Combining Fault Localization with Information Retrieval: an Analysis of Accuracy and Performance for Bug Finding

Nadhratunnaim Nasarudin

PhD

University of York

Computer Science

January 2022

Abstract

There has been a significant amount of study in developing and enhancing fault localization techniques, which are used in assisting developers to locate faults within a body of code. However, identifying fault locations using individual techniques is not always effective; combining different techniques, which represent distinct forms of analysis, might help to overcome this issue. There has been a very limited amount of research that suggests that combining more than one approach to fault localization may have benefits, principally because information from different sources is included in the localization process. In this thesis, I attempt to more precisely address the question of whether combining different fault localization techniques can more effectively and efficiently find faults in code, when contrasted with a single technique. To answer this, I have carried out experiments that combine the use of three fault localization techniques: *Information Retrieval (IR), Spectrum Based Fault Localization (SBFL)*, and *Text Based Search*. These techniques are representative of both dynamic and static fault localization. My hypothesis is that a combination of dynamic and static fault localization analysis can assist developers in better fault localization. I have evaluated the various combinations of techniques in identifying faults against real-world programs, Defects4j, where 395 faults and bug reports have been analyzed. The experimental results demonstrate that the combination of three techniques (SBFL, Text Search, and IR) is the most accurate, with 86.84% accuracy for 343 faults located from a total of 395. This finding contributes positively towards concretely recommending techniques for assisting developers in locating faults in code. Guidelines are provided on which combination of techniques, with maximal accuracy of result, should be applied especially when there is no prior knowledge about the fault.

TABLE OF CONTENTS

List of Tables

List of Figures

[فَإِنَّ مَعَ ٱلۡعُسۡرِ يُسۡرًا ﴿٥﴾ إِنَّ مَعَ ٱلۡعُسۡرِ يُسۡرٗا]

So, indeed with hardship comes ease (5). Indeed with hardship comes ease (6)

(Quran: 94:5-6)

Acknowledgement

I would like to extend my appreciation and gratitude to my supervisors, Professor Richard Paige and Dr. Nicholas Matragkas for their patience, motivation, support, and knowledge. They have guided me during my years of study at the University of York and gave me countless valuable inputs. Thank you from the bottom of my heart for always being helpful and not giving up on me. Besides my supervisor, I would like to thank my internal assessor, Dr. Rob Alexander, and Dr. Silvia Abrahão, my PhD thesis examiner. Thank you for the insightful and constructive comments, and the encouragement that had helped me to widen my views in this research.

The love of my life, my husband and my family for supporting me spiritually, emotionally, physically, and financially throughout this journey and my life in general. Even though it's a cliché to say that's what family do but I want to dedicate this thesis to all of you and let you guys know that you guys did it wonderfully. May God and only God can repay all your sacrifices and kindness. I would also thank my colleagues and friends in the university for sharing their pearls of wisdom during this journey and inspirations to keep my enthusiasm in finishing what I have started. I could not imagine my life at York without all of you.

Finally, above all I owe it all to Almighty God for granting me the wisdom, health, and strength to undertake this journey and enabling me to its completion. Although, this journey has been the most incredible and exciting experience in my life, it also has been unbeknownst to me extremely challenging, filled with self-doubts, self-criticism, and fear of rejection.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Nadhratunnaim Nasarudin

April 2023

1 Introduction

This chapter is a brief introduction to the research, including a problem statement, research objectives, motivation, research hypothesis and research questions.

1.1 Problem Statement

It is difficult to write code that does not contain faults. A fault is a characteristic of a program that prevents the program from meeting its (implicit or explicit) specification [1]. A fault itself, when the program is executed, can lead to a failure, and these failures can be observed by programmers. Programmers wish to identify and remove the faults leading to observable failures. Thus, finding faults is an important part of the software engineering process, but this process can be challenging, because of complex interactions between software components (e.g., client code, APIs, the operating system, device controllers etc.), or because the specification of the program is itself unclear in some way, or for a variety of other reasons.

Numerous techniques have been proposed and evaluated for managing faults in code. Two key aspects of dealing with faults is *identifying* the fault (i.e., making an observation that demonstrates that the program does not meet its specification), and then *locating* the fault (i.e., finding the line number or line numbers in the program that include the fault). It is the debugging process that is most frequently used to try to identify and locate a fault. Debugging is an action taken following a program failure; more precisely, debugging is defined as "determining what runtime faults led to a runtime failure or what software errors were responsible for those runtime faults and modifying the code to prevent the runtime faults from occurring" [2]. Examples of debugging activities include practices such as inserting print statements and break-points, checking the stack trace, and reverifying failing test cases [3].

Debugging is a process where developers identify what caused the fault, locating where in the code the fault arose, and thereafter preventing future occurrences by modifying, adding or deleting the (presumably problematic) code [4]. The first and second activities are generally referred to as *fault localization*, while the third activity is called fault repair or bug fixing. Though both activities are important and have been studied extensively, this thesis will only be focusing on fault localization, since it is a prerequisite for fault repair: one cannot repair a fault without knowing where it is located.

Because debugging is such a widespread and important technique in programming, many recommendations have been made to produce more effective and efficient debugging techniques and tools [5], [6], [7], [8], [9], [10] , [11], [12], [13], [14], [15]. However, despite this effort, debugging is still considered to be an expensive [16] [17] and time-consuming activity [18] [19] [20] . This thesis is focuses on attempts to identify improvements to fault localization techniques; as we shall see, improvements to the accuracy (fault location) and performance (time taken to allocate faults) of fault localization techniques can have a substantial improvement in the overall effort required to be spent on debugging.

There have been numerous techniques proposed for fault localization, and in Chapter 2 the most significant research results, methods and tools will be reviewed and analyzed. Some of the most recent research, e.g., by [21] – which presents a detailed comparison between all fault localization techniques – suggests that the spectrum-based fault localization (SBFL) techniques are the most accurate in fault localization, whereas information retrieval (IR) techniques are amongst the fastest techniques. SBFL techniques can, however, provide poor performance, whereas (as we shall see) IR techniques can struggle with accuracy. As such, we would argue that it is worthwhile to determine if a combination of IR and SBFL techniques can provide better performance and accuracy, and if so, under which circumstances.

In this thesis, I have combined two state-of-the-art techniques in fault localization (presented in [22]and [23]) that apply Information-Retrieval Based Fault Localization and Spectrum Based Fault Localization respectively. Since these techniques are using different information sources for carrying out their respective analyses, we wish to determine if the complementarity of their information sources leads to complementary or compatible results: that is, will a combination lead to better accuracy and performance? If so, when, and under which conditions? As we shall see in later chapters, there are some clear situations where the two techniques produce better results, but also some situations where the results are not demonstrably better. In these latter situations, we investigate whether *Text Search*, an approach that has been widely used for identifying meaningful strings in a corpus of text, can provide further complementary results and actually improve on combined fault localization techniques. The Semantic Web [24], bioinformatics [25] and 3D modelling [26] are just a few of the domains where Text

Search has been successfully applied; the investigations in this thesis consider its use in fault localization.

The question of whether combining fault localization techniques can be valuable has very recently been noticed – at least conceptually – by other researchers, who have argued that in principle, considering multiple techniques could significantly outperform any standalone technique in fault localization [21]. However, this body of research did not consider a combination of different fault localization techniques that rely on completely different data sources (like source code and bug reports, as is the case for SBFL and IR). As such, we argue that there are still open questions about the efficacy, utility and difficulty of combining disparate techniques in fault localization. We discuss this previous research further in Chapter 2.

1.2 Research Objective

The main objective of this thesis, therefore, is to report on observations regarding the relationship between SBFL and IR when used for fault localization. In other words, the thesis investigates how the techniques can be combined, and what results can be produced through the combination. The importance of combining these techniques is:

a) When fault localization results using IR techniques are unable to precisely identify fault location, the SBFL technique may provide more precise information.
b) When fault localization results using SBFL techniques are unable to localize faults, the results may be supplemented with analysis from the IR technique.

c) When both IR and SBFL techniques are unable to localize faults, the Text Search technique might help to shed light on the location of faults.

In particular, this thesis intends to assess the combination of these techniques via experiment, particularly focusing on accuracy (i.e., how close is the predicted location of the fault to the actual line or lines of code containing the fault) and performance (i.e., time to deliver a result)– which in turn can help us understand the benefits gained from using these techniques together.

1.3 Motivation

Every year, trillions of dollars are lost[1] due to software failure [2]; in some cases, failures have led to fatalities especially in domains such as avionics/aerospace, and medical devices (e.g., Boeing-737 MAX 8 crashes[3]). Because of the cost of managing and mitigating software failures is so substantial, techniques that can help reduce that cost quickly and efficiently are needed – hence the research interest in fault localization (and fault detection). In an ideal world, better fault localization techniques may lead to more productive developers and reduced need for organizations to hire more developers to manage faults and failures.

Though much research has focused on the challenge of identifying and eliminating faults in code before deployment, in general fault localization is still considered to be one of the most expensive [27], [28], [29], [17], tedious and time consuming activities

[1] Cost of software debugging, 2019. URL https://goo.gl/okoj21.
[2] Report: Software failure caused $1.7 trillion in financial losses in 2017. URL https://tek.io/2FBNl2i.
[3] https://goo.gl/GwXv6H

in the debugging activity [28] , [20], possibly since this debugging task is very often carried out manually, without support of efficient and effective tools. Despite much research that aims to automate aspects of fault detection and localization, as mentioned earlier, most fault localization in practice is carried out manually, e.g., where the programmer has to insert print statements and breakpoints (i.e., checkpoints, pause-on-value) check the stack trace, and reverify failing test cases [3] to help to localize faults.

For more than two decades, the automation of fault localization research tools has been an active research area [30] [31] where it has been hypothesized to lead to efficiency and productivity improvements in the overall debugging process. Advanced automated tools under active investigation include Spectrum Based Fault Localization (SBFL), Mutation Based Fault Localization (MBFL), Program Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization [17] [21]; these are discussed in more detail in Chapter 2. Across these different techniques, there is use of different information sources; some rely on evaluation of code via tests whereas others rely on exploiting supplementary information sources, such as code comments and documentation, or bug reports that contain a textual description of the fault [32]. Similarly, different outputs are produced by each technique, which can eventually be used to hypothecate fault locations.

In SBFL, the location of the fault is determined after *suspiciousness* results are generated. The suspiciousness value for each line of code is calculated using probability formulae that are based on test cases. The calculation of pass/fail test cases will be counted using the formula to generate the probabilistic result which are also called suspiciousness values. The idea behind this is that the line of code with the highest suspiciousness value holds the greatest chance that this line of code contains a fault. This idea has been a state-of-the-art technique for some time and has been the basis of substantial research. We explain more about how SBFL and suspiciousness values work in Chapter 2. What is interesting to observe about this research is that the suspiciousness value generated from the SBFL technique for fault localization is not always accurate, in part because it is based on the proportion of pass/fail tests, and tests may be inaccurate, incomplete, or even wrong.

As such, various extensions to SBFL have been proposed, such as [33], who considers an output of a fault localization tool to be *effective* if the root cause of the fault appears in the Top 10 most suspicious program elements. Some of the most recent research on SBFL has proposed further extensions and changes, e.g., using the concept of "Top N" of the highest scores of suspiciousness to improve overall accuracy [34], [35], [36] , [37] where "N" represent a limit (e.g., N=10) such that the correct answer (i.e., the location of the fault) is assumed to be within this limit. What this all suggests is that there are still open questions about the accuracy of SBFL, and many indications that by itself, it can sometimes be inaccurate in predicting the location of a fault.

Research made by [21] performed an experiment to compare all standalone techniques in the fault localization family, including Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization (MBFL), Program Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization. These experiments found that SBFL is the most effective standalone fault localization technique compared with the other individual techniques. In particular, in these experiments, SBFL managed to localize about 44% and 43% faults in the Top 10 result. These results, while interesting, demonstrate that there are still opportunities to improve on the accuracy of fault localization.

This “Top 10 approach” has been used as an important concept in many other research papers. I have used this indicator in this thesis to assess the suspiciousness list generated. However, a problem arises when the actual location of the fault is not in the Top 10 of the suspiciousness results. SBFL techniques aim to pinpoint faulty program elements by sorting them only by their suspiciousness scores; developers tend to resort to a different debugging strategy if they do not find the fault in the first positions of a suspiciousness list [3]. The question is “How can a developer determine the location of the fault if it does not appear in the suspiciousness result, or it does appear but not in the Top 10 list?”. What additional information can they use to assist them in fault localization especially if it involves a large program? This will be discuss further in chapter 4.

Manually finding fault in code, typically line by line, using standard output statements, is tedious and time consuming. It may be feasible if it involves small programs or simple ones, but for large-scale programs, this approach is impractical. Though there is no precise definition of what constitutes a small or large program, lines of code (Loc) or number of lines, is a software metric that is often used as a measure for program size [38]. [39] proposed that a large program is one that contains at least 10,000 Loc, while a small program is one that contains less than 2000 Loc. All programs containing in the between the Loc stated are considered to be medium-sized programs.

The accuracy of fault localization, as argued by Xuan et al [16], may be further enhanced by considering extra information sources (beyond source code). This is the motivation for my research – i.e., to study when the combination of more than one fault localization technique has a positive enhancement on fault localization performance.

1.4 Research Hypothesis

The *hypothesis* of my research is therefore: the combination of more than one fault localization technique increases the accuracy and performance in locating faults. IR-based techniques assist programmers in sorting through vast amounts of data or information as quickly and efficiently as possible [40]; these techniques measure, for example, the textual similarity between a bug report and the source files. As such, IR-based techniques take a bug report and source file as input, rather than a set of test cases, and generate a list of relevant source code files as output based on the bug report query [41].

IR-based fault localization techniques are static: they do not require program execution information, such as passed and failed test cases [36]. As such, they are complementary to SBFL-based techniques which are dynamic. Therefore, the combination of IR and SBFL may provide value, and give additional advantage, as they are based on different types of information.

Just as was the case with SBFL, IR-based techniques may be unable to accurately locate a fault; this is true of all fault localization techniques – sometimes they simply cannot with any degree of accuracy locate the fault. With IR-based techniques, what sometimes happens is the topic set generated from bug reports is insufficiently precise to locate the fault; this is perhaps analogous to SBFL not identifying the actual fault in the Top 10 potential suspicious locations.

As such, we argue that SBFL and IR techniques are complementary – both in terms of the information sources they operate on, and in the way in which they operate (i.e., dynamic versus static), and potentially could be beneficially combined. In a sense, our hypothesis fits within the body of work demonstrating that combining static and dynamic analysis can be beneficial [42].

1.5 Research Questions

Though the combination of the most accurate technique (SBFL) and the fastest technique (IR) [21] for fault localization may produce a promising result, there may be situations in which the combination does not produce better results than the individual techniques (or, indeed, may produce worse results). In this situation, adding another analysis approach, after combining two sophisticated techniques, may unduly increase the time and expense of fault localization - unless any further, additional technique is uncomplicated and inexpensive to apply.

To this end, I will further investigate the addition of text search techniques with the combination since text search not only compatible with SBFL and IR, it also complements them, as it is a fast and accurate analysis. Text search techniques exploit the same information sources as SBFL and IR – namely source code and bug reports – but operate differently, and produce different results, and we wish to investigate whether this combination of three complementary techniques has advantages. This kind of combination (SBFL, IR and Text Search) has, to the best of my knowledge, never been done before.

I therefore propose the following research questions:

RQ1: How accurate is the process of localizing faults using SBFL, IR and Text Search techniques individually?

Answering this question provides a baseline to understand the accuracy and performance of widely-used techniques in fault localization. The formula used for each technique will be explained in details in Chapter 3. However, the accuracy are based on the Top 10 results as explained earlier.

RQ2: How accurate is the process of localizing faults using the combinations of SBFL, IR and Text Search?

Answering this question will verify the possible combination of different techniques in fault localization and evaluates the performance and accuracy of the combined techniques.

RQ3: What is the execution runtime performance of the standalone techniques and combined techniques? And how long is the average of execution time for each technique on one bug?

The previous questions concerned the fault localization standalone and combined technique accuracy, while this question considers the performance or execution time for each of the techniques. The time taken to execute all experiment are recorded to measure the performance.

RQ4: Which combination of techniques is the most accurate and performant (fastest in runtime)?

This research question compares the accuracy and execution time of the combination technique to determine which combination is the most accurate and performant.

To find the answer to these research questions, a set of experiments using Defects4j datasets will be performed to observe the individual results for SBFL, IR and Text Search, the combination of techniques that complement each other, and to identify which combination's performance is the best in term of accuracy and execution time. We intend to cover all types of faults that developers can encounter (types of faults are discussed in Chapter2) in order to make maximum use of available datasets.

1.6 Summary and structure of the thesis

Chapter 2 briefly explains the fundamental definitions and basic terminology that will be used in this thesis, and the background reviews including related work of research in fault localization. Each fault localization technique will be discussed and previous research on hybrid techniques will also be highlighted.

Chapter 3 is where the experiment methodology will be explained in detail. It will include the Experiment plan and Experiment operation in detailed manner.

In Chapter 4, the result and analysis of each technique either individually or as a combination will be discussed. Threats to validity of this research will also be highlighted.

Chapter 5 summarizes the thesis, draws some conclusions, and proposes future work.

2 Background and related work

This chapter provides an overview of the background, as well an analysis of related work underpinning this thesis at the end of this chapter (in discussion of related work subsection). The discussion of related work also includes an overview of research on the integration or combination of different fault localization techniques, along with the context in which these integrations have been considered.

2.1 Fundamental definitions and terminology: fault, error, failure and test

To avoid confusion in further discussions, I will first present the fundamental definitions and terminology used in this thesis. According to IEEE Standard [43], a *fault* (which is sometimes described as a "bug" or "defect") is an inaccurate step, procedure, or data definition in a computer program introduced by a developer or a programmer of the program. According to [44] in 2006, an ISTQB Glossary recognizes that a "bug" and a "fault" have the same meaning.

According to De Souza, an *error* is frequently used to show a wrong state throughout the implementation of the program [39], whereas [45] defines error as a form of source code issue that inhibits successful compilation or execution. *Failure* is the term used to indicate a system's inability to work according to expectations [43] including unexpected output or incorrect data.

To this point, "fault" and "bug" are synonyms. This situation can also be seen with "error" and "failure" terms where both are also referring to the same issues only with different words. In short, faults or bugs in the code may cause an error or failure in program execution. To ensure clarity and avoid confusion, the interpretation that will be used for this thesis is the term “fault” (or sometimes “bug” as it reflects the dataset in Defects4j). “*Fault localization*” is a process where developers identify what caused the fault and attempt to locate where in the code the fault occurs [4].

Hristova [46] did significant research in analysing and understanding common errors or failures that occur amongst novice programmer; these errors have been divided into three categories: syntax errors, semantic errors and logic errors. Syntax errors refers to mistakes in the spelling, punctuation and order of words in the program, while semantic errors occur from a mistaken idea of how the language, or its execution engine, interprets certain instructions. Logic errors are general errors that may cause unintended results, sometimes even without failed execution.

The purpose of this explanation is to clarify the types of behaviour and phenomena that are usually faced by a developer and at the same time to give a reader a precise indication of terminology. However, we have not precisely prescribed which types of faults are under consideration, as we cannot predict the fault that occurs in a real world environment. We intend to cover all types of faults and failures that developers could encounter, since narrowing the faults to a certain type only will make the datasets that are available and useful for our experiments very limited. Fault localization tools are typically generic and are designed to detect or allocate all types of faults regardless of the types.

A test, or the process of testing, is one of the main sources of information for debugging, as well as a very important process in fault localization. Testing is defined as "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [47]. Testing is performed to make sure that the intended function of the program under a test is as expected. At the same time the test requirements (e.g., test cases, subject, etc.) are used as a guarantee that the code is widely tested, and most (if not all) of the program elements are executed [3].

2.2 Bug reports and source code

I will now briefly explain on the definitions of bug report and source code and how these are generated. *Bug reports* can be defined as a document that usually contains a detailed description of a failure [48]. A fault location can potentially be detected from a bug report, since it contains information that can link or related to a fault. However, since a bug report is usually written by a user of a program from various backgrounds and expertise levels, the quality of a bug report really varies. Some fault localization activities are slowed down by a bug report that contains inadequate or incorrect information, whereas locating the fault from a poorly written bug report takes more time and reducing the accuracy of fault localization.

Unlike *source code* - instructions for a computer [49] that is written by programmers [50] and normally written in structured manner - anybody (not restricted to a programmer) that uses a program who happens to encounter a fault or a bug can write a *bug report*. This diversity of potential creators of bug reports is one reason why there is much debate on bug report quality; past papers [37] [51] have highlighted the importance of a bug report that contains a quality content. However, there is as of yet no agreement on how we define a good or high quality bug report. I will briefly discuss on bug report quality in this subsection and will elaborate more in Chapter 3 and Chapter 4 with some examples. The examples will include a comparison of a good quality bug report with an incomplete bug report and it's results.

A complete and precise bug report is specified using a combination of bug report's title and its description [42]. However, [48] argues that a high quality bug report is one that also includes a code attachment or code snippets. To investigate this further, [37] evaluated the importance of specific program constructs (such as class names and method names) in bug reports, and argued that the greater use of program constructs increased the quality in bug reports and may lead to improved fault localization. In a different experiment, [52] found that the best results from their IR-based fault localization depends on there being similar textual characteristics between bug reports and source code.

A bug report is important so that programmers can use it to attempt to map the issues raised in the report to the fault location in the source code. Consequently, insufficient or inadequate information in a bug report may prevent accurate results being generated in an IR based fault localization process. Source code – produced by a programmer – typically consists of a set of executable commands and comments. Research carried by [53] found that comments and literals play an important role in the performance of the fault localization tools.

As a summary, in general a bug report is a document that can be written by anybody that uses the program or software, ranging from a user that may have little to no background of programming (i.e. admin or user for a system or website), to a developer at any level of expertise. As for my experiments, the Defects4j dataset will be used including its bug reports and source code. Source code normally is written in a structured manner that depends on the programming language used. Although the writer of the programs is oftentimes more than one person (i.e. a team of programmers), we can still encourage programmers to include comments and string literals to the code for better understanding.

By contrast, a bug report, which is not bound to any programming language structure, is written by not one but a variety of people with different levels of expertise and knowledge: the person who uses a program is not necessarily the person who wrote it. It is beneficial to recommend a template for bug reports (e.g. including title, descriptions, code attachment or code snippets) as the fault localization accuracy depends on them; however the decision to follow a recommended bug report template is really up to the individual filing the report. Both bug reports and source code will be discussed further in Chapter 3 including examples and suggestions for writing a bug reports that are suitable to use as part of a fault localization process.

2.3 Fault Localization Techniques

As was previously mentioned, the process of fault localization involves the developers attempting to find the source of the error and pinpointing its location in the code [4].

Usually, fault localization is performed manually, wherein developers would examine or search the source code for errors after failing test cases. This covers procedures such as adding breakpoints and print statements, inspecting the stack trace, and reverifying failed test cases [3].

Much research has been done to improve fault localization processes and practices, including providing a deeper grasp of the underlying theories and principles as well as the development of more robust tools to enhance automation fault localization. Today, full automation of fault localization is still an ongoing goal in terms of its applicability in an industrial environment, as the gap between a real-world environment and a controlled experiment environment are substantial. An example of automated fault localization tools that is quite popular and being used in debugging is Chrome DevTools[4] which is used in web development to test a web application, and GDB[5], a GNU debugger typically used to support fault localization in C or C++ projects.

In this section, I will briefly explain the popular categories of fault localization techniques. An introduction and discussion on wide variety of technique in Fault Localization is to inform reader of the breadth of FL techniques that are available. Different techniques of FL make use of different sources of information, and have different execution processes. As mentioned in previous chapter, SBFL is considered the most accurate technique, while IR is considered to be the fastest. Text search, on the other hand, has previously been chosen because of the simplicity of its execution.

[4] https://developer.chrome.com/docs/devtools/

[5] https://www.sourceware.org/gdb/

This will be explained further in Chapter 3,4 and 5. Other than the chosen techniques such as SBFL, IR and Text Search techniques, there are techniques such as program slicing, mutation based fault localization (MBFL), predicate switching and delta debugging. We now briefly explain representative instances of these techniques and the reason why they are not evaluated further in the experiments.

2.3.1 Spectrum Based Fault Localization (SBFL)

In the literature, a wide variety of fault localization techniques have been studied. A technique that relies on analysing execution patterns is called spectrum-based fault localization (SBFL), sometimes known as coverage-based fault localization; it is a dynamic debugging analysis technique that has attracted much attention [54]. It has received this attention in part due to its ease of use and effectiveness, as it has been shown to be one of the most effective individual techniques.

SBFL is a broad category of fault localization techniques. One of the first occurrences of it being discussed in the literature is [55] which proposed that program spectra (or sometimes are called program spectrum), such as code coverage, testing information, execution trace, execution path, path profile, and execution profile [56] [39] can be utilized in finding fault locations. This can be done by comparing the elements in the program spectrum after test execution, to infer and identify the fault location.

Building on this idea, SBFL analyses testing information (i.e. failures and passes of test executions) for a program based on execution of test cases to identify suspicious locations that may contains a fault. By using formula (e.g., Tarantula, Ochiai, Naish, etc.) or equations that involve a probability rule, testing information will be used to calculate a value that determine the possibility of fault location. This is called *suspiciousness*; suspiciousness range in value from 0 to 1, where the value 1 represents the most suspicious or high possibility to be the fault location and vice versa.

To explain this in simpler terms, when one statement is more likely to have faults or faulty elements than another, that statement usually has a higher suspiciousness score [57]. The suspiciousness values are computed based on how frequently the statements are used in passing or failing test cases. An element is considered suspicious if it is run more frequently in failed tests and less frequently in successful tests.

Numerous approaches to spectrum-based fault localization have been proposed in the literature [27], [58], [59], [10], [11], [60], [61], [62] and many of these approaches suggest various formulas that can be used to calculate the suspiciousness of a program [33]. In terms of evaluation, SBFL technique are generally assessed in terms of their *accuracy* and *performance*. We discuss these evaluation criteria further later in the thesis.

For example, Jones and Harrold presented the Tarantula formula [8] [27]. Later, Abreu et al proposed Ochiai [58], another formula in SBFL which in the experiments showed that they can outperform Tarantula and other formulae such as Jaccard [63] and AMPLE [64] in terms of accuracy (i.e. an increase in detection of fault location).

Additionally, research has been done to investigate the SBFL technique's drawbacks in order to better understand why it hasn't been widely used by practitioners. Much research has been done in optimizing and improving the formula to enhance fault localization including combining the fault localization technique which will be explained further in Subsection 2.4, Discussion of related work.

As mentioned in Chapter 1, although [21] found that SBFL is the most accurate technique in locating faults, unfortunately it is obvious when dealing with large programs with varies type of faults that the aforementioned requirements are unlikely to be met [65]. Though several improvements have been made on the formulas, which have been claimed to be better than the previous or existing versions, [66] demonstrated that the Tarantula [8] formula and the popular Ochiai formula [58] are both grounded in statistical approaches, specifically correlation. This in turn led them to conclude that focusing on improvement of the formulas only might not help SBFL produce optimum or even better results. As a result, it is from this paper that an argument came to light for combining SBFL with other fault localization techniques might lead to greater improvements in accuracy and/or performance especially when it involves a large program.

In 2014, Lucia et al. [62] investigated forty different association measures, and as a result proposed a formula used to quantify the strength of the relationship between two variables of interest. This study highlighted the fact that there is no one best measure that works in every situation and case. This finding strengthens our thesis motivation in combining different techniques in fault localization as it considers different success measure from different sources.

2.3.2 Information retrieval (IR) based techniques

In this section I will briefly explain IR-based techniques in general, and how they have been used in fault localization. Information retrieval (IR) based techniques aim to determine the location of the fault by using information retrieved from the program and its context (i.e., source code, bug report, etc) without running a test case execution; this makes it a static analysis technique, unlike SBFL which requires execution of test cases. IR techniques were initially used to index text and search for useful documents in a collection [67]. IR rapidly emerged from a narrow area that has been used by librarians and information experts to a mature state since the introduction of World Wide Web and has been applied in a variety of domains, including debugging [68].

IR has grown beyond its primary goal when after several years later, it was discovered that it may assist developer in debugging. To this end, the work of [69] paved the way by demonstrating that Latent Semantic Indexing (LSI) – a specific IR technique that could be used to find a fault in source code (fault localization). LSI is an approach that derives similarity measures between source code elements, and it has been used to map concepts expressed in natural language by the programmer to relevant parts of the source code, i.e., to assist in debugging and traceability.

IR, as applied to fault localization, is a static debugging approach that statically locates faults using different types of analyses, including bug models, reports and source code file, instead of relying on execution of a set of test cases, (i.e., running a test to analyse and allocate fault), without any actual execution of the program [37], [70], [71]. To be precise, IR fault localization techniques do not require program execution information, such as passed and failed test cases [36], but as a result, executing an IR technique for fault localization generates list of relevant source code files as output [41]; this should be contrasted with what is produced by SBFL techniques, which attempt to produce a more fine-grained and precise output, e.g., a line number or a ranked list of statements that are considered to be suspicious.

