies on program execution information (program spectra) from previous runs (passed and failed) to match up the software components with the observed failures and determine the odds of each component being faulty. Program spectra, also known as code coverage, can be defined as a set of program entities covered during test execution. The different covered entities include statements blocks, predicates, function or method calls.

The input of SFL is constituted by the program spectra and previous runs (see Figure 3.2). The program spectra is expressed in terms of M × N activity matrix A.

$$
M \text{ spectra} \quad \overset{N \text{ components}}{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix}} \quad \overset{\text{errors}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}}
$$

Figure 3.2: SFL Input - based on (R. Abreu, P. Zoeteweij, and A. J. C. van Gemund 2007)

The program spectra of M runs constitute a binary matrix, whose columns correspond to N different components (e.g., functions, statements). The error vector has a length of M is constituted by the information in which runs an error was detected. In SFL one measures the statistical similarity between the error vector e and the activity profile column A_{*j} for each component c_j. It essentially consists in identifying the part whose column vector resembles the error vector most. To achieve this, SFL relies of similarity coefficients like $_{ST}$ (Tarantula) or $_{SO}$ (Ochiai).

Considering the fault program 3.2, that has six program entities $\langle N_1, N_2, N_3, N_4, N_5, E \rangle$, where E is the specification.

```
int GreatestNumber() // N1
{
    int num1, num2, num3, biggest;

    if (num1 > num2)
    {
        if (num1 > num3)
        {
            biggest = num1; // N2
        }
        else
        {
            biggest = num1; // N3 ( Bug ! )
        }
    }
    else if (num2 > num3)
        biggest = num2;  // N4
    else
        biggest = num3; // N5

    Console.Write("The number {0} is the greatest.
   ", biggest);
    Console.ReadKey();
}
```

Listing 3.1: Greatest Number (C#).

The program fails to satisfy the specification num1 < num3. The reason for the failure is the bug at N_3, which should be an assignment from num2 to greatest, instead of num1 to greatest. To locate the fault, there was collected coverage data from ten test cases M_1 to M_6. One of them fail and five pass. The coverage matrix for these test cases is given in table 3.1.

Table 3.1: Coverage matrix for GreatestNumber.cs (3.2)

Input		**N_1**	**N_2**	**N_3**	**N_4**	**N_5**	**E**

num1 = 1;
num2 = 2;
num3 = 3;
	M_1	1	1	0	0	0	1

num1 = 1;
num2 = 3;
num3 = 2;
	M_2	1	0	0	0	0	1

num1 = 2;
num2 = 1;
num3 = 3;
	M_3	1	0	1	0	0	0

num1 = 2;
num2 = 3;
num3 = 1;
	M_4	1	0	0	0	1	1

num1 = 3;
num2 = 1;
num3 = 2;
	M_5	1	0	0	0	0	1

num1 = 3;
num2 = 2;
num3 = 1;
	M_6	1	0	0	1	0	1

The coverage matrix 3.1 uses the following nomenclature: 1 indicates if a program entity N was executed in tests, and 0 indicates if the program entity N was not executed in tests. If the specification (E) was also hit that means the program exited successfully. This coverage matrix could be used to calculate, for example, the Ochiai ranking metric (see table 3.2). In other words, if we focus on the statement N_3, which is the faulty statement, we could state that the Ochiai ranking metric for this statement was:

$$\frac{1}{\sqrt{(1+0)\,(1+0)}} \tag{3.1}$$

The result of the above equation (3.1) is 1, which results of the statement N_3 being 100% faulty.

Metric-based Techniques

Metric-based techniques are those that use ranking metric formulas to pinpoint faulty program entities (Souza, Chaim, and Kon 2016). These ranking metrics use program spectrum information derived from testing as input to determine correlations between program entities and test case results. Each program entity receives a suspiciousness score that indicates how likely it is to be faulty. The rationale is that program entities frequently executed in failing test cases are more suspicious. Therefore, the frequency in which entities are executed in failing and passing test cases is analyzed to calculate its suspiciousness score.

Tarantula (J. A. Jones, M. J. Harrold, and John Stasko 2002) is a tool used to debug projects written in C that uses this SFL technique. For each program entity, Tarantula calculates the frequency in which a program entity is executed in all failing test cases, divided by the frequency in which this program entity is executed in all failing and passing test cases. In addition to Tarantula, other techniques based on similarity formulas have been proposed in the last years.

Table 3.2: Ranking metrics for fault localization. Based on (J. A. Jones, M. J. Harrold, and John Stasko 2002)

Ranking metric	**Formula**
Tarantula	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}}$
Jaccard	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}}$
O^p	$c_{ef}-\frac{c_{ep}}{c_{ep}+c_{np}+1}$
Kulczynski2	$\frac{1}{2}\cdot\left(\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ef}}{c_{ef}+c_{ep}}\right)$
DStar	$\frac{c_{ef}^{*}}{c_{nf}+c_{ep}}$
Ochiai	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}}$
Zoltar	$\frac{c_{ef}}{c_{ef}+c_{np}+c_{ep}+10000\cdot\frac{c_{nf}c_{ep}}{c_{ef}}}$
O	-1 if c_{nf} >0 , otherwise c_{np}
McCon	$\frac{c_{ef}^{2}-c_{nf}c_{ep}}{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}$
Minus	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}}-\frac{c-\frac{c_{ef}}{c_{ef}+c_{nf}}}{1-\frac{c_{ef}}{c_{ef}+c_{nf}}+1-\frac{c_{ep}}{c_{ep}+c_{np}}}$

In table 3.2 there are shown some of the main ranking metrics used for SFL. Table 3.2 uses the following nomenclature: c_{ef} indicates the number of times a program entity (c) is executed (e) in failing test cases (f), c_{nf} is the number of times a program entity is not executed (n) in failing test cases, c_{ep} is the number of times a program entity is executed by passing test cases (p) and c_{np} represents the number of times a program entity is not executed by passing test cases.

Statistics-based Techniques

Statistical techniques are used in the same manner as metric-based techniques. However, each statistical technique uniquely deals with testing information (Souza, Chaim, and Kon 2016).

There was made a study where the use of unusual inference for fault localization aiming to enhance fault localization by separating unusual effects that occur between program entities in a program dependence graph (Baah, Podgurski, and Mary Harrold 2010). This separation can improve the values assigned to faulty entities by reducing noise caused by other program elements in the appearance of failures. They used unusual graphs concepts to recognize entities that are independent in a program dependence graph. It was also suggested a linear regression model to calculate the unusual effect of statements on failures.

A technique for using a non-parametric statistical model for fault localization was also suggested (Z. Zhang et al. 2011). To calculate the wariness of predicates, it was considered the difference between the probability density function of a random variable of passing test cases and a failing test case. W. E. Wong, V. Debroy, and D. Xu proposed the use a crosstab-based technique for fault localization (W. E. Wong, V. Debroy, and D. Xu 2012).

The chi-square test was used to calculate the null hypothesis that an execution is independent of the coverage of a statement. Chi-square is applied to deal with categorical variables; in this work, such values are the test case results (pass and fail), and the presence of a statement in an execution (executed or not). Thus, they measured the dependency between an execution and a statement (W. E. Wong, V. Debroy, and D. Xu 2012).

It was proposed to use maximum likelihood estimation and linear regression to tune in SFL lists (Y. Zhang et al. 2012). They used previously known lists and bug positions, assuming a symmetric distribution of bug positions in such lists. Thus, they estimated a position to adjust the lists. In 2013, it was applied the *odds ratio* to SFL (Xue and Siami Namin 2013). This ratio is a statistical technique frequently used in text mining problems and classification, and it measures the weakness or the strength of a variable (a statement executed or not executed) associated with an event (a passing or a failing test case).

Meanwhile it was suggested the use of Probabilistic Cause-effect Graphs (PCEG), which is obtained using program dependence graph of failing test cases (J. Xu, R. Chen, and Du 2015). The technique applies PCEG (a Bayesian network variation) to calculate the chance of statements to be faulty. The statements more suspicious are the ones that are more fully connected to other suspicious statements.

Artificial Intelligence-based Techniques

Artificial Intelligence (AI) [1] techniques have been successfully used in many areas of software engineering. In fault localization, these techniques use program spectrum data as input for classifying suspicious program elements.

The use of association measures for fault localization was evaluated (Lucia et al. 2014). 20 association measures commonly used in data mining and statistics were evaluated. They modeled the problem as the strength of association between each entity's execution (and non-execution) and failures.

It was suggested a technique based on *Feature Selection* from Machine Learning (Roychowdhury and Khurshid 2011). Two methods were used, *RELIEF* [2] and *RELIEF-F* [3], to classify statements according to their relevant potential to be faulty. The coverage matrix obtained from the test execution is applied to *RELIEF* and *RELIEF-F*. A feature is considered each executed statement, and each coverage of a certain test case is a sample. In consequence, the technique captures the diversity of these statements' behaviors as they relate to bug execution (Souza, Chaim, and Kon 2016).

It was applied *Markov Logic Network* [4] from machine learning to fault localization (S. Zhang and C. Zhang 2014). They modeled the problem by considering program spectra, static information (control-flow and data-flow dependence), and prior bug knowledge (locations of similar bugs found in the past). The technique was developed for single-bug programs.

[1] In this dissertation Artificial Intelligence is considered a broad area, which includes Data Mining and Machine Learning techniques.

[2] Algorithm for feature selection in Machine Learning

[3] Algorithm for feature selection in Machine Learning

[4] Approach to combine first-order logic and probabilistic graphical models in a single representation

Program Dependence-based Techniques

Some techniques use additional program analysis information to highlight faulty code sections, such as program dependence data, program slicing, and assignment of weights to differentiate program entities (Souza, Chaim, and Kon 2016).

E. Alves et al. used dynamic slicing and *change impact analysis* [5] to reduce the number of statements in the ranking lists provided by fault localization techniques (E. Alves et al. 2011).

Three techniques were proposed to obtain this reduction:

- **T1** - Ranks only statements that appear in the dynamic slice;
- **T2** - Applies a change impact analysis before the dynamic slicing and considers all statements in the dynamic slice;
- **T3** - Identical to **T2** but with only changed statements ranked.

A technique that calculates suspiciousness of predicates was presented (Wang Tiantian et al. 2011). To assign values that indicate the feasibility that predicates will be faulty is used F-score. From a predicate, the technique generates control-flow and data-flow information on demand for each predicate by constructing a Procedure Dependence Graph (PDG) for the procedure that includes the predicate. Next, backward and forward slices from this PDG are obtained (Souza, Chaim, and Kon 2016).

It was proposed a technique that combines SFL and program slicing (W. Wen 2012). It consists in removing all program elements that do not belong to any failing executions slices. To calculate the suspiciousness of the remaining elements, it was considered the execution frequency of each element in each test case and the contribution of an element in a test case. The contribution is the percentage in which an element is executed, considering all executed elements. Neelofar et al. suggested to combine dynamic and static analysis for fault localization (Neelofar et al. 2017). Statements were categorized in several categories (e.g., assignment, control, function call). These categories are used to weight suspiciousness scores obtained using SFL.

Execution Models

This approach consists in to build execution models from test executions. This models represent patterns of failing and/or passing executions. Execution models are used to identify entities that meet or flee from an expected pattern. There are also models represented by graphs of execution (Souza, Chaim, and Kon 2016).

It was proposed a behavioral model that is constructed using two different coverage types: objects of an OO program and calls occurred inside each object (Wan, Xiaoguang Mao, and Ziying Dai 2012). This model consists in a sequence of objects and calls in execution traces of failing and passing test cases. The model contains two levels: the objects, and its internal calls. Two values from each entity are used to compose a suspicious value: coverage and violation. The violation means that entities from failing executions that are not present in the model are more suspicious, and entities which are present in the model from passing executions are less suspicious (Souza, Chaim, and Kon 2016).

[5] To analyzing the impact of changes in the deployed product or application

A probabilistic model of state dependency was also suggested (Gong et al. 2015). This state dependency relates to predicate statement, which can be true or false. The probability that a predicate is true or false in passing and failing executions is calculated. Two probability models are produced: one for passing executions, and another for failing executions. The probability of control dependent statements is based on the probability of their parent statements. The doubtful value for each statement is then calculated using the models' probability values, and the result is a list of the most doubtful statements (Souza, Chaim, and Kon 2016).

A technique that indicates methods more likely to be faulty was presented in 2016 (Laghari, Murgia, and Demeyer 2016). This technique uses information from integration testing, capturing patterns of method calls for each executed method through the use of *closed itemset mining algorithm*.

3.1.2 Model-Based Diagnosis

The ground of Bodel-based Diagnosis (MBD) is comparing the model, i.e. the description of the system's behavior, with the actually observed behavior (Mayer and Stumptner 2008). In other words, its a reasoning approach based on models. Ning Ge, Nakajima, and Pantel, in 2013 proposed a MBD approach in integration testing (Ning Ge, Nakajima, and Pantel 2013). They suggested a method based on Hidden Markov Model [6] which core is a fault localization algorithm that gives out the set of suspect faulty components and a backward algorithm that calculates the matching degree between the HMM and the real system to evaluate the confidence degree of the localization conclusion (Mayer and Stumptner 2008).

It was proposed the use of multiple model based fault diagnosis to determine the fault location and size (Skeli and Weidemann 2018). The means and the covariances calculated by the mode-conditioned filters are mixed component-wise. Their approach does not mix the means and covariances but rather the probability density functions calculated by the mode-conditioned filters.

3.1.3 Other References

There was proposed a new technique called Historical Spectrum Based Fault Localization (HSFL) based on the version history that aims to be more precise locating faults than SFL (M. Wen et al. 2019). This technique starts by finding the first commit in the version history from which the bug-revealing test cases start to fail, and finally it tries to trace their evolutions to the target version for fault localization. HSFL builds a Historical Spectrum for each suspicious code entity introduced in the bug-inducing commits (M. Wen et al. 2019).

A study was made by hybridizing Mutation based testing with Spectrum based fault localization (Dutta and Godboley 2021). They start to create statements (i.e. mutants) and drive along with the test cases to produce spectra for each statement (i.e. mutant). These generated spectra for all statements are supplied to fault localization techniques (see Table 3.2) to generate the statement ranking sequence for each mutant. They achieved 36.48% improvement over existing fault localization techniques (Dutta and Godboley 2021).

[6] Abstraction of system's component to simulate component's behavior.

3.2 Delta Debugging

Delta Debugging (DD) is an automated technique which takes a test case that causes a bug and turns the bug reports into minimal test cases where every part of the input would be significant in reproducing the failure thereby producing simplified bug reports (Zeller and Hildebrandt 2002). In other words, it's method that is used to find out what states appear to be related to system specific behaviors. Is an algorithm for reducing the size of failing test cases. DD algorithms common core is to use a variation on binary search to remove individual components of a failing test case to produce a new test case *t1min* satisfying two properties: (1) *t1min* fails and (2) removing any component from *t1min* results in a test case that does not fail (Christi et al. 2018). DD has two basic algorithms, Simplification and Isolation. The Simplification algorithm is used to locate a faulty interaction and the Isolation algorithm is used to locate an element that is relevant to a system failure (J. Li, Nie, and Lei 2012).

3.3 Program Slicing

Program slicing consists in finding all statements in a program that directly or indirectly affect the value of a variable occurrence, in other words, focuses on those parts of a program that could have contributed to the failure (Weiser 1984). Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that faulty behaviour. That subset of the program, called "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior (Weiser 1984). Slices are based on dependencies between statements: A statement S2 depends on a statement S1, if S1 can influence the program state accessed by S2 (W. Wong and Vidroha Debroy 2009). Starting from a statement, the transitive closure over all dependencies forms a program slice.

It is important to make a distinction between static and dynamic slicing. Further details about this subject will be presented below.

3.3.1 Static Slicing

The static slicing approach was proposed by Weiser in 1984. Slicing tries to find the fault by analyzing program dependencies. It only presents to the developer that subset of the program that may have some effect on the value of an erroneous variable, i.e. statements of assignments that can affect a certain variable. A static slice applies to all possible runs, and therefore is computed without making any assumptions about a concrete (failing) program run. It focus the user's attention only on the code segments that are relevant to the fault (Weiser 1984).

```
1    int GreatestNumber()
2    {
3        int num1, num2, num3, greatest;
4
5        Console.Write("Input the 1st number :");
6        num1 = Convert.ToInt32(Console.ReadLine());
7        Console.Write("Input the  2nd number :");
8        num2 = Convert.ToInt32(Console.ReadLine());
9        Console.Write("Input the 3rd  number :");
10       num3 = Convert.ToInt32(Console.ReadLine());
11
```

```
12          if (num1 > num2)
13          {
14              if (num1 > num3)
15              {
16                  greatest = num1;
17              }
18              else
19              {
20                  greatest = num1; // Bug ! (Num3)
21              }
22          }
23          else if (num2 > num3)
24          {
25                  greatest = num2;
26          }
27          else
28          {
29                  greatest = num3;
30          }
31
32          Console.Write("The number {0} is the greatest.", greatest);
33          Console.ReadKey();
34      }
35
```

Listing 3.2: Static Slicing - Greatest Number (C#).

The code presented in 3.2 contains a bug marked by a comment, the correct assignment should be *num3*, resulting wrong results for some inputs. Static slicing will try to find out how the result was wrong by analyzing program dependencies that can affect the variable *greatest*. All the statements and assignments that static slicing analyses are highlighted in red in the code.

3.3.2 Dynamic Slicing

Korel and Laski extended Weiser's static slicing algorithms based on data-flow equations for the dynamic case (Korel and Laski 1988). In 1993, Agrawal and Horgan proposed the use of dynamic slicing for debugging. Dynamic slicing and its underlying dynamic dependence analysis have been extensively studied and used as the foundation for numerous automated-debugging techniques (X. Li and A. Orso 2020). This technique examines the failed tests but only analyses the dependencies that were exercised in that execution, in other words, a dynamic slice just applies to the failing run and thus is more precise.

```
1   int GreatestNumber()
2   {
3       int num1, num2, num3, greatest;
4
5       Console.Write("Input the 1st number :");
6       num1 = Convert.ToInt32(Console.ReadLine());
7       Console.Write("Input the  2nd number :");
8       num2 = Convert.ToInt32(Console.ReadLine());
9       Console.Write("Input the 3rd  number :");
```

```
10        num3 = Convert.ToInt32(Console.ReadLine());
11
12         if (num1 > num2)
13         {
14             if (num1 > num3)
15             {
16                 greatest = num1;
17             }
18             else
19             {
20                 greatest = num1; // Bug ! (Num3)
21             }
22        }
23        else if (num2 > num3)
24        {
25                greatest = num2;
26        }
27        else
28        {
29                greatest = num3;
30        }
31
32         Console.Write("The number {0} is the greatest.", greatest);
33        Console.ReadKey();
34    }
35
```

Listing 3.3: Dynamic Slicing - Greatest Number (C#).

As shown in 3.3 dynamic slicing only executed the statements that were executed when the result failed. All the statements and assignments that dynamic slicing analyses are highlighted in red in the code.

In 2014, P. Zhang et al., used JSlice [7] in six representative Java programs to prove that dynamic slicing can effectively locate faults and JSlice is an effective dynamic slicing tool (P. Zhang et al. 2014). In 71 faulty versions, they could find 61 faulty versions.

X. Li and A. Orso suggested a more accurate dynamic slicing, which is relevant for all programming languages that can be used to compute memory address (i.e. C, C++, C# or Java) (X. Li and A. Orso 2020). This approach proposes a dynamic analysis algorithm for accurately computing Potential Memory-address Dependence (PMD)s, and defines a dynamic-slicing technique that takes this kind of dependences into account. PMD is referred as the potential dependencies between two instruction instances *s1* and *s2*, such that *s1* directly or indirectly affects the computation of a memory-address *ma* and *s2* reads the value of a memory location *m* different from that identified by *ma* (X. Li and A. Orso 2020).

3.4 Tools

One challenge for many empirical studies on software fault localization is that they require appropriate tool support for automatic or semi-automatic data collection and suspiciousness

[7] Dynamic slicing tool available for Java

computation (W. Eric Wong et al. 2016). In this section, there are present some of the tools researched including a brief description.

3.4.1 Aletheia

Aletheia [8] consists in a failure diagnosis toolchain, which aims to help developers and testes to reduce failure analysis time (Golagha et al. 2018). This tool is coded in C# and it is also available in a separate Visual Studio Extension [9]. Aletheia only analyzes C++ projects with unit tests developed in GTest framework.

Aletheia is composed by three components which are: data generation, failure clustering and fault localization (see figure 3.3) (Golagha et al. 2018). Each one of this can be used separately or the output of each one can be used as the input for the next one.

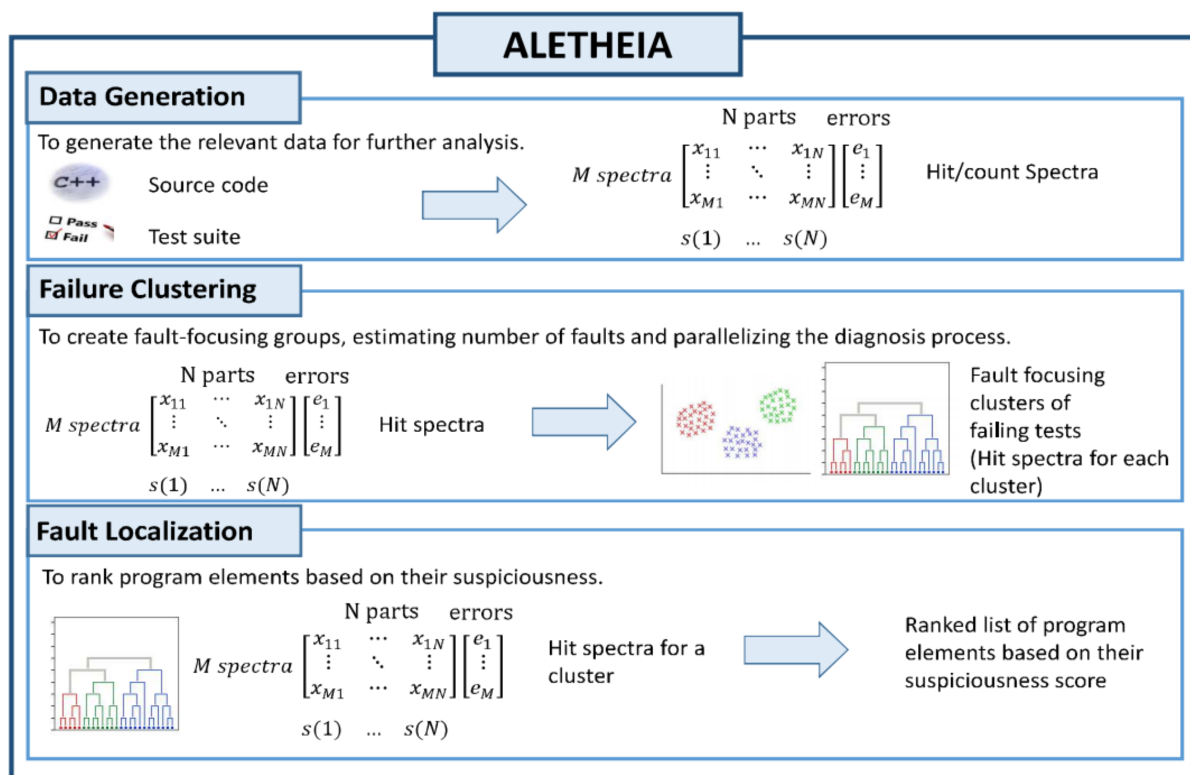

Figure 3.3: Aletheia with its 3 main components. Based on: (Golagha et al. 2018)

The first component, data generation, aims to generate and prepare data for further analysis. To do this, the source-code ant its test suite are needed. Then, the tool runs the test, collects coverage information and aggregates them into hit/count spectra.