Akbar highlighted in his research that there are three generations of IR fault localization techniques that have been identified over a fifteen-year of research span [72]. The first generation of technique is based on simple bag-of-words (BoW) assumptions [69], [73], [71], [34], where in BoW concepts, the relevance score for a file to a query is based on the frequencies of individual query terms that appear in the files containing code. After that, the files are ranked according to their relevance scores before generated as output.

The second generation utilized richer and more diverse information to improve BoW-based techniques [36], [74], [75] [75], [76]. The information that was derived from structural information (e.g., source code and bug report), software-evolution related information (e.g., version histories), or evolution of the software project (e.g., historical bug report and code change) all bear an important role in making a richer set of information available to support fault location. Research carried out in this generation demonstrated that the performance of fault localization is improved by utilising structural information encoded in source code files [77], [37], [75], [78] such as method names and class names, and in bug reports [79], [75], [78], [80] such as execution stack traces and source code patches.

The most recent third generation focuses more on the term order as well as semantic relationship modelling; these have been taken into consideration to improve the IR tools in general [81], [82], [83], [84], [85], [86]. In IR, due to the nature of language use, the terms that represent a topic are often semantically related [87], [88], for example if the topic of the bug report's results is about "period", so the source code that contains topic "period" as the highest result might be the location of fault. This might help in contributing greater accuracy in fault localization using SBFL.

The tools used in the research by [89] used word embeddings based on *word2vec* modelling [90] of textual data to incorporate contextual semantics. This research uses the Markov modelling ideas first established in the text retrieval community [91] to exploit the term order in the context model. The third generation of IR technique also make use of hybrid techniques in IR, which has led to the inclusion of machine learning and deep learning. This will be explained further (in Section 2.4) where I give a discussion of related work.

In order to locate the components or part of the program that need to be adjusted in order to fix a fault, IR techniques are used in fault localization. As a result, they help programmers navigate through massive amounts of data or information as quickly and effectively as possible [40]. These techniques do not attempt to pinpoint every component of the program that must be fixed. Instead, they aim to locate a starting point from which the bug can be fixed. They are thus possibly complementary to other SBFL techniques, which aim to more precisely narrow down faulty code in the program as this technique is the fastest in locating fault compared to other techniques [21]. Conceivably, then, one could envisage a workflow where coarse-grained fault localization is carried out by IR, and then more fine-grained assessment could be carried out by SBFL based techniques, amongst others.

In the next subsection, I will explain further details on different IR including those that will be used in my experiment.

2.3.2.1 Text retrieval techniques

Very often, debugging and fault localization starts from a bug report filed by an external user. Advanced knowledge and thorough understanding of how a system is built and its various components interact are required in manually attempting fault localization from the information provided in a bug report. Various works have considered techniques to aid programmers in decreasing the human effort spent in fault localization activity. I'm using this technique from the IR technique as it is easily applicable, since it relies on documents such as source code, bug reports and comments. I will explain the approach in details in Chapter 3.

In simple terms, the text retrieval technique is one in which the system's source code is indexed into a search space and then queried for code related to a given bug report [92]. This is a typical IR approach that aims to locate faulty files by comparing the bug reports with the source files [93]. Numerous IR fault localization approaches that employ techniques to calculate the similarity between a bug report and a program element (e.g., a source code file) have been proposed [34], [94], [95], [35], [36], [37], [77], [96].

Lukins et al. used a *topic modeling algorithm* named Latent Dirichlet Allocation (LDA) for bug localization [94]. In the IR community, historically, the Vector Space Model (VSM) was proposed early on and is considered a mature retrieval technique [97]. Its development was followed by many other IR techniques, including Unigram Model (UM), Latent Semantic Indexing (LSI) [69] and Latent Dirichlet Allocation Model (LDA) [87].

UM and VSM would be considered to be the simplest way to represent documents for the purpose of IR; these techniques determine the similarity between two objects (i.e., documents) [97] using algorithms such as Cosine Similarity, Euclidian and etc. LSI is used to cluster source code in order to identify abstract data types in procedural code and clones [69] so as in order to calculate similarity measurements between source code elements. On the other hand, LDA is a probabilistic and fully generative topic model for extracting latent, or hidden, topics from a series of documents and modelling each document as a finite mixture over the set of topics [87] where each topic in this collection is a probability distribution over the terms that make up the document collection's vocabulary.

Rao et al, evaluated the performance of several standard IR techniques for bug localization including VSM and UM and LDA [34] by conducting a comparative study between the tools on fault localization task. In the study, they found that simple text models such as VSM and UM perform better than more sophisticated models such as LDA where in general almost 50% of the total fault of relevant file at the rank of 10 can be retrieved. Since then, numerous works have improved the effectiveness of standard IR models by considering more information, applying advanced techniques, and refining queried bug reports.

Zhou et al proposed an approach named BugLocator that includes a specialized VSM (named rVSM) that considers the similarities amongst bug reports to localize bugs [36] then ranks it. However, [37] by using a different dataset with [36] proposed an approach that find the similarities of the structure of source code files and bug reports and employs structured retrieval for fault localization. They found that structured information retrieval based on code constructs, such as class and method names, increase the accuracy in fault localization. The experiments also show that it performs better than BugLocator.

2.3.2.2 Stack trace techniques

Stack trace is a technique in IR where the traces of test case execution are used to find the fault location of the program. In order to try and find a fault using the stack trace technique, the test cases must first be run at least once. When the test cases are executed, the list of active stack frames are generated during the programme execution; this list is known as a stack trace. Each stack frame corresponding to a function call that is still in progress and not yet returned. Developers can find important information from stack traces while doing debugging activities. When the system crashes, the stack trace indicates the currently active function calls to the developer and point where the crash happens [21].

Usually, bug reports that are generated automatically from an error also contain this stack trace information; the same holds for an error log that is attached to a bug report. Searching for text in an error log that is attached to a bug report to identify the location of fault is reasonably accurate indicator. Unfortunately, not all automatically generated bug reports include an attached error log; it is only present in a small percentage of bug reports [80]. Segmentation and stack-trace analysis were suggested as a way to enhance fault localization performance by [75], and [79] proposed the Lobster technique, which computes the similarity between the code elements and the source code of the programs using the stack trace information that is recorded in the bug report. In other words, instead of using bug report like in text retrieval technique discussed in section 2.3.2.1, to find the similarities in source code, stack traces information in a bug report is used to find the fault in source code documents (similarities).

The latest research by [98] presented the first approach to computing stack trace similarity based on deep learning based techniques; their study demonstrated that their approach outperforms state-of-the-art approaches on both private JetBrains dataset and open-source NetBeans data. However, stack trace analysis usually works on crash faults - a fault generated from system crash (e.g., caused by syntax errors, semantic errors and logic errors) - and have also been shown to be less accurate for other type of faults, hence it will not be used in my thesis.

2.3.2.3 History-based techniques

History-based fault localization is a technique that only needs to examine the development history of the code [21], which are usually used for fault prediction. – for example, by tracking densities of faults across Git repository branches. This technique ranks the elements in a program by their likelihood to be defective. Generally, fault prediction and fault localization are considered as different techniques; moreover fault prediction is typically executed before any failure has been discovered [99]. Unfortunately, this technique will not be used as it involved a history of the code which are quite impossible for me to access due to time and other restriction.

Dallmeier [100] introduced a tool named iBUGS, a tool that semi-automatically extracts benchmarks for fault localization from the history of a project by collecting all past successes and failures of the project. These benchmarks are useful for both static and dynamic fault localization tools. [35] also proposed a version history aware fault localization technique which considers past buggy files to predict the likelihood of a file to be buggy and uses this likelihood along with VSM to localize fault.

Wen [81] proposed an approach named "Locus" to locate bugs from software changes; their approach was evaluated on six large open-source projects. While [101] proposed an approach to prioritizing test cases based on historical data where the priorities of test cases are identified based on requirement priorities and then are calculated dynamically according to historical data in regression testing. [102] proposed a technique that utilizes the information of bug fixes across projects in the development history to effectively guide and drive a program repair process.

Zou [21] made an empirical study of fault localization techniques and found that the SBFL technique is the highest in accuracy where it successfully locates up to 68% of bugs in Top 10 result generated in experiments, compared to other techniques. They also propose that a combination of several techniques in fault localization can potentially increase the accuracy of the results, but do not explore this idea in further detail. Nevertheless they are one of the originators of the concept that has led to this thesis.

As a conclusion, the IR technique has much potential in assisting programmer in fault localization, as it allows exploitation of many variations of data such as bug report, source code, stack traces, and history records. However, executing IR techniques alone is not enough in achieving an accuracy of fault localization results as an information such as a post-mortem analysis program execution is also one of many pieces of information that are not only critical but also can contribute in finding the bug location, hence increasing the fault localization accuracy. Combining the IR technique with other techniques that contain different types of information (e.g. SBFL) might address this issue faced in IR fault localization analysis.

2.3.3 Program slicing techniques

A program slice is a segment or some part of the program [103]; slicing itself is a technique that is used to build a slice of a program that will be explicitly tested under specific assumptions. As a debugging tool, program slicing was developed to condense a program to its bare minimum while preserving a given designated behavior [5]. There are two types of slicing that are widely mentioned in the literature: static and dynamic slicing. Static slicing involves larger program slices than dynamic slicing since it takes into account every conceivable execution of the program during testing. It solely requires the source code and particular inputs for all possible program executions.

Static slicing gave rise to the improved approach known as dynamic slicing where it only takes into account a specific program execution; as a result, it slices all statements that really modify a variable's value for the specified program inputs. Therefore, dynamic slicing concentrates on a single execution for a particular input [6]. A set of variables at a program location that might have unexpected or undesirable values are called a *slicing criterion*. In other words, the main distinction between dynamic and static slicing is that static slicing includes executed statements for all potential inputs, while dynamic slicing only includes executed statements for the specific input. Dynamic slices are more efficient in term of performance and accuracy compared to static slices for debugging since dynamic slicing are substantially more specific and focused [104].

Just like the SBFL technique, a slicing technique requires a tracing execution of the test cases at least once. The primary distinction between these two techniques (SBFL and slicing) is that a slicing technique only has to trace a failed test case, while SBFL must trace all test cases involved (either failed or passed test cases) in order to calculate the suspiciousness value. The empirical analysis of the relationship between dynamic slicing and the SBFL technique for fault localization conducted by [105], revealed that while dynamic slicing can accurately allocate faults for program that contain a single fault, the SBFL technique performs better in terms of accuracy for program that contain multiple faults or more than one faults. However, in the same literature, it was also suggested that a combination of technique in fault localization could produce an optimal outcome in terms of fault localization accuracy. Since the availability of relevant tools is very poor, this technique is not included in the combination of fault localization experiment for this thesis.

2.3.4 Mutation-Based Fault Localization (MBFL) techniques

Initially, Mutation-Based Fault Localization (MBFL) was proposed by [106] and [107]. The main concept behind MBFL is to mutate a program to simulate faults and then observe the effects of the mutation on test results. When a program is subjected to a mutation operator in MBFL, a mutant version of the original program is produced. There are various types of mutations incorporated in the mutation operators included statement mutations, operation mutations, variable mutations, and continuous mutations. These operators are designed to mimic the errors that programmers make while choosing constants and identifiers for expressions, composition expressions function, and composition function utilizing iterative and conditional statements [108] .

In contrast to how a program is typically executed, MBFL use information from mutation analysis [109] and links the results with the program's fault location. MBFL strategies determine whether the execution of a statement affects the result of a test by injecting mutants while in SBFL technique, it is assessed whether a statement is executed or not. In most circumstances, a mutant modifies an expression or a statement by substituting one operand or expression for another [110]. If a program statement affects failed tests more frequently and passes tests less frequently, it is more suspicious to be the location of the fault.

Constraint-based debugging is a method that [111] suggested is to limit the number of potential bugs. In a way, this helps to reduce the time spend to execute test cases for MBFL as this technique are well known in taking the longest time to compute the result since the it will involve all possible bug mutation. Mutants that make the failing test cases pass are used to suggest possible faulty sites. Later, [112] suggested a method that modifies both faulty and non-faulty statements using mutation. According to this logic, a faulty statement is more likely to be faulty if a mutant is inserted into it and the number of test cases that fail can be reduced.

Conversely, a mutant inserted in a correct statement which generates more failing test cases is less likely to be faulty. [113] proposed a similar approach for multilingual programs. Mutation testing is also used to seed faults for experiments, and to suggest fixes for program repair [114] [115]. [116] used mutation testing to generate faults and shows that these faults are similar to real faults.

In addition, MBFL requires a significant amount of time to complete since it involves mutating as many mutans as possible to increase fault localization accuracy. It also requires a significant amount of technology and software for operation and setup. Therefore, MBFL technique is typically the slowest technique compared to other fault localization techniques, and it is debatable as to whether it is generally practical, especially in light of SBFL and IR's significantly faster execution times and this is the reason why MBFL are not included as one of the techniques to be included in my thesis experiment though the result might be promising.

2.3.5 Predicate switching techniques

Though predicate switching [117] is similar to MBFL, this approach is designed to identify faults in control flow, or faults that relate to it. A predicate, sometimes also called a conditional statement, directs the execution of various branches. If a failed test case can be made to pass by changing the evaluated result of a predicate, the predicate is said to be a critical predicate, and it is argued that this could be the source of the defect.

Predicate switching begins by tracing the failed test's execution and identifying all instances of branch predicates. The tests will be re-run several times, where each time forcing a different predicate's outcome. If switching a predicate yields the correct output, the predicate is called a critical predicate since it potentially could be the cause of the error.

Predicate switching and MBFL approaches are similar in that they both use mutations and look at how the execution results change. However, because predicate switching alters the control flow rather than the program itself, we treat it as a separate family of techniques. Furthermore, as far as we are aware, predicate switching is not included in earlier work by [110], [118] as an MBFL technique or part of it. If a conditional expression has been tested numerous times during program execution in predicate switching, it only flipping reverses one evaluation at a time rather than run all evaluations, so in a time wise, predicate switching is faster than MBFL but not SBFL and IR technique as it also involves mutation operations.

2.3.6 Delta Debugging techniques

Delta debugging is a technique for simplifying an input of a failing program to facilitate debugging process [60] . Through series of experiments, it iteratively reduces the size of a failing input until it finds the smallest possible component of it that also leads to failure. The base of delta debugging is that a smaller input will covers less code, thus less debugging work is required. By simplifying input delta debugging has shown to be effective in identifying the root causes of failures.

The work in [119] narrows down the variables that contribute to failure in debugging. It makes use of memory graphs [120] to identify the shared characteristics between successful and unsuccessful runs. Then, it compares the program states that were brought by the shared variables in the two executions.

Burger [121] described a method for reproducing an observed failure in the simplest possible way. The interaction between program objects is captured and reduced from a single failing run to a simple unit test consisting of a set of method calls that reliably reproduces the failure. JINSI, a tool developed by the authors and based on the record/replay mechanism, delta debugging, and program slicing, enables the suggested technique.

The disadvantage of this technique is that that it can change the program state (or input) in ways that are impossible to achieve in the original context leading to unfeasibility of state or input and at the same time might obstruct debugging process.

2.3.7 Text Search

Text search is a widely used idea [122] that is supported by a variety of extremely powerful tools that allow for the search of text, images, signals, and audio [123]. The Semantic web [24], bioinformatics [25] and 3D modelling [26], have all effectively implemented automated text search, which has opened up previously unimaginable levels of information access. Historically, keyword based search has always been the primary method for finding and accessing information [124] and string matching algorithms are common and widely being used with the text search technique to recognise a specific word or sequence of words in a huge text document [125]. For the purpose of standardisation, text search technique are using a steps that are implemented in IR techniques such as stemming, lemmatization, and word embeddings in addition to indexes to find pertinent texts based on a search query [32].

To achieve high quality search results, improvement on text search has been made to use information retrieval to automatically improve queries as they can connect various information retrieval algorithms and analyse query based data sets [126]. Though text search has been successfully applied on various and different discipline and domain, it has received little attention in fault localization literature.

One recent research paper by [32] investigates generally how text search engines can improve existing fault localization approaches. This paper found that text search does improve fault localization performance and it is a useful extension to existing approaches. In this work, [32] introduced the Broccoli tools that combined several techniques in IR such as version history, report similarity, structure and stack traces with text search to improve fault localization performance. The performance of the proposed approach has been evaluated against seven state-of-the-art fault localization algorithms of IR technique on open source projects in two data sets.

My research expands on this idea by combining text search technique with two different state-of-the-art fault localization techniques: SBFL and IR technique. This highlight the contribution of my research where the improvement for fault localization are made and are different from any other existing approaches. Table 2.1 below shows the summary of fault localization techniques.

Techniques	Explanations
SBFL	SBFL techniques are using formulas that contain a probabbility calculation of the success and failed of test execution where the calculated value are called suspiciousness where it represent the location of fault in program.
IR	IR techniques use information to find fault location that retrieved from a program using source code and bug reports, without running a test case execution.
Program slicing	Program slicing is a part or segment (slice) of a program that can be explicitly tested under specific conditions in order to determine fault location. The hypothesis is that testing a slice is easier and cheaper than testing the entire program.
MBFL	MBFL inject mutation operators (statement mutation, operation mutation, variable mutation, continuous mutation etc.) of faults in a program and use the information from mutation analysis to allocate fault location.
Predicate switching	Similar to MBFL, predicate switching techniques use mutations to generate changes in output results, however, predicate switching only alters the control flows rather than the program itself
Delta Debugging	A technique that simplifies an input from failing program until it finds a smallest possible components in the program that leads to failure.
Text Search	This is a keyword based search approach for finding and accessing information using string matching algorithms in order to recognise a specific word or sequence of words in a huge text document.

Table 2.1: Summary of fault localization techniques.

2.4 Related work

Over the last decade, there has been research that combines different techniques in fault localization, typically to address technical or performance limitations. By necessity, techniques can be combined only if there is a common basis for integration, e.g., operating on the same or similar artefacts, producing the same or similar outputs. SBFL and IR technique are generally considered the two leading approaches in fault localization due to their extensive investigations in the literature [127]. Both SBFL and IR techniques ultimately generate a ranked list of program elements that likely contain a fault; however, they each only consider one source of information: either program spectra, or bug reports. [128] argue that this is less than optimal.

Different techniques in the SBFL family may contain strongly correlated information on real-world projects. To further improve fault localization effectiveness, extra information sources could be introduced [21]. Since this thesis focuses on SBFL and IR techniques for fault localization, further explanation and elaboration will be made on their combination. This subsection will explain research that has considered SBFL and IR techniques, and also the combination of various fault localization techniques with each other.

There has been much research that has been done in the last decade regarding fault localization; one key observation is that it seems to be more promising in terms of improving accuracy and performance - to find new information sources to consider when carrying out fault localization, instead of further optimizing use of existing information sources [21].

One of early research projects that combined techniques was made by [129] where they proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution, Latent Semantic Indexing (LSI) technique and a scenario-based probabilistic ranking (SPR) technique to improve accuracy of fault localization. They later rank the program elements based on their corresponding weighted sums. The result conclude that LSI and SPR, based on different analysis methods and data, complement each other, and the accuracy results obtained with the combined techniques are better than those of any of the techniques used independently in fault localization.

Liu [130] proposed an approach named SITIR that combined information from two different sources which is an execution traces and the comments and identifiers from the source code. SITIR filters program elements returned by LSI approaches, an IR technique if they are not executed in a failing execution trace . The results from the combinations experiment indicate that SITIR has higher accuracy in fault localization compared with individually fault localization technique and they planned to extend their works by combining other possible combination of information sources to support fault localization accuracy improvements.

Le [95] built and later extended [33] an automated oracle based on machine learning techniques that could predict whether the output of a fault localization tool was plausible before the developer proceed to localize fault. The successful combination of SBFL and machine learning in this research shows an improvement in fault localization accuracy. At the same time, this situation shows that capturing different dimensions of information or data from one entity (e.g., fault) can improve the existing technique as we might discover many potential and interesting data entities within it -

we just need an appropriate approach to execute it in providing increasing accuracy of fault localization.

Later, [131] proposed LDA-GA, a combination of Latent Dirichlet Allocation (LDA) technique and a Genetic Algorithm (GA) where [132] later on extended the works by the LDA-GA technique in TraceLab[6]. The LDA-GA approach used a genetic algorithm to determine a near-optimal configuration for LDA in the context of three different tasks including fault localization. The results demonstrate that LDA-GA has higher accuracy on all tasks as compared to LDA technique alone.

Another hybrid technique to allocate bugs has been considered in research by [16] and [102]; the authors separately combine SBFL and learning-based techniques. Learning-based technique is a family of supervised machine learning techniques that solves ranking problems [133] especially in IR technique. In [16] research, they proposed MULTRIC, a learning-based approach that combining multiple ranking metrics in SBFL technique, while on the other hand, [102] proposed *Savant* that also employs learning-based strategy that use elements such as suspiciousness score and likelihood of being a root cause of a failure to rank fault location. The result from both shows that MULTRIC and *Savant* successfully localize faults more accurate than state-of-art technique, such as Tarantula, Ochiai, and Ample.

Sohn [15] go further by combining learning-based technique, genetic programming (GP) and linear rank Support Vector Machines (SVMs). All these techniques are

[6] https://www.cs.wm.edu/semeru/data/tefse13/

combined with existing SBFL technique that has been extended by adding code and change metrics such as size, age, and code churn which have been examined in the context of defect prediction where all these features are used along with suspiciousness values from SBFL techniques to localize fault. They proposed FLUCCS, the first technique to use code and change metrics for fault localization, connecting automated debugging to the field of defect prediction for the first time. The results show significant improvement over the state-of-the-art SBFL technique in fault localization in term of fault localization accuracy.

There is also research by [118] where they agree that even though the SBFL technique is the most intensively investigated fault localization approach, SBFL may have limited effectiveness since a code element executed by a failed test may not necessarily have an impact on the test outcome and cause the test failure. They bridge the gap by proposing TraPT, an automated learning-based technique to combined with MBFL technique. TraPT explore the obtained mutation information to achieve precise fault localization and the result from the study shows that TraPT outperform individually state-of-the-art MBFL and SBFL.

Wen [134] argue that SBFL accuracy is still limited as the test coverage information leveraged to construct the spectrum does not reflect the root cause directly and SBFL also suffers from the tie issue (same rank of *suspiciousness* value) so that the buggy code entities cannot be well differentiated from non-buggy ones. They suggested that a combination of SBFL technique with History-based technique to improve fault localization accuracy. The approach records the version histories on how bugs are introduced to software projects. This information reflects the root cause of bugs directly, whereas at the same time, the evolution histories of code can also assist to differentiate those suspicious code entities when ranked as a tie by SBFL. As a result, the HSFL approach outperforms the state-of-the-art SBFL techniques significantly in fault localization accuracy.

Next, it was suggested by Yoo et. al. [135] to use genetic programming (GP) to create new suspiciousness score formulas that would perform better than the human-designed ones now in use. They claimed that no human has ever been able to construct a formula that can perform better than the one developed by GP, according to theoretical and empirical evidence presented in their study. They found that in the single failure case, GP outperformed the best-known human-designed formula when applied to SBFL. It also at the same time serves as an illustration of the limitations of spectrum-based procedures and a plea for fault localization methods to consider signs other than program spectrum as pursuing the greatest formula is no longer a viable research goal for SBFL. Yoo also suggest that in the future, all work that are related to fault localization are encouraged to designing a formulae that are effective in certain contexts, such as a specific project or a particular type of faults (specialisation).

In a systematic empirical investigation on the application of Statistical Debugging (SD) techniques with the combination with SBFL, [18] proposed PREDFL (Predicate-based Fault Localization). The results of the experiment demonstrate that PREDFL can further enhance the accuracy of state-of-the- art fault localization technique. While [136] provide a method for merging MBFL and SBFL to increase localization accuracy, the combination managed to surpass state-of-the-art MBFL (MUSE and Metallaxis) and SBFL (Dstar and Ochiai) techniques.

In a different approach by [137] they merged numerous IR techniques with learning-based approach in their research to improve the accuracy of fault localization. They combined textual similarity with other helpful IR information, such as version history, comparable bug reports, source code structure, stack trace, and other features, using learning-based techniques, to improve fault localization. As a result, their method out-performs BLUiR [37] and AmaLgam [101] two other hybrid IR tools. Tools like BLUiR and AmaLgam leverage hybrid information from IR, like stack traces, version histories, text structures, similar reports, etc., to increase the accuracy in fault locali-zation.

It is interesting to know how closely various procedures are connected to one another since the information sources used by each techniques in different families are vary. The authors of the research [21] discovered that several SBFL family technique may contains highly strong connected data on real world projects. According to the research, more information sources should be added in fault localization process in order to increase the effectiveness of fault localization rather than just focusing on the one family only. This is a key argument for combining SBFL with other techniques, and is a motivation for us considering the combination of SBFL with IR in this thesis. This also has been supported by many recent studies that integrating more information sources significantly outperforms any techniques of fault localization alone, and the combined techniques can significantly outperform any standalone technique. Table 2.2

below shows a summary of related work regarding hybrid or combination in fault localization research.

Author (Year)	**Fault localization Technique**	**Datasets**	**Size of program (LOC)**	**Result**
Poshyvanyk (2007)	LSI (IR)+ SPR (PROMESIR)	Eclipse (v2.1.3)	2.4 M	PROMESIR improved fault location ranking result compared to LSI and SPR techniques alone.
		Mozilla (v1.5.1)	3.7 M	
Liu (2007)	LSI (IR) + Execution trace (SITIR)	JEdit (v4.2)	88, 000	SITIR improved ranking result of fault location compared to other combined techniques (PROMESIR, DFT, etc.
		Eclipse (v2.1.3)	2.4 M	
Le (2015)	SBFL + Machine Learning	Siemens test suite (print_token, print_token2, replace, schedule, schedule2, tcas, tot_info)	2,563	Precision, recall, and F-measure of up to 74.38 %, 90.00 % and 81.45 %, respectively.
		space	6,218	
		Nano XML	4,782	
		XML-Security	22,318	
Panichella (2013)	LDA + GA	EasyClinic system	20,000	The results of the empirical studies demonstrate that LDA-GA is able to identify robust LDA configurations.
		eTour system	45,000	
		JEdit (v4.3)	104,000	
		ArgoUML (v0.22)	149,000	
		JHotDraw	29,000	

		eXVantage	28,000	
Xuan (2014)	SBFL + Learning based (MULTRIC)	Daikon (v4.6.4) Eventbus (v1.4) Jaxen (v1.1.5) Jester (v1.37b) JExel (v1.0.0b13) JParsec (v2.0) AcCodec (v1.3) AcLang (v3.0) Draw2d (v3.4.2) HtmlParser (v1.6)	unknown	MULTRIC localizes faults more effectively than state-of-art metrics, such as Tarantula, Ochiai, and Ample.
Le (2016)	SBFL + Learning based (Savant)	Defects4J (Chart, Closure, Math, Time, Lang)	321,000	Savant can successfully locate 57.73%, 56.69%, and 43.13% more bugs at Top 1, Top 3, and Top 5 positions.
Sohn (2017)	Learning based + GP + SVM + SBFL (FLUCCS)	Defects4J (Closure, Math, Time, Lang)	225,000	FLUCCS ranks the faulty at the Top 1 for 106 faults, and within the Top 5 for 173 faults compared to SBFL only manage to rank 49 fault for Top 1 and 127 for Top 5.
Li (2017)	Learning based + MBFL (TraPT)	Defects4J (Chart, Closure, Math, Time, Lang)	321,000	TraPT localizes 65.12% and 94.52% more bugs within Top 1 compared to MBFL and SBFL.
Wen (2018)	SBFL + History based (HSFL)	Defects4J (Chart, Closure, Math, Time, Lang)	321,000	Allocates and ranks at Top 1 for 77.8% more bugs as compared with SBFL, and 33.9% more bugs at Top 5

Jiang (2019)	SBFL + SD (PREDFL)	Defects4J (Chart, Closure, Math, Time, Lang)	321,000	PREDFL can improve the fault localization up to 20.8% improvement where the faults successfully located at Top 1.
Cui (2020)	SBFL + MBFL (SMFL)	Defects4J (Chart, Mockito, Closure, Math, Time, Lang)	332,000	SMFL techniques detect at least 2.36 times more faults than two SBFL techniques (DStar and Ochiai) and detect at least 1.86 times more faults than two MBFL techniques (MUSE and Metallaxis) in the Top 1 suspiciousness.
Shi (2017)	Learning based + IR	SWT ZXing Eclipse-3.1.	Not stated	Outperforms two state-of-the-art localization tools in IR which is BLUiR and AmaLgam.
Li (2020)	IR + SBFL + MBFL (IRBFL)	Defects4j (Chart, Lang, Math, Mockito, Time)	242,000	IRBFL can locate faults more accurately than the other five SBFL techniques (FLSF, Ochiai, Op2, Tarantula, and DStar).
Yoo (2017)	SBFL + GP	SIR datasets (flex, grep, gzip, sed , space)	32,284	The empirical study shows that GP-evolved formulae are the most practically useful in finding fault location compared to state-of-the-art formulae.