The failure clustering component receives a hit spectrum (provided by the first component or by other tools such as GZoltar (see section 3.4.4) for Java programs). First, it generates a hierarchical tree of failing tests then it cuts the tree into some clusters utilizing spectrum-based fault localization.

[8] Aletheia Source Code `https://github.com/tum-i4/Aletheia`
[9] Aletheia Extension Source Code `https://github.com/tum-i4/AletheiaPlugin`

The last component, fault localization, using the information summarized in a hit spectrum (of all tests or one cluster), returns a ranked list of program elements based on their suspiciousness score to be faulty. Aletheia tool, implements three of the most popular metrics, Tarantula, Ochiai and Jaccard (see section 3.1.1)

3.4.2 AutoFLox

AutoFLox [10] stands for "Automatic Fault Localizer for Ajax" capable to of performing automatic fault localization of client-side JavaScript errors in Ajax-based applications. Specifically, it can localize code-terminating DOM-related JavaScript errors, which make up over 90% of bug reports we have studied for well-known web applications (Ocariza, Pattabiraman, and Mesbah 2012).

Its approach consists of two phases: trace collection, and trace analysis. The trace collection phase involves crawling the web application and gathering traces of executed JAVASCRIPT statements until the occurrence of the error that halts the execution. After the traces are collected, they are parsed in the trace analysis phase to find the direct DOM access (Ocariza, Pattabiraman, and Mesbah 2012).

3.4.3 FLAVS

FLAVS, meaning a Fault Localization Add-in for Visual Studio, is a tool developed to help developers using Microsoft Visual Studio to debug and test programs with complex bugs (Wang et al. 2015).

When the software under test executes, FLAVS records arguments, parameters, and standard inputs for replaying. During the software execution, it continuously logs the execution information including statement coverage, call stack traces, environmental factors and so forth. After a software run finishes, the developer is asked to mark the status of the program run, i.e., successful or failed. Then, it uses a fault localization module to incrementally update the suspiciousness of each program statement being related to fault, lists out the executed statements in the FLAVS window, and highlights the most suspicious ones in appropriate colors (Wang et al. 2015).

3.4.4 GZoltar

GZoltar is a graphical debugger interface that uses Zoltar's output, as well as a generated code dependency graph and a project hierarchy tree as input (Riboira and Rui Abreu 2010).

Zoltar consists in a toolset for automatic fault localization that can predict, the localization of software faults with a high success rate (Janssen, Rui Abreu, and Gemund 2009). Zoltar hosts a range of spectrum-based fault localization techniques featuring BARINEL [11] (RF Abreu, Zoeteweij, and A. van Gemund 2009). This tool provides data collected at runtime which is subsequently analyzed to return a ranked list of diagnosis candidates (Riboira and Rui Abreu 2010).

[10] `http://www.ece.ubc.ca/~frolino/projects/autoflox/`

[11] BARINEL stands for Bayesian AppRoach to dIagnose iNtErmittent fauLts.

GZoltar it's a library for automating the testing and debugging processes of the software life-cycle. This project aims to test and debug JAVA projects and it is an on-going open-source project [12]. This library is available also as:

- Command line interface
- Ant tasks
- Maven plug-In
- Visual Studio Code Extension

This tool analyzes all software's modules and these are colored differently to represent their failure probability. At all times, the developer would be able choose to see the dependency graph of a given module. This would help the developer to analyze the possibility of having errors propagated by the modules due to its dependency (Riboira and Rui Abreu 2010).

3.4.5 MZoltar

MZoltar is graphical debugger interface that uses Zoltar's that aims to detect faults in mobile applications. This tool follows the same strategy as GZoltar (see section 3.4.4) and also provides a graphical representation of the diagnostic report to aid the developers in the process of locating the defects in the code (Machado, Campos, and Rui Abreu 2013).

It relies on the Android testing framework and JaCoCo [13] to run the application's tests and acquire coverage information. This tool is only available as a plugin for the Eclipse IDE, and it also relies on the abstractions provided by the ADT plugin to perform some of the tasks (Machado, Campos, and Rui Abreu 2013).

3.4.6 Tarantula

Tarantula is fault localization tool that uses the SFL technique Tarantula that implements visual mapping. This tool is coded in Java and consists of 3600 lines of code. It takes as input a software system's source code and the results of executing a test suite on the system. It displays an interactive view of the system according to a variety of visual mappings (J. Jones, M.J. Harrold, and J. Stasko 2002).

3.4.7 UnitFL

UnitFL is a tool that can be run in two modes: test mode and fault localization mode (C. Chen and Wang 2016). Firstly, it uses program slicing and dynamic program instrumentation techniques to cut down the program execution overhead. It then provides different program granularities for fault localization analysis to cut down the overhead during program analysis. The next step, is to evaluate each unit test performance to uncover underlying bugs and finally it provides seamless guidance to explore fault-related program units (C. Chen and Wang 2016).

This tool is coded in C# and is implemented as an add-in in Microsoft Visual Studio 2013 [14] and supports to develop projects written in C#, C++, Visual Basic and several other CLR languages.

[12] https://github.com/GZoltar/gzoltar
[13] JaCoCo homepage http://www.eclemma.org/jacoco/,2013.
[14] https://marketplace.visualstudio.com/items?itemName=Wangnangg.UnitFL

3.5 Conclusion

An evaluation was on a range of automated debugging techniques by comparing its EXAM score (Zou et al. 2021). The EXAM score measures the relative position of the faulty element in the ranked list. In other words, its a measure that is given by n/N, in which n is the rank of a defective statement in the list and N is the number of statements in the program P. The score ranges between 0 and 1, and smaller numbers are better. The study evaluates the technique's performance in a dataset called Defects4J [15] which contains 357 faults minimized from real-world faults in five open-source Java projects.

Table 3.3: The Performance of Standalone Techniques on All 357 Faults (based on (Zou et al. 2021))

Family	Technique	$E_{inspect}$				Exam
		@1	@3	@5	@6	
SBFL	Ochiai	16 (4%)	81 (23%)	111 (31%)	156 (44%)	**0.033**
	DStar	17 (5%)	84 (24%)	111 (31%)	155 (43%)	**0.033**
MBFL	Metallaxis	23 (6%)	78 (22%)	103 (29%)	129 (36%)	0.118
	MUSE	24 (7%)	44 (12%)	58 (16%)		0.304
Slicing	Union	5 (1%)	33 (9%)	58 (16%)	84 (24%)	0.207
	Intersection	5 (1%)	35 (10%)	55 (15%)	71 (20%)	0.222
	Frequency	6 (2%)	39 (11%)	58 (16%)	84 (24%)	0.208
HSFL	Bugspots	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0.465

The numbers marked in bold in table 3.3 indicates the best-performing techniques. The $E_{inspect}$ is a measure that calculates the expected rank when multiple faulty elements are presented in ties. From this point on, its concluded that the best techniques to pursue are SFL since its EXAM score is higher and there is more relevant and recent information about it.

After researching and analyzing the different available automated fault localization tools, a brief review was made (see table 3.4). The available tools were filtered in order to select the ones that implement SFL techniques, alongside having open source code. For this reason, and to obtain a better outcome with an already studied language, the selected tool was Aletheia.

Table 3.4: Comparative Analysis of Automated Fault Localization Tools

Tool	Brief Description	Language	Availability
Aletheia	An automated testing and debugging tool	C#	Yes
AutoFLox	An automated testing and debugging tool	Java	Yes
FLAVS	Fault Localization Add-in for Visual Studio	C#	No
GZoltar	An automated testing and debugging framework	Java	Yes
MZoltar	Automatic debugging tool for android applications	Java	No
Tarantula	Fault localization tool using Tarantula	Java	No
UnitFL	An automated testing and debugging tool	C#	No

[15] https://homes.cs.washington.edu/ mernst/pubs/bug-database-issta2014.pdf

Chapter 4

Value Analysis

Automated debugging has been improving over the years, bringing value with many forms for different entities. This chapter addresses automated debugging potential and it affects its value as a technique.

4.1 Business and Innovation Process

When developing a new product, framework or idea, the innovation process, according to Koen (2004), can be divided into three parts (see Figure 4.1): The Fuzzy Front End (FFE) or Front End of Innovation (FEI), New Product Development (NPD) and Commercialization.

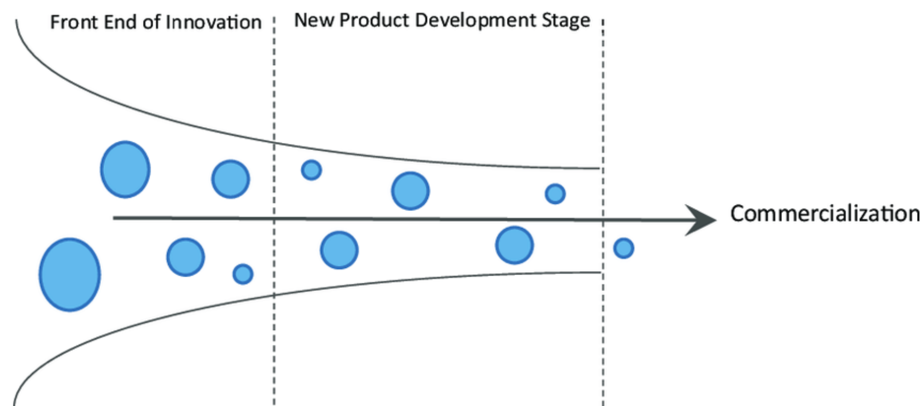

Figure 4.1: The Innovation Process (Koen, 2004)

This activities are described as follows:

- **The Fuzzy Front End (FFE) or Front End of Innovation (FEI) -** Activities that are less structured and less predictable that come before the formal and well structured NPD.
- **The New Product Development Stage (NPD) -** Disciplined and goal oriented with a project plan, structured with a formalized and stipulated set of tasks and questions.
- **Commercialization -** Final step of innovation process.