Zou (2019)	SBFL + MBFL + Dynamic slicing + Stack Trace + Predicate switching + IRFL + HBFL (CombineFL)	Defects4J (Chart, Closer, Math, Time, Lang)	321,000	CombineFL improves performance significantly: 200% (Top1), 63% (Top 3), 51% (Top 5) 31% (Top 10) increased in localized faults compared to the best standalone technique.
Le (2015)	SBFL + IR (AML)	AspectJ	>300,000	AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs when developers inspect the Top 1, Top 5, and Top 10 methods, respectively.
		Ant	>300,000	
		Lucene	>300,000	
		Rhino	<100,000	
Li (2021)	SBFL + MBFL + Deep learning (DL) (DEEPRL4FL)	Defects4J (Chart, Closer, Math, Time, Lang, Mockito)	332,000	DEEPRL4FL approach improves FL statement level baselines Top 1 results from 173.1% to 491.7% and FL method level baselines Top 1 results from 15.0% to 206.3%.
Li (2018)	SBFL + MBFL (MURE)	Siemens test suite (print_token, print_token2, replace, schedule, schedule2, tcas, tot_info)	2,563	Improved effectiveness by showing a 30% accuracy improvement over SBFL.

Table 2.2 Summary of related work regarding hybrid or combination in fault localization research

I’m also aware on a recent paper that has been published by Chen [138] on fault localization. Indeed it is a very inspirational paper however, the paper research are slightly different route than mine as it investigate the use of code change information for fault localization in continuous integration (CI) systems. They perform an empirical study utilising a data from real-world CI systems and compare the effectiveness of different fault localization techniques that use code change information with those that do not. As the conclusion, the results from the experiment show that code change information can be useful for fault localization, and they also concluded that CI systems can benefit from incorporating this information in their fault localization processes. They also emphasise that more research and study is needed to fully comprehend the impact of code change information on fault localization in CI systems.

2.5 Summary

In this chapter we have reviewed the key literature: foundations of different fault localization techniques, including SBFL, IR and text search; some of the key empirical results; and the fundamental research that has investigated combinations of techniques for fault localization. We observe that there has been no detailed empirical investigation combining SBFL and IR, as well as text search, to clarify whether they provide improved accuracy and/or execution time. This is the focus of this thesis. We continue this discussion in the next chapter, where we give an overview of our experimental methodology, specifically, how we will test the hypothesis and answer the research questions from Chapter 1.

3 Experimental methodology

This chapter will explain in detail the experimental methodology used in this thesis. This chapter consists of two subsections, Experimental Plan and Experimental Operation. The experimental plan section focuses on the experiment set up, including definitions of experiment context, hypothesis, selection of variables, and selection of subjects. The experimental design and the experimental instrumentation will also be explained in this section. The Experimental Operation section is a detailed explanation on overall experiment preparation and execution, including data analysis and validation.

3.1 Experiment Plan

The experiment will be conducted using the Goal-Question-Metric (GQM) framework [139] which is a de facto standard for experimental Software Engineering research. This framework has also been used in many other experimental research papers and has been referenced frequently. Before conducting the experiment, the goal needs to be defined first; in other word, what will be achieved after the experiment is complete?

The goal of my experiment is to observe the relationship between disparate techniques in fault localization, and to determine what (ideally beneficial) effects arise when combining different techniques. From the combination of different techniques, I must analyze which combinations are the best in terms of accuracy to localize a fault. The ranked results of the combinations will be summarized at the end of the experiments and the execution time also measured.

3.1.1 Research Context

In software engineering, an experiment or controlled experiment can be defined as an empirical investigation that modifies one element or variable of the research environment [139]. According to Wohlin [139] most experiments are conducted in a laboratory setting, which offers a high degree of control; in such a setting the goal is to change one or more variables while keeping the other variables at predetermined levels in order to determine if there is a measurable/observable effect.

The experiment in this thesis is a technology-oriented experiment based on simulation; there are no human subjects was involved in the experiment. The experiment is done off-line and will be run by an individual (research student) tackling a real problem in fault localization. The experiment will be using a real-world issues and problems (Defects4J programs) including faults. The goal of the experiment is to compare the combinations of techniques with individual fault localization techniques and highlight the best technique in term of accuracy.

Due to time limitations, there will be only a single experiment procedure, which will be carried out in three sessions by using different individual settings for each session. Each session in the experiment is using three well-known existing techniques (SBFL, IR and Text Search) in fault localization research. The results from each individual session will be recorded and the runtime for each session is also being measured. Each session in the experiment can be run in parallel or sequential as each technique is different from others and not related to each other. In short, each technique has different inputs and execution process however it generates a same output, which is a location of the faults. Further details on this will be elaborated in Section 3.1.5 Experiment Design.

Several tests were carried out before the real experiment took place to ensure that the results are consistent. For replicability and reusability purposes, the code, and the guidelines to run the experiment are uploaded on the author's GitHub[7] site so that in the future, this research on fault localization can potentially be continued by others. As the current fault localization in practice is carried out manually, these actions are also important to ensure the potential reproducibility of the results generated in improve fault localization.

3.1.2 Research Hypothesis

This subsection will reiterate and explain further on the research hypothesis and research questions for the experiment as mentioned in Chapter 1. My experiments aim to understand "What combination of techniques provides greater accuracy in fault localization?" and "How long is the runtime taken for each technique?". The hypothesis of my research is therefore: *the combination of more than one fault localization techniques increases the accuracy and performance in locating faults in fault localization activities.*

My research questions are as follow:

RQ1: How accurate is the process of localizing faults using SBFL, IR and Text Search technique individually?

[7] https://github.com/Nadhratunnaim/thesisNadhra

RQ2: How accurate is the process of localizing faults using the combinations of SBFL, IR and Text Search?

RQ3: What is the execution runtime of the standalone techniques and combined techniques? And how long is the average execution time for each technique on one bug?

RQ4: Which combination of techniques is the most accurate and performant (fastest in runtime)?

3.1.3 Variable selection

To run an experiment, variable selection must be done first. The variables must consist of dependent and independent variables. The independent variables are variables that we can control and modify in the experiment, while the dependent variables are the outputs derived from the experiments. For the variable selection for this experiment, the independent variables are the four combinations of the techniques, while the dependent variables are the accuracy of the fault localization technique(s) under investigation, and their execution time. The metrics used to measure them are firstly the accuracy, by ranking the *suspiciousness* values, and secondly the runtime in seconds taken for each technique to identify faults.

3.1.4 Selection of Subjects

In an experiment, selecting a subject to be executed is crucial especially when the data might reflect the generalization of the results. There are many datasets available that could potentially be used for experiments in fault localization and, more generally, debugging. However, some of the datasets are using seeded faults [140] [33] such as SIR (Software-artifact Infrastructure) datasets, and the use of a toy program [141] (i.e., Triangle program) in the experiment. This might not reflect real world software engineering environments and might provide results that are not indicative of the behavior of a real fault in a real program. SIR datasets have been considered before for such the experiment as they have been used by over 50% of software testing literature [142]. These datasets are, however, written in different dialects of the C programming language datasets, and most are far too obsolete to compile and run correctly though the latest versions of fault localization tools, including the tools considered in this thesis.

The experiment in this thesis will be executed on the Defects4j[8] framework [143], version 1.5.0. See Table 3.1 for details, including the name of each program in the framework, its purpose, number of faults and the line of code (LOC) that each programs contains which also determine the size of each program [38]. Defects4j program have been used for all sessions in the experiments to ensure comparability with other research, as well as uniformity. Defects4j has been used as a supporting resource for professionals in both software testing and debugging studies [144]. It has been used to evaluate the effectiveness of automated test generation and corresponding fitness function [145], automated program repair [146], [147], and fault localization [110] research.

[8] https://github.com/rjust/defects4j

Program	*Description*	*Number of Faults*	*Line of codes (Loc)*
Time	A standard date and time library for Java	27	28,000
Mockito	A mocking framework to write tests in Java	38	11,000
Math	A lightweight mathematics and statistics library for Java	106	85,000
Lang	A complement library for java.lang	65	22,000
Chart	An open source framework for Java to create chart	26	96,000
Closure	A tool to optimize JavaScript source code	133	90,000
Total		395	332,000

Table 3.1: Defects4J Dataset (version 1.5.0) with each program description.

There are other datasets than Defects4j, for example, the Bench4BL dataset [32] [127], which are also being used for IR experiments. However, the Bench4BL dataset does not have information other than bug reports that are relevant to fault localization; that is, insufficient information is available to calculate suspiciousness values and thus perform a fair comparison in the experiments. In other words, if the Bench4BL dataset is used, there are high possibility that the result for SBFL cannot be generated and it will be hard to compare with IR and Text Search, thus making it difficult if not impossible to combine all three techniques.

Defects4j is one of many benchmarks for real-world programs, and it provides an extensible set of reproducible bugs derived from Java software systems in the real world, along with a supporting infrastructure to use these bugs aims at advancing software engineering research [143], [144]. The Defects4j database contains of 357 bugs from 5 programs such as Chart, Closure, Lang, Math and Time (version 1.0.1) initially, and since then has grown into 835 bugs from 17 programs (Version 2.0.0).

With the total of 332,000 Loc for Defects4j datasets (version 1.5.0), each program under consideration in this thesis can be classified as a *large program* as it contains more than 10,000 Loc. [39] define a large program as one that contains 10,000 Loc or more. Many previous studies on fault localization have used Defects4J as their benchmarks [110], [102] [148]. Most importantly, Defects4J provides a faulty version and a fixed version of each project, something that is critical in assessing the accuracy of each fault localization technique.

Defects4j contains a suite of test cases for each fault, and this suite contains at least one failed test case that triggers the fault. The bug reports associated with failed test cases and faults replicate traditional real-world considerations in bug reports, and includes real-world information such as example code, output, and context. As such, it is a comprehensive set that allows us to (a) execute each individual fault localization technique; (b) execute combinations of fault localization techniques; and (c) compare the results of all executions fairly, given a common basis for comparison.

Due to time limitations, the latest version of Defects4j (which is version 2.0.0) was not used in the experiments; this version is quite new and there is little research that has as of yet been published using it. We instead use version 1.5.0 to enable more accurate comparison with other research, and to guarantee that we can reproduce failures and faults that are exhibited in previous research. Version 1.5.0 contains 6 programs with 395 bugs from Defects4j because of its stability since many researchers and experiment in Fault Localization has been used this version. This version has one additional program Mockito compared to the previous version (version 1.0.1).

3.1.5 Experiment Design

The purpose of this section is to explain how the experiment for my research will be carried out based on hypothesis and research questions. I will first explain the experiments for the individual SBFL, IR and Text Search techniques, then the experiments associated with combinations of techniques. The structure of this subsection consists of an individual experiment on the SBFL technique, an individual experiment on the IR technique, and an individual experiment on Text Search technique. Finally, I will describe the experiment that consist of the combination of SBFL + IR technique, SBFL + Text Search technique, Text Search + IR technique, and SBFL + IR + Text Search technique.

As mentioned before in subsection 3.1.1 Research Context, several tests have been carried out before the real experiment take place to ensure that the results generated are consistent across run. There also will be only one experiment that being executed; however, it will be run in three sessions (SBFL, IR and Text Search). For replicability and reusability purpose, the code and the guidelines to run the experiment are already uploaded on the GitHub site so that in the future, research on fault localization can be continued by others. We aim to support reproducibility and consistency across runs to provide greater confidence in the validity of the experimental results.

There are no individual tools that I can easily run for the experiment sessions, since some of the tools that are available to support these techniques are not accessible (i.e., the version is obsolete, no replication package included), and some of them proved to be difficult to use in a modern environment (i.e., their execution was not easy to reproduce and involving multiple setups with various programming languages). To ensure uniformity and the ease of use, I have developed and designed the tools for each technique from scratch based on my understanding of the architecture of the tools from research papers.

For consistency, all the techniques implemented were programmed in the Python language. The main reason why Python was chosen is because of the vast libraries support that Python has where I can find all the functions needed for the tasks in my experiment and there are no dependencies on external libraries. Python is a high-level programming language with a relatively straightforward syntax which makes Python code easier to read, write and learn than some alternatives (e.g., C++). It does give an advantage to people who already have a basic understanding of programming to understand the code; also, frequently, Python needs fewer lines of code to perform the same task as compared to other major languages such as C/C++ and Java. At the same time, I also intended to reduce any threats to validity of the experiment results, and enhance reproducibility by using a single language, as the setups with different programming language for an experiment might raise some issues that can be countered. Using a heterogeneous set of languages may introduce unexpected errors or inconsistencies due to unpredictable interfacing behavior between components written in different languages.

In summary, using a single language for implementing all experiments is simpler, more likely to enhance reproducibility and validity, and may result in code that is easier to read and understand.

3.1.5.1 SBFL Individual Experiment

This subsection is an explanation of the experiment that has been carried out to evaluate the accuracy and execution time of SBFL as a standalone technique. As mentioned earlier, amongst all fault localization techniques, SBFL is generally argued to be the most effective (accurate, fastest); amongst the family of SBFL techniques, Ochiai, one of SBFL formula to calculate *suspiciousness* has been shown to provide the best performance against all metrics [147], [21]. According to Vancsics et al, Ochiai obtained the best results [149] particularly with respect to locating individual bugs [150] but it was also effective at identifying multiple faults [151].

Unlike other formulae in SBFL, Ochiai is considered to be more effective for object-oriented programs [68]; thus, most SBFL based detection (and repair) tools also use Ochiai. Since all programs in Defects4j are Java based, and because of the robust and up-to-date nature of the framework, Ochiai has been chosen as the representative tool for the SBFL technique in this thesis. More specifically, I will use Ochiai formula for both individual applications of SBFL against Defects4j programs, as well as in combination with other (IR, text search) based techniques.

Most SBFL formulae including Ochiai are inspired from the probabilistic and statistical based causality models that consist of a program spectrum. In 1986 through his findings, [152] proposed that a program spectrum could be used for fault localization, where it contains the execution information details of a program, and it can be used to track a program behavior. For example, when the execution fails, the spectrum of information (i.e., failed test, success or passed test, total failed and passed, code coverage, etc.) can be used in fault localization to identify the suspicious location in the code that is responsible for the failure.

The Ochiai formula used in this experiment will be used to calculate the suspiciousness values for each line of code no matter either the test has passed the execution or failed. Then the suspiciousness values will be ranked from the highest values to the lowest values to narrow down the search for the faulty component that made the execution fail. The more frequently an element is executed in failed tests, and the less frequently it is executed in passed tests, the more suspicious the element is [21]. In short, the line of code or statement that has the highest suspiciousness values have the highest probability of containing the fault's location. The formula used for calculation of suspiciousness values proposed by [23], which is at the heart of Ochiai, is shown below.

$$\boxed{Ochiai(e) = \frac{\boldsymbol{failed}(e)}{\sqrt{\boldsymbol{totalfailed} \cdot (\boldsymbol{failed}(e) + \boldsymbol{passed}(e))}}}$$

Ochiai *suspiciousness* formula

In the formula, the following notation is used. A program *E* is a set of elements. Given a program element *e* ∈ *E*, we define the following notations:

• *failed(e)* denotes the number of failed test cases that cover program element *e*.

• *passed(e)* denotes the number of passed test cases that cover program element *e*.

• *totalfailed* denotes the number of all failed test cases.

Ochiai calculates passed, failed and total failed test cases in the execution, in order to calculates suspiciousness values to enable a list of ranking. Ochiai assigns a suspiciousness value to each statement in the program based on the number of passed and failed test cases in a test suite that executed that statement [23]. The intuition for this approach to fault localization is that statements in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases [27]. Figure 3.1 is an illustration of SBFL and the idea how it works based on above explanation on source code, program spectrum, Ochiai formula and the ranking results.

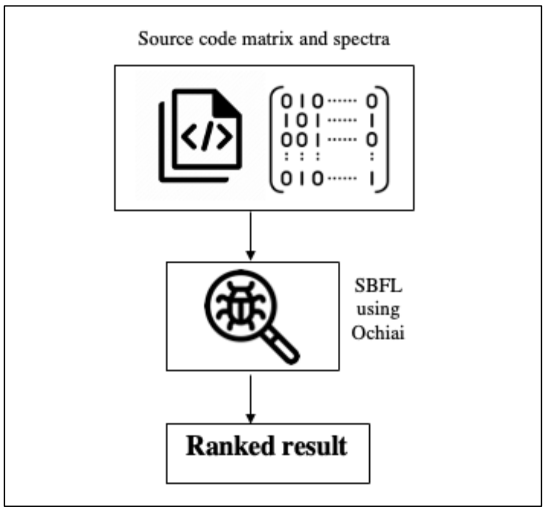


Figure 3.1: Spectrum Based Fault Localization technique (SBFL) framework

Figure 3.2 shows a snippets example of output for bug 5 in Lang program (L5) in the experiment, where on the left side of the Figure 3.2 is a result that generated from the experiment that displays every line of statement test case after SBFL experiment are executed using Ochai formula, while on the right side is a Top 10 ranking for the suspiciousness results after sorting the statement based on the suspiciousness values on the left side. The suspiciousness value is where the highest value is 1 and the lowest value is 0. As the full output raw data is quite big to be displayed in this section, it will be included in the appendix in this thesis.

Suspiciousness results output after execution		Ranking suspiciousness results output	
Statement and line number	Suspiciousness	Statement and line number	Suspiciousness
org.apache.commons.lang3.LocaleUtils $SyncAvoid#288	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#99	0.5773502691896258
org.apache.commons.lang3.LocaleUtils $SyncAvoid#295	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#99	0.5773502691896258
org.apache.commons.lang3.LocaleUtils $SyncAvoid#296	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#89	0.4472135954999579
org.apache.commons.lang3.LocaleUtils $SyncAvoid#297	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#92	0.4472135954999579
org.apache.commons.lang3.LocaleUtils $SyncAvoid#298	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#93	0.4472135954999579
org.apache.commons.lang3.LocaleUtils#57	0.0	org.apache.commons.lang3.LocaleUtils#96	0.4472135954999579
org.apache.commons.lang3.LocaleUtils#58	0.0	org.apache.commons.lang3.LocaleUtils#97	0.4472135954999579
org.apache.commons.lang3.LocaleUtils#42	0.2773500981126146	org.apache.commons.lang3.LocaleUtils#98	0.4472135954999579
org.apache.commons.lang3.LocaleUtils#46	0.2773500981126146	org.apache.commons.lang3.LocaleUtils$Syn-cAvoid#288	0.2773500981126146
org.apache.commons.lang3.LocaleUtils#89	0.4472135954999579	org.apache.commons.lang3.LocaleUtils$Syn-cAvoid#295	0.2773500981126146
org.apache.commons.lang3.LocaleUtils#90	0.0	org.apache.commons.lang3.LocaleUtils$Syn-cAvoid#296	0.2773500981126146
org.apache.commons.lang3.LocaleUtils#92	0.4472135954999579		

Figure 3.2: The snippet examples of suspiciousness results for L5 bug after execution and after ranking the suspiciousness to Top 10

Based on the aforementioned research questions and goal of the experiment, an implementation of the SBFL technique will return the result of execution as the *statement* which contains line number and file name or method name of the fault, along with the *suspiciousness* values. The suspiciousness values and corresponding program elements will be ranked in numerical order, typically from most to least suspicious as illustrated on the right side of the Figure 3.2 above.

The real location of fault for bug L5 is at “LocaleUtils” file, so as we can see from the SBFL output, the location is suspected to be at line 99 in “LocaleUtils” file with “0.5774” *suspiciousness* score calculation, in other word the highest suspiciousness amongst other line of statement. Unlike other techniques, SBFL can give a very detailed and fine-grained information regarding the fault up until the line number of statements, though in practice the real fault usually does not exist on the exact suspected line. This is quite possibly because of the difference between fault and failure: the fault is the mistake that we have made in writing our code, whereas the failure is what we observe in output. Sometimes the fault location and the code generating output that reveals the failure are different, leading to the SBFL algorithm not giving the exact line number containing the fault. There are likely other explanations, too.

3.1.5.2 IR Individual Experiment

This subsection is an explanation of the experiment that has been carried out to evaluate the accuracy and execution time of IR (information retrieval technique) as a standalone fault localization technique. Applying an IR to a fault localization exploits a variety of information, and this can be divided into several components such as document similarity, version history, structure, stack traces, and bug report comments [32].

In this thesis, the IR experiment I carry out makes use of the first option, i.e., document similarity, because it is one of the most widely used techniques, produces ranked results, and is generally the fastest approach amongst the many options available. Of course, there are many different document similarities approaches available in IR. I have chosen to use a Vector Space Model (VSM), one of earliest techniques in finding document similarity proposed by [97]. I chose VSM not only because it is a popular IR approach that has the advantages of being simple to implement and fast, but also it has a ranking system that is considered to be as accurate as a vast variety of alternatives [153].

Cosine similarity is a traditional cosine measure [154] that is often employed in information retrieval to determine the similarity between two objects (i.e. documents) that are represented as vectors. In this experiment, cosine similarity formula will be used to measure the document similarity between the bug report and source code text (represented as a text file document). Below shows the cosine similarity equation that is being used in this experiment.

$$\boxed{\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}},}$$

Cosine similarity from VSM text model formula

We define the notations above as below:

• **A** and **B** are the vector of object (e.g., Bug report and source code).

• n is the number of words.

In this calculation, each word in a document is notionally assigned a different dimension. A document is characterized by a vector where the value in each dimension corresponds to the number of times the word appears in the document. The cosine distance measures the distance for each word in the vector and the computed result from this calculation which contains the highest value of cosine are representing the most similar file. The most similar source code file to the bug report document is assumed as the location of fault. Other factors that also count as pivotal role in determining which IR technique to use for the experiments in this thesis include information availability (i.e., the ease of availability of bug reports and source code of the programs). Figure 3.3 shows the overall approach to running IR specific experiments in this thesis and I will elaborate further.

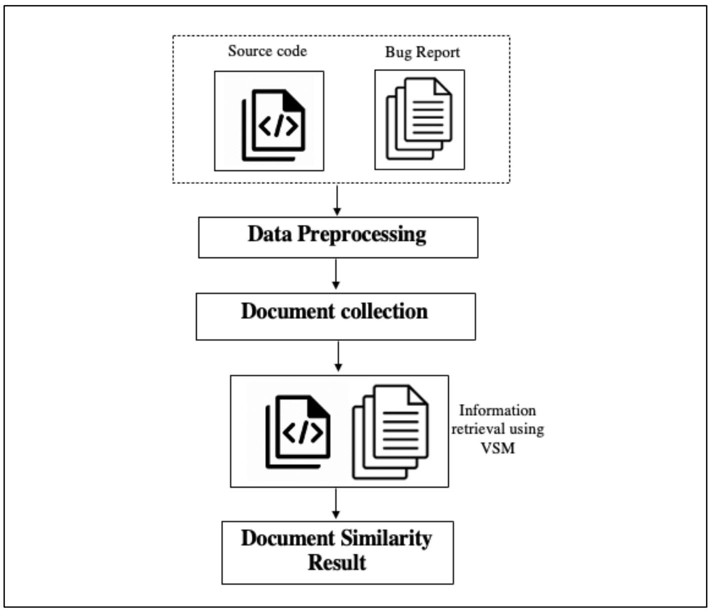


Figure 3.3: Information Retrieval (IR) technique using VSM framework

As described earlier, bug report information will be used to localize faults in source code files in the IR specific experiments. Unlike SBFL techniques, IR techniques do not require program coverage information, though their generated ranking is based solely on source code files [41]. As such, before conducting the experiments, the buggy version of the source code and the bug report of the Defects4j programs such as Time, Mockito, Math, lang, Closure and Chart are required.

Research made by [53] found that comments and literals play an important role in the performance of the fault localization in one of their IR research projects. Later also in their research, they grouped the source code into three group: identifier, comments and string literals. As well, combinations from these three groups are noted as being capable of generating more accurate results. We now briefly explain these categories, and their importance, as this impacts on the quality of the results we obtain in our experiments and allows us to control one potential threat to validity in our experiments: the quality of the source code and bug reports that are input to IR techniques.

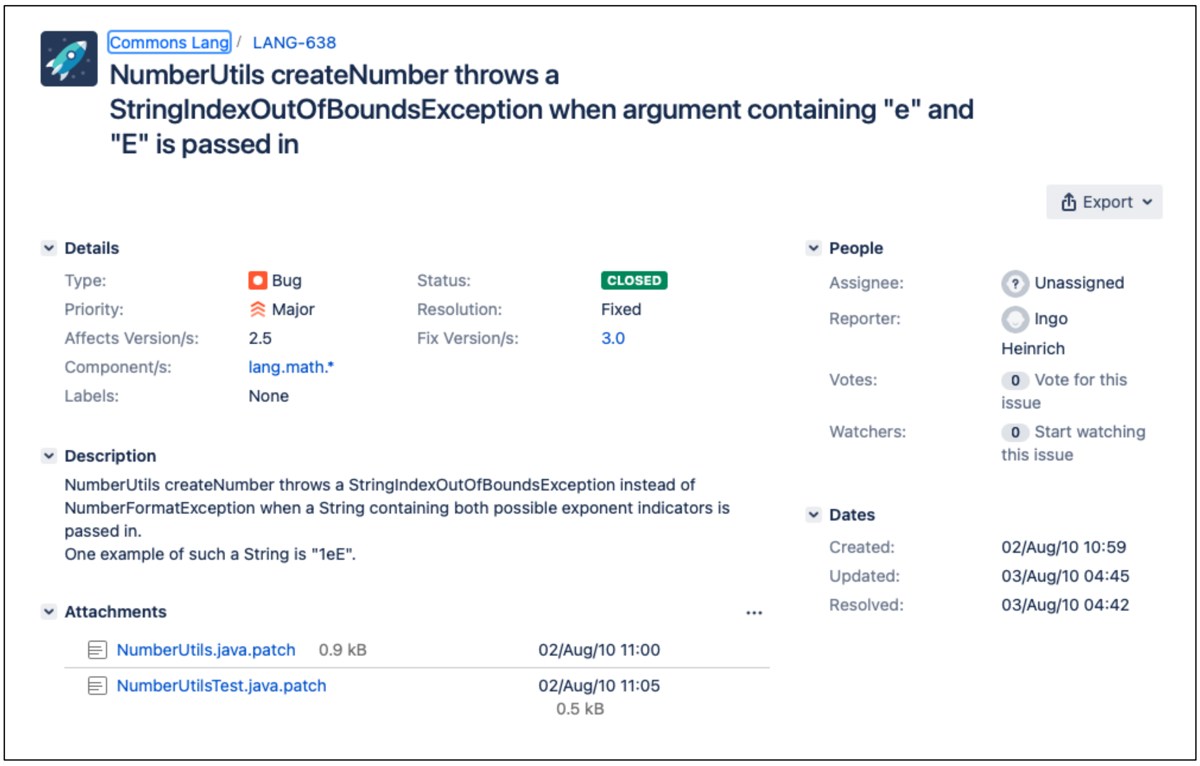
- **Identifiers** are defined to be a class name, attribute name, method name, parameter name, local variable name, enumeration constant name, label name, or a generic/template parameter name [155] [53].
- **Comments** generally are used either to map requirements to code or to describe the code [156].
- **String Literals** generally are used either to convey information to the end user such as an error message, which usually contains domain information, or to the

developer such as a debugging message which usually contains implementation information. Copyright information is also included into string literals [53].

All three types of information could potentially be input to an IR technique for the purposes of fault localization. However, in this thesis, I only use Identifier and Comments sources, since string literal information is not generally and consistently available across different programs, except potentially for the copyright information. Copyright information is typically not to be indexed by IR techniques, as it adds no information about program purpose or behavior. Besides source code, the key source of knowledge for developers to understand a fault is a bug report. In a report, the summary gives a concise overview of the issue [157], potentially including the steps of execution that led to the failing behavior.

Therefore, a bug report is important, so that programmers can use them to attempt to map the issues raised in the report to the fault location in the source code. Effectively, this is what IR techniques attempt to automate in fault localization: the mapping of issues to source code locations. Consequently, insufficient or inadequate information in a bug report may prevent accurate results being generated in an IR based fault localization process. As such, it is preferable to apply IR based techniques to high quality (complete and precise) bug reports for high success rate in fault localization which is not always the case.

A complete and precise bug report is specified using a combination of bug report's title and its description [42]. However, [48] argues that a high quality bug report is one that also includes a code attachment or code snippets. To investigate this further, [37] evaluated the importance of specific program constructs (such as class names and method names) in bug reports, and argued that the greater use of program constructs in bug reports may lead to improved fault localization. In a different experiment, [52] found that the best results from their IR-based fault localization depends on there being similar textual characteristics between bug reports and source code. Taking from these guidelines, Figure 3.4 and Figure 3.5 below show an example of a good quality bug report.


Commons Lang / LANG-638

NumberUtils createNumber throws a StringIndexOutOfBoundsException when argument containing "e" and "E" is passed in

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	2.5	Fix Version/s:	3.0
Component/s:	lang.math.*		
Labels:	None		

Description

NumberUtils createNumber throws a StringIndexOutOfBoundsException instead of NumberFormatException when a String containing both possible exponent indicators is passed in.
One example of such a String is "1eE".

Attachments

NumberUtils.java.patch	0.9 kB	02/Aug/10 11:00
NumberUtilsTest.java.patch	0.5 kB	02/Aug/10 11:05

People

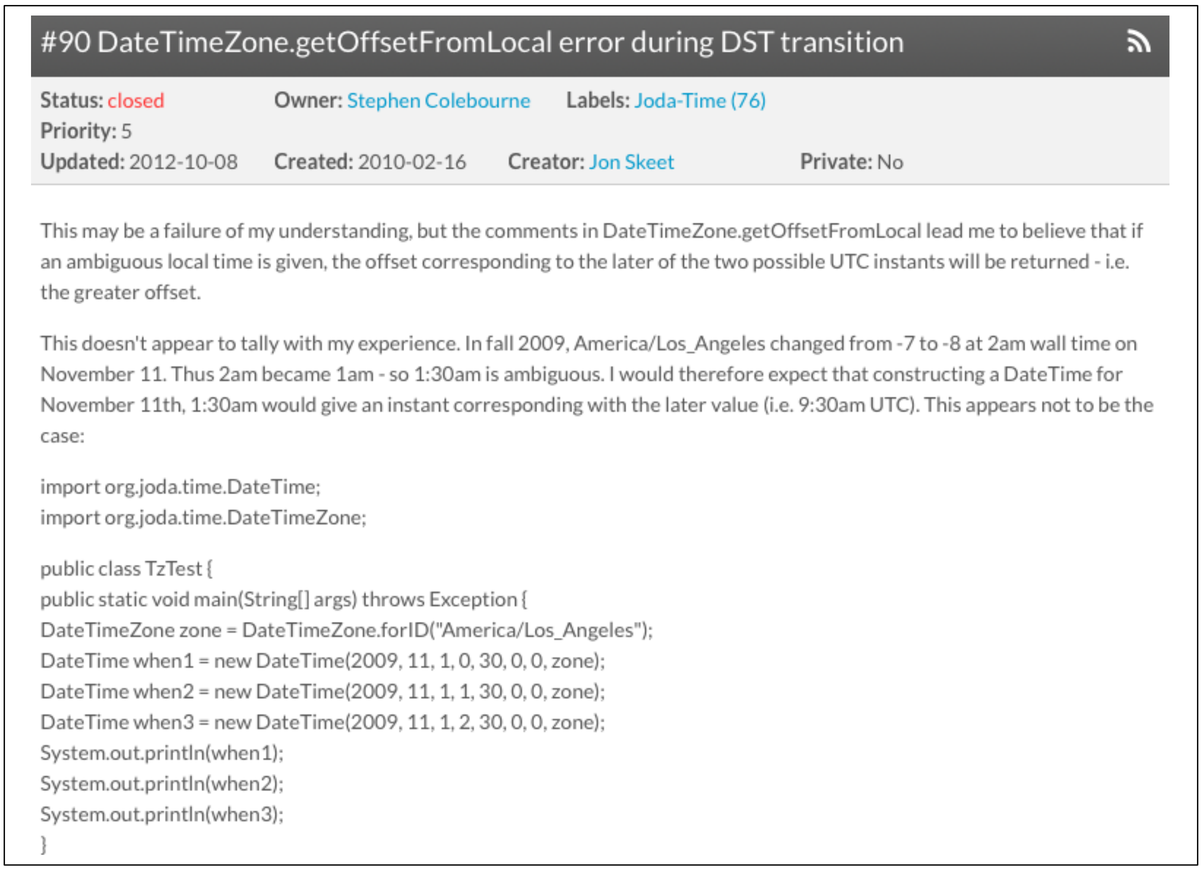
Assignee:	Unassigned
Reporter:	Ingo Heinrich
Votes:	0 Vote for this issue
Watchers:	0 Start watching this issue

Dates

Created:	02/Aug/10 10:59
Updated:	03/Aug/10 04:45
Resolved:	03/Aug/10 04:42

Figure 3.4: Example of a good quality bug report L27, that contains bug report title, description and code attachment that related to the bug.

Both figure 3.4 and figure 3.5 shows an example of bug 27 from Lang[9] program (L27) and bug 25 from Time[10] program (T25) as a good quality bug report as highlighted in previous literature which it should contains a title of the bug report, descriptions of the bug and a code attachment or code snippets that related to the bug.



#90 DateTimeZone.getOffsetFromLocal error during DST transition

Status: closed **Owner:** Stephen Colebourne **Labels:** Joda-Time (76)
Priority: 5
Updated: 2012-10-08 **Created:** 2010-02-16 **Creator:** Jon Skeet **Private:** No

This may be a failure of my understanding, but the comments in DateTimeZone.getOffsetFromLocal lead me to believe that if an ambiguous local time is given, the offset corresponding to the later of the two possible UTC instants will be returned - i.e. the greater offset.

This doesn't appear to tally with my experience. In fall 2009, America/Los_Angeles changed from -7 to -8 at 2am wall time on November 11. Thus 2am became 1am - so 1:30am is ambiguous. I would therefore expect that constructing a DateTime for November 11th, 1:30am would give an instant corresponding with the later value (i.e. 9:30am UTC). This appears not to be the case:

```
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;

public class TzTest {
public static void main(String[] args) throws Exception {
DateTimeZone zone = DateTimeZone.forID("America/Los_Angeles");
DateTime when1 = new DateTime(2009, 11, 1, 0, 30, 0, 0, zone);
DateTime when2 = new DateTime(2009, 11, 1, 1, 30, 0, 0, zone);
DateTime when3 = new DateTime(2009, 11, 1, 2, 30, 0, 0, zone);
System.out.println(when1);
System.out.println(when2);
System.out.println(when3);
}
```

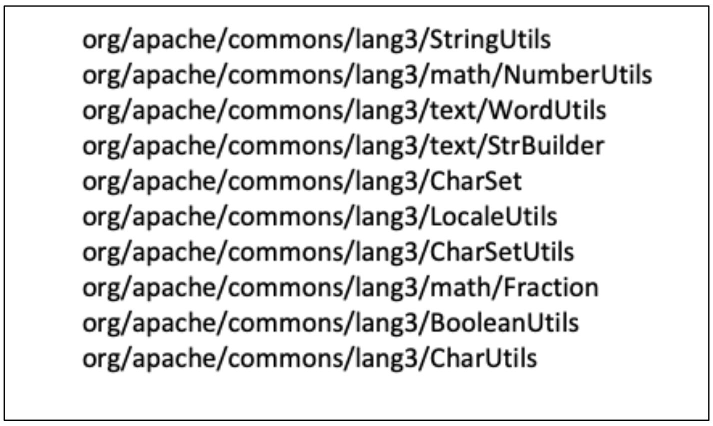
Figure 3.5: Example of a good quality bug report T25, that contains bug report title, description and code snippet that related to the bug.

Though L27 and T25 are considered to be a good quality of bug report, however, in Chapter 4 we will explain further in details the outcome of the experiment either the result are affected with this quality of bug factor or not. All bugs that do not have any

[9] https://issues.apache.org/jira/browse/LANG-638
[10] https://sourceforge.net/p/joda-time/bugs/90/

bug report will be excluded, as IR solely relies on the bug report information to allocate the fault. However, the bug reports that do not have complete information, will still be executed using the IR technique to fully assess and demonstrate how the incomplete or insufficient information in the bug report affects the results. In this experiment, we do find several bugs that do not have any bug report; this will be explained further in Chapter 4.

As shown in Figure 3.6 below, the output results from IR execution are generated in a file name or method name of the fault which is the same level as statement output in SBFL technique. The output are sorted based on the cosine similarity in the VSM method to identify the similarity between documents. Since we are using L5 for output example in SBFL technique, we are also using the same bug which is L5 for IR technique to enable readers to comprehend and catch up. As we know in the previous subsection, the real location of fault for bug L5 is at "LocaleUtils" file.



```
org/apache/commons/lang3/StringUtils
org/apache/commons/lang3/math/NumberUtils
org/apache/commons/lang3/text/WordUtils
org/apache/commons/lang3/text/StrBuilder
org/apache/commons/lang3/CharSet
org/apache/commons/lang3/LocaleUtils
org/apache/commons/lang3/CharSetUtils
org/apache/commons/lang3/math/Fraction
org/apache/commons/lang3/BooleanUtils
org/apache/commons/lang3/CharUtils
```

Figure 3.6: The output result for L5 from IR execution

The location of the Top 10 suspected or potentially contains fault are as in Figure 3.6, and as we can see in the Figure 3.6, the "LocaleUtils" file is included in the Top 10 list. However, unlike SBFL (which provides fine grained results), the IR technique can only identify a file that contains a similarity to the bug report. As a result, the process of making the list of file locations with highlighting the source code to identify a fault location is quite challenging. That is why we mentioned earlier that the importance of a quality bug report in assisting programmer to identify fault in IR technique though it is understandable why it is not always available.

3.1.5.2.1 Preprocessing of data

Before execution of the IR process, each source code and bug report information will be extracted including information such as comments and identifiers. The extraction is called preprocessing which includes *stemming*, *normalizing*, *removing stop words* and *splitting*.

Stemming is a process is where we strip suffixes to reduce words to their stems; for example “changing” becomes “chang”. This typically uses the *Porter stemmer algorithm* [158]. *Normalizing* is the process of replacing each upper case letter with the corresponding lower case letter, while *filtering* is removing common English language stop words such as “the”, “it”, “on” “an”. Filtering also applies to the programming language keywords such as “if”, “while”, which are also removed. *Splitting* is done by removing all punctuation, numbers including characters related to the syntax of the programming language such as “&&”, “->”.

The main idea behind applying these steps is to capture the semantics of the developer’s intentions, which are thought to be encoded within the identifier names and comments in the source code [129]. These preprocessing steps should be applied to both source code and the bug reports, where all information of the bug report such as title, descriptions, codes attachment or code snippets, are preprocessed. In this thesis, I apply stemming, normalizing and filtering as described above, but implement splitting with a slight variation: we do not split words and retain the original (unsplit) tokens in compound identifiers, such as “AgeCalculator”, “PeriodType”, “LocalDate” that represent class name or method name to avoid or reduce this vocabulary mismatch issue. This is because, [34] found number of obstacles to overcome when attempting to adapt IR for fault localization retrieval, one of which is vocabulary mismatch. The usage of abbreviations and concatenations of variable names and identifiers by programmers during code development causes a vocabulary mismatch problem. These words are known as “hard words”.

During a small test, I found that this course of action tends to preserve a token’s original name until at the end of the IR process, and this can enable the developer to understand the context of the result of fault localization better if the uniformity is applied (e.g., if unsplit, apply it to all documents such as source code and bug report, and vice versa). At the same time, it also helps the accuracy of fault localization. This action is also aligned with research made by [159], indicating that any splitting in preprocessing will suffice and does not make a significant difference as long as uniformity across the whole experiment is maintained.

After the preprocessing steps, the source code is saved as a text file, where each document represents one class that contains one or more methods of the source code; this is because the techniques we are comparing and integrating operate at the granularity level of methods and classes. Unlike source code that contains hundreds of classes and methods for each program, each bug report only represents one specific bug so only one document will be saved as a text file for each bug. Both text file documents from source code and bug reports now are in the same format, which will simplify the next process of IR, which is to find the similarity of the documents by using Vector Space Model (VSM) .

3.1.5.3 Text Search Individual Experiment

This subsection is an explanation of the Text Search experiment that has been carried out to evaluate the technique's accuracy and execution time. Text Search uses search tools to finds documents in a collection that are good matches to specified queries [122]. For the experiments in this thesis, source code and bug reports are undergoing the same kind of data preprocessing as in the IR technique described in Preprocessing of data subsection (Section 3.1.5.2.1) above.

Figure 3.7 describes how text search is being used in this thesis, and how I am intending to use it in my experiments. As mentioned earlier in Chapter 2 (subsection 2.6), I am making use of *keyword-based text search*, which is one of the main use cases for finding and accessing information [124]. Text search makes use of string matching algorithms, which have been demonstrated to provide significant accuracy and efficiency when applied in various domain [123]. In this experiment, the preprocessed bug report title and description will be used to search and detect matching words in the source code.

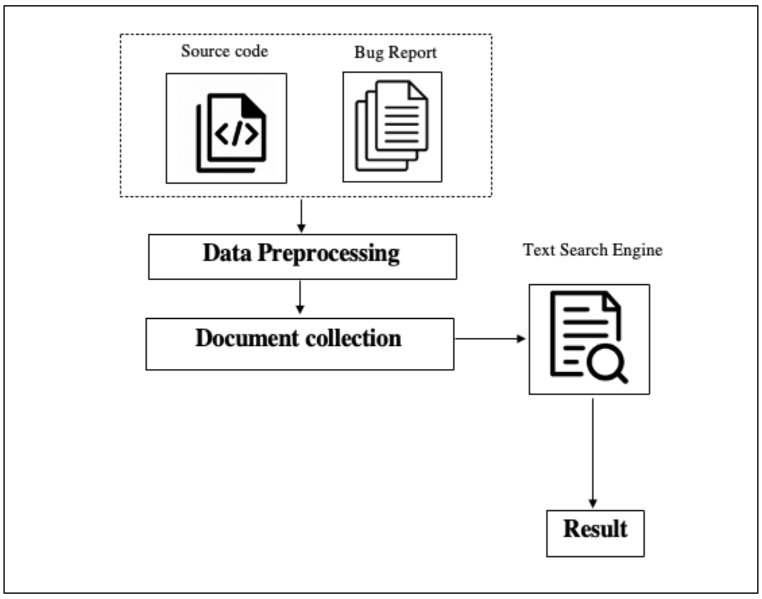
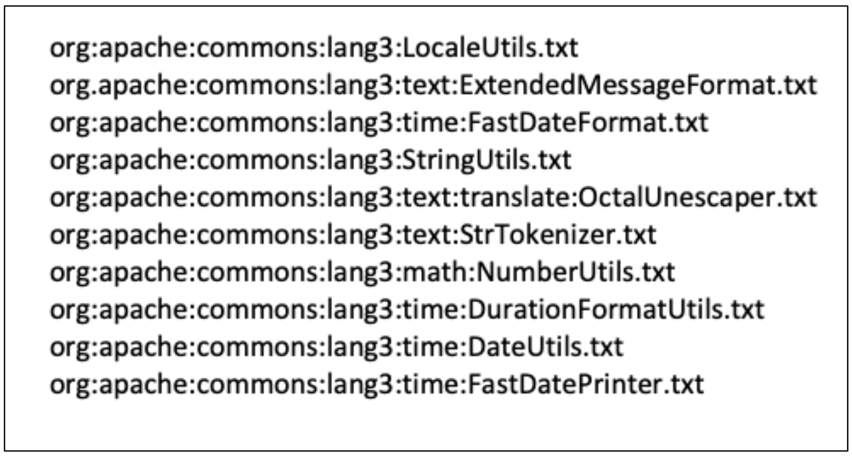


Figure 3.7: Text Search framework

As with the SBFL and IR technique mentioned earlier, faults are considered localized when the file name (class name or method name) of the fault appears in Top 10 search result. Figure 3.8 below is an output of the same bug as previous example before which is L5, that are generated and ranked after Text Search execution. “LocaleUtils” the exact location for L5 bug and it does included in the Top 10 list of output below.



```
org:apache:commons:lang3:LocaleUtils.txt
org.apache:commons:lang3:text:ExtendedMessageFormat.txt
org:apache:commons:lang3:time:FastDateFormat.txt
org:apache:commons:lang3:StringUtils.txt
org:apache:commons:lang3:text:translate:OctalUnescaper.txt
org:apache:commons:lang3:text:StrTokenizer.txt
org:apache:commons:lang3:math:NumberUtils.txt
org:apache:commons:lang3:time:DurationFormatUtils.txt
org:apache:commons:lang3:time:DateUtils.txt
org:apache:commons:lang3:time:FastDatePrinter.txt
```

Figure 3.8: The output result for L5 from Text Search execution

3.1.5.4 Experiment on combinations of techniques

The previous subsections in this chapter are an explanation on how I will carry out individual experiments on the SBFL technique (subsection 3.1.5.1), IR technique (subsection 3.1.5.2), and Text Search technique (subsection 3.1.5.3). This subsection is an explanation on the combination of different techniques: the combination of SBFL and IR (SBFL + IR), the combination of SBFL and Text Search (SBFL + Text Search), the combination of Text Search and IR (Text Search + IR), and the combination of SBFL, IR and Text Search (SBFL + IR + Text Search).

As mentioned previously, Rao et al argued that combining IR fault localization tools with dynamic fault localization could significantly improve the state-of-the-art in fault localization [34]. In fault localization research, VSM has been shown to outperform many other IR approaches [160] [34] while Ochiai is the best in performance compared to other technique [21] in SBFL. These are the reasons why I chose the Ochiai technique for SBFL, and the VSM technique for IR, because of their outstanding performance. I will now explain how the experiments for the different combinations of techniques are to be carried out.

3.1.5.4.1 The combination of SBFL and IR technique

The first experiment of the combination is between the state-of-the-art technique in fault localization, i.e., SBFL and IR. Figure 3.9 below show the architecture framework of the SBFL and IR technique combination. The procedure for SBFL and IR technique in this combination are the same as the individual technique for SBFL and IR mentioned in the previous subsections. The results of fault localization for each individual technique can only be combined after the individual techniques have been executed; as such, we argue that the individual techniques can be executed in any order. However, in the Result and Analysis section (chapter 4), we will explain further what we believe to be a recommended order of the combination of techniques, based on the experimental results.

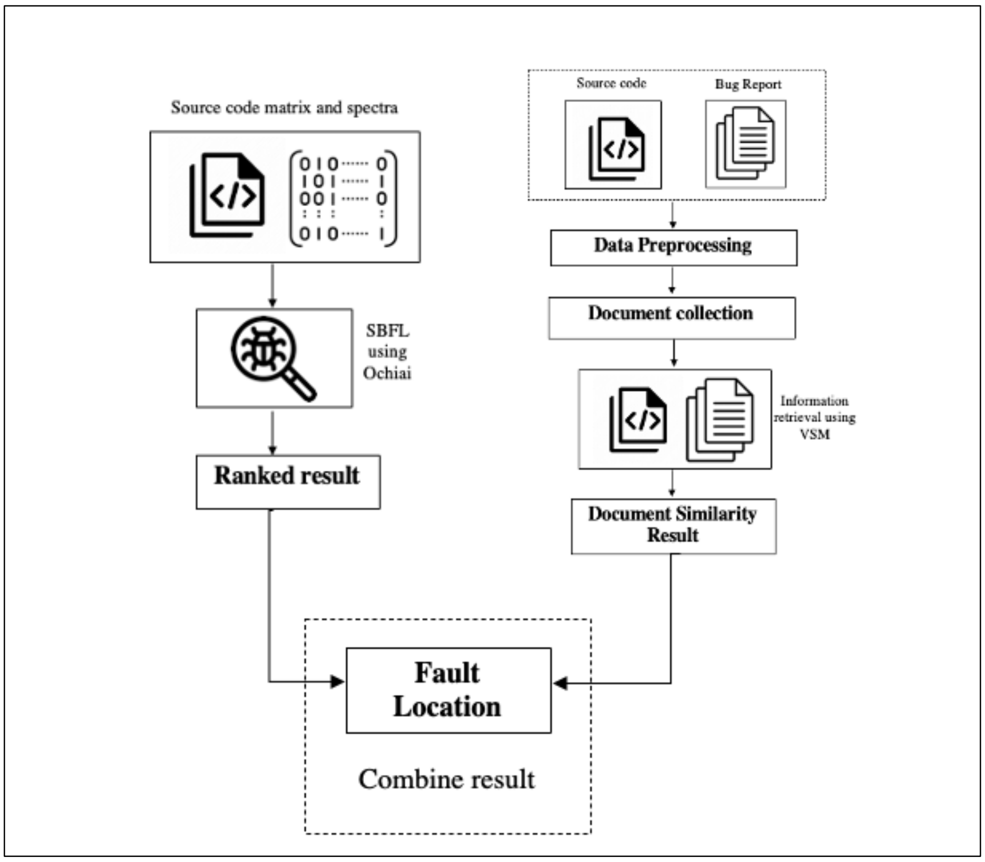


Figure 3.9: The combination of SBFL and IR technique architecture framework

As mentioned before this in individual experiment, the vital result that is generated for both techniques is the file location of the fault, since that is the only way we could identify where is the fault located. For SBFL, the result is the location of the fault, are *statements,* which is the file name with line number and suspiciousness value;

however, the suspiciousness values calculation is intended to rank the location based on the probability calculation only.

While the line number generated from SBFL result might assist the programmer to find the fault, in a way it does give advantage to combine the SBFL technique with IR since the IR technique output's does not have the line number though the line number are not always the exact location of the fault. Figure 3.10 below shows the Top 10 output combination of SBFL and IR technique and the exact location for L5 bug which is "LocaleUtils" does included in the Top 10 list of output below.

Output SBFL technique	Output IR technique
org.apache.commons.lang3.LocaleUtils#99	org/apache/commons/lang3/StringUtils
org.apache.commons.lang3.LocaleUtils#89	org/apache/commons/lang3/math/NumberUtils
org.apache.commons.lang3.LocaleUtils#92	org/apache/commons/lang3/text/WordUtils
org.apache.commons.lang3.LocaleUtils#93	org/apache/commons/lang3/text/StrBuilder
org.apache.commons.lang3.LocaleUtils#96	org/apache/commons/lang3/CharSet
org.apache.commons.lang3.LocaleUtils#97	**org/apache/commons/lang3/LocaleUtils**
org.apache.commons.lang3.LocaleUtils#98	org/apache/commons/lang3/CharSetUtils
org.apache.commons.lang3.LocaleUtils$SyncAvoid#288	org/apache/commons/lang3/math/Fraction
org.apache.commons.lang3.LocaleUtils$SyncAvoid#295	org/apache/commons/lang3/BooleanUtils
org.apache.commons.lang3.LocaleUtils$SyncAvoid#296	org/apache/commons/lang3/CharUtils

Figure 3.10: The combination of output result for L5 bug from SBFL and IR execution

3.1.5.4.2. The combination of SBFL and Text Search technique

The next experiment evaluates a combination of SBFL and Text Search. Figure 3.11 below show an overview of how we combine these techniques. The results of fault localization for each technique are only to be combined after the individual techniques have been executed to completion; nevertheless, this means that the SBFL and Text

Search can be run in parallel or executed sequentially in any order. After execution terminates, the results are combined.

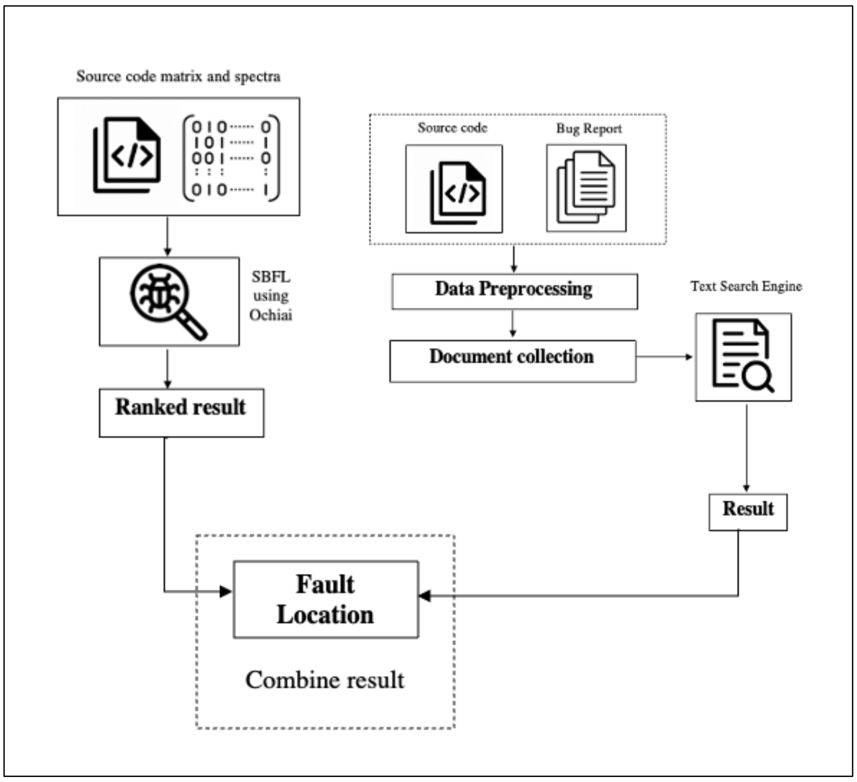


Figure 3.11: The combination of SBFL and Text Search technique architecture framework

While Figure 3.12 shows the Top 10 output combination for L5 bug from SBFL and Text Search execution. Again, the location for L5 bug which is "LocaleUtils" does included in the Top 10 list of output below showing that the location of the fault.

Output SBFL technique	Output Text Search technique
org.apache.commons.lang3.LocaleUtils#99	**org:apache:commons:lang3:LocaleUtils.txt**
org.apache.commons.lang3.LocaleUtils#89	org.apache:commons:lang3:text:ExtendedMessageFormat.txt
org.apache.commons.lang3.LocaleUtils#92	org:apache:commons:lang3:time:FastDateFormat.txt
org.apache.commons.lang3.LocaleUtils#93	org:apache:commons:lang3:StringUtils.txt
org.apache.commons.lang3.LocaleUtils#96	org:apache:commons:lang3:text:translate:OctalUnescaper.txt
org.apache.commons.lang3.LocaleUtils#97	org:apache:commons:lang3:text:StrTokenizer.txt
org.apache.commons.lang3.LocaleUtils#98	org:apache:commons:lang3:math:NumberUtils.txt

org.apache.commons.lang3.LocaleUtils$SyncAvoid#288	org:apache:commons:lang3:time:DurationFormatUtils.txt
org.apache.commons.lang3.LocaleUtils$SyncAvoid#295	org:apache:commons:lang3:time:DateUtils.txt
org.apache.commons.lang3.LocaleUtils$SyncAvoid#296	org:apache:commons:lang3:time:FastDatePrinter.txt

Figure 3.12: The combination of output result for L5 bug from SBFL and Text Search execution

3.3.4.3. The combination of IR and Text Search technique

The next experiment evaluates the combination of IR and Text Search. Figure 3.13 below shows how we combine IR and Text Search.

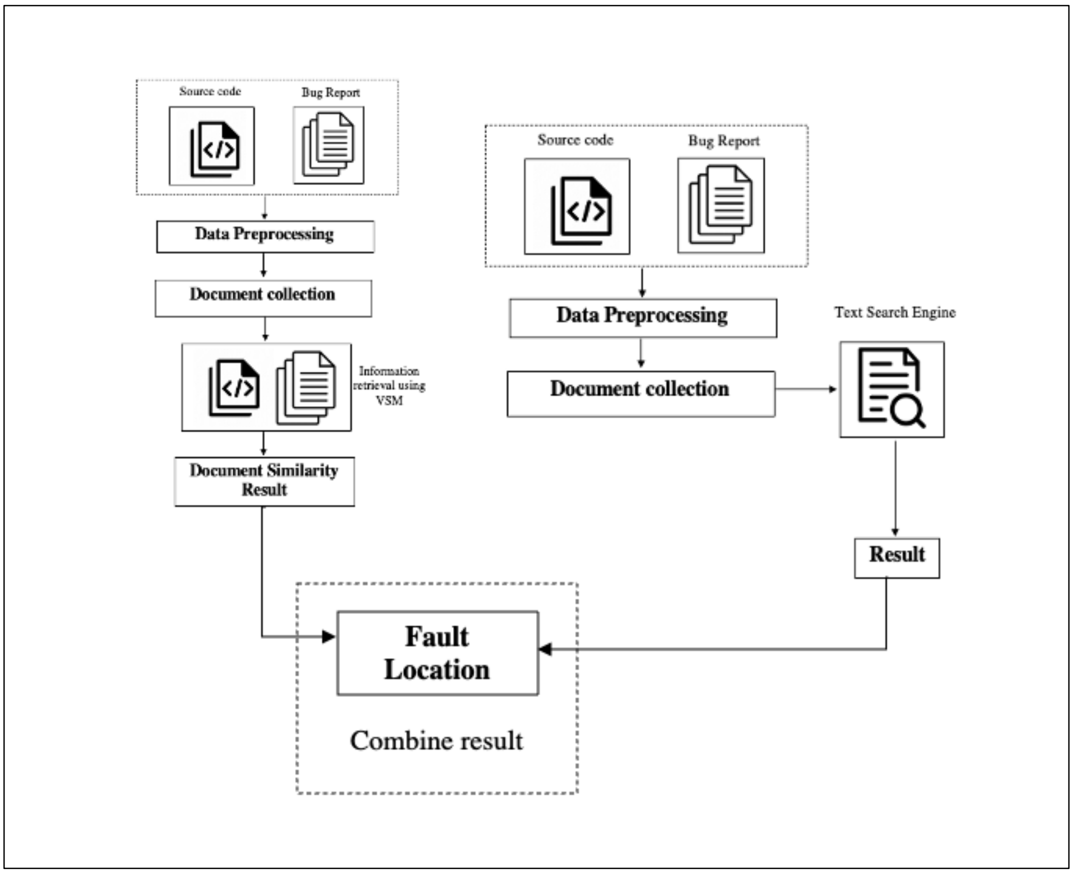


Figure 3.13: The combination of IR and Text Search technique architecture framework

Similarly, to the other experiments, IR and Text Search results are combined sequentially. In practice, IR and Text Search can be executed in any order (or in parallel), and the results of the individual techniques are thereafter combined. Figure 3.14 shows the

Top 10 output combinations for L5 bug from IR and Text Search execution. Again, the location for L5 bug which is “LocaleUtils” does included in the Top 10 list of output below showing that the location of the fault.