The New Concept Development (NCD) is a model that provides a common language to understand the key components of the Front End of Innovation, and it is divided in three parts (see Figure 4.2):

- The **engine** that provides power to the innovation process;
- The **wheel**, or inner spoke area, of the NCD model, defines the five activity elements: opportunity identification and analysis, idea selection and enrichment and concept definition;
- The **rim** that consists in external factors that influence the engine and inner activity elements;

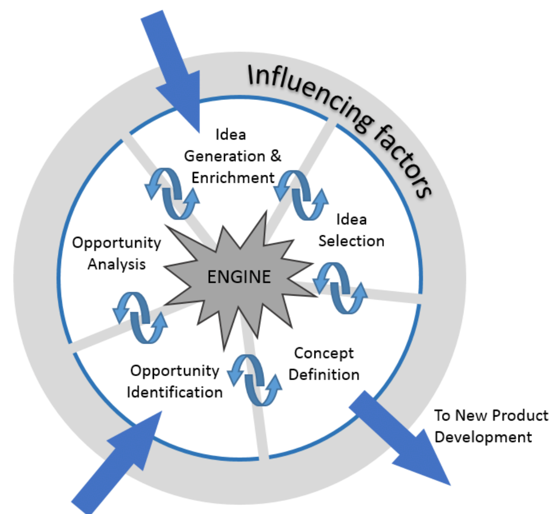

Figure 4.2: The New Concept Development NCD - model. (Koen et al., 2002)

The arrows pointing into the model represent the starting points for projects, which may originate by either opportunity identification or idea generation and enrichment. In other hand, the arrows pointing out the model represent projects that leave the front end of innovation process and enter the NPD.

It is important to consider the five activity elements of the wheel:

1. **Opportunity Identification:** Where large or incremental business and technological chances and opportunities are identified, by design or default, in a more or less structured way. It's where the organization identifies the opportunities that the company might want to pursue.
2. **Opportunity Analysis:** This activity involves gathering the additional information required to translate the identified opportunities into specific business and technology opportunities for the company.
3. **Idea Generation and Enrichment:** Described as the birth, development and maturation of the opportunity into a concrete idea.
4. **Idea Selection:** Activity that aims to choose which ideas to pursue in order to achieve the most business and consumer value.
5. **Concept Definition:** Development of a business case based on estimates of market potential, customer needs, investment requirements, competitor assessments, technology unknowns, and overall project risk.

These five activities mentioned above were applied to this dissertation theme, as follows in the sections bellow.

4.1.1 Opportunity Identification

It was already stated that, debugging is a notoriously difficult, requires human-intensive activity and it is extremely time consuming. Recent studies show that this activity can consume up to 50% of the development and maintenance effort (M. Wen et al. 2019). For this reason, several automated techniques and tools where developed for supporting various debugging tasks. Although potentially useful, most of these techniques have yet to fully demonstrate their practical effectiveness (A. Orso 2011). Many of these techniques rely on a set of strong assumptions on how developers behave while debugging, focus mainly on trying to reduce the number of statements to examine, and mostly ignore the importance of identifying relevant inputs.

The adversities of debugging is a problem that can be seen as an opportunity to improve an automated debugging technique and reduce several debugging tasks.

4.1.2 Opportunity Analysis

After identifying the opportunities, it is necessary to proceed with their assessment to see if they are worth the investment. The concept of "Automated Debugging" has a current growing emphasis and there are several techniques to make this concept a reality and bring major improvements to the debugging process, thus helping to develop applications with grater reliability and quality in less time. This dissertation involves an initial bring-up of the existing techniques and their potential. The techniques to be analyzed are those that have been approached with greatest emphasis in the state-of-the-art (Chapter 3).

4.1.3 Idea Generation and Enrichment

The selected idea was based on the most relevant factors of a technique and tool, improving:

- The success rates of defect identification of each one, in order to maximize a possible final solution, thus extracting the positive aspects and avoiding the mistakes that were once made.
- The validation of such techniques and tools on real systems.

4.1.4 Concept Definition

The final phase of formalization of the ideal already chosen previously.

4.2 Value Analysis

During software development, software faults are continuously introduced and removed no matter how much effort is put on testing or how well the software is designed. Automated debugging techniques aim to detect as many faults as possible using the least resources. For this reason, it is important to define the key concept of **value**, which can be "... in different theoretical contexts as need, desire, interest, standard/criteria, beliefs, attitudes, and preferences" (Nicola, Ferreira, and Pinto Ferreira 2012).

4.2.1 Customer Value

Customer value is the perception of what a product or service is worth to a customer versus the possible alternatives. Worth means whether the customer feels that he or she received benefits and services over what was paid. [1] In this dissertation, the costumers are independent software developers and software development companies, as the main focus of this dissertation is to **propose improvements to a automated debugging technique and applying it to an existing tool**: the product. The value for the costumer translates into a reliable and fast product to reduce their time of debugging processes and its complexity.

4.2.2 Perceived Value

Perceived value is the costumer's belief that a product or a service meets their needs or expectations. This belief can impact demand and pricing for a product.

"Perception is reality. If you are perceived to be something, you might as well be it because that's the truth in people's minds." (Steve Young)

So, if a customer inherently believes a product is valuable, they may be more willing to pay a premium and/or experience enjoyment from purchasing or using the product. This concept is further detailed in table 4.1.

[1] https://customerthink.com/what-is-customer-value-and-how-can-you-create-it/

Table 4.1: Value for the costumer based on (Woodall 2003)

Benefits		Sacrifices
Attributes	**Outcomes**	
Perceived Quality	Functional Benefits	Price
Product Quality	Utility	Market Price
Quality	Use Function	Monetary Costs
Technical Quality	Operational Benefits	Costs
Functional Quality	Economy	Costs Of Use
Performance Quality	Logistical Benefits	Perceived Costs
Service Performance	Product Benefits	Search Costs
Service	Strategic Benefits	Acquisition Costs
Service Support	Financial Benefits	Opportunity Costs
Special Service Aspects	Results For The Customer	Delivery and Installation Costs
Additional Services	Social Benefits	Costs Of Repair
Core Solution	Security	Training and Maintenance Costs
Customization	Convenience	Non-monetary Costs
Reliability	Enjoyment	Non-financial Costs
Product Characteristics	Appreciation From Users	Relationship Costs
Product Attributes	Knowledge	Psychological Costs
Features	Self-expression	Time
Performance	Personal Benefits	Human Energy
	Association With Social Groups	Efforts
	Affective Arousal	

The table above (4.1) details the benefits and sacrifices related to perceived value, which is derived from different perspectives from the costumer. The sacrifices listed in the third column can be categorized either as monetary, or non-monetary costs, and the benefits (both attributes and outcomes) listed represents benefits received for the price paid. From this perspective, there is a clear trade-off regarding customer benefits and sacrifices.

For this dissertation, the benefits and sacrifices are represented in the following list.

Benefits

- **Ease of Use -** Being simple to use for a developer.
- **Dependency Reduction -** A developer can be less dependent of a debugging tool.
- **Utility -** A developer can use to complement the testing process.
- **Dedicated Support -** There should be dedicated support for the developer.

- **Open Source -** The automated debugging technique should be available to an open source community.
- **Responsiveness -** Once a problem is found, and due to the framework's open sourced nature, the developer should get immediate feedback.

Sacrifices

- **Time/effort/energy -** The time and effort spent testing (by open source community).
- **Errors/Bugs -** More prone to design and implementation bugs than Commercial Off-the-shelf (COTS) solutions.
- **Certificate -** Not certificated by an higher entity.

4.2.3 Value Proposal

A value proposal describes what is the company's offer and why the customers should buy the product. In other words, it is a promise of value stated by a company that summarizes how the product's benefit or service will be delivered, experienced, and acquired. To represent this concept regarding the product of this dissertation, a business model Canvas was used to help design the system based on a perspective of objectives business and sustainability.

4.3 Analytic Hierarchy Process

Analytic Hierarchy Process (AHP), developed by mathematician Thomas Saaty in the 1970s, is a method for assessing and prioritizing options. Its primary use is to offer solutions to decision and estimation problems in multivariate environments. The AHP establishes priority weights for alternatives by organizing objectives, criteria, and sub criteria in a hierarchic structure (Bernasconi, Choirat, and Seri 2009).

In order to make a decision in an organized way to generate priorities, the decision must be decompose into 4 steps (Saaty 2008):

1. "Define the problem and determine the kind of knowledge
2. Structure the decision hierarchy from the top with the goal of the decision, then the objectives from a broad perspective, through the intermediate levels (criteria on which subsequent elements depend) to the lowest level (which usually is a set of the alternatives)
3. Construct a set of pairwise comparison matrices. Each element in an upper level is used to compare the elements in the level immediately below with respect to it.
4. Use the priorities obtained from the comparisons to weigh the priorities in the level immediately below. Do this for every element. Then for each element in the level below add its weighed values and obtain its overall or global priority. Continue this process of weighing and adding until the final priorities of the alternatives in the bottom most level are obtained. "

The objective of this section is to apply AHP to **determine the best automated debugging technique** based on the following criteria:

- **Accuracy -** The amount of faults it can detect.

- **Reliability -** The ability of performing consistently well.
- **Performance -** The amount of time it takes to find all faults.

The three alternative techniques are:

- **Fault Localization**
- **Delta Debugging**
- **Program Slicing**

The generated AHP can be seen bellow in figure 4.3:

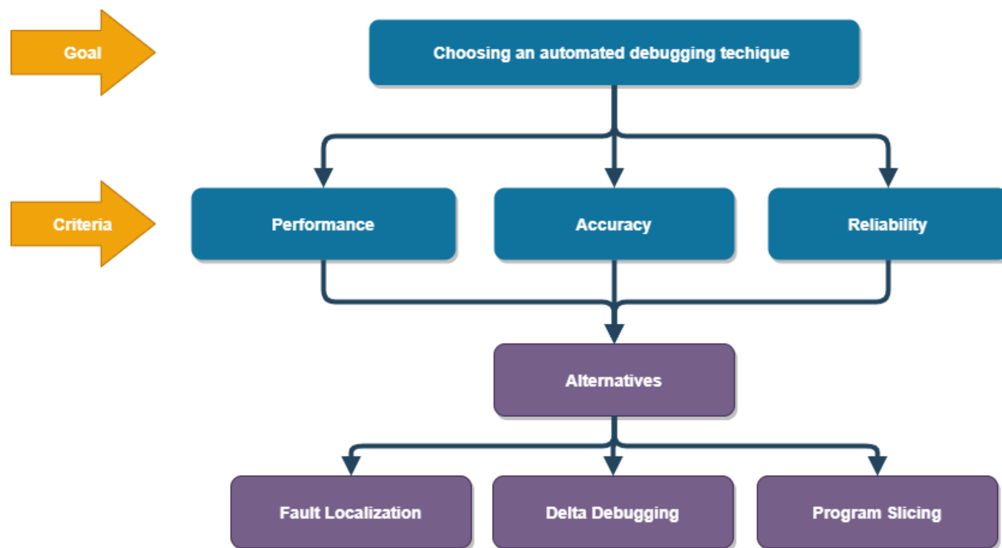

Figure 4.3: Hierarchical Decision Tree

4.3.1 Pairwise Comparisons

Each node (rectangle in the hierarchy — see Figures 4.3) will be derived from a series of measurements: pairwise comparisons involving all the nodes.

The next step is to complete step 3, which is to compare each level, two by two, regarding to their contribution to the nodes directly above them. The results are then entered into a matrix with the goal of assigning priorities for all the nodes on the level.

In this case, there are three automated debugging techniques, or alternatives (Fault Localization, Delta Debugging and Program Slicing). These are then compared each one with the remaining two. Then, pairwise comparisons are made with respect to each criterion: Fault Localization VS Delta Debugging , Fault Localization VS Program Slicing and Delta Debugging vs Program Slicing. For each comparison, it will be judged which member of the pair is weaker regarding the criterion under evaluation. Then, a relative weight is assigned to the other automated debugging technique.

The AHP fundamental scale for pairwise comparisons is shown in table 4.2.

Table 4.2: The fundamental scale for pairwise comparisons (Saaty 2008)

Intensity of Importance	**Definition**	**Explanation**
1	Equal Importance	Two activities contribute equally to the same objective
2	Week or slight	
3	Moderate importance	Experience and judgement slightly favor one activity over another
4	Moderate plus	
5	Strong importance	Experience and judgement strongly favor one activity over another
6	Strong plus	
7	Very strong or demonstrated importance	An activity is favored very strongly over another; its dominance demonstrated in practice
8	Very, very strong	
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
Reciprocals of above	If activity i has one of the above non-zero numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i	A reasonable assumption
1.1-1.9	If activities are very close	May be difficult to assign the best value but when compared with other contrasting activities the size of small numbers would not be too noticeable yet they can still indicate the relative importance of the activities

By comparing pairs of automated debugging techniques, AHP will be used to originate a scale that measures their strength regarding the criteria:

- Accuracy
- Reliability
- Performance

Ultimately, by using the AHP fundamental scale (4.2) a weight is assigned to each of the criteria mentioned above to the other automated debugging technique. In addition, each criteria is also compared in pairwise operations (see table 4.3). For example, performance is compared with accuracy and reliability. The same is applied for every other criteria.

Table 4.3: AHP criteria comparison

	Accuracy	Reliability	Performance
Accuracy	1	5	7
Reliability	1/5	1	1/5
Performance	1/7	5	1

Accuracy The accuracy was established to be the most important criteria, the reliability being the second, and the least important the performance. The accuracy is the main decision criterion because the among of bugs the technique can find is fundamental to the success of this dissertation. Table 4.4 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.4: AHP comparison for accuracy

Accuracy	Fault Localization	Delta Debugging	Program Slicing
Fault Localization	1	5	3
Delta Debugging	1/5	1	1/5
Program Slicing	1/3	5	1

Reliability The reliability criterion is important to distinguish which automated debugging techniques have the ability of performing consistently well for the development of this dissertation. Table 4.5 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.5: AHP comparison for reliability

Reliability	Fault Localization	Delta Debugging	Program Slicing
Fault Localization	1	5	3
Delta Debugging	1/5	1	1/5
Program Slicing	1/3	5	1

Performance Performance is fundamental but less important than accuracy and reliability criteria for the technique selection. Table 4.6 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.6: AHP comparison for performance

Performance	Fault Localization	Delta Debugging	Program Slicing
Fault Localization	1	3	2
Delta Debugging	1/3	1	5
Program Slicing	1/2	1/5	1

Results To decide which automated debugging technique is the best option, an AHP tool [2] (developed in JavaScript language) was used to calculate all the previous weight tables, as can be seen in Figure 4.7. This AHP tool generates consistent weight tables by introducing only the necessary comparison numbers between criteria and alternatives.

Table 4.7: Ranking matrix for every criterion

	Fault Localization	Delta Debugging	Program Slicing
Accuracy	0.607	0.090	0.303
Reliability	0.607	0.090	0.303
Performance	0.503	0.348	0.148

[2] https://github.com/airicyu/ahp

Fault Localization had the best rating with 58.3% Program Slicing in second place with a rating of 26.8% and in the last Delta Debugging with 14.9% (see table 4.8).

Table 4.8: Ranking matrix

	Score
Fault Localization	0.583
Program Slicing	0.268
Delta Debugging	0.149

Note: the complete source code of the JavaScript file used in the AHP matrices calculation where the alternatives, criteria and ranking metrics are specified, is available in the attachment A.

Chapter 5

Design and implementation

To improve the localization of faults in the software, Aletheia was extended to integrate more fault localization techniques and a symmetry coefficient was applied to all of these techniques. This chapter therefore describes the practical components of Aletheia that were improved with the main goal of increasing the efficiency of the tool and the symmetry coefficient is detailed and applied to several fault localization techniques.

5.1 Aletheia

Aletheia was the selected tool that served as base for this implementation. It's a console application coded in C# and it is located in an public open-source repository [1]. For the development of this dissertation there was created a fork in GitHub [2], which consists in a new server-side repository copy from the original project. Aletheia consists in a failure diagnosis toolchain, which aims to help developers and testers to reduce failure analysis time (see chapter 3.4.1).

Aletheia is composed by four components which are: data generation, failure clustering, fault localization and the new component added, Exam score calculation. This component was developed in order to allow Aletheia to perform the EXAM score calculation of every SFL automated debugging technique (see figure 3.3). Each one of this can be used separately or the output of each one can be used as the input for the next one. Data generation, failure clustering and fault localization are described in the section 3.4.1. In this section, the data generation, fault localization and the Exam score calculation components are fully detailed.

5.1.1 Data Generation

To generate the hit spectra of an application, Aletheia requires as an input a C++ project application and tests made in Google Test [3]. The process of generating the Hit Spectra occurs mainly in the file *Spectralizer.cs*. The steps executed to generate the hit spectra (see figure 5.1) are described bellow.

[1] https://github.com/tum-i4/Aletheia
[2] https://github.com/csampaio26/Aletheia
[3] https://github.com/google/googletest

Figure 5.1: Aletheia - Steps to Generate Hit Spectra

Aletheia starts by checking if the host computer contains OpenCPP installed. This software needs to be installed so Aletheia can know the executed lines in a program, therefore, this is the first thing to be checked.

The next step is to validate the input parameters introduced by the user. This operation is built on a method that returns true if all the input parameters are valid, it returns false if it doesn't. If all these requirements are valid, Aletheia then executes all the test cases in a multithread environment, analyzes the openCppCoverage files, parses all the test case touched functions, and finally build the hit spectra data table so it can export then the .csv files with the generated hit spectra.

To be able to simplify the tests made to this component, a simple test project in C++ with google tests were added to the GitHub repository. The test application includes the algorithm used to exemplify an hit spectra in the section 3.1.1. After running the following command:

```
1  Aletheia.exe do=GenerateHitSpectra separator=; project_path=Aletheia\
   TestApplication\TestApplication.vcxproj source_directory=Aletheia\
   TestApplication
2  gtest_path=Aletheia\x64\Debug\TestApplication.exe
```

Aletheia then prints on the console what test cases passed and what test cases failed (see figure 5.2).

```
| M2.InputGreatestNumber1     | PASSED |
| M6.InputGreatestNumber5     | PASSED |
| M3.InputGreatestNumber2     | FAILED |
| M1.InputGreatestNumber0     | PASSED |
| M4.InputGreatestNumber3     | PASSED |
| M5.InputGreatestNumber4     | PASSED |
```

Figure 5.2: Hit Spectra output in console

The tool exported the generated hit spectra .csv files with all the test coverage of the software. The selected information to this dissertation was the statement hit spectra, which contains all the statements blocks, predicates, function or method calls that were hit during the tests execution. The tool goes through every line of code executed in the tests and then generates an .csv file with 0, if the statement was not hit or 1 if the statement was hit. The last column, result, indicates if the test failed(0) or passed(1). The obtained .csv file is available in the attachment B.

5.1.2 Fault Localization

To complete this component, Aletheia receives as an input a .csv file with the hit spectra. The .csv file can be generated by Aletheia itself of by any other application. It is possible to choose the fault localization technique to be executed or to execute them all.

After running the following command:

```
1  Aletheia.exe do=cluster separator=; input_path=C:\HitSpectras\
   GENERATEHITSPECTRA_2021-05-16_13-04-12\TestCoverage_2021-05-16_13
   -04-12\statement_hit_spectra.csv clustering_method=maxclust
2  linkage_method=average linkage_metric=euclidean
```

Aletheia then calculates the suspiciousness value of each statement executed with the given ranking metric. This value goes from 0 to 1, being 1, 100% probability to be faulty. To

achieve this goal, the tool goes through each executed and non executed statement in the hit spectra and their match to the final result, if the test passed or failed. For example, if the statement was executed and the test failed, Aletheia adds it to a variable *covered-Failed*. This calculation is made in a method called *calculateSuspiciousnessRanking* in the file FaultLocalizer.cs (see attachment 5.3).

The steps executed to calculate the suspiciousness ranking (see figure 5.1) are described bellow.

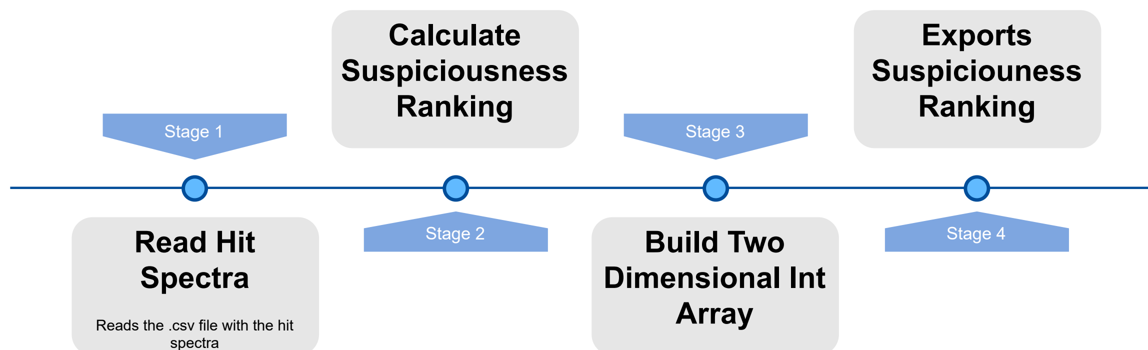

Figure 5.3: Aletheia - Steps to Calculate the Suspiciousness Ranking

Aletheia starts by checking if the input .csv file is valid and then, parses the .csv hit spectra file into a data table. Next, enters the *Detective.cs* file to create a 2D integer array from hit spectra. Then it calls *FautLocalizer.cs* to calculate the suspiciousness ranking of the hit spectra. Finally, it exports to .csv files with all the fault localization techniques selected with the correspondent suspiciousness ranking.

5.1.3 Exam score calculation

This component was developed so it can be easier for the developer to calculate the Exam score of all the fault localization techniques with and without the symmetry coefficient applied to them. Aletheia only receives as an input the folder where the generated fault localization .csv files .

After running the following command:

```
Aletheia.exe do=examScore separator=; input_path=C:\HitSpectras\
  FAULTLOCALIZATION_2021-06-24_17-12-34
```

Aletheia then calculates the Exam score of each file found in the input directory. The formula of the Exam score is given by n/N, in which n is the rank of a defective statement in the list and N is the number of statements in the program P. The score ranges between 0 and 1, and smaller numbers are better. The rank value of each statement is incremented as the suspiciousness value is higher (see attachment D). This calculation is made in a method called *CalculateExamScore* in the file *ExamScoreCalculation.cs* . Aletheia starts by checking if the directory exists. Then it builds a data table with all of the data in each file found on that directory (see attachment E).

To calculate the Exam score, Aletheia divides the rank of every statement by the number of existing statements. Then it performs an average off all statements to obtain the result Exam

score of the fault localization technique applied to that software. After the calculation of the Exam score of each technique, it finally exports to a .csv file with all the fault localization techniques selected with the correspondent Exam score.