Output IR technique	Output Text Search technique
org/apache/commons/lang3/StringUtils	**org:apache:commons:lang3:LocaleUtils.txt**
org/apache/commons/lang3/math/NumberUtils	org.apache:commons:lang3:text:ExtendedMessageFormat.txt
org/apache/commons/lang3/text/WordUtils	org:apache:commons:lang3:time:FastDateFormat.txt
org/apache/commons/lang3/text/StrBuilder	org:apache:commons:lang3:StringUtils.txt
org/apache/commons/lang3/CharSet	org:apache:commons:lang3:text:translate:OctalUnescaper.txt
org/apache/commons/lang3/LocaleUtils	org:apache:commons:lang3:text:StrTokenizer.txt
org/apache/commons/lang3/CharSetUtils	org:apache:commons:lang3:math:NumberUtils.txt
org/apache/commons/lang3/math/Fraction	org:apache:commons:lang3:time:DurationFormatUtils.txt
org/apache/commons/lang3/BooleanUtils	org:apache:commons:lang3:time:DateUtils.txt
org/apache/commons/lang3/CharUtils	org:apache:commons:lang3:time:FastDatePrinter.txt

Figure 3.14: The combination of output result for L5 bug from IR and Text Search execution

3.3.4.4. The combination of SBFL, IR and Text Search technique

The final experiment evaluates the combination of all three of SBFL, IR and Text Search. Figure 3.15 below depicts how we combine these three techniques. As with the previous combinations mentioned, the result of fault localization for each technique are combined after the individual techniques have been executed. The individual techniques can be executed in any order, as it is the results that are combined. However, I will discuss further in Chapter 4, Results and Analysis, regarding optimal sequences of analysis that should be done.

Figure 3.16 below shows the Top 10 output combination for L5 bug from SBFL, IR and Text Search execution. Again, the location for L5 bug which is “LocaleUtils” does included in all Top 10 list result of output from all technique showing that the location of the fault. All technique are run separately, only the result are combined and even though a new architecture or tools are involved, each technique still need to be run separately on their own and once the results are generated, then from the result probably a mechanism can be added in choosing the technique. I will discuss further regarding this in Chapter 5.

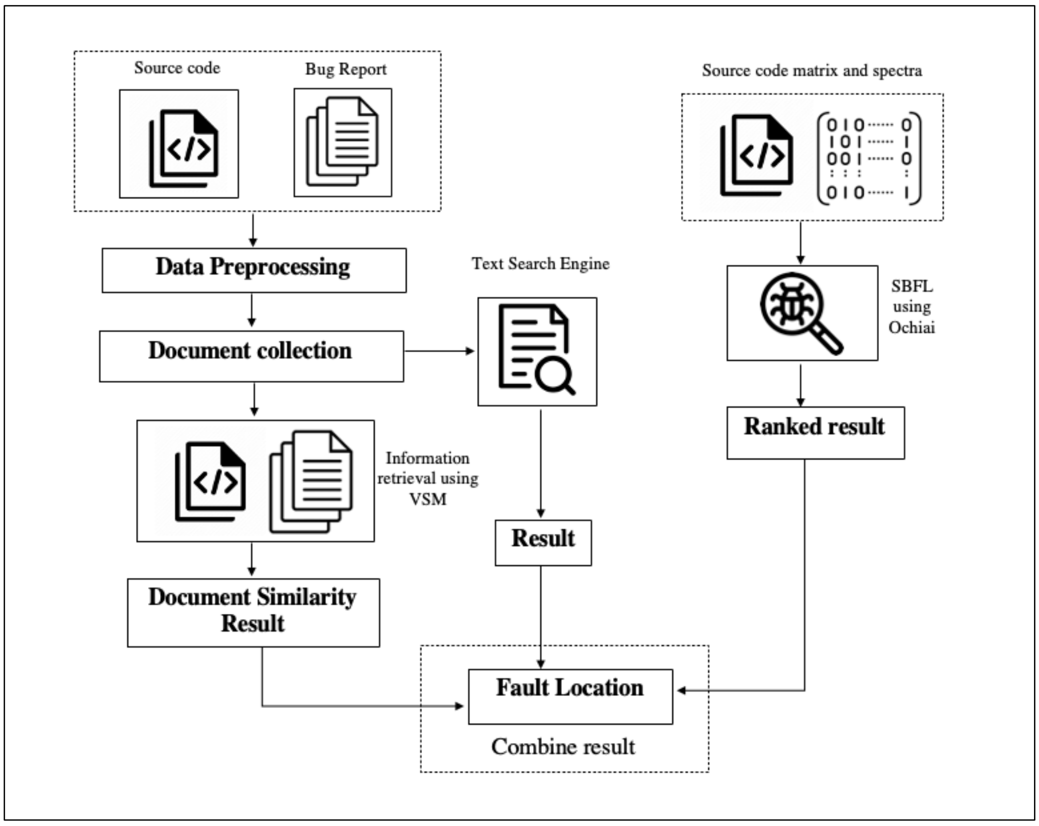


Figure 3.15: The combination of SBFL , Text Search and IR technique architecture framework

Output SBFL technique	Output IR technique	Output Text Search technique
org.apache.commons.lang3.LocaleUtils#99	org/apache/commons/lang3/StringUtils	**org:apache:commons:lang3:LocaleUtils.txt**
org.apache.commons.lang3.LocaleUtils#89	org/apache/commons/lang3/math/NumberUtils	org.apache:commons:lang3:text:ExtendedMessageFormat.txt
org.apache.commons.lang3.LocaleUtils#92	org/apache/commons/lang3/text/WordUtils	org:apache:commons:lang3:time:FastDateFormat.txt
org.apache.commons.lang3.LocaleUtils#93	org/apache/commons/lang3/text/StrBuilder	org:apache:commons:lang3:StringUtils.txt
org.apache.commons.lang3.LocaleUtils#96	org/apache/commons/lang3/CharSet	org:apache:commons:lang3:text:translate:OctalUnescaper.txt
org.apache.commons.lang3.LocaleUtils#97	**org/apache/commons/lang3/LocaleUtils**	org:apache:commons:lang3:text:StrTokenizer.txt
org.apache.commons.lang3.LocaleUtils#98	org/apache/commons/lang3/CharSetUtils	org:apache:commons:lang3:math:NumberUtils.txt
org.apache.commons.lang3.LocaleUtils$SyncAvoid#288	org/apache/commons/lang3/math/Fraction	org:apache:commons:lang3:time:DurationFormatUtils.txt
org.apache.commons.lang3.LocaleUtils$SyncAvoid#295	org/apache/commons/lang3/BooleanUtils	org:apache:commons:lang3:time:DateUtils.txt
org.apache.commons.lang3.LocaleUtils$SyncAvoid#296	org/apache/commons/lang3/CharUtils	org:apache:commons:lang3:time:FastDatePrinter.txt

Figure 3.16: The combination of output result for L5 bug from SBFL, IR and Text Search execution.

3.2 Experimental Operation

This subsection presents the experimental operation. It highlights the preparation of the experiment before executing, and the execution of the experiment as well.

3.2.1 Preparation of experiment

The experiments are running on the macOS 11 operating system (Big Sur version) with Apple M1 chip processor. Apple M1 chip are using 8-core CPU with 4 performance cores and 4 efficiency cores, 8GB unified memory with 256GB storage (SSD). Substantial main memory is needed to execute the experiment since this involves substantial data, and these specifications are sufficient to run the experiment. The experiment took around 8 months to complete, including the analysis phase. I'm running my experiment using Python language and this adds value to my research since many research before this did their research in other language such as C++ and Java.

3.2.2 Execution of experiment

As mentioned in the research context subsection, there will be only one experiment that is being executed, but it will be running in three sessions. The experiment will be using a real-world dataset where six program from Defects4J are used. All six program with the capacity of size with 332,000 lines of codes that contains 395 bugs will be running for three session of fault localization, which is Spectrum Based Fault Localization (SBFL), Information Retrieval Fault Localization (IR) and Text Search where it will be analysed for seven times in total including when the method being combined. The experiment contains the total of approximately 1600 text file of source code, 373 document of bug report and 395 bug with a set of test cases for each bug from Defects4j programs.

Before running the experiment, three testing session have been done to make sure that the results obtained are consistent. Guidelines for experiment execution are included to allow replication. This experiment is using a qualitative approach where no statistical calculations are involved. This is largely since the experiment itself is a lot of work, involving inventing new methods and tools and the evaluation on accuracy and performance (time taken to execute) of the tools.

The results of the new methods then are compared with the existing tools where this means that, those existing tools are also being executed and evaluated. It is this largely qualitative comparison with existing methods and tools that is the basis of our empirical analysis; adding statistical evaluation will provide little further benefit given that, as we will see, the comparison of performance results are significant and compelling.

In a study made by [33], while running their experiments, they consider an output of a fault localization tool to be *effective* if the root cause appears in the Top 10 most suspicious program elements. I have applied this principle in my experiments as an indicator for assessing the quality of the output generated from different tools (i.e., the root cause of a fault appears in the Top 10 list of generated results). This Top 10 decision analysis will also be explained further in Chapter 4 Results and Analysis.

Initially SBFL, IR and Text Search techniques will be executed individually using the same datasets to assess their respective accuracy and execution time. Then all the individual results of the technique will be combined to evaluate how different the combined results are compared to individually executed techniques. I examine three individual technique and four combinations of techniques: the first combination is the combination of SBFL and IR (SBFL + IR) techniques, the second one is the combination of SBFL and Text Search (SBFL + Text Search) techniques, the third combination is the combination of Text Search and IR (Text Search + IR) techniques; while the final one is the combination of SBFL, IR and Text Search (SBFL + IR + Text Search) techniques.

4 Results and Analysis

This chapter will summarize and explain the results and analysis from the experiments, which have been designed to answer the research questions (as presented in subsection 3.1.2, Research Hypothesis).

I have embarked on a set of experiments evaluating SBFL, IR and text search, where initially each technique is executed independently. The accuracy of each technique is measured, and their execution time is recorded. After these individual experiments, all distinct pairwise combinations of these three techniques are evaluated in their own experiments, where the accuracy and the execution time are determined. At the end of the combination of pairwise technique experiments, an experiment that combined all the three techniques is carried out, once again calculating accuracy and execution time. This last collection of results allows us to compare individual techniques with all three techniques, to help understand where one technique might be preferable to a combination of pairs or all three techniques.

While the results of the experiments are quantitative, and this permits quantitative comparison of the results, I do not include further statistical tests, as the direct comparison of the experiment results is sufficient for identifying the strengths and weaknesses of the individual and combined techniques. Adding statistical tests does not contribute sufficient further understanding to justify the extra effort and expense involved in setting up and carrying out those tests.

I now summarize the research questions and discuss how the experimental results address them.

4.1 *RQ1: How accurate is the process of localizing faults using SBFL, IR and Text Search technique individually?*

The first research question aims at understanding the accuracy of widely used techniques individually in fault localization. This defines a baseline for the remaining experiments on combinations of techniques. The first set of results and analysis can be found in Table 4.1 below; it contains the accuracy result for individual techniques in the Top 1, Top 5 and Top 10 for six programs in the Defects4j dataset. As explained earlier in Chapter 1 regarding Top N concept, Top 1 represent the suspicious location of the fault that appears in Top 1 in the list result after experiment execution. Meaning that, if the location of the fault that appear as number 1 in the result ranking list is true, then the result will be count as accurate. Same concept also applies to Top 5 and Top 10. The boldface indicates the technique demonstrating the greatest accuracy for each program. As mentioned in Subsection 1.2, most researchers argues that the accuracy of SBFL is high when the location of fault is in between Top 1, Top 5 and 10 result generated. As for my experiment and taking this as a guideline, the accuracy result of Top 1, Top 5 and Top 10 of each technique either individual or combined technique are measured and analyzed.

The results for SBFL, shown in Table 4.1, indicates that the accuracy for Top 1 is 47.6%, Top 5 is 59.56% while for Top 10 is 64.56%. For IR technique, Table 4.1

shows that for Top 1, the accuracy is 18.87%, Top 5 is 39.09%, and Top 10 with 48.5% accuracy. Finally, the Text search technique where the accuracy for Top 1 is 23.59%, Top 5 is 51.2%, and Top 10 is 67.29%.

Program	*SBFL*			*IR*			*Text Search*		
	Top 1	Top 5	Top 10	Top 1	Top 5	Top 10	Top 1	Top 5	Top 10
Time	44.67%	59.56%	67%	27%	46.28%	54%	15.37%	57.63%	**73%**
Lang	85.87%	90.47%	92%	42.84%	74.58%	84.1%	52.39%	87.82%	**94%**
Mockito	39.6%	55.44%	**66%**	10.81%	13.51%	16.22%	5.27%	15.81%	29%
Chart	69.54%	81.13%	85%	37.5%	62.5%	87.5%	62.5%	87.5%	**100%**
Math	44.34%	54.72%	56.60%	24.5%	47.12%	55.66%	26.41%	59.43%	**75.47%**
Closure	30.29%	46.19%	53%	2.29%	19.84%	31.3%	11.46%	32.85%	**55%**
Accuracy	47.6%	59.75%	64.56%	18.87%	39.09%	48.5%	23.59%	51.2%	**67.29%**

Table 4.1: Result for individual technique accuracy in Top 1, Top 5 and Top 10 for 6 programs with 395 faults in Defects4j dataset.

As we can see, the accuracy result for Top 10 is the highest and outperform Top 1 and Top 5 results in all individual techniques. The Top 10 result usually defined as a "sufficient accuracy" [94], as this are seen as a feasible number for programmer to find the faults location and to investigate it.

We carried out an additional experiment calculating results if we take the Top 30 and Top 100, instead of Top 10. This further experiment demonstrates that the accuracy result will increase, since this also means an increase in the results range value, however, we argue that it will be useless as the higher *N* value also means that more noise and potentially an increased number of false positives will be involved. In this context, therefore, a larger number does not mean a better (i.e., more accurate) result. That is why by taking all this information into consideration, we have decided that the best value to measure accuracy in this experiment is the Top 10 returned results.

The Top 10 results in Table 4.1 also show that with respect to the accuracy and for individual techniques, Text Search is the highest, with 67.29% of accuracy in fault localization for all six Defects4j program. The SBFL technique had a result of 64.56% in accuracy, where both Text Search and SBFL techniques accuracy is greater than the IR technique, which only scored 48.5%. This result is unexpected, as we predicted that both state-of-the-art fault localization tools, which is SBFL and IR or either one of it to be the one that high in accuracy. In other word, this also mean that Text Search technique, a simple tool used for searching purpose task have outperform both state of the art technique in fault localization which are SBFL and IR technique.

If we look closer at Table 4.1, the Text Search technique applied to the Time program led to 73% accuracy compared to 67% for SBFL. The Lang program scores 94% compared to 92% for SBFL; the Math program scores 75.47% compared to 56.60% only for SBFL. The Closure program scores 55% compared to 53% accuracy for SBFL. From these results, we determine that Text Search technique tends to provide more accurate results than SBFL or IR in general largely because of its simplicity, and because of the nature of the text search tool itself, where it manages to find the related files in the source codes using keywords from the bug report. Nevertheless, the success of the text search method hinges on having high quality data (especially appropriate keywords) as input.

For example, consider bug 25 in Time program (T25). Even though all bug reports are considered to be of good quality – the reports contain title, description and details such as code snippets [11] - the text search technique managed to find the location of the fault even when SBFL and IR technique were unable to do so, by only using the title of the T25 bug (“#90 DateTimeZone.getOffsetFromLocal error during DST transition”). This is one of the reasons why human (developer) involvement is important as only they understand the semantic issues they face in debugging the code in context. Moreover, this is evidence that details in bug reports are indeed important (i.e., a developer should describe the fault as accurately as possible) and by doing so, this can help in fault localization. Relying solely on the SBFL technique might result in difficulties for developers to localize faults accurately.

In the T25 example, the developer includes all important keywords that in their view precisely represent the error, but using only titles; this means that carrying out fault localization by Text Search tools using only bug report title information is possible. This finding is confirmed by results from [52], who carried out experiments on bug reports in the Eclipse code base containing only titles (but no descriptions). These were demonstrated to be sufficient in achieving the best accuracy, whereas for experiments involving Mozilla code base, both title and description were needed to provide the best results. [37] also found that the use of text length or long queries (e.g., when using the

[11] https://sourceforge.net/p/joda-time/bugs/90/

bug report's description field) can obscure key search terms. They also highlighted the important of program construct such as class name and method name to be present in bug report might improve fault localization accuracy. Consequently, all these experiments demonstrate that the bug report length does not impact on the results of IR based fault localization but the quality of the information that matters (e.g., the class name, or class method are included in the bug report).

I infer that the quality of the bug report is important; a high quality bug report will include as much detail as possible, including a title, details of the bug, and a source code patch or snippets. However, many bug reports are lacking in these aspects, and further research likely needs to be carried out on both assessing existing bug reports and providing templates to write better ones (for the purposes of fault localization). This is an example of situation that has been mentioned before in chapter 1 (subsection 1.1) where the condition "when both IR and SBFL techniques are unable to localize faults, Text Search might help to shed light on the location of faults".

Turning now to other programs in the experiment, Table 4.1 shows that the Chart program scored 100% accuracy for the Text search technique, however, if we look at Table 4.2 (that summarized 395 faults in Defects4j dataset that manage to execute using individual techniques), the experiment executed only 8 bugs out of 26, as the omitted 18 bugs did not have bug report references and could proceed.

Program	*SBFL*			*IR*			*Text Search*			*Total Bug*
	✓	×	O	✓	×	O	✓	×	O	

Time	18	9	-	14	12	1	19	7	1	27
Lang	60	5	-	53	10	2	61	3	1	65
Mockito	25	13	-	6	31	1	11	27	-	38
Chart	22	4	-	7	1	18	8	-	18	26
Math	60	46	-	59	47	-	80	26	-	106
Closure	70	63	-	41	90	2	72	59	2	133
Total	255	140	-	180	191	24	251	122	22	395
		395			371			373		

Table 4.2: 395 faults in Defects4j dataset that were executed using Individual techniques.

From the summary in Table 4.2, it shows that overall, SBFL managed to allocate (✓) faults of 255 bugs from the total of 395 bugs compared to Text Search, which managed to allocate (✓) faults for 251 bugs from the 373 bugs executed; note that 22 bugs are unable to be executed (O) since there is no bug report to enable the search. This also means that SBFL and Text Search were unable (✕) to localize faults of 140 bugs and 122 bugs respectively. By contrast, the IR technique managed to localize (✓) only 180 bugs from the total 371 bugs executed; 24 bugs are unable (✕) to be executed where from the 24 bugs, 22 bugs do not have a bug report while another 2 bugs contains report title that is too short leading to inability to identify the document similarity process. This means that the IR technique was unable to localize faults of 191 bugs from the total of 371 bugs that have been executed.

From Table 4.1, the SBFL technique is more accurate than IR technique for each Defects4j program, except for the Chart program with 87.5% compared to 85% for SBFL technique. However, as mentioned before and shown in Table 4.2, the Chart program only managed to run 8 bugs as another 18 from the total of 26 bugs did not have any information regarding a bug report to proceed with. The overall results show that the Text Search technique is more accurate than SBFL technique in each Defects4j programs except for Mockito program; in this program, SBFL performs the best for Mockito with 66% fault localization accuracy compared to only 29% and 16.22% for Text Search and IR technique respectively.

One of the explanations for this situation is because of the low quality of bug report provided; for example, the title for bug 24 in Mockito program (M24), is very short - “fix some rawtype warnings in tests #467” [12] and as a result the fault cannot be located with both IR and Text Search techniques without provision of additional information. The M24 bug contains only a title that is not only very short, but also does not contain important keyword that enables the Text Search technique to search the location of the fault. The bug is also lacking in any description or other details such as code snippets or attachment, which makes it very difficult for the IR technique to unable to find the similarity between documents. This is an example of situation that has been mentioned in Chapter 1 (Subsection 1.1) where the condition of “when fault localization results using IR techniques are unable to precisely identify fault location, SBFL technique may provide more precise information”. This situation also applies to 22 bugs that do not have a bug report, leading to a situation where IR has been unable to localize faults, as the document similarity process cannot successfully complete.

[12] https://github.com/mockito/mockito/pull/467

Bug 27 from the Lang program (L27) is one example of the situation “when fault localization results using SBFL techniques are unable to localize faults, the results may be supplemented with analysis from IR technique”. The L27 [13] bug report contains a title, a simple description that is sufficient (though it is not a particularly lengthy report), and finally the attachment of the source code. The SBFL technique is unable to detect the location of the fault for L27 bug, and by using IR technique, the location for bug L27 is identified. The L27 bug also could be localized by using text search only by using the title “NumberUtils createNumber throws a StringIndexOutOfBoundsException when argument containing "e" and "E" is passed in”; this title does express the exact issues of the fault accordingly. All these findings from the experiments on individual fault localization techniques suggest to us that the combinations of techniques may enhance accuracy in localizing fault.

As a conclusion based on the data from Table 4.1 and Table 4.2, Text search provides the best accuracy in fault localization with the highest score 67.29%, where 251 bugs can be located from 373 bugs. This also includes a program such as Math (where 80 bugs can be located from 106 total bugs) and Closure (72 bugs can be located from the total of 131 bugs) with 75.47% and 55% of fault localization accuracy respectively.

4.2 RQ2: How accurate is the process of localizing faults using the combinations of SBFL, IR and Text Search?

The second research question considers different combinations of fault localization techniques and evaluates the behavior of each combination in terms of accuracy. The

[13] https://issues.apache.org/jira/browse/LANG-638

results of executing each combination of technique against the chosen corpus are presented in Table 4.3 below, which shows the accuracy of combinations of SBFL, IR and Text Search on all 395 faults in all six Defects4j programs. The boldface indicates the combined technique demonstrating the greatest accuracy for each program.

Program	*SBFL + Text Search*			*SBFL + IR*			*Text Search +IR*			*SBFL + IR + Text Search*		
	Top 1	Top 5	Top 10	Top 1	Top 5	Top 10	Top 1	Top 5	Top 10	Top 1	Top 5	Top 10
Time	51.85%	81.48%	88.89%	55.55%	70.36%	77.77%	42.31%	76.92%	88.46%	62.96%	85.18%	**92.59%**
Lang	90.77%	95.39%	98.46%	89.22%	93.84%	95.38%	75%	95.31%	98.43%	92.31%	96.92%	**100%**
Mockito	42.10%	65.78%	**73.68%**	42.11%	60.53%	65.79%	13.16%	23.69%	31.58%	44.73%	68.42%	**73.68%**
Chart	69.23%	84.62%	88.46%	69.23%	80.77%	88.46%	62.5%	87.5%	**100%**	69.23%	84.61%	88.46%
Math	62.26%	82.07%	91.51%	59.43%	73.58%	78.30%	40.57%	73.59%	85.85%	64.15%	85.85%	**93.4%**
Closure	37.68%	57.25%	73%	31.58%	57.15%	65.41%	12.98%	43.51%	64.12%	38.34%	63.15%	**77.44%**
Accuracy	56.45%	74.43%	83.80%	53.67%	70.38%	76.20%	34.59%	62.20%	75.60%	58.48%	78.23%	**86.84%**

Table 4.3: Result for combination technique accuracy in Top 1, Top 5 and Top 10 for 6 programs with 395 faults in Defects4j dataset.

Previously (in subsection 4.1) I have already discussed the motivation for choosing Top 10 accuracy as a "sufficient accuracy", I will only summarize the Top 10 result accuracy for further discussion in this section. However, I will still include the result for Top 1 and Top 5 results for each combined technique for further insight. In addition, the total time taken to execute each experiment either individual or combined technique does not affect by the Top N selection as the experiment will be executed to generate all of the results first, then from the result generated, the location of the fault will be sorted according to Top N.

As expected, Table 4.3 shows that the highest accuracy for combinations of techniques is SBFL combined with IR and Text Search. This combination scores 86.84% accuracy on fault localization. On the other hand, Table 4.4 below shows the 395 faults in Defects4j dataset that could be processed using a combination of techniques. In Table 4.4, 343 bugs can be localized using this combined technique out of a total of 395 bugs. We anticipated this result: the three fault localization techniques are conceptually and largely independent (operating on different data inputs - specifically source code, bug reports, and keywords) and thus their combination, we expected, would offer a greater accuracy than any individual technique.

Program	*SBFL + Text Search*			*SBFL + IR*			*IR + Text Search*			*SBFL + Text Search + IR*			*Total Bug(s)*
	✓	×	O	✓	×	O	✓	×	O	✓	×	O	
Time	24	3	-	21	3	-	23	3	1	25	2	-	27
Lang	64	1	-	62	3	-	63	1	1	65	-	-	65
Mockito	28	10	-	25	13	-	12	26	-	28	10	-	38
Chart	23	3	-	23	3	-	8	-	18	23	3	-	26
Math	97	9	-	83	23	-	91	15	-	99	7	-	106
Closure	97	36	-	87	46	-	84	47	2	103	30	-	133
Total	331	64	-	301	91	-	282	91	22	343	52	-	395
		395			395			373			395		

Table 4.4: 395 faults in Defects4j dataset that could be processed using a combination of techniques.

As shown in Table 4.3, the Time programs led to a score of 92.59% for fault localization, the Math programs score 93.4% while the Closure program scores 77.4%. The combination of SBFL and Text Search technique result is the second-best fault localization technique in terms of accuracy. This rather unexpected finding is indicated by an accuracy score of 83.80% where 331 bugs were localized from the total 395 bugs.

Before carrying out the experiment I expected that the second-best technique (in terms of accuracy) would be the combination of SBFL and IR, since these are considered the state of the art in fault localization. However, it appears that at least for the Defects4j programs – the simple mechanisms offered by Text Search provide unexpectedly accurate results, indeed more so than SBFL+IR together. This probably because of the input or bug report to the Text Search technique are using a keyword that reflect the fault locations, and it is somewhat surprising to see a relatively simple Text Search combined with SBFL providing much more accurate results than SBFL combined with an IR. This is compelling in terms of improving the state of the art in fault localization – Text Search may be able to play a richer role in fault localization than it has up to now – but it also suggests that more experiments need to be carried out to understand in depth whether these results are repeatable for buggy programs other than Defects4j, perhaps where more variable bug reports and inserted keywords arise.

Another question arises from this result: "if combining two techniques are not much different from combining three techniques in terms of accuracy, when might a combination of three techniques provide value?". The next experiment on runtime execution aims to answer this.

The Lang programs managed to allocate faults for all 65 bugs in the program; that is, it successfully scored 100% using the combination of SBFL, IR and Text Search. Lang program even manage to perform excellently with individual technique experiment where it scored 92% for SBFL and 94% for Text Search as shows on Table 4.1 above, though the score is not 100% accurate, it is among the highest in individual technique results for a program. Lang is an example of a program where it is possible to allocate all faults when a good test case coverage and a high-quality bug report are combined.

For the Mockito program, the greatest accuracy is the same for the combination of SBFL with Text Search and the combination of SBFL with IR and Text Search; both scores are 73.68% in accuracy. This is probably because as mentioned in the previous subsection (4.1), the quality of the bug report plays a huge role in fault localization, especially for the techniques rely on it such as IR and Text Search. However, for the Mockito program many of the bug reports cannot be classified as good, as they did not contain all the elements of a high-quality report that were specified in Chapter 3. This is why the scores of both techniques are the same: IR does not contribute more than Text Search since both techniques are referring to elements of the same low quality bug reports. This can also be seen in the results of the combination of Text Search and IR for Mockito, which provides the worst score in accuracy with 31.58%. This is because both techniques solely depend on the bug reports. The individual technique results are as we described previously (refer to Table 4.1 above).

For the Chart program, as with the Mockito program, the accuracy for the combination of IR and SBFL is the same as both the combination of SBFL and Text Search, and the combination of SBFL, IR and Text Search: all three combinations scores 88.46%. Interestingly, the combination of Text Search and IR managed to achieve 100% accuracy, however, as mentioned earlier (if we refer to Table 4.4), this is because there are missing bug reports for 18 bugs, and since both IR and Text Search technique are using bug reports for fault localization, only 8 bugs are included in the experiment. This is a good example demonstrating that sometimes adding more techniques does not help much in terms of an increase in accuracy, because the technique in question may simply be inapplicable to the fault localization problem in question.