5.2 Symmetry Coefficient

As measures, or coefficients, of similarity commonly used in calculating the proximity between two entities are intended for symmetric data, or for data asymmetric. In other words, comparing the statements covered in the passed tests to the uncovered statements in the failed tests, the symmetry between this variables is null. In this cases, the discordant pares are normally ignored in the calculation of the similarity between two entities. A. Sampaio, I. Sampaio, and G. R. Alves state that there is possible to find a solution that consists in finding a parameter that adjusts the measure of similarity to the amount of asymmetry that is known (A. Sampaio, I. Sampaio, and G. R. Alves 2007).

The proposed solution consists in finding a parameter that adjusts the measure of similarity to the quantity of asymmetry that is known. It associates the symmetry coefficient (CS), see 5.1, to the uncovered statements in the failed tests (d), in which n is the number of variables and NS a parameter that variates proportionally to the quantity of the global asymmetry presented in the data (A. Sampaio and I. Sampaio 2006).

$$CS = \frac{n - NS}{n} \tag{5.1}$$

The symmetry coefficient was applied in the Aletheia tool to all of the fault localization techniques. Since the level of symmetry is unknown, or in another words, is not possible to know the number of asymmetric characters and/or its symmetry, the symmetry Coefficient can take the value of 0.5 (A. Sampaio, I. Sampaio, and G. R. Alves 2007). The 0.5 value for the symmetry coefficient was selected because the level of symmetry is unknown, or in another words, is not possible to know the number of asymmetric characters and/or its symmetry.

Applying this coefficient to the fault localization techniques, it will result in a set of coefficients in which is called adjustable symmetry measures. In the table 5.1, there are represented all the fault localization techniques applied with symmetry coefficient (CS).

Table 5.1: Fault Localization Techniques with Symmetry Coefficient

Fault Localization Technique	Formula with Symmetry Coefficient
Tarantula	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}*CS}}{\frac{c_{ef}}{c_{ef}+c_{nf}*CS}+\frac{c_{ep}}{c_{ep}+c_{np}}}$
Jaccard	$\frac{c_{ef}}{c_{ef}+c_{nf}*CS+c_{ep}}$
DStar	$\frac{c_{ef}^{*}}{c_{nf}*CS+c_{ep}}$
Ochiai	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf}*CS)(c_{ef}+c_{ep})}}$
Rojers Tanimoto	$\frac{c_{ef}+c_{np}}{c_{ef}+c_{np}+c_{ep}+c_{nf}*CS}$
Sokal Sneath	$\frac{2*(c_{ef}+c_{np})}{2*(c_{ef}+c_{np})+c_{ep}+(c_{nf}*CS)}$

5.3 Integration of Aletheia with Symmetry Coefficient

As mentioned in section 5.1, Aletheia is offered as an independent automated fault localization application coded in C#. As this tool takes advantage of OpenCPP to find the executed lines in a program, the debugging effort is reduced. Also, the provided cues try to be as explicit and well-aimed as possible, to help the user debug accurately.

As Aletheia has already different classes to each ranking metric, the integration with the symmetry coefficient, was more ease to implement. In the figure 5.1 there is shown a excerpt of the implemented code with the tarantula ranking metric applied with the symmetry coefficient.

```
1  namespace Aletheia.Clustering.FaultLocalization.SimilarityMetrics
2  {
3      public class TarantulaCS : IRankingStrategy
4      {
5          public double calculateSuspiciousness(int coveredFailed,
       int uncoveredFailed, int coveredPassed, int uncoveredPassed)
6          {
7              double totalFailed = coveredFailed + uncoveredFailed
       * 1.5;
8              double totalPassed = coveredPassed + uncoveredPassed;
9              double failRatio = (double)coveredFailed /
       totalFailed;
10             double passRatio = (double)coveredPassed /
       totalPassed;
11             double result = failRatio / (failRatio + passRatio);
12
13             return result;
14         }
15     }
16 }
```

Listing 5.1: Code extract from Aletheia with the Tarantula fault localization technique with Symmetry Coefficient (C#).

5.4 Improvements To Aletheia

Before running the experiments that will evaluate the improved automated fault localization techniques, the following changes needed to be applied:

- **Fix project input arguments** - Aletheia had several bugs with the input arguments in the console. The major issue with the tool was that the software could only be executed in the same path as the test project path. This would not be efficient since it was forced the user to change Aletheia path every time he wanted to change the test project. This issue was fixed in Aletheia, by using the *source_directory* input argument. Several other minor changes were made to allow the tool to work properly with the selected input arguments.
- **Several fault localization techniques at once** - It was only possible to execute one fault localization technique at time. It was made a change on Aletheia that allows the user to export all the fault localization techniques in two ways: not adding the input argument *ranking_metric* input or by adding the input argument *ranking_metric* = *all* .
- **Improve code quality** - In order to improve Aletheia's performance, the code quality was refined. Several variables where deleted, several imports were deleted, and some statements were improved.

Chapter 6

Evaluation

Upon improving an automated debugging technique, there must be some form of evaluating if the improvements reached the proposed objectives and goals. This chapter presents the solution evaluation and the discussion of its results. It starts by describing the methodology, the metrics in which the results will be measure, detailing the investigation hypotheses and the evaluation indicators and the information sources are presented. After, the results obtained from applying the symmetry coefficient to the fault localization techniques are compared against the original fault localization techniques. A summary of the different results concludes the chapter.

6.1 Methodology

It was used two evaluation methodologies, one from the automated debugging technique and the other from the development point of view. From the automated debugging technique point of view, the EXAM score metric was used to evaluate the solution.

From the development point of view, the automated debugging tool was submitted to several functional tests which aim to guarantee a controlled development by the needs and problem requirements, as well as ensuring that the developed software is high quality. The automated debugging tool must meet all the development good practices, in order to guarantee its reliability and maintainability, while providing the ability to expand the solution.

6.2 Metrics

To measure the success of the SFL automated debugging techniques we use the EXAM score metric, which measures the relative position of the faulty element in the ranked list. This metric allows to compare the obtained results between all the techniques, the lower it is the final EXAM score, the more effective the automated debugging technique is.

In addition to that, this will be published as an open source project in GitHub getting the following metrics:

- Number of users using the tool
- Overall user satisfaction with the application

6.3 Investigation Hypotheses

As it was already mentioned throughout this thesis, its objective is to improve an automated debugging technique with a focus on fault localization. In order to evaluate whether the objective was met, the following hypotheses have been defined:

- H1: EXAM score - **Lower applying symmetry coefficient** - Success of the improvement of the automated debugging technique applying the coefficient symmetry comparing to the original fault localization techniques.
- H2: EXAM score - **Higher applying symmetry coefficient** - Failure of the improved solution regarding the appliance of the coefficient symmetry to all the introduced fault localization techniques.

6.4 Evaluation Indicators and Information Sources

This section presents what indicators will be to evaluate the hypothesis, and also what information sources will provide such indicators, so that each of them is dispatched appropriately. To evaluate the automated debugging technique in terms of localization, the solution will be tested using developed C++ software applications with GTests. The EXAM score will be measure and compared with recent measures obtained in studies about the respective technique.

In addition to the indicators mentioned above, the following information sources will be used:

- **Open Source Community:** Due to the framework's open sourced nature, its possible to obtain the developers opinions and satisfaction regarding the solution.
- **Real-life Scenario Implementation:** Apply the solution in a real-life scenario and evaluate if significant improvements are observed.

6.5 Evaluation Results

The design and implementation in chapter 5 represents the improvement of an automated debugging tool implemented with several fault localization techniques applied with and without the symmetry coefficient. The real value of the solution can only be obtained when the EXAM score is compared between all of the fault localization techniques.

In order to perform a case study, an example C++ software was developed with several bugs and unit tests. Also, as mentioned in 5.1 the fault localization tool was improved to provide all the necessary data to obtain this results. This process was followed by an analysis of the different fault localization techniques that were applied to the tool to identify which one provides better results over the adopted software. The obtained results with the EXAM score of each technique applied with and without the symmetry coefficient are compared in this chapter.

6.5.1 Aletheia

As mentioned in 3.5, Aletheia was the selected automated debugging tool to the development of this dissertation. The tool was submitted to several tests to ensure that its behavior was the expected. The tool performs any of the components (Data Generation, Fault

Localization and Exam score calculation) in less than two seconds. This amount of time to go through a project and perform analysis and retrieve the results is considered a good result.

6.5.2 Example Test Software

Measuring the solution's performance in a test software is essential to evaluate the improved fault localization tool. Due to the nature of the solution's architecture, this test software was coded in C++ with several statements and unit tests and it's located at the Aletheia repository. This is an example of the projects that Aletheia can analyze and obtain results from.

A significant amount of bugs were placed in this software in order to obtain a more precise result from Aletheia. Additionally, for every method and statement there were build multiple unit tests built with the framework GTest.

6.5.3 Fault Localization Techniques Applied

To Aletheia there were applied several fault localization techniques in order to obtain a more precise result between them. The symmetry coefficient were also added to this techniques in order to compare the results from the original fault localization techniques against the fault localization techniques with the symmetry coefficient applied to them. The techniques studied are shown in the table 6.1.

Table 6.1: Fault Localization Techniques Added to Aletheia

Fault Localization Technique	Formula
Tarantula	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}}$
Tarantula with Symmetry Coefficient	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}*CS}}{\frac{c_{ef}}{c_{ef}+c_{nf}*CS}+\frac{c_{ep}}{c_{ep}+c_{np}}}$
Jaccard	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}}$
Jaccard with Symmetry Coefficient	$\frac{c_{ef}}{c_{ef}+c_{nf}*CS+c_{ep}}$
DStar	$\frac{c_{ef}^{*}}{c_{nf}+c_{ep}}$
DStar with Symmetry Coefficient	$\frac{c_{ef}^{*}}{c_{nf}*CS+c_{ep}}$
Ochiai	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ef}+c_{ep})}}$
Ochiai with Symmetry Coefficient	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf}*CS)(c_{ef}+c_{ep})}}$
Rojers Tanimoto	$\frac{c_{ef}+c_{np}}{c_{ef}+c_{np}+c_{ep}+c_{nf}}$
Rojers Tanimoto with Symmetry Coefficient	$\frac{c_{ef}+c_{np}}{c_{ef}+c_{np}+c_{ep}+c_{nf}*CS}$
Sokal Sneath	$\frac{2*(c_{ef}+c_{np})}{2*(c_{ef}+c_{np})+c_{ep}+(c_{nf})}$
Sokal Sneath with Symmetry Coefficient	$\frac{2*(c_{ef}+c_{np})}{2*(c_{ef}+c_{np})+c_{ep}+(c_{nf}*CS)}$

6.5.4 EXAM score Results

To obtain the EXAM score results, all of the Aletheia components detailed in the section 5.1 were applied. The test software was executed by Aletheia to extract the hit spectra, calculate the suspiciousness and rank of each statement using every fault localization technique and finally the EXAM score was obtained. Table 6.2 illustrates the obtained results.