The combination of IR with SBFL technique results does not have a substantial difference to the combination of IR with Text Search technique, with accuracy scores of 76.20% and 75.60% respectively. As a conclusion, the accuracy of fault localization improved impressively by 19.55% when combined technique are being used compared to individual technique. As such, the combination of SBFL, IR and Text Search is the best combination in achieving the highest accuracy, in fault localization by 86.84% accuracy from only 67.29% accuracy using Text search technique alone. However, it is important to consider execution time to get a greater indication of the resources involved in applying the combination of techniques; this is the motivation to drive us to the next question for this thesis.

4.3 RQ3: What is the execution runtime of the standalone techniques and combined techniques? How long is the average execution time for each technique on one bug?

The previous research questions, RQ1 and RQ2, concerned accuracy of different combinations of fault localization techniques. RQ3 considers the time taken for each individual and combined technique as well as the time spent on each bug in fault localization. Table 4.5 below shows the comparison on total time taken for SBFL, IR and Text Search as well as the time spent for each bug in individual technique.

Program	SBFL		IR		Text Search	
	Total time (s)	Time per bug (s)	Total time (s)	Time per bug (s)	Total time (s)	Time per bug (s)
Time	31. 35	1.16	74.73	2.87	1.50	**0.06**
Mockito	5.26	0.14	89.42	2.42	2.49	**0.07**
Lang	0.73	**0.01**	191.66	3.04	4.11	0.06
Math	4.04	**0.04**	258.89	2.44	14.86	0.14
Chart	2.85	0.11	21.75	2.72	0.49	**0.06**
Closure	643.72	4.84	405.37	3.10	13.93	**0.11**
Overall time	687.95	1.74	1041.82	2.81	**37.38**	**0.10**

Table 4.5: Overall program runtime and time spent on each bug for SBFL, IR and Text Search technique.

Table 4.5 shows that Text Search is the fastest technique individually, where the overall time taken to execute all six Defects4j programs that contains 395 bugs is 37.38 seconds; each bug only needs around 0.10 seconds or less to be localized. Time, Mockito, Chart and Closure program are the fastest in fault localization when using Text Search, where the total time taken to localize faults for each program are 1.5 seconds, 2.49 seconds, 0.49 seconds and 13.93 seconds respectively.

The Time program only needs an average time around 0.06 seconds to localize faults, with 0.07 seconds for Mockito program, 0.06 seconds for Chart program and 0.11

seconds for the Closure program. On the other hand, Lang and Math are the fastest with SBFL technique though both programs contain large number of bugs with 65 and 106 bugs respectively. Each Lang and Math program score 0.73 seconds and 4.04 seconds of total execution time, with each bug only needing an average of 0.01 seconds and 0.04 seconds respectively, to localize faults. In addition, the overall time spent with SBFL to localize all faults is 687.95 seconds, while IR is the slowest to localize all faults, requiring around 1041.82 seconds to process all six Defects4j programs.

With respect to combinations of technique, Table 4.6 shows the total time taken for each combination of techniques to be executed and as well as the time spent for locating each bug for the combined technique. Table 4.6 shows that the combination of SBFL and Text Search runtime are the fastest; in my opinion this is because of the simplicity of Text Search, where it is using a simple string matching algorithm, while IR needs to construct more sophisticated models in finding similarities between documents.

The time taken to execute all six programs in Defects4j is 725.33 seconds, the fastest among all combined techniques, where it only took 6.74 seconds per bugs to be localized. The time taken to execute all six programs for combination of Text Search and IR technique is 1079.2 seconds with average 16.91 seconds per bugs to be allocate while the combination of IR and SBFL spend 1729.77 seconds with average 20.7 seconds time taken for each bug to be localize. The slowest technique for combination is the combination of SBFL, IR and Text Search technique that took 1767.15 seconds with average calculation for each bug executed are around 20.27 seconds per bug.

Program	SBFL + IR		SBFL + Text Search		SBFL + IR + Text Search		Text Search + IR	
	Total time (s)	Time per bug (s)	Total time (s)	Time per bug (s)	Total time (s)	Time per bug (s)	Total time (s)	Time per bug (s)
Time	106.08	3.93	32.85	1.22	107.58	3.98	76.23	2.93
Mockito	94.68	2.49	7.75	0.20	97.17	2.56	91.91	2.42
Lang	192.39	2.96	4.84	0.07	196.5	3.02	195.77	3.06
Math	262.93	2.48	18.9	0.18	277.79	2.62	273.75	2.58
Chart	24.6	0.95	3.34	0.13	25.09	0.97	22.24	2.72
Closure	1049.09	7.89	657.65	4.94	1063.02	7.99	419.3	3.20
Overall time	1729.77	20.7	**725.33**	**6.74**	1767.15	20.27	1079.2	16.91

Table 4.6: Overall runtime for combination techniques and time spent for each bug

The combination of SBFL and Text Search technique for Time program only took 32.85 seconds with average of 1.22 seconds for each bug to localize fault, Mockito program about 7.75 seconds with average time 0.2 seconds per bugs to localize faults, Lang program took 4.84 seconds with 0.07 seconds for each bug to localize fault, Math program took 18.9 seconds with 0.18 seconds for each bug to localize fault and Chart program took 3.34 seconds with 0.13 seconds for each bug to localize. For the Closure program, the fastest combination technique for this program is the combination of Text Search and IR technique with total time taken is 419.3 seconds and time taken for each bug to be localized is 3.20 seconds.

4.4 RQ4: Which combination of techniques is the most accurate and performant (fastest in runtime)?

This research question aims to compare and identify which combination of techniques is the most accurate and shortest in runtime. As mentioned in the earlier research question (RQ1), SBFL alone manages to allocate 255 bugs (64.56%), IR allocates 180 bugs (48.52%) while Text Search allocates 251 bugs (67.29%) from the total of 395 bugs (see Table 4.1). However, in RQ2 the combination of SBFL, IR and Text Search managed to allocate up to 86.84% or 343 bugs from 395 bugs compared to by only using individual technique. This positive outcome from the combined technique reduced the unlocalized fault to only 52 bugs (see Table 4.4) compared to 140 bugs if localize fault using SBFL alone, 122 bugs if using Text Search and 191 bugs if using IR (see Table 4.2).

In essence, as shown in Table 4.7 below, a combination of fault localization techniques demonstrably provides more accurate results compared to individual fault localization techniques. Table 4.7 below shows the summary of overall accuracy and runtime for individual and combinations of techniques; this summary consists of the accuracy result for each technique on all Defects4j program, and as well as the time spent on each bug in fault localization.

Technique	Accuracy	Runtime per bug(s)
SBFL	64.56%	1.74s
IR	48.5%	2.81s
Text Search	67.29%	0.10s
SBFL + Text Search	83.80%	6.74s
SBFL + IR	76.20%	20.7s
IR + Text Search	75.60%	16.91s
SBFL + Text Search + IR	**86.84%**	**20.27s**

Table 4.7 : Summary of overall accuracy and runtime for individual and combination technique for all program in Defects4j.

The main motivation of fault localization is to identify faults that caused a visible failure; thus it is important to measure the accuracy first. There is arguably no point in using the fastest runtime technique when the result is not accurate. From the executed experiment, we found that the combination of three techniques, i.e., SBFL + Text Search + IR, is the most accurate in fault localization technique compared to others, however it is the longest time taken in runtime per bug with 20.27 seconds. There are undoubtedly situations where the runtime is a concern (e.g., localizing faults in extremely large codebases), and for these, the accuracy might be lesser weighted. In such cases, the developer or programmer might want to consider the combination of SBFL + Text Search technique, where it is the fastest runtime technique, with a second best place in accuracy of fault localization for combination technique.

4.5 Threats to validity

This subsection will discuss the different threats that can threaten the validity of my experiment. Validity evaluation is very important in demonstrating that the result for the experiment is valid. There are four types of threats that are usually being discussed after experiment execution which are: threats to construct validity, threats to conclusion validity, threats to internal validity and threats to external validity.

4.5.1 Threats to construct validity

Due to time constraints, only three techniques - SBFL, IR and Text Search - are being considered. Some of the tools available to support these techniques proved to be difficult to use in a modern environment (i.e., their execution was not easy to reproduce and involved multiple setups with different programming languages), and some of them are not accessible (i.e., the version is obsolete, no replication package included). To ensure uniformity, reliability and repeatability of experiments, the tools used in the experiments have been designed and built from scratch by using my understanding of existing tools from research papers. I reimplemented all the algorithms and tools using Python.

It is possible that I have introduced minor errors or flaws in my implementations of the Text Search, SBFL and IR algorithms, but as I have tested these implementations extensively, and am applying the implementations against widely accepted benchmarks, these errors are likely to have only minimal or trivial effects. I also agree that hardware (e.g., type of processor, machine, RAM, memory) and software settings (e.g., programming language, software versions) factors contribute a slightly difference in result (e.g., time). The syntax of programs, error type or bug type are also might affect the result where the runtime might be slightly different but should not affect accuracy results.

4.5.2 Threats to internal validity

There are so many techniques that are available for fault localization with variety of combinations possibilities. Again, due to time constraints, availability, and accessibility issues, only 4 combinations are managed to be executed in this experiment. As noted earlier there are many IR techniques that have been applied in Fault localization research, including bug histories and stack traces. I chose the document similarity technique given the availability of bug reports for many programs in Defects4j, but of course by using other IR techniques, probably it may provide different results for the fault location. Nevertheless, document similarity is a very widely used IR technique and as such I expect the results are useful representatives of the utility of IR in the context of sophisticated fault localization workflows.

4.5.3 Threats to conclusion validity

Different results may be obtained if datasets other than Defects4j were to be used. The reason why I choose Defects4j datasets for this experiment, as mentioned in Chapter 3 before this, it is because Defects4j is a standard dataset that are being used for fault localization research for SBFL. Moreover, Defects4j already being used as a benchmark to understand fault localization in real-world environments. The Defects4j (version 1.5.0) sample size contains 332,000 Loc, 6 programs and 395 faults are considered as a large sample size program, and arguably it is large enough to be used in this.

I am aware that for IR technique, many researchers are using the Bench4BL dataset [127] [32]in their experiments, however, I also noticed that this dataset does not have information (other than bug report) that may be used and be useful by the SBFL technique, such as program spectrum and metrics to enable suspiciousness values calculation in the experiment execution. In other word, If I was to use the Bench4BL dataset, the results for SBFL cannot be generated and it will be hard to compare with IR and Text Search.

4.5.4 Threats to external validity

I now discuss threats to external validity. Many past researchers relied on experiments with seeded faults [140] [33] and a toy program [141], which are not representative of a real-world environment. More recently, real-world or more realistic experiments [143] [144] have increased in volume, but more research is needed. Therefore, the selection of a real-world datasets such as Defects4j is crucial, as that was my concern before executing the experiments. I agree that a controlled environment in experiment might produce different results in a real-world environment though the selection of Defects4j datasets is generally considered to be state-of-the-practice in mimicking real-world fault behaviors. As such, we argue that we did our best to imitate and copy a real-world environment by selecting a real-world data set. It is important to execute an experiment under a controlled environment to generate reliable and valid results. Taking this as an important factor that might affect our experiment results, that is why the selection of sample on a real-world environment is being considered.

4.6 Results summary

As mentioned earlier, in fault localization the accuracy is the most important element in fault localization to dictate either the technique is reliable or not. From what we have seen and experienced in a real-world experience, the difference in seconds does not really matters as it will give a very minimum impact. In other words, the importance of accuracy of the technique outweighs the runtime period, as long as the latter is not too long (e.g., days, weeks). As mentioned earlier, currently most fault localization in practice is carried out manually, and the time to allocate faults are varies (i.e. sometimes take days and weeks), speeding up the fault localization process with high accuracy result does make a difference and counted as a contribution in fault localization researches.

Though the execution time of each of the techniques has been measured, current research can also be seen as setting a benchmark for future research considering a runtime for an experiment might be different depends on many factors such as software and hardware factor, development environment, program size, etc. There are cases that led to very similar accuracy results, thus at this point, the runtime execution in fault localization process can be used to help to identify the best combined technique.

The Venn Diagram in Figure 4.1 below is a summary of overall fault localization result that are generated from the experiment using SBFL, IR and Text Search technique which are represent in the three subsets. In the diagram below, ‘E’ is the total number of elements, and in this case; the total number of 395 fault from six program in Defects4j that are used in this experiment. From 395 total faults, 52 faults are unable to be located by using any techniques including combination techniques.

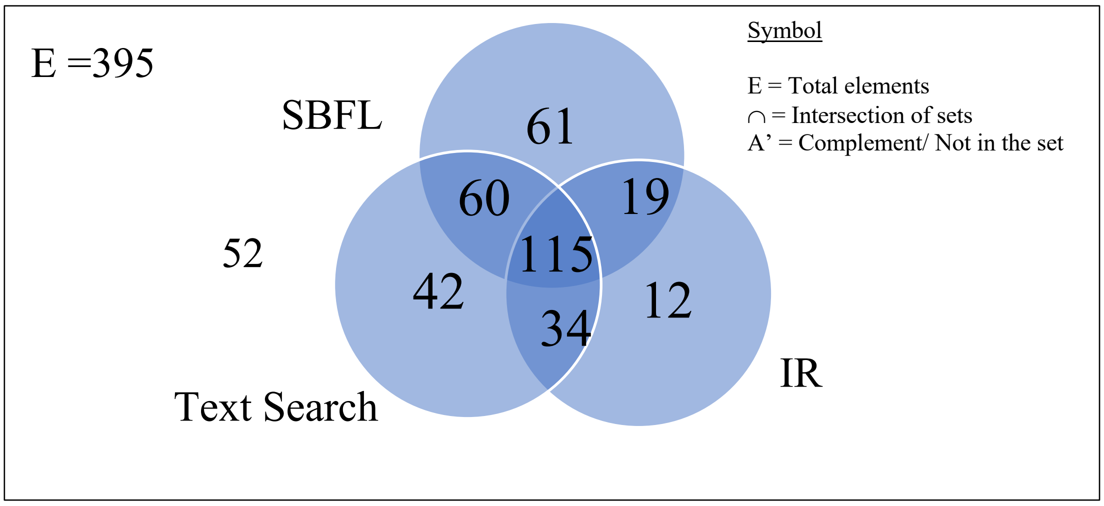


Figure 4.1 : The Venn Diagram of fault localization result that are generated from the experiment using SBFL, IR and Text Search technique in Defects4j.

As we can see in subsection 4.1 (RQ1) and subsection 4.2 (RQ2) where Table 4.2 and Table 4.4 that shows the total number of faults that each individual technique managed to allocate where though SBFL, IR and Text Search technique alone manage to allocate 255, 180 and 251 faults respectively, we must agree that the combination of techniques manage to increase the accuracy of fault localization to 331 faults (SBFL + Text Search), 301 faults (SBFL+IR), 282 faults (IR + Text Search), and 343 faults (SBFL + Text Search + IR). The varies value in intersection and union values strengthen the idea of combination of technique does increases the accuracy in fault localization.

From the Venn Diagram above also, we can see that using the combination of more technique (SBFL + Text Search + IR) benefits the most (highest accuracy) and avoiding using it may cause developer hard time to identify the fault location, for example if SBFL + Text Search (fastest combination of two techniques with 6.74s) are used, only 331 faults are managed to be allocated while another 12 faults lost the chance to be localized. Same condition applies with the situation if SBFL+IR and IR + Text Search combination are used, the optimal fault localization results can only go to 301 faults with 42 faults missed chance to fault localization and 282 faults with 61 faults missed chance fault location respectively.

From the data that has been collected, we suggest that ideally the best sequence of techniques is to run the most accurate technique first. The general approach taken in this research allows for developer to choose which technique that they prefer to begin first, since we believe that the familiarity with certain or specific fault localization technique might make it easier for them to apply it. However, we also believe that by following the sequence and applying the most accurate technique first can speed up the fault localization process, because there are opportunities for optimization and caching of results. For example, in a production debugging environment, the fault that has been localized in the first technique does not need to be localized again by the second technique, because its location is already predicted. The developer can proceed to focus on undetected location only hence why the most accurate technique sequence role contributes to this case. Further optimizing this process by caching fault localization results and eliminating bugs or bug reports from consideration is a direction for further research.

Finally, I also want to highlight the importance of quality of the bug report; there may be situations where a fault location cannot be identified as the information provided in the bug report is not enough. Though the effectiveness of the IR and Text Search techniques is likely to depend heavily on the quality of the bug reports, [51] unfortunately notes that sufficiently high-quality bug reports that contain important information are not always available. This makes it challenging to carry out repeatable and plausible experiments on the utility of combinations of different techniques in fault localization. Fortunately, we have the Defects4j database as a benchmark.

5 Conclusions and Future Work

5.1 Conclusion

In this thesis, I analyzed 395 faults in real programs using fault localization techniques; specifically, using individual techniques as well as a combinations of multiple techniques, particularly SBFL, IR, and Text Search. Based on the results of experiments, which assessed both the accuracy and the run time of the individual and combined techniques, I demonstrated that a combination of two or more techniques is indeed complementary with respect to fault localization, and different combinations of techniques provide different profiles of improved accuracy, with a price to be paid in terms of increased execution time.

From the analysis of the data that was collected, ideally the best sequencing of techniques is to run the most accurate technique first. As mentioned earlier, in fault localization the accuracy is the most important characteristic; it dictates whether the technique is considered reliable and repeatable (or not). From what we have observed in the experiments, a performance difference in seconds or milliseconds has little impact on the utility of the fault localization technique. In other words, for fault localization, the accuracy of the technique outweighs the execution time taken, as long as it is not taking too much time (e.g., days or weeks).

One observation from the experimental results is that though the time spent in applying individual techniques is slightly less than the combinations of techniques, the combination of techniques significantly outperforms standalone techniques in term of accuracy. The amount of time taken by each technique does not define its accuracy. Even though the combination of techniques takes longer compared to individual techniques, when it comes to critical situations (e.g., critical bugs in important code) the execution time for the fault localization technique may not be as important as finding the approximate location of the fault.

As mentioned in Chapter 4, the results of the experiments are quantitative, and this permits quantitative comparison of the results; hence, I do not include further statistical tests, as the direct comparison of the experiment results is sufficient for identifying the strengths and weaknesses of the individual and combined techniques. Adding statistical tests does not contribute sufficient further understanding to justify the extra effort and expense involved in setting up and carrying out those tests.

I will provide a guideline for researchers and practitioners on how to combine existing techniques for fault localization based on the results collected in the experimentation. I anticipate that this guideline might be complementary to fault localization technique research in the future, as it could further improve individual techniques accuracy, while showing that the combinations of techniques can be beneficial. Fault localization techniques have comprehensively been utilized and evaluated individually. This research demonstrated how relatively straightforward it is to combine even the most disparate fault localization approaches.

5.1.1 Guideline for combinations of fault localization

A guideline on how to combine the techniques based on the evidence collected in the experimentation is provided below. This ideally will help enhance reproducibility in future research.

Step 1: Identify fault localization techniques

Firstly, select the fault localization technique to combine. As mentioned early in this thesis, it is beneficial to use techniques from different families and those that use complementary input sources (e.g., source code and bug reports), as this does help in improving accuracy. Taking as an example from the experiment executed in this thesis, 3 fault localization techniques that are used to localize faults are SBFL, IR and Text Search.

Step 2: Sequence - Run the most accurate first

When sequencing the techniques to run, execute the techniques in order of accuracy. Arguably the less accurate techniques will incrementally improve the outputs from the most accurate techniques. For example, based on the experiments in this thesis, I would first execute the Text Search technique, followed by SBFL and IR techniques.

Step 3: Combine the results

The setting for each experiment is to execute the program until finish to achieve maximal approaches. As the results accumulate from each technique, they may overlap, but as long as they produce a consistent result, this is not problematic. It would be more efficient to customize techniques after the first to try to localize faults not localized by the first technique to avoid redundant results and decrease execution time. For example, a fault that has been localized from Text Search does not need the be included in second execution of SBFL technique. Similarly, it would be beneficial to target execution of IR, so that its execution will target faults on not localized in Text Search and SBFL.

This research is complementary to existing research on fault localization techniques, as it could further improve the state-of-the-art by providing indications of how to combine existing techniques, while showing that the combination can be productive. Fault localization techniques have always been utilized and evaluated individually. This research demonstrates the challenges and costs associated with combining even the most different fault localization approaches.

A key recommendation of this research is that developers should combine different fault localization strategies while perhaps placing time limits on how long they will spend on fault localization, rather than using one technique alone and waiting perhaps arbitrary amounts of time for a result. This implies that it is more important to understand how individual techniques contribute when combined with other techniques, rather than solely understanding the performance of each technique in isolation. Moreover, the data presented in Chapter 4 on execution time gives some indication of threshold execution times that may be helpful in guiding users in understanding how long they may need to wait for a useful result from combinations of techniques.

An additional observation is that the combination of techniques is reliable when localizing faults; that is, it produces repeatable and deterministic results when run on the same code. However, when using the IR technique in combination with other approaches, if the bug reports do not meet the quality/requirement (highlighted in Chapter 3) the result may not support the developer in locating faults. Our results shows that there is no single technique that can be effective in locating faults, so it is our recommendation to nevertheless use multiple technique as a strategy in fault localization. The main contributions of this thesis can be summarized as follows.

- An empirical study that compares a specific set of fault localization techniques on real faults, specifically SBFL, IR and Text Search techniques.
- Observations on the relationship between SBFL, IR and Text Search technique behavior when applied to a real-world and large-scale dataset.
- A guideline for different combinations of fault localization techniques that are configurable based on the accuracy and time spent when localizing faults.
- Infrastructure for evaluating and combining fault localization techniques for future research.

5.2 Future work and outlook

It would be beneficial if the studies in this thesis were replicated on a different dataset of Java programs or other programming language to understand the results different if any, in order to improve fault localization. There needs to be further work on the full automation of the combination of techniques, for example, by setting up workflows or pipelines of fault localization techniques, where individual cells in the pipeline can be turned off or on as needed. One could also envision combining such a workflow-based approach with machine learning or hyper heuristics, where it could be learned which techniques to turn off (or on) depending on configuration data from the user, or from direct analysis of the input data itself. As a concrete example, suppose that a subset of bug reports is of "poor quality" according to the criteria discussed in Chapter 3; a machine learning configuration model could "learn" this fact and trigger a configuration element for the workflow that turns off an IR cell/module (as this would provide poor or erroneous results given poor quality bug reports). In other words, "smart" workflows for fault localization may be interesting to explore.

Carrying out such experiments on diverse datasets *in the context of* a debugging process may be interesting as well, to understand the feed forward and feedback between fault localization and debugging actions of programmers. Though much research on prediction and root cause analysis has been made, applying it to a real-world environment might face difficulties as every source code structure is different. As well, every programmer's knowledge and experience are also different. Understanding how combinations of fault localization techniques interact with programmer experience and abilities would be interesting to explore.

The results for this experiment in an increase of fault localization accuracy is expected since many combinations of fault localization experiment is moving to a positive outcome (in term of accuracy that increased compared to using single technique only). By executing the experiment and providing guidelines to pave a way or an idea for future research. In the future it will beneficial if the combination of fault localization techniques is using a majority consensus approach to dictate the location of the fault after execution of fault localization techniques. Finally, it would be interesting and challenging to investigate the effectiveness of combinations of fault localization techniques in software product lines, to understand the relationship between faults and product instances.

[وَلَسَوْفَ يُعْطِيكَ رَبُّكَ فَتَرْضَىٰٓ]

And soon will thy Guardian-Lord give thee (that wherewith) thou shalt be well-pleased (Quran: 93:5)

Appendix

Suspiciousness results output after execution	Ranking suspiciousness results output
Statement, Suspiciousness	org.apache.commons.lang3.LocaleUtils#99,0.5773502691896258
org.apache.commons.lang3.LocaleUtils$SyncAvoid#288,0.2773500981126146	org.apache.commons.lang3.LocaleUtils#89,0.4472135954999579
org.apache.commons.lang3.LocaleUtils$SyncAvoid#295,0.2773500981126146	org.apache.commons.lang3.LocaleUtils#92,0.4472135954999579
org.apache.commons.lang3.LocaleUtils$SyncAvoid#296,0.2773500981126146	org.apache.commons.lang3.LocaleUtils#93,0.4472135954999579
org.apache.commons.lang3.LocaleUtils$SyncAvoid#297,0.2773500981126146	org.apache.commons.lang3.LocaleUtils#96,0.4472135954999579
org.apache.commons.lang3.LocaleUtils$SyncAvoid#298,0.2773500981126146	org.apache.commons.lang3.LocaleUtils#97,0.4472135954999579
org.apache.commons.lang3.LocaleUtils#57,0.0	org.apache.commons.lang3.LocaleUtils#98,0.4472135954999579
org.apache.commons.lang3.LocaleUtils#58,0.0	org.apache.commons.lang3.LocaleUtils$SyncAvoid#288,0.2773500981126146
org.apache.commons.lang3.LocaleUtils#42,0.2773500981126146	org.apache.commons.lang3.LocaleUtils$SyncAvoid#295,0.2773500981126146
org.apache.commons.lang3.LocaleUtils#46,0.2773500981126146	org.apache.commons.lang3.LocaleUtils$SyncAvoid#296,0.2773500981126146
org.apache.commons.lang3.LocaleUtils#89,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#90,0.0	
org.apache.commons.lang3.LocaleUtils#92,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#93,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#94,0.0	
org.apache.commons.lang3.LocaleUtils#96,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#97,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#98,0.4472135954999579	
org.apache.commons.lang3.LocaleUtils#99,0.5773502691896258	
org.apache.commons.lang3.LocaleUtils#101,0.0	
org.apache.commons.lang3.LocaleUtils#102,0.0	
org.apache.commons.lang3.LocaleUtils#104,0.0	
org.apache.commons.lang3.LocaleUtils#105,0.0	
org.apache.commons.lang3.LocaleUtils#107,0.0	
org.apache.commons.lang3.LocaleUtils#108,0.0	
org.apache.commons.lang3.LocaleUtils#110,0.0	
org.apache.commons.lang3.LocaleUtils#111,0.0	
org.apache.commons.lang3.LocaleUtils#112,0.0	
org.apache.commons.lang3.LocaleUtils#114,0.0	
org.apache.commons.lang3.LocaleUtils#115,0.0	
org.apache.commons.lang3.LocaleUtils#116,0.0	
org.apache.commons.lang3.LocaleUtils#118,0.0	
org.apache.commons.lang3.LocaleUtils#119,0.0	
org.apache.commons.lang3.LocaleUtils#121,0.0	
org.apache.commons.lang3.LocaleUtils#122,0.0	
org.apache.commons.lang3.LocaleUtils#124,0.0	
org.apache.commons.lang3.LocaleUtils#125,0.0	
org.apache.commons.lang3.LocaleUtils#127,0.0	
org.apache.commons.lang3.LocaleUtils#144,0.0	
org.apache.commons.lang3.LocaleUtils#166,0.0	
org.apache.commons.lang3.LocaleUtils#167,0.0	
org.apache.commons.lang3.LocaleUtils#168,0.0	
org.apache.commons.lang3.LocaleUtils#169,0.0	
org.apache.commons.lang3.LocaleUtils#170,0.0	
org.apache.commons.lang3.LocaleUtils#172,0.0	
org.apache.commons.lang3.LocaleUtils#173,0.0	
org.apache.commons.lang3.LocaleUtils#175,0.0	
org.apache.commons.lang3.LocaleUtils#176,0.0	
org.apache.commons.lang3.LocaleUtils#179,0.0	
org.apache.commons.lang3.LocaleUtils#193,0.2773500981126146	
org.apache.commons.lang3.LocaleUtils#207,0.0	
org.apache.commons.lang3.LocaleUtils#218,0.2773500981126146	
org.apache.commons.lang3.LocaleUtils#232,0.0	
org.apache.commons.lang3.LocaleUtils#233,0.0	
org.apache.commons.lang3.LocaleUtils#235,0.0	
org.apache.commons.lang3.LocaleUtils#236,0.0	
org.apache.commons.lang3.LocaleUtils#237,0.0	
org.apache.commons.lang3.LocaleUtils#238,0.0	
org.apache.commons.lang3.LocaleUtils#239,0.0	
org.apache.commons.lang3.LocaleUtils#240,0.0	
org.apache.commons.lang3.LocaleUtils#241,0.0	
org.apache.commons.lang3.LocaleUtils#243,0.0	
org.apache.commons.lang3.LocaleUtils#246,0.0	
org.apache.commons.lang3.LocaleUtils#247,0.0	
org.apache.commons.lang3.LocaleUtils#248,0.0	
org.apache.commons.lang3.LocaleUtils#250,0.0	
org.apache.commons.lang3.LocaleUtils#264,0.0	
org.apache.commons.lang3.LocaleUtils#265,0.0	
org.apache.commons.lang3.LocaleUtils#267,0.0	
org.apache.commons.lang3.LocaleUtils#268,0.0	
org.apache.commons.lang3.LocaleUtils#269,0.0	
org.apache.commons.lang3.LocaleUtils#270,0.0	
org.apache.commons.lang3.LocaleUtils#271,0.0	
org.apache.commons.lang3.LocaleUtils#272,0.0	
org.apache.commons.lang3.LocaleUtils#273,0.0	
org.apache.commons.lang3.LocaleUtils#276,0.0	

```
org.apache.commons.lang3.LocaleUtils#279,0.0
org.apache.commons.lang3.LocaleUtils#280,0.0
org.apache.commons.lang3.LocaleUtils#281,0.0
org.apache.commons.lang3.LocaleUtils#283,0.0
'org.apache.commons.lang3.LocaleUtils#135,0.5773502691896258',
'org.apache.commons.lang3.LocaleUtils#138,0.5773502691896258',
'org.apache.commons.lang3.LocaleUtils#127,0.5',
'org.apache.commons.lang3.LocaleUtils#130,0.5']
```