Table 6.2: Result EXAM score

Fault Localization Technique	**EXAM score**
Tarantula	0.015
Tarantula with Symmetry Coefficient	0.015
Jaccard	0.017
Jaccard with Symmetry Coefficient	0.017
DStar	0.017
DStar with Symmetry Coefficient	0.017
Ochiai	0.017
Ochiai with Symmetry Coefficient	0.017
Rojers Tanimoto	**0.013**
Rojers Tanimoto with Symmetry Coefficient	0.014
Sokal Sneath	**0.013**
Sokal Sneath with Symmetry Coefficient	0.014

The numbers marked in bold in table 6.2 indicates the best-performing techniques. After analyzing the results, its concluded that adding the symmetry coefficient doesn't make significant changes to the SFL automated debugging techniques. Also, the added techniques (Rojers Tanimoto and Sokal Sneath) presents more accurate results than the most used techniques. Comparing this results with the ones studied in 3.5, although the testing dataset used is different from the one applied in 3.5, Aletheia seems to be a more accurate and precise tool.

In order to obtain more accurate results about the appliance of the symmetry coefficient, Aletheia should be tested with more software's with several symmetry levels.

6.5.5 Open Source Results

For the development of this dissertation there was created a fork from the original Aletheia project in GitHub [1]. Since it wasn't received any feedback, it's not possible to obtain results in these metrics.

[1] https://github.com/csampaio26/Aletheia

Chapter 7

Conclusion

This chapter concludes the development of this dissertation as well as the achieved requirements and the future work.

The development of this dissertation tackled different knowledge areas and solidified skills that will surely transpose into further projects. Knowledge areas included automated debugging techniques and tools and how to use it to ease software development issues.

Regarding the work developed on this dissertation, the selected automated debugging tool, Aletheia, was very precise on the task of finding all the existing bugs in a developed test software. In a first stage, the different automated debugging techniques and tools were studied in order to understand their accuracy and effectiveness finding the errors in a software. The study of the different automated debugging techniques and tools allowed to select the type of automated debugging techniques to implement and an automated debugging tool to improve and at the same time contribute to an open-source community Aletheia was the selected automated debugging tool and on which a large part of the work reported on this dissertation was referred to.

As seen in the chapter 3.5, SFL was considered to be a more valued approach to this work, since its EXAM score is higher and there is more relevant and recent information about it. Besides the work developed in Aletheia, two SFL automated debugging techniques were added, Rojers Tanimoto and Sokal Sneath. In addition, all of the automated debugging techniques where implemented with the symmetry coefficient to obtain results of the EXAM score against the original automated debugging techniques.

Although the addition of the symmetry coefficient did not make major improvements on the SFL automated debugging techniques, the results were considered positive. Aletheia had a very significant accuracy finding all the bugs in the test software and its time to analyze the project was fast.

7.1 Achieved Requirements

In this dissertation, the identification and analyze of the existing automated debugging techniques and tools they use, was made in order to select the best automated debugging techniques and tools to pursue.

It was also made an analyze on the pros and cons of the tools and techniques already in use, including as success rates of localization of faults of each one, with the intention of maximizing a possible final solution, extracting, in this way, the positive aspects and avoiding the mistakes that were once made.

As referred in 3.5, the selected automated debugging techniques were the SFL. These techniques were successfully applied with the symmetry coefficient and it was possible to compare results between them.

The selected automated debugging tool to improve in order to provide more accurate results was Aletheia. Aletheia was selected in order to obtain a better outcome with an already studied language. This tool was successfully improved in order to provide all the information needed to analyze this dissertation results. Although Aletheia should be tested with more software's with several symmetry levels, it's considered that the results were satisfactory.

7.2 Future Work

Concerning possible future work, there are several improvements and additions that can enhance the general performance and usability of the selected automated debugging tool.

Firstly, it is decisive to remove the dependency of OpenCppCoverage tool. OpenCppCoverage doesn't provide coverage for many type of projects. As a result, its not possible to obtain the hit spectra of many projects.

Secondly, Aletheia should also contain all of the SFL automated debugging techniques to compare them against each other. Therefore, the results would be more complete.

Finally, the dependency of MatLab should be removed. Aletheia uses MatLab to cluster failing tests with respect to hypothesized faults. Moreover, while clustering, every time the MatLab libraries are loaded into RAM it takes a few minutes. Replacing MatLab library with something else might speed up Aletheia a bit in the cluster component.

Regarding the test software, there are some improvements that could be explored. One of the improvements its to expand its amount of known bugs and difficulty. More accurate results can be obtained by achieving this.

Bibliography

Abreu, R., P. Zoeteweij, and A. J. C. van Gemund (2007). "On the Accuracy of Spectrum-based Fault Localization". In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98. doi: 10.1109/TAIC.PART.2007.13.

– (2009). "Spectrum-Based Multiple Fault Localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 88–99. doi: 10.1109/ASE.2009.25.

Abreu, RF, P Zoeteweij, and AJC van Gemund (2009). "A Bayesian Approach to Diagnose Multiple Intermittent Faults". Undefined/Unknown. In: *Proceedings of the Twentieth International Workshop on Principles of Diagnosis (DX'09)*. Ed. by E Frisk et al. null ; Conference date: 14-06-2009 Through 17-06-2009. Linkoping University, pp. 27–33. isbn: -.

Adragna, P. (2008). "Software debugging techniques". In: *Inverted CERN School of Computing (iCSC2006)*, pp. 71–86.

Agrawal, Hiralal and Joseph R. Horgan (1990). "Dynamic Program Slicing". In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. PLDI '90. White Plains, New York, USA: Association for Computing Machinery, pp. 246–256. isbn: 0897913647. doi: 10.1145/93542.93576. url: https://doi.org/10.1145/93542.93576.

Alves, E. et al. (2011). "Fault-localization using dynamic slicing and change impact analysis". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 520–523. doi: 10.1109/ASE.2011.6100114.

Baah, George, Andy Podgurski, and Mary Harrold (Jan. 2010). "Causal inference for statistical fault localization". In: pp. 73–84. doi: 10.1145/1831708.1831717.

Beller, M. et al. (2018). "On the Dichotomy of Debugging Behavior Among Programmers". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 572–583. doi: 10.1145/3180155.3180175.

Bernasconi, Michele, Christine Choirat, and Raffaello Seri (Jan. 2009). "The Analytic Hierarchy Process and the Theory of Measurement". In: *University of Venice "Ca' Foscari", Department of Economics, Working Papers* 56. doi: 10.2307/27784145.

Bourque, P. and R.E. Fairley (2014). "Guide to the Software Engineering Body of Knowledge, Version 3.0". In: *ISO/IEC/IEEE 24765:2017(E)*.

Chen, Cheng and Nan Wang (2016). "UnitFL: A fault localization tool integrated with unit test". In: *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pp. 136–142. doi: 10.1109/ICCSNT.2016.8070135.

Christi, Arpit et al. (Oct. 2018). "Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization". In: pp. 184–191. doi: 10.1109/ISSREW.2018.00005.

Dubrova, Elena (Mar. 2013). "Fault-Tolerant Design". In: pp. 87–136. isbn: 978-1-4614-2112-2. doi: 10.1007/978-1-4614-2113-9_5.

Dutta, Arpita and Sangharatna Godboley (2021). "MSFL: A Model for Fault Localization Using Mutation-Spectra Technique". In: *Lean and Agile Software Development*. Ed. by

Adam Przybyłek et al. Cham: Springer International Publishing, pp. 156–173. isbn: 978-3-030-67084-9.

Golagha, Mojdeh et al. (2018). "Aletheia: A Failure Diagnosis Toolchain". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 13–16.

Gong, Dandan et al. (2015). "State dependency probabilistic model for fault localization". In: *Inf. Softw. Technol.* 57, pp. 430–445.

Hernández, Gonzalo Osco (2002). *Debugging like a pro*. [Online; accessed 11-November-2020]. url: https://medium.com/@gonzaloohk/debugging-like-a-pro-ec8556e220e0.

"ISO/IEC/IEEE International Standard" (2017). In: *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541. doi: 10.1109/IEEESTD.2017.8016712.

Janssen, Tom, Rui Abreu, and Arjan J.C. van Gemund (2009). "Zoltar: A Toolset for Automatic Fault Localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 662–664. doi: 10.1109/ASE.2009.27.

Jones, J.A., M.J. Harrold, and J. Stasko (2002). "Visualization of test information to assist fault localization". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 467–477. doi: 10.1145/581396.581397.

Jones, James A., Mary Jean Harrold, and John Stasko (2002). "Visualization of Test Information to Assist Fault Localization". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: Association for Computing Machinery, pp. 467–477. isbn: 158113472X. doi: 10.1145/581339.581397. url: https://doi.org/10.1145/581339.581397.

Kernighan, B. W. and P. J. Plauger (1978). "The elements of programming style". In: [Online; accessed 11-November-2020]. url: http://www2.ing.unipi.it/~a009435/issw/extra/kp_elems_of_pgmng_sty.pdf.

Korel, Bogdan and Janusz Laski (1988). "Dynamic program slicing". In: *Information Processing Letters* 29.3, pp. 155–163. issn: 0020-0190. doi: https://doi.org/10.1016/0020-0190(88)90054-3. url: http://www.sciencedirect.com/science/article/pii/0020019088900543.

Laghari, G., A. Murgia, and S. Demeyer (2016). "Fine-tuning spectrum based fault localisation with frequent method item sets". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 274–285.

Landsberg, David et al. (2015). "Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme". In: *Fundamental Approaches to Software Engineering*. Ed. by Alexander Egyed and Ina Schaefer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 115–129. isbn: 978-3-662-46675-9.

Laprie, Jean-Claude (1995). "Dependability — Its Attributes, Impairments and Means". In: *Predictably Dependable Computing Systems*. Ed. by Brian Randell et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 3–18. isbn: 978-3-642-79789-7.

Li, J., C. Nie, and Y. Lei (2012). "Improved Delta Debugging Based on Combinatorial Testing". In: *2012 12th International Conference on Quality Software*, pp. 102–105. doi: 10.1109/QSIC.2012.28.

Li, X. and A. Orso (2020). "More Accurate Dynamic Slicing for Better Supporting Software Debugging". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 28–38. doi: 10.1109/ICST46399.2020.00014.

Lucia, Lucia et al. (Feb. 2014). "Extended comprehensive study of association measures for fault localization". In: *Journal of Software: Evolution and Process* 26. doi: 10.1002/smr.1616.

Machado, Pedro, José Creissac Campos, and Rui Abreu (Aug. 2013). "MZoltar: automatic debugging of Android applications". In: *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile - DeMobile*. ACM. Saint Petersburg, Russian Federation: ACM, 9–16.

Mayer, W. and M. Stumptner (2008). "Evaluating Models for Model-Based Debugging". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 128–137. doi: 10.1109/ASE.2008.23.

Moore, D., E. Benson, and Raymond Budd (2007). "Professional Rich Internet Applications: AJAX and Beyond". In.

Neelofar, Neelofar et al. (Mar. 2017). "Improving spectral-based fault localization using static analysis: SPECTRAL-BASED FAULT LOCALIZATION USING STATIC ANALYSIS". In: *Software: Practice and Experience* 47. doi: 10.1002/spe.2490.

Nicola, Susana, Eduarda Ferreira, and João José Pinto Ferreira (May 2012). "A novel framework for modeling value for the customer, an essay on negotiation". In: *International Journal of Information Technology & Decision Making* 11.

Ning Ge, S. Nakajima, and M. Pantel (2013). "Hidden Markov model based automated fault localization for integration testing". In: *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pp. 184–187. doi: 10.1109/ICSESS.2013.6615284.

Ocariza, Frolin, Karthik Pattabiraman, and Ali Mesbah (Apr. 2012). "AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript". In: doi: 10.1109/ICST.2012.83.

Orso, A. (2011). "Automated Debugging: Are We There Yet?" In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 596–596. doi: 10.1109/ICSTW.2011.16.

Parnin, Chris and Alessandro Orso (Jan. 2011). "Are Automated Debugging Techniques Actually Helping Programmers?" In: doi: 10.1145/2001420.2001445.

Pearson, S. et al. (2017). "Evaluating and Improving Fault Localization". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 609–620. doi: 10.1109/ICSE.2017.62.

Riboira, André and Rui Abreu (2010). "The GZoltar Project: A Graphical Debugger Interface". In: *Testing – Practice and Research Techniques*. Ed. by Leonardo Bottaci and Gordon Fraser. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 215–218. isbn: 978-3-642-15585-7.

Roychowdhury, Shounak and Sarfraz Khurshid (2011). "Software Fault Localization Using Feature Selection". In: *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*. MALETS '11. Lawrence, Kansas, USA: Association for Computing Machinery, pp. 11–18. isbn: 9781450310222. doi: 10.1145/2070821.2070823. url: https://doi.org/10.1145/2070821.2070823.

Saaty, Thomas (Jan. 2008). "Decision making with the Analytic Hierarchy Process". In: *Int. J. Services Sciences Int. J. Services Sciences* 1, pp. 83–98. doi: 10.1504/IJSSCI.2008.017590.

Sampaio, Alberto and I. Sampaio (Sept. 2006). "Proposta de Coeficientes de Associação de Simetria Ajustável e sua Avaliação Parcial". In: XIV Congresso Anual da Sociedade Portuguesa de Estatística. Covilhã, Portugal: XIV Congresso Anual da Sociedade Portuguesa de Estatística, p. 1.

Sampaio, Alberto, I. Sampaio, and G. R. Alves (2007). "Avaliação de Algumas Medidas de Dissimilaridade de Simetria Ajustável". In.

Skeli, E. and D. Weidemann (2018). "Multiple-Model Based Fault-Diagnosis: An Approach to Heterogeneous State Spaces". In: *2018 23rd International Conference on Methods Models in Automation Robotics (MMAR)*, pp. 815–820. doi: 10.1109/MMAR.2018.8485836.

Souza, H. A. D., M. L. Chaim, and Fabio Kon (2016). "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges". In: *ArXiv* abs/1607.04347.

Wambugu, Geoffrey Mariga and Kevin Mwiti Njeru (Nov. 2017). "Automatic Debugging Approaches: A literature Review." In: *Automated Debugging Approaches* 12, pp. 2–3. url: https://www.researchgate.net/publication/319543813_Automatic_Debugging_Approaches_A_literature_Review.

Wan, X., Xiaoguang Mao, and Ziying Dai (2012). "Fault localization via behavioral models". In: *2012 IEEE International Conference on Computer Science and Automation Engineering*, pp. 472–475. doi: 10.1109/ICSESS.2012.6269507.

Wang, Nan et al. (May 2015). "FLAVS: A Fault Localization Add-in for Visual Studio". In: doi: 10.1109/COUFLESS.2015.8.

Wang Tiantian et al. (2011). "Comprehension oriented software fault location". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1, pp. 340–343. doi: 10.1109/ICCSNT.2011.6181971.

Weiser, M. (1984). "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4, pp. 352–357. doi: 10.1109/TSE.1984.5010248.

Wen, M. et al. (2019). "Historical Spectrum based Fault Localization". In: *IEEE Transactions on Software Engineering*, pp. 1–1. doi: 10.1109/TSE.2019.2948158.

Wen, W. (2012). "Software fault localization based on program slicing spectrum". In: *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1511–1514. doi: 10.1109/ICSE.2012.6227049.

Wong, W and Vidroha Debroy (Jan. 2009). "A Survey of Software Fault Localization". In.

Wong, W. E., V. Debroy, and D. Xu (2012). "Towards Better Fault Localization: A Crosstab-Based Statistical Approach". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.3, pp. 378–396. doi: 10.1109/TSMCC.2011.2118751.

Wong, W. Eric et al. (2016). "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42.8, pp. 707–740. doi: 10.1109/TSE.2016.2521368.

Woodall, Tony (Jan. 2003). "Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis". In: *Academy of Marketing Science Review* 12.

Xu, Junjie, Rong Chen, and Zhenjun Du (Aug. 2015). "Probabilistic reasoning in diagnosing causes of program failures". In: *Software Testing, Verification and Reliability* 26. doi: 10.1002/stvr.1583.

Xue, Xiaozhen and Akbar Siami Namin (June 2013). "Measuring the Odds of Statements Being Faulty". In: vol. 7896, pp. 109–126. doi: 10.1007/978-3-642-38601-5_8.

Zeller, Andreas and Ralf Hildebrandt (Feb. 2002). "Simplifying and Isolating Failure-Inducing Input". In: *IEEE Trans. Softw. Eng.* 28.2, pp. 183–200. issn: 0098-5589. doi: 10.1109/32.988498. url: https://doi.org/10.1109/32.988498.

Zhang, P. et al. (2014). "Fault localization based on dynamic slicing via JSlice for Java programs". In: *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pp. 565–568. doi: 10.1109/ICSESS.2014.6933631.

Zhang, Sai and Congle Zhang (May 2014). "Software bug localization with markov logic". In: *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*. doi: 10.1145/2591062.2591099.

Zhang, Y. et al. (2012). "Wielding Statistical Fault Localization Statistically". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pp. 189–194. doi: 10.1109/ISSREW.2012.51.

Zhang, Zhenyu et al. (2011). "Non-parametric statistical fault localization". In: *Journal of Systems and Software* 84.6, pp. 885–905. issn: 0164-1212. doi: https://doi.org/10.1016/j.jss.2010.12.048. url: http://www.sciencedirect.com/science/article/pii/S0164121211000045.

Zou, D. et al. (2021). "An Empirical Study of Fault Localization Families and Their Combinations". In: *IEEE Transactions on Software Engineering* 47.2, pp. 332–347. doi: 10.1109/TSE.2019.2892102.

Appendix A

AHP Javascript Code

```
1
2        const AHP = require('ahp')
3
4        var ahpContext = new AHP()
5        ahpContext.addItems([
6                'Fault Localization',
7                'Delta Debugging',
8                'Program Slicing',
9            ])
10       ahpContext.addCriteria(['Accuracy', 'Reliability', 'Performance'
    ])
11
12       ahpContext.rankCriteriaItem('Accuracy', [
13               ['Delta Debugging', 'Program Slicing', 1 / 5],
14               ['Fault Localization', 'Program Slicing', 3],
15               ['Fault Localization', 'Delta Debugging', 5],
16           ])
17       ahpContext.rankCriteriaItem('Reliability', [
18               ['Delta Debugging', 'Program Slicing', 1 / 5],
19               ['Fault Localization', 'Program Slicing', 3],
20               ['Fault Localization', 'Delta Debugging', 5],
21           ])
22       ahpContext.rankCriteriaItem('Performance', [
23               ['Delta Debugging', 'Program Slicing', 5],
24               ['Fault Localization', 'Program Slicing', 2],
25               ['Fault Localization', 'Delta Debugging', 3],
26           ])
27       ahpContext.rankCriteria([
28               ['Accuracy', 'Reliability', 5],
29               ['Accuracy', 'Performance', 7],
30               ['Reliability', 'Performance', 1 / 5],
31           ])
32
33       let analyticContext = ahpContext.debug()
34       for (let key in analyticContext) {
35               console.log(`${key}:\n`, analyticContext[key], '\n')
36           }
```

Appendix B

Hit spectra generated for example Application

	Line															
TestCase	7	10	12	14	15	18	20	21	22	24	26	27	31	32	34	35
M1	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1
M2	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1
M3	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	1
M4	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1
M5	1	1	1	1	1	0	1	0	0	0	1	1	1	1	1	1
M6	1	1	1	1	1	0	1	0	0	0	1	1	1	1	1	1

	Line																	
TestCase	37	38	39	41	42	43	45	46	47	49	50	51	53	54	55	57	58	59	**Result**
M1	1	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
M2	1	0	0	1	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1
M3	1	0	0	1	0	0	1	1	1	1	0	0	1	0	0	1	0	0	0
M4	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1	0	0	1
M5	1	0	0	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1
M6	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	1	1

Appendix C

Method to calculate the suspiciousness ranking

```
1
2     public List<Item> CalculateSuspiciousnessRanking()
3     {
4         List<Item> suspiciousnessList = new List<Item>();
5
6         int dim1 = testcaseMatrix.GetLength(0);
7         int dim2 = testcaseMatrix.GetLength(1);
8         int indexResult = testcaseMatrix.GetUpperBound(1);
9
10        for (int j = 0; j < dim2 - 1; j++)
11        {
12            string functionName = functionNames[j];
13
14            if (j < testcaseMatrix.GetUpperBound(1))
15            {
16                int coveredFailed = 0;
17                int uncoveredFailed = 0;
18                int coveredPassed = 0;
19                int uncoveredPassed = 0;
20
21                for (int i = 0; i < dim1; i++)
22                {
23                    if (testcaseMatrix[i, indexResult] == 0)
24                    {
25                        if (testcaseMatrix[i, j] == 0) uncoveredFailed
    ++;
26                        else coveredFailed++;
27                    }
28                    else
29                    {
30                        if (testcaseMatrix[i, j] == 0) uncoveredPassed
    ++;
31                        else coveredPassed++;
32                    }
33                }
34
35                double suspiciousnessValue = rankingStrategy.
    calculateSuspiciousness(coveredFailed, uncoveredFailed, coveredPassed
    , uncoveredPassed);
36                suspiciousnessList.Add(new Item(functionName,
    suspiciousnessValue));
37            }
38        }
```

```
39
40          suspiciousnessList.Sort();
41          suspiciousnessList.Reverse();
42
43          return suspiciousnessList;
44      }
45
```

Appendix D

Code extract from Aletheia to calculate the rank of each statement(C#)

```
1
2     int rank = 1;
3     double value = tmpList.ElementAt(0).Suspiciousness;
4     foreach (Item item in tmpList)
5     {
6         if (Math.Abs(value - item.Suspiciousness) > 0.0000001) rank++;
7         output += Convert.ToString(rank) + separator + item.ItemName +
      separator + item.Suspiciousness + "\n";
8         value = item.Suspiciousness;
9     }
10
```

Appendix E

Code extract from Aletheia to perform the Exam score calculation in all the fault localization files (C#)

```
1
2      DirectoryInfo d = new DirectoryInfo(inputPath);
3    foreach (var file in d.GetFiles("*.csv"))
4    {
5        FaultLocalizationCsvSheetReader reader = new
      FaultLocalizationCsvSheetReader(file.FullName, separator);
6        reader.ParseSheet();
7        DataTable dataTable = reader.getDataTable();
8
9        var allStatements = dataTable.AsEnumerable()
10       .Select(x => x.Field<string>(0))
11       .ToList();
12
13       double examScore = allStatements.Sum(x => double.Parse(x) /
      allStatements.Count) / allStatements.Count;
14
15       DataRow row = ExamScoreDatatable.NewRow();
16       row["Ranking Metric"] = file.Name;
17       row["ExamScore"] = examScore;
18       ExamScoreDatatable.Rows.Add(row);
19   }
20
21   CsvSheetWriter writer = new CsvSheetWriter(Path.Combine(destination, "
      ExamScore.csv"), separator, ExamScoreDatatable);
22   writer.WriteToWorkSheet();
23
```