Figure 3.2 Full data on output

Result summary for Defects4j program:

Program	**SBFL**		**IR**		**SBFL + IR**		**Text Search**		**SBFL + Text Search**		**SBFL+IR+ Text Search**	
	E	**Time**	**E**	**Time**	**E**	**Time**	**E**	**Time**	**E**	**Time**	**E**	**Time**
Time	67%	31. 35s	54%	74.73s	78%	106.08s	73%	1.50s	89%	32.85s	93%	107.58s
Mockito	66%	5.26 s	16%	89.42s	66%	94.68s	29%	2.49s	74%	7.75s	74%	97.17s
Lang	92%	0.73 s	86%	191.66s	95%	192.39s	94%	4.11s	98.5%	4.84s	100%	196.5s
Math	70.8%	4.04s	55.6%	258.89s	77.4%	262.93s	79%	14.86s	92.5%	18.9s	94.3%	277.79s
Chart	85%	2.85 s	87.5%	21.75s	88.5%	24.6s	100%	0.49 s	88.5%	3.34s	88.5%	25.09s
Closure	53%	643.72s	31.3%	405.37s	65.4%	1049.09s	55%	13.93s	73%	657.65s	78.2%	1063.02s

Result summary for SBFL + Text Search:

Program	SBFL		Text Search		SBFL Top 10 +Text Search		SBFL Top 30 + Text Search		SBFL Top 100 + Text Search	
	E	Time	E	Time	E	Time	E	Time	E	Time
Time	67%	31. 35s	73%	1.50s	89%	32.85s	100%	32.85	-	-
Mockito	66%	5.26 s	29%	2.49s	74%	7.75s	86.8%	7.75s	100%	7.75s
Lang	92%	0.73 s	94%	4.11s	98.5%	4.84s	-	4.84s	100%	4.84s
Math	70.8%	4.04s	79%	14.86s	92.5%	18.9s	96.2%	18.9s	99%	18.9s
Chart	85%	2.85 s	100%	0.49 s	88.5%	3.34s	92.3%	3.34s	-	
Closure	53%	643.72s	55%	13.93s	73%	657.65s	75.2%	657.65s	85.7%	657.65s

*E = Effectiveness
*CS = Cosine Similarity
*IR = Information Retrieval
*SBFL = Spectrum Based Fault Localization
*T = Title
*C = Combination (Title and Description)
*BR = Bug report

Top 10 Result analysis:

<u>Time</u>

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	✓	✓(C)	✓	× (T,C)	✓	✓	✓
2	✓	✓ (C)	✓	× (T,C)	✓	✓	✓
3	✓	× (C)	✓	✓ (T)	✓	✓	✓
4	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
5	×	✓ (C)	✓	× (T,C)	×	✓	✓
6	✓	✓ (C)	✓	✓ (C)	✓	✓	✓
7	✓	× (C)	✓	✓ (T)	✓	✓	✓
8	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

9	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
10	×	× (C)	××	✓ (C)	✓	✓	✓
11	×	✓ (C)	✓	✓ (T)	✓	✓	✓
12	✓	✓ (T)	✓	× (T)	✓	✓	✓
13	×	× (C)	××	× (T,C)	××	××	×
14	✓	× (C)	✓	× (T,C)	✓	✓	×
15	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
16	✓	× (C)	✓	✓ (T)	✓	✓	✓
17	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
18	✓	× (C)	✓	✓ (T)	✓	✓	✓
19	×	✓ (C)	✓	✓ (T)	✓	✓	✓
20	✓	× (C)	✓	✓ (C)	✓	✓	✓
21	✓	No BR	✓	No BR	✓	✓	No BR
22	×	× (C)	××	✓ (T)	✓	✓	✓
23	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
24	×	× (C)	××	× (T,C)	××	××	×
25	×	× (C)	××	✓ (T)	✓	✓	✓
26	×	× (C)	××	✓ (T)	✓	✓	✓

27	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
Result	**67% (18/27)**	**54% (14/26)**	**77.77% (21/27)**	**73% (19/26)**	**88.89% (24/27)**	**92.59% (25/27)**	**88.46% (23/26)**
Time (sec-onds)	**31. 35s**	**74.73s**	**106.08s**	**1.50s**	**32.85s**	**107.58s**	**76.23s**
	SBFL -2	IR - 1		TS - 4			

Lang

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	✓	✓(C)	✓	✓ (T)	✓	✓	✓
2	✓	No BR	✓	No BR	✓	✓	No BR
3	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
4	✓	✓ (C)	✓	× (T,C)	✓	✓	✓
5	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
6	×	× (C)	××	✓ (T)	✓	✓	✓
7	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
8	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
9	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

10	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
11	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
12	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
13	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
14	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
15	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
16	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
17	✗	✓ (C)	✓	✗ (T,C)	✗	✓	✓
18	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
19	✓	✗ (C)	✓	✓ (C)	✓	✓	✓
20	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
21	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
22	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
23	✗	✗ (C)	✗✗	✓ (T)	✓	✓	✓
24	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
25	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
26	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
27	✗	✓ (C)	✓	✓ (T)	✓	✓	✓

28	✓	× (C)	✓	× (T,C)	✓	✓	×
29	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
30	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
31	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
32	✓	× (C)	✓	✓ (T)	✓	✓	✓
33	✓	✓ (T)	✓	✓ (T)	✓	✓	✓
34	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
35	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
36	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
37	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
38	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
39	✓	× (C)	✓	✓ (T)	✓	✓	✓
40	✓	× (C)	✓	✓ (T)	✓	✓	✓
41	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
42	✓	× (C)	✓	✓ (C)	✓	✓	✓
43	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
44	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
45	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
46	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

47	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
48	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
49	✓	Title too short	✓	✓ (T)	✓	✓	✓
50	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
51	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
52	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
53	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
54	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
55	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
56	✕	✕ (C)	✕✕	✓ (T)	✓	✓	✓
57	✓	✕ (C)	✓	✓ (T)	✓	✓	✓
58	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
59	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
60	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
61	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
62	✓	✓ (C)	✓	✓ (C)	✓	✓	✓
63	✓	✓ (C)	✓	✓ (C)	✓	✓	✓
64	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

65	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
Re-sult	**92% (60/65)**	**84.13% (53/63)**	**95.38% (62/65)**	**94% (61/64)**	**98.46% (64/65)**	**100% (65/65)**	**98.43% (63/64)**
Time (sec-onds)	**0.73 s**	**191.66s**	**192.39s**	**4.11s**	**4.84s**	**196.50s**	**195.77s**
	SBFL - 2	IR - 1		TS - 3			

<u>Mockito</u>

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
2	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
3	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
4	✓	✕ (C)	✓	✓ (T)	✓	✓	✓
5	✕	✕ (C)	✕✕	✓ (T)	✓	✓	✓

6	✓	✓ (C)	✓	× (T,C)	✓	✓	✓
7	✓	× (C)	✓	✓ (T)	✓	✓	✓
8	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
9	×	× (C)	××	✓ (T)	✓	✓	✓
10	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
11	✓	✓ (C)	✓	✓ (T)	✓	✓	✓
12	✓	× (C)	✓	× (T,C)	✓	✓	×
13	×	× (C)	××	× (T,C)	×	×	×
14	×	× (C)	××	× (T,C)	×	×	×
15	×	× (C)	××	× (T,C)	×	×	×
16	×	× (T)	××	✓ (T)	✓	✓	✓
17	✓	× (C)	✓	× (T,C)	✓	✓	×
18	✓	× (C)	✓	✓ (T)	✓	✓	✓
19	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

20	✕	✕ (C)	✕✕	✕ (T,C)	✕	✕	✕
21	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
22	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
23	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
24	✓	Title too short	✓	✕ (T,C)	✓	✓	✕
25	✕	✕ (C)	✕✕	✕ (T,C)	✕	✕	✕
26	✕	✕ (C)	✕✕	✕ (T,C)	✕	✕	✕
27	✕	✕ (C)	✕✕	✕ (T,C)	✕	✕	✕
28	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
29	✓	✕ (C)	✓	✕ (T,C)	✓	✓	✕
30	✓	✓ (C)	✓	✓ (T)	✓	✓	✓

31	✓	× (C)	✓	× (T,C)	✓	✓	×
32	✓	× (C)	✓	× (T,C)	✓	✓	×
33	×	× (C)	××	× (T,C)	×	×	×
34	×	× (C)	××	× (T,C)	×	×	×
35	✓	× (C)	✓	× (T,C)	✓	✓	×
36	×	× (C)	××	× (T,C)	×	×	×
37	✓	× (C)	✓	× (T,C)	✓	✓	×
38	✓	× (C)	✓	× (T,C)	✓	✓	×
Re-sult	**66% (25/38)**	**16.22% (6/37)**	**65.79% (25/38)**	**29% (11/38)**	**73.68% (28/38)**	**73.68% (28/38)**	**31.58% (12/38)**
Time (Sec-onds)	**5.26 s**	**89.42s**	**94.68s**	**2.49s**	**7.75s**	**97.17s**	**91.91s**
	SBFL - 15			TS - 2			

Chart

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	✓	✓	✓	✓ (T)	✓	✓	✓
2	✗	✓	✓	✓ (T)	✓	✓	✓
3	✓	No BR	✓	No BR	✓	✓	No BR
4	✓	No BR	✓	No BR	✓	✓	No BR
5	✓	✓	✓	✓ (T)	✓	✓	✓
6	✓	No BR	✓	No BR	✓	✓	No BR
7	✓	No BR	✓	No BR	✓	✓	No BR
8	✓	No BR	✓	No BR	✓	✓	No BR
9	✓	✓	✓	✓ (T)	✓	✓	✓
10	✓	No BR	✓	No BR	✓	✓	No BR
11	✓	✓	✓	✓ (T)	✓	✓	✓
12	✓	✓	✓	✓ (T)	✓	✓	✓

13	✓	No BR	✓	No BR	✓	✓	No BR
14	✗	No BR	✗✗	No BR	✗	✗✗	No BR
15	✓	No BR	✓	No BR	✓	✓	No BR
16	✓	✗	✓	✓ (C)	✓	✓	✓
17	✓	✓	✓	✓ (T)	✓	✓	✓
18	✓	No BR	✓	No BR	✓	✓	No BR
19	✓	No BR	✓	No BR	✓	✓	No BR
20	✓	No BR	✓	No BR	✓	✓	No BR
21	✓	No BR	✓	No BR	✓	✓	No BR
22	✓	No BR	✓	No BR	✓	✓	No BR
23	✓	No BR	✓	No BR	✓	✓	No BR
24	✓	No BR	✓	No BR	✓	✓	No BR

25	×	No BR	××	No BR	×	××	No BR
26	×	No BR	××	No BR	×	××	No BR
Re-sult	**85% (22/26)**	**87.5% (7/8)**	**88.46% (23/26)**	**100% (8/8)**	**88.46% (23/26)**	**88.46% (23/26)**	**100% (8/8)**
Time	**2.85 s**	**21.75s**	**24.6s**	**0.49s**	**3.34**	**25.09s**	**22.24s**
	SBFL - 15						

Math

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	×	✓	✓	✓ (T)	✓	✓	✓
2	×	✓	✓	✓ (C)	✓	✓	✓
3	✓	✓	✓	✓ (T)	✓	✓	✓
4	✓	✓	✓	✓ (T)	✓	✓	✓
5	✓	✓	✓	✓ (T)	✓	✓	✓
6	✓	×	✓	× (C,T)	✓	✓	×

7	✓	✓	✓	✓ (T)	✓	✓	✓
8	✓	✓	✓	✕ (C,T)	✓	✓	✓
9	✓	✓	✓	✓ (T)	✓	✓	✓
10	✓	✕	✓	✓ (T)	✓	✓	✓
11	✕	✓	✓	✕ (C,T)	✕	✓	✓
12	✕	✕	✕✕	✕ (C,T)	✕	✕✕	✕
13	✕	✕	✕✕	✓ (T)	✓	✓	✓
14	✕	✕	✕✕	✕ (C,T)	✕	✕✕	✕
15	✓	✓	✓	✓ (T)	✓	✓	✓
16	✓	✓	✓	✓ (T)	✓	✓	✓
17	✓	✓	✓	✓ (T)	✓	✓	✓
18	✓	✕	✓	✓ (T)	✓	✓	✓
19	✕	✕	✕✕	✓ (T)	✓	✓	✓
20	✓	✕	✓	✓ (T)	✓	✓	✓
21	✕	✕	✕✕	✓ (T)	✓	✓	✓
22	✓	✕	✓	✕ (C,T)	✓	✓	✕
23	✕	✕	✕✕	✓ (T)	✓	✓	✓
24	✕	✕	✕✕	✓ (T)	✓	✓	✓
25	✓	✕	✓	✓ (T)	✓	✓	✓
26	✓	✓	✓	✓ (T)	✓	✓	✓

27	✓	✓	✓	✓ (T)	✓	✓	✓
28	×	×	××	✓ (C)	✓	✓	✓
29	✓	✓	✓	✓ (T)	✓	✓	✓
30	×	✓	✓	✓ (T)	✓	✓	✓
31	×	×	××	× (C,T)	×	××	×
32	✓	×	✓	✓ (T)	✓	✓	✓
33	✓	×	✓	✓ (T)	✓	✓	✓
34	✓	✓	✓	✓ (T)	✓	✓	✓
35	✓	✓	✓	✓ (C)	✓	✓	✓
36	×	✓	✓	✓ (T)	✓	✓	✓
37	×	✓	✓	✓ (T)	✓	✓	✓
38	✓	×	✓	✓ (T)	✓	✓	✓
39	×	×	××	✓ (T)	✓	✓	✓
40	×	✓	✓	✓ (C)	✓	✓	✓
41	✓	✓	✓	✓ (T)	✓	✓	✓
42	×	✓	✓	✓ (C)	✓	✓	✓
43	✓	×	✓	✓ (T)	✓	✓	✓
44	×	✓	✓	✓ (C)	✓	✓	✓
45	✓	✓	✓	✓ (C)	✓	✓	✓
46	×	✓	✓	× (C,T)	✓	✓	✓

47	✕	✓	✓	✕ (C,T)	✓	✓	✓
48	✕	✕	✕✕	✓ (T)	✓	✓	✓
49	✓	✓	✓	✓ (T)	✓	✓	✓
50	✓	✕	✓	✓ (T)	✓	✓	✓
51	✕	✕	✕✕	✓ (T)	✓	✓	✓
52	✓	✓	✓	✓ (T)	✓	✓	✓
53	✕	✓	✓	✓ (T)	✓	✓	✓
54	✕	✓	✓	✓ (T)	✓	✓	✓
55	✓	✓	✓	✕	✓	✓	✓
56	✓	✓	✓	✓ (T)	✓	✓	✓
57	✓	✓	✓	✕	✓	✓	✓
58	✕	✓	✓	✕	✕	✓	✓
59	✕	✕	✓	✓ (C)	✓	✓	✓
60	✕	✕	✕✕	✓ (T)	✓	✓	✓
61	✕	✕	✕✕	✓ (T)	✓	✓	✓
62	✕	✕	✕✕	✕ (C,T)	✕	✕✕	✕
63	✓	✕	✓	✕ (C,T)	✓	✓	✕
64	✓	✕	✓	✓ (T)	✓	✓	✓
65	✓	✓	✓	✓ (C)	✓	✓	✓
66	✕	✕	✕✕	✓ (T)	✓	✓	✓

67	✗	✗	✗✗	✓ (T)	✓	✓	✓
68	✓	✗	✓	✓ (T)	✓	✓	✓
69	✓	✗	✓	✗ (C,T)	✓	✓	✗
70	✓	✓	✓	✓ (T)	✓	✓	✓
71	✓	✗	✓	✗ (C,T)	✓	✓	✗
72	✓	✓	✓	✓ (T)	✓	✓	✓
73	✓	✓	✓	✓ (T)	✓	✓	✓
74	✗	✗	✗✗	✓ (T)	✓	✓	✓
75	✓	✓	✓	✓ (T)	✓	✓	✓
76	✓	✓	✓	✗ (C,T)	✓	✓	✓
77	✗	✗	✗✗	✓ (T)	✓	✓	✓
78	✗	✓	✓	✓ (T)	✓	✓	✓
79	✗	✗	✗✗	✗ (C,T)	✗	✗✗	✗
80	✓	✗	✓	✗ (C,T)	✓	✓	✗
81	✓	✗	✓	✓ (T)	✓	✓	✓
82	✓	✓	✓	✗	✓	✓	✓
83	✗	✓	✓	✓ (C)	✓	✓	✓
84	✗	✗	✗✗	✗ (C,T)	✗	✗✗	✗
85	✗	✗	✗✗	✗ (C,T)	✗	✗✗	✗
86	✗	✓	✓	✓ (T)	✓	✓	✓

87	×	✓	✓	✓ (T)	✓	✓	✓
88	×	✓	✓	✓ (C)	✓	✓	✓
89	✓	✓	✓	✓ (C)	✓	✓	✓
90	✓	✓	✓	✓ (C)	✓	✓	✓
91	×	✓	✓	✓ (T)	✓	✓	✓
92	✓	×	✓	× (C,T	✓	✓	×
93	✓	✓	✓	× (C,T	✓	✓	✓
94	✓	✓	✓	× (C,T	✓	✓	✓
95	✓	×	✓	✓ (T)	✓	✓	✓
96	✓	✓	✓	✓ (T)	✓	✓	✓
97	✓	✓	✓	✓ (T)	✓	✓	✓
98	✓	×	✓	✓ (T)	✓	✓	✓
99	✓	×	✓	× (C,T)	✓	✓	×
100	✓	✓	✓	✓ (T)	✓	✓	✓
101	✓	✓	✓	✓ (T)	✓	✓	✓
102	✓	×	✓	✓ (T)	✓	✓	✓
103	×	×	××	✓ (T)	✓	✓	✓
104	✓	✓	✓	✓ (T)	✓	✓	✓
105	✓	✓	✓	✓ (T)	✓	✓	✓
106	✓	✓	✓	✓ (T)	✓	✓	✓

Re-sult	**56.60% 60/106**	**55.66% (59/106)**	**78.30% (83/106)**	**75.47% (80/106)**	**91.51% (97/106)**	**93.4% (99/106)**	**85.85% (91/106)**
Time	**4.04s**	**258.89s**	**262.93s**	**14.86s**	**18.90s**	**277.79s**	**273.75s**
	SBFL-8	IR - 4		TS -17			

<u>Closure</u>

Bug	SBFL	IR	SBFL + IR	Text Search	SBFL + Text Search	SBFL+IR +Text Search	IR + Text Search
1	✗	✗	✗	✗	✗	✗	✗
2	✓	✓	✓	✓	✓	✓	✓
3	✗	✓	✓	✗	✗	✓	✓
4	✓	✗	✓	✗	✓	✓	✗
5	✗	✓	✓	✗	✗	✓	✓
6	✓	✓	✓	✗	✓	✓	✓
7	✓	✗	✓	✗	✓	✓	✗
8	✗	✗	✗	✓	✓	✓	✓
9	✓	✓	✓	✗	✓	✓	✓
10	✓	✗	✓	✓	✓	✓	✓
11	✗	✗	✗	✓	✓	✓	✓
12	✗	✗	✗	✗	✗	✗	✗

13	×	×	×	×	×	×	×
14	√	×	√	×	√	√	×
15	×	×	×	×	×	×	×
16	×	√	√	×	×	√	√
17	√	√	√	√	√	√	√
18	×	√	√	√	√	√	√
19	√	×	√	√	√	√	√
20	×	√	√	√	√	√	√
21	√	×	√	×	√	√	×
22	√	×	√	×	√	√	×
23	√	×	√	√	√	√	√
24	√	√	√	√	√	√	√
25	×	√	√	√	√	√	√
26	√	√	√	√	√	√	√
27	√	×	√	×	√	√	×
28	×	×	×	×	×	×	×
29	×	×	×	×	×	×	×
30	×	×	×	×	×	×	×
31	×	√	√	√	√	√	√
32	√	√	√	√	√	√	√

33	✓	×	✓	×	✓	✓	×
34	✓	×	✓	✓	✓	✓	✓
35	✓	✓	✓	✓	✓	✓	✓
36	×	×	×	×	×	×	×
37	✓	×	✓	×	✓	✓	×
38	✓	×	✓	✓	✓	✓	✓
39	✓	×	✓	×	✓	✓	×
40	×	✓	✓	×	×	✓	✓
41	×	×	×	✓	✓	✓	✓
42	✓	×	✓	✓	✓	✓	✓
43	✓	×	✓	✓	✓	✓	✓
44	×	×	×	×	×	×	×
45	×	×	×	×	×	×	×
46	✓	×	✓	×	✓	✓	×
47	×	×	×	✓	✓	✓	✓
48	✓	✓	✓	✓	✓	✓	✓
49	×	×	×	×	×	×	×
50	×	×	×	✓	✓	✓	✓
51	×	×	×	×	×	×	×
52	✓	×	✓	✓	✓	✓	✓

53	×	×	×	×	×	×	×
54	✓	✓	✓	✓	✓	✓	✓
55	×	×	×	×	×	×	×
56	✓	×	✓	✓	✓	✓	✓
57	✓	×	✓	×	✓	✓	×
58	✓	×	✓	×	✓	✓	×
59	×	✓	✓	✓	✓	✓	✓
60	✓	✓	✓	✓	✓	✓	✓
61	×	✓	✓	✓	✓	✓	✓
62	✓	✓	✓	×	✓	✓	✓
63	✓	No BR	✓	No BR	✓	✓	No BR
64	✓	✓	✓	✓	✓	✓	✓
65	✓	×	✓	✓	✓	✓	✓
66	✓	✓	✓	✓	✓	✓	✓
67	×	×	×	×	×	×	×
68	✓	✓	✓	×	✓	✓	✓
69	×	×	×	✓	✓	✓	✓
70	×	×	×	✓	✓	✓	✓
71	✓	×	✓	✓	✓	✓	✓
72	×	×	×	×	×	×	×

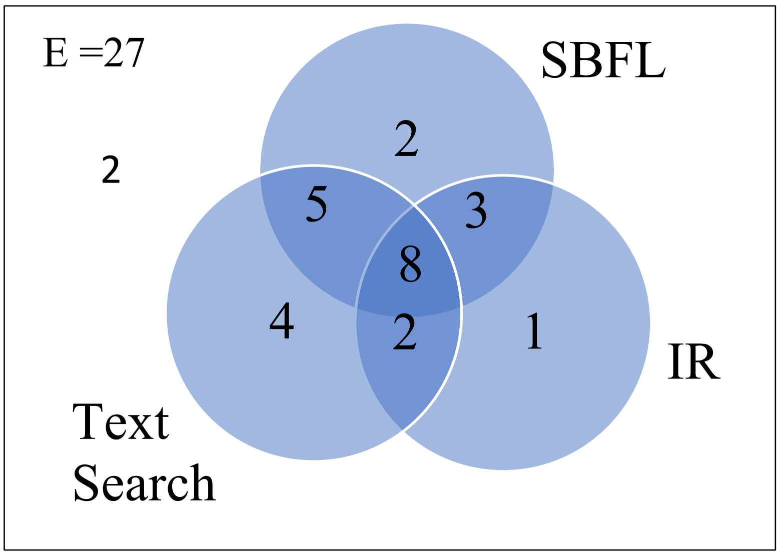
73	✓	✕	✓	✓	✓	✓	✓
74	✓	✕	✓	✕	✓	✓	✕
75	✓	✕	✓	✓	✓	✓	✓
76	✕	✕	✕	✕	✕	✕	✕
77	✓	✓	✓	✓	✓	✓	✓
78	✓	✕	✓	✓	✓	✓	✓
79	✕	✓	✓	✕	✕	✓	✓
80	✓	✕	✓	✓	✓	✓	✓
81	✓	✕	✓	✓	✓	✓	✓
82	✓	✕	✓	✓	✓	✓	✓
83	✓	✓	✓	✓	✓	✓	✓
84	✓	✕	✓	✓	✓	✓	✓
85	✕	✕	✕	✕	✕	✕	✕
86	✓	✕	✓	✓	✓	✓	✓
87	✕	✕	✕	✓	✓	✓	✓
88	✕	✕	✕	✕	✕	✕	✕
89	✕	✓	✓	✓	✓	✓	✓
90	✓	✕	✓	✓	✓	✓	✓
91	✕	✕	✕	✕	✕	✕	✕
92	✕	✓	✓	✓	✓	✓	✓

93	✗	No BR	✗	No BR	✗	✗	No BR
94	✓	✗	✓	✓	✓	✓	✓
95	✗	✗	✗	✓	✓	✓	✓
96	✗	✓	✓	✓	✓	✓	✓
97	✗	✗	✗	✓	✓	✓	✓
98	✗	✗	✗	✗	✗	✗	✗
99	✗	✗	✗	✗	✗	✗	✗
100	✓	✗	✓	✓	✓	✓	✓
101	✗	✓	✓	✓	✓	✓	✓
102	✗	✓	✓	✗	✗	✓	✓
103	✗	✗	✗	✓	✓	✓	✓
104	✓	✗	✓	✗	✓	✓	✗
105	✗	✗	✗	✗	✗	✗	✗
106	✗	✗	✗	✗	✗	✗	✗
107	✗	✓	✓	✓	✓	✓	✓
108	✓	✗	✓	✓	✓	✓	✓
109	✓	✓	✓	✓	✓	✓	✓
110	✓	✓	✓	✓	✓	✓	✓
111	✓	✗	✓	✗	✓	✓	✗
112	✓	✗	✓	✓	✓	✓	✓

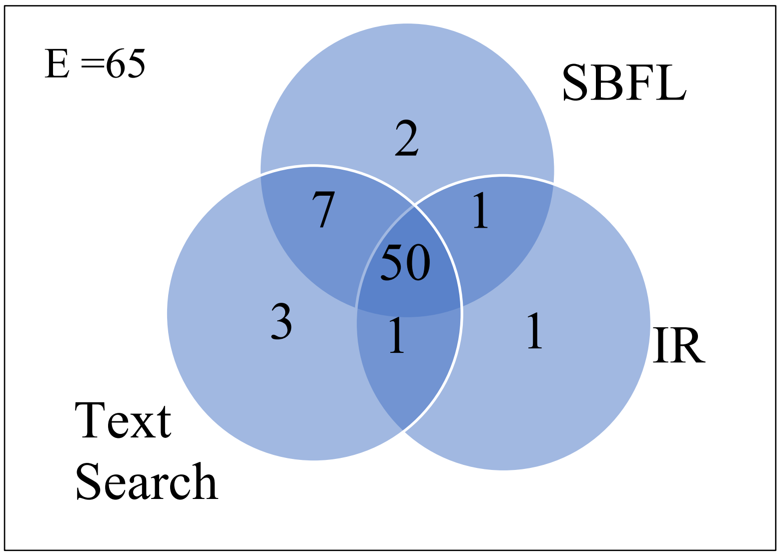
113	✓	✓	✓	✓	✓	✓	✓
114	✓	×	✓	×	✓	✓	×
115	✓	×	✓	✓	✓	✓	✓
116	✓	×	✓	✓	✓	✓	✓
117	✓	✓	✓	×	✓	✓	✓
118	✓	×	✓	×	✓	✓	×
119	×	×	×	✓	✓	✓	✓
120	×	×	×	×	×	×	×
121	×	×	×	×	×	×	×
122	✓	✓	✓	✓	✓	✓	✓
123	✓	×	✓	✓	✓	✓	✓
124	✓	×	✓	×	✓	✓	×
125	✓	✓	✓	✓	✓	✓	✓
126	×	×	×	✓	✓	✓	✓
127	×	×	×	×	×	×	×
128	×	×	×	✓	✓	✓	✓
129	×	×	×	×	×	×	×
130	×	×	×	✓	✓	✓	✓
131	×	×	×	×	×	×	×
132	×	×	×	✓	✓	✓	✓

133	✓	✓	✓	×	✓	✓	✓
Result	**53% (70/133)**	**31.3% (41/131)**	**65.41% (87/133)**	**55% (72/131)**	**73% (97/133)**	**77.4% (103/133)**	**64.12% (84/131)**
Time	**643.72s**	**405.37s**	**1049.09s**	**13.93s**	**657.65s**	**1063.02s**	**419.3s**
	SBFL-19	IR-6		TS-16			

Time bugs



Lang bugs



Mockito bugs

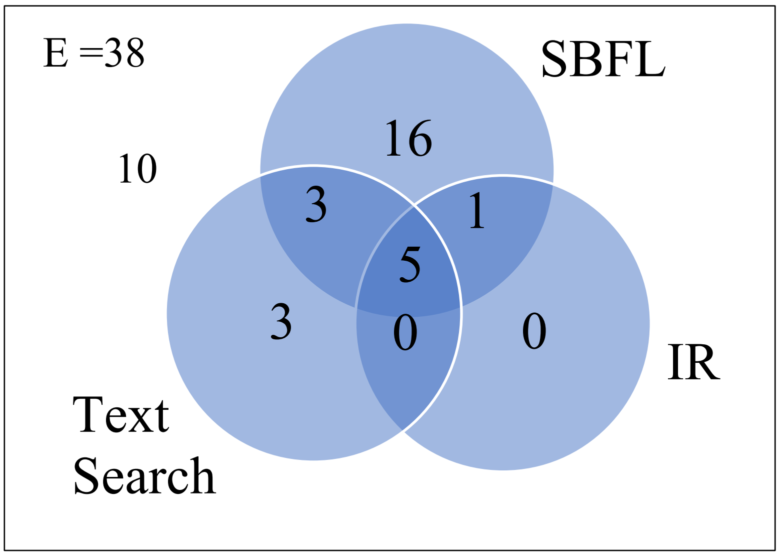
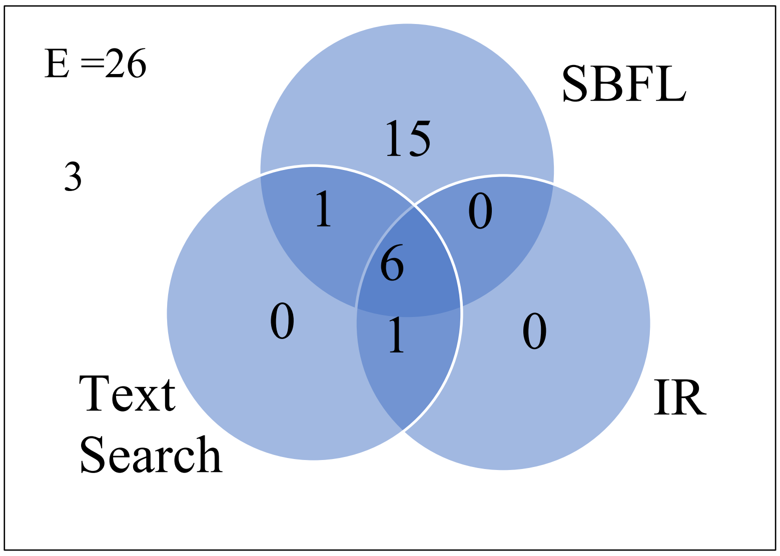
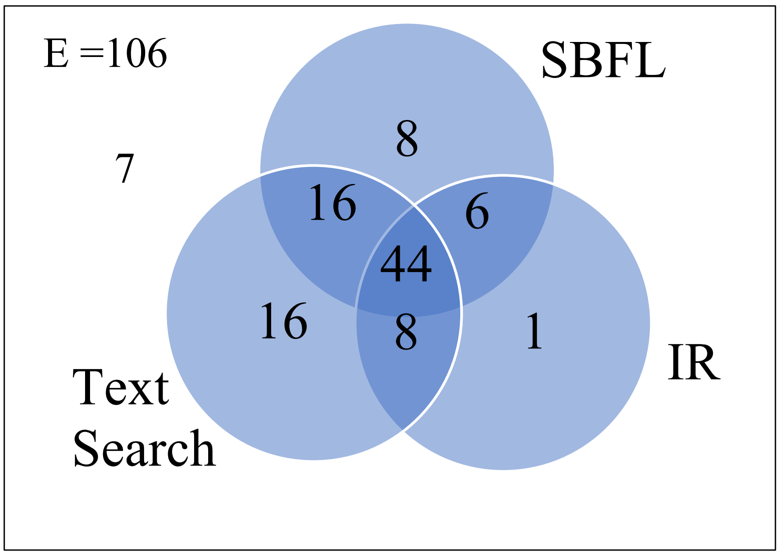


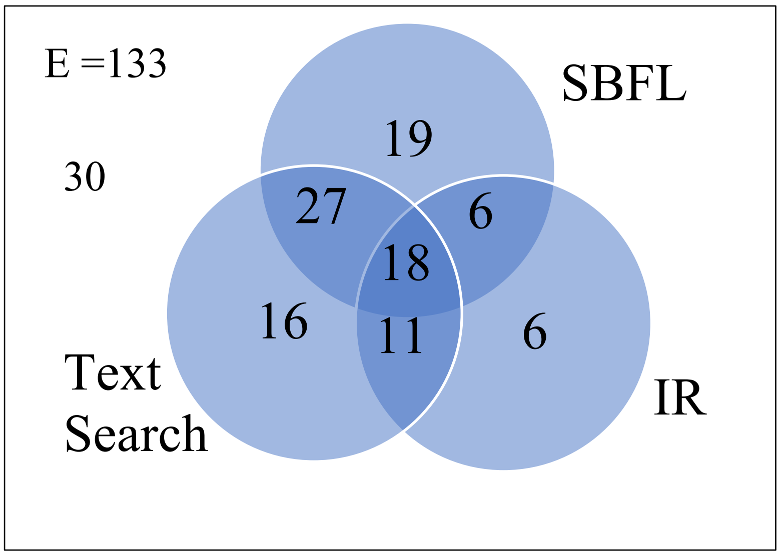
Chart bugs



Math bugs



Closure bugs



Bibliography

[1] S. Desikan and G. Ramesh, "Software testing : principles and practice", Bangalore, India: Dorling Kindersley (India), 2006.

[2] S. Fitzgerald, L. Gary, R. McCauley, L. Murphy, B. Simon, L. Thomas and C. Zander, "Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers," *Computer Science Education,* vol. 18, no. Issue 2: Debugging by Novice Programmers, pp. 93-116, 2008.

[3] H. A. De Souza, D. Mutti, M. L. Chaim and F. Kon, "Contextualizing spectrum-based fault localization," *ScienceDirect,* p. 245–261, 2017.

[4] W. Masri, "Automated Fault Localization: Advances and Challenges," *Advances in Computers,* vol. 99, pp. 103-156, 2015.

[5] M. Weiser, "Program slicing," *In Proceedings of the 5th International Conference on Software Engineering,* p. 439–449, 9-12 March 1981.

[6] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters,* vol. 29, no. 3, pp. 155-163, 1988.

[7] H. Cleve and A. Zeller, "Finding failure causes through automated testing," *Proceedings of the Fourth International Workshop on Automated Debugging,* 28-30th August 2000.

[8] A. J. Jones, J. M. Harrold and T. J. Stasko, "Visualization of test information to assist fault localization," *ICSE,* pp. 467-477, 2002.

[9] M. Renieres and P. S. Reiss, "Fault localization with nearest neighbor queries," *In 18th IEEE International Conference on Automated Software Engineering,* pp. 30-39, 2003.

[10] B. Liblit, M. Naik, X. A. Zheng, A. Aiken and I. M. Jordan, "Scalable statistical bug isolation," *PLDI,* pp. 15-26, 2005.

[11] C. Liu, X. Yan, L. Fei, J. Han and P. S. Midkiff, "Sober: Statistical model-based bug localization.," *SIGSOFT, Softw. Eng. Notes,* vol. 30, no. 5, pp. 286-295, 2005.

[12] E. W. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering,* vol. 19, no. 4, p. 573– 597, 2009.

[13] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," *In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11,* pp. 199-209, 2011.

[14] T. Shu, T. Ye, Z. Ding and J. Xia, "Fault localization based on statement frequency," *Information Sciences,* pp. 43-56, 2016.

[15] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," *In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA,* pp. 273-283, 2017.

[16] J. Xuan and M. Monperrus, "Learning to Combine Multiple Ranking Metrics for Fault Localization," *International Conference on Software Maintenance and Evolution,* pp. 191-200, 2014.

[17] C. Sun and T. Huang, "Using mutation in fault localization," *Computer Science Information Technology,* p. 59–66, 2016.

[18] J. Jiang, R. Wang, Y. Xiong, X. Chen and L. Zhang, "Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study," *International Conference on Automated Software Engineering (ASE),* pp. 502-514, 2019.

[19] L. Zhang, Z. Zhang and J. Zhang, Improving Fault-Localization Accuracy by Referencing Debugging History to Alleviate Structure Bias in Code Suspiciousness, IEEE, 2020.

[20] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: a literature review," *ACM SIGSOFT Software Engineering Notes,* vol. 39, no. 5, p. 1–5, 2014.

[21] D. Zou, J. Liang, Y. Xiong, M. D. Ernst and L. Zhang, "An Empirical Study of Fault Localization Families and Their Combinations," 2019.

[22] D. Poshyvanyk, Y.-G.¨. Gue´he´neuc, A. Marcus, G. Antoniol and V.´. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Transactions on software engineering,* vol. 33, no. 6, pp. 420-432, 2007.

[23] R. Abreu, p. Zoeteweij and V. Gemund, "On the accuracy of spectrum- based fault localization," *In Testing: Academic and Industrial Conference Practice and Research Techniques,* p. 89–98, 2007.

[24] Y. Lei, V. Uren and E. Motta, "Semsearch: A search engine for the semantic web," *In International conference on knowledge engineering and knowledge management ,* p. 238–245., 2006.

[25] I. Letunic, T. Doerks and P. Bork, "Smart 7: recent updates to the protein domain annotation resource," *Nucleic acids research,* p. D302–D305, 2012.

[26] T. Funkhouser, P. Min, M. Kazhdan, J. Chen, A. Halderman, D. Dobkin and D. Jacobs, "A search engine for 3d models.," *ACM Transactions on Graphics (TOG,* vol. 22, no. 1, p. 83–105, 2003.

[27] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique.," *International Conference on Automated Software Engineering (ASE 2005),* p. 273–282, 2005.

[28] E. W. Wong and V. Debroy, "Software fault localization," *Encyclopedia of Software Engineering,* p. 1147–1156, 2010.

[29] M. Srivastav, Y. Singh and D. S. Cauhan, "An Optimized Approach of Fault Distribution for Debugging in Parallel," *Journal of information Processing System,* vol. 6, no. 4, pp. 537-552, December 2010.

[30] N. Shahmehri, M. Kamkar and P. Fritzson, "Semi-automatic bug localization in software maintenance," pp. 30-36, 1990.

[31] E. Y. Shapiro, Algorithmic Program Debugging, Cambridge, MA: MIT Press, 1983.

[32] B. Ledel and S. Herbold, "Broccoli: Bug localization with the help of text search engines," *2021 Association for Computing Machinery,* vol. 1, no. 1, October 2021.

[33] T.-D. B. Le, D. Lo and F. Thung, "Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness," *Empirical Software Engineering,* vol. 20, no. 5, pp. 1237-1274, 2015.

[34] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," *In Proceeding of the 8th working conference on Mining software repositories,* pp. 43-52, 2011.

[35] B. Sisman and A. C. Kak, "Incorporating Version Histories in Information Retrieval Based Bug Localization," 2012.

[36] J. Zhou, H. Zhang and D. Lo, "Where Should the Bugs Be Fixed?," *ICSE,* pp. 14-24, 2012.

[37] R. K. Saha, M. Lease, S. Khurshid and D. E. Perry, "Improving Bug Localization using Structured Information Retrieval," *ASE,* pp. 345-355, 2013.

[38] K. Bhatt, V. Tarey and P. Patel, "Analysis Of Source Lines Of Code(SLOC) Metric", vol. 2, IJETAE, 2012.

[39] H. De Souza, M. Chaim and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges", 2016.

[40] D. Lillis and M. Scanlon, "On the Benefits of Information Retrieval and Information Extraction Techniques Applied to Digital Forensics," 2016.

[41] E. W. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," *IEEE,* 2016.

[42] B. Dit, M. Revelle, M. Gethers and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," 2011.

[43] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," 1990.

[44] M. Krawiec, "Terminological discrepancies in the field of software testing: A case of mistake, error, bug, defect, fault, and failure in the specialist language of IT," pp. 71-81, 2018.

[45] D. McCall and M. Kölling, "Meaningful Categorisation of Novice Programmer Errors," 22-25 Oct 2014.

[46] M. Hristova, A. Misra, M. Rutter and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," *SIGCSE,* 19-23 Febuary 2003.

[47] IEEE, "IEEE Standard for Software and System Test Documentation," 2008.

[48] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schro ̈ter and C. Weiss, "What Makes a Good Bug Report?," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 36, no. 5, pp. 618-643, September/October 2010.

[49] J. Krysa and G. Sedek, "Source Code", 10.7551/mitpress/9780262062749.003.0034, Ed., 2008.

[50] D. Lin, M. Sag and R. Laurie, "Source Code versus Object Code: Patent Implications for the Open Source Community," vol. 18, no. 2, 2002.

[51] Q. Wang, C. Parnin and A. Orso, "Evaluating the usefulness of IR-Based Fault Localization Techniques," *ISSTA'15,* 13-17 July 2015.

[52] C. Tantithamthavorn, S. L. Abebe, A. E. Hassan, A. Ihara and K. Matsumoto, "The Impact of IR-based Classifier Configuration on the Performance and the Effort of Method-Level Bug Localization," April 2018.

[53] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn and N. A. Kraft, "Configuring latent Dirichlet allocation based feature location," *Empir Software Engineering,* vol. 19, pp. 465-500, 2014.

[54] X. Xie, T. Y. Chen and F.-C. Kuo, "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization," vol. 22, no. 4, p. 40, 2013.

[55] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," *in National Computer Conference.,* p. 539, 1986.

[56] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," vol. 10, no. 3, pp. 171-194, 2000.

[57] M. A. Alipour, "Automated Fault Localization Techniques; A survey," 2012.

[58] R. Abreu, P. Zoeteweij and A. J. v. Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing,* vol. 12, pp. 39-46, 2006.

[59] Lucia, D. Lo, L. Jiang and A. Budi, "Comprehensive Evaluation of Association Measures for Fault Localization," *International Conference on Software Maintenance,* 2010.

[60] A. Zeller and H. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 28, no. 2, p. 183–200, February 2002.

[61] H. Cleve and A. Zeller, "Locating Causes of Program Failures," *ICSE,* 15-21 May 2005.

[62] Lucia, D. Lo, L. Jiang, F. Thung and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process,* p. 172–219, 2014.

[63] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *In Proceedings of the 2002 International Conference on Dependable Systems and Networks,* p. 595–604, 2002.

[64] V. Dallmeier, C. Lindig and A. Zeller, "Lightweight defect localization for Java," *In A. P. Black, editor, ECOOP 2005 : 19th European Conference,* vol. 3568 of LNCS, p. 528–550, 25-29 July 2005.

[65] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. V. Hoorn, Lucia and D. Lo, "A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a

Large-Scale Software System", Prague: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017.

[66] G. K. Baah, A. Podgurski and M. J. Harrold, "Causal Inference for Statistical Fault Localization," *Proceedings of the 19th International Symposium on Software Testing and Analysis,* pp. 73-83, 12-16 July 2010.

[67] R. Baeza-Yates and B. Ribeiro-Neto, "Modern Information Retrieval", ACM press, 1999.

[68] M. Motwani and Y. Brun, "Automatically Repairing Programs using both test and bug reports," 2020.

[69] A. Marcus, A. Sergeyev, V. Rajlich and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *11th Working Conference on Reverse Engineering,* p. 214–223, Nov 2004.

[70] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *OOPSLA'04,* 24-28 OCT 2004.

[71] S. K. Lukins, N. A. Kraft and H. L. Etzkorn, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation," *15th Working Conference on Reverse Engineering,* pp. 155-164, 2008.

[72] S. A. Akbar, "Source Code Search For Automatic Bug Localization," 2020.

[73] A. Kuhn, S. Ducasse and T. Grba, "Semantic clustering: Identifying topics in source code," *12th Working Conference on Reverse Engineering,* vol. 49, no. 3, p. 230 – 243, 2007.

[74] B. Sisman and C. Kak, "Assisting code search with automatic query refor- mulation for bug localization," *in Proceedings of the 10th Working Conference on Mining Software Repositories,* p. 309–318, 2013.

[75] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang and H. Me, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," 2014.

[76] L. Moreno, G. Bavota, S. Haiduc, D. M. Penta, R. Oliveto, B. Russo and A. Marcus, "Query-based configuration of text retrieval solutions for software engineering tasks," *in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering,* p. 567–578, 2015.

[77] S. Wang and D. Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," *ICPC,* pp. 53-63, 2-3 June 2014.

[78] K. C. Youm, J. Ahn, J. Kim and E. Lee, "Bug Localization Based on Code Change Histories and Bug Reports," *Asia-Pacfic Software Engineering Conference,* 2015.

[79] L. Moreno, J. J. Treadway, A. Marcus and W. Shen, "On The Use of Stack Traces to Improve Text Retrieval-based Bug localization," *International Conference on Software Maintenance and Evolution,* pp. 151-160, 2014.

[80] S. Davies and R. Marc, "Bug localisation through diverse sources of information," *in 2013 IEEE International Symposium on Software Reliability Engineer- ing Workshops (ISSREW),* p. 126–131, 2013.

[81] M. Wen, R. Wu and S.-C. Cheung, "Locus: Locating Bugs from Software Changes," *ASE'16,* pp. 262-273, 3-7 September 2016.

[82] M. M. Rahman and C. K. Roy, "Improving IR-Based bug localization with context-aware query reformulation," *in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* p. 621–632, 2018.

[83] S. A. Akbar and A. C. Kak, "Scor: Source code retrieval with semantics and order," *Proceedings of the 16th International Conference on Mining Software Repositories, ser. MSR '19,* pp. 1-12, 2019.

[84] Y. Xiao, J. Keung, B. E. Bennin and Q. Mi, "Improving bug localization with word embedding and enhanced T convolutional neural networks," 2018.

[85] A. N. Lam, A. T. Nguyen, H. A. Nguyen and T. N. Nguyen, "Bug Localization with Combination of Deep Learning and Information Retrieval," *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC),* p. 218–229, 2017.

[86] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen and T. N. Nguyen, "Combining Word2Vec with Revised Vector Space Model for Better Code Retrieval," *2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion (ICSE-C),* p. 183–185, May 2017.

[87] D. M. Blei, A. Y. Ng and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research,* pp. 993-1022, 2003.

[88] T.-H. Chen, S. W. Thomas and A. E. Hassan, " A Survey on the Use of Topic Models when Mining Software Repositories," *Empirical Software Engineering,* 2015.

[89] B. Sisman, . S. A. Akbar and A. C. Kak, "Exploiting spatial code proximity and order for improved source code retrieval for bug localization," *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS,* 18 April 2017.

[90] T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *in Advances in Neural Information Processing Systems 26,* p. 3111–3119, 2013.

[91] D. Metzler and W. B. Croft, "A Markov Random Field Model for Term Dependencies," *in Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval,* p. 472–479, 2005.

[92] C. Mills, E. Parra, J. Pantiuchina, G. Bavota and S. Haiduc, "On the relationship between bug reports and queries for text retrieval-based bug localization," *Empirical Software Engineering,* no. 25, pp. 3086-3127, 2020.

[93] H. Zhong and H. Mei, "Learning a graph-based classifier for fault localization," *Science China - Information Sciences,* 2020.

[94] S. K. Lukins, N. A. Kraft and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology,* pp. 972-990, 2010.

[95] T.-D. B. Le and D. Lo, "Will Fault Localization Work For These Failures ? An Automated Approach to Predict Effectiveness of Fault Localization

Tools," *International Conference on Software Maintenance,* pp. 310-319, 2013.

[96] X. Ye , R. Bunescu and C. Liu, "Learning to Rank Relevant Files for Bug Reports using Domain Knowledge," *FSE,* pp. 689-699, 16-21 November 2014.

[97] G. Salton, A. Wong and C. S. Yang, "A vector space model for automatic indexing," vol. 18, no. 11, pp. 613-620, November 1975.

[98] A. Khvorov, R. Vasiliev and G. Chernishev, "S3M: Siamese Stack (Trace) Similarity Measure," 18 March 2021.

[99] S. Kim, T. Zimmermann, J. E. Whitehead and A. Zeller, "Predicting Faults from Cached History," *IEEE,* 2007.

[100] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," *ASE,* 4-9 November 2007.

[101] S. Wang and D. Lo, "AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization," *Journal of Software Evolution and Process,* pp. 921-942, 10 October 2016.

[102] T. B. Le, D. Lo, C. Le Goues and L. Grunske, "A learning-to- rank based fault localization approach using likely invariants", in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM,, 2016, pp. 177-188.

[103] B. Xu, J. Qian, X. Zhang, Z. Wu and L. Chen, "A Brief Survey Of Program Slicing," *ACM SIGSOFT Software Engineering Notes,* vol. 30, no. 2, March 2005.

[104] X. Zhang, N. Gupta and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empir Software Eng,* no. 12, pp. 143-160, 2007.

[105] E. Soremekum, L. Kirschner, M. Bo¨hme and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering (2021),* vol. 26, no. 51, 2021.

[106] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," vol. 11, no. 4, pp. 34-41, 1978.

[107] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *Software Engineering, IEEE Transactions ,* vol. 3, no. 4, pp. 279-290, 1977.

[108] Z. Li, L. Yan, Y. Liu, Z. Zhang and B. Jiang, "MURE:," *IEEE International Conference on Software Quality, Reliability and Security Companion,* pp. 65-63, 2018.

[109] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification & Reliability,* vol. 25, no. 5-7, 2013.

[110] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang and B. Keller, "Evaluating and improving fault localization," 2017.

[111] M. Nica, B. Peischl and F. Wotawa, "Constraint-based configuration of embedded automotive software," 2010.

[112] S. Moon, Y. Kim, M. Kim and S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization," *IEEE,* pp. 153-162, 2014.

[113] S. Hong , T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim and M. Kim, "MUSEUM: Debugging real-world multilingual programs using mutation analysis," *Elsvier,* 2016.

[114] Weimer, Westley, "Patches as Better Bug Reports," pp. https://web.eecs.umich.edu/~weimerw/2015-6610/lectures/weimer-gradpl-genprog2.pdf, 2006.

[115] V. Debroy and E. Wong, "Combining Mutation and Fault Localization for Automated Program Debugging," *Journal of System and Software,* vol. 90, no. 1, 2013.

[116] S. Ali, J. H. Andrews, T. Dhandapani and W. Wang, "Evaluating the Accuracy of Fault Localization Techniques," *IEEE,* pp. 76-87, 2009.

[117] X. Zhang, N. Gupta and R. Gupta, "Locating Faults Through Automated Predicate Switching," *ICSE,* 20-28 May 2006.

[118] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization", Proceedings of the ACM on Programming Languages, vol. 1, 2017, p. 92.

[119] A. Zeller, "Isolating cause-effect chains from computer programs," *Symposium on Foundations of Software Engineering (FSE),* pp. 1-10, 2002.

[120] T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," *Proceedings of the Inter- national Seminar on Software Visualization,* p. 191–204, 2001.

[121] M. Burger and A. Zeller, "Minimizing Reproduction of Software Failures," *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11),* pp. 221-231, 17-21 July 2011.

[122] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," vol. 38, no. 2, 2006.

[123] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan and M. Imran, "Exact String Matching Algorithms: Survey, Issues, and Future Research Directions," vol. 7, 2019.

[124] S. Jonassen, "Efficient Query Processing in Distributed Search Engines," January 2013.

[125] M. N. Kabir, Y. M. Alginahi and O. Tayan, "Efficient Search of a sequence of words in a Large Text File," 2014.

[126] A. Białecki, R. Muir, G. Ingersoll and L. Imagination, "Apache lucene 4," *In SIGIR 2012 workshop on open source information retrieval,* p. 17, 2012.

[127] J. Lee, D. Kim, T. F. Bissyandé, W. Jung and Y. L. Traon, "Bench4BL: Reproducibility Study on the Performance of IR-Based Bug Localization," *ISSTA'18,* 16–21 July 2018.

[128] T. Hoang, R. J. Oentaryo, T.-D. B. Le and D. Lo, "Network-Clustered Multi-Modal Bug Localization," *IEEE Transactions on Software Engineering,* vol. 45, no. 10, October 2019.

[129] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," 2007.

[130] D. Liu, A. Marcus, D. Poshyvanyk and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," *ASE'07,* pp. 234-243, 5-9 November 2007.

[131] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk and A. D. Lucia, "How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms," 2013.

[132] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. D. Penta and D. Poshyvanyk, "Configuring Topic Models for Software Engineering Tasks in TraceLab," 2013.

[133] T. -Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval,* vol. 3, no. 3, p. 225–331, 2009.

[134] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao , S. Han and S.-C. Cheung, "Historical Spectrum based Fault Localization," 2018.

[135] S. Yoo, X. Xiaoyuan, F.-C. Kuo and M. Harman, "Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis," vol. 26, no. 1, June 2017.

[136] Z. Cui, M. Jia, X. Chen, L. Zheng and X. Liu, "Improving Software Fault Localization by Combining Spectrum and Mutation," vol. 8, 2020.

[137] Z. Shi, J. Keung, K. E. Bennin and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," 2017.

[138] A. R. Chen, T. Chen and J. Chen, "How Useful is Code Change Information for Fault Localization in Continuous Integration?," *ASE 2022,* 2022.

[139] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell and A. Wesslen, "Experimentation in Software Engineering", Springer, 2012.

[140] M. Pezze` and M. Young, Software Testing and Analysis: Process, Principles, and Techniques, John Wiley & Sons, Inc, 2007.

[141] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus and B. Baudry, "A Snowballing Literature Study on Test Amplification," 27 August 2019.

[142] J. L. Min , N. Rajabi and A. Rahmani, "Comprehensive study of SIR: Leading SUT repository for software testing", vol. 1869, Annual Conference on Science and Technology (ANCOSET 2020), Journal of Physics: Conference Series, IOP Publishing, 2020, pp. 1-7.

[143] R. Just, J. Darioush and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs", In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014, pp. 437-440.

[144] G. Gay and R. Just, "Defects4J as a Challenge Case for the Search-Based Software Engineering Community," 2020.

[145] U. Rueda, R. Just, J. P. Galeotti and T. E. J. Vos, "Unit Testing Tool Competition — Round Four," *9th International Workshop on Search-Based Software Testing,* 16-17 May 2016 .

[146] M. Martinez, T. Durieux, R. Sommerard, J. Xuan and M. Monperrus, "Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the

Defects4J Dataset," *Empirical Software Engineering,* vol. 22, no. 4, pp. 1936-1964, 2017.

[147] M. Motwani, M. Soto, Y. Brun, R. Just and C. Le Goues, "Quality of Automated Program Repair on Real-World Defects," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* 2020.

[148] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner and R. Ramler, "GRT: An Automated Test Generator Using Orchestrated Program Analysis," *International Conference on Automated Software Engineering (ASE),* pp. 842-847, 2015.

[149] B. Vancsics, A. Szatmári and Á. Beszédes, "Relationship between the Effectiveness of Spectrum-Based Fault Localization and Bug-Fix Types in JavaScript Programs," *SANER,* pp. 308-319, 2020.

[150] C. Oo and H. M. Oo, "Spectrum-Based Bug Localization of Real-World Java Bugs," 2020.

[151] Y. Xiaobo, B. Liu and W. Shihai, "An Analysis on the Negative Effect of Multiple-Faults for Spectrum-Based Fault Localization," 2018.

[152] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," *National Computer Conference,* p. 539, 1986.

[153] V. M. Ngo, T. H. Cao and T. M. Le, "Combining Named Entities with WordNet and Using Query-Oriented Spreading Activation for Semantic Text Search," *2010 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF). ,* 2010.

[154] G. Salton, "Automatic text processing," 1988.

[155] S. L. Abebe, S. Haiduc, P. Tonella and A. Marcus, "Lexicon Bad Smells in Software," *2009 16th Working Conference on Reverse Engineering,* 2009.

[156] B. Vinz and L. Etzkorn, "A synergistic approach to program comprehension," *In: Proc of the 14th IEEE int'l conf on program comprehension,* p. 69–73, 2006.

[157] D. Kim, Y. Tao, S. Kim and A. Zeller, "Where Should We Fix This Bug? A Two-Phase Recommendation Model," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 39, no. 11, November 2013.

[158] F. M. Porter, "An algorithm for suffix stripping," vol. 14, no. 3, pp. 130-137, 1980.

[159] B. Dit, L. Guerrouj, D. Poshyvanyk and G. Antoniol, "Can Better Identifier Splitting Techniques Help Feature Location?," 2011.

[160] S. Wang, D. Lo and J. Lawall, "Compositional Vector Space Models for Improved Bug Localization," pp. 171-180, 2014.

[161] J. Ko and A. Myers, "Designing the whyline: A debugging interface for asking questions about program behavior.," *SIGCHI Conference on human factors in computing systems, CHI,* pp. 151-158, 2004.

[162] I. Zayour and A. Hamdar, "A qualitative study on debugging under an enterprise IDE.," *Information and software technology,* pp. 130-139, 2016.

[163] M. Layman, "Information Needs of Developers for Program Comprehension During Software Maintenance Tasks," *PhD Thesis,* 2009.

[164] J. A. Ko, A. B. Myers, J. M. Coblenz and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks.," *IEEE Transactions on Software Engineering,* vol. 32, no. 12, pp. 971-987, 2006.

[165] J. C. Munson, A. P. Nikora and J. S. Sherif, "Software faults: A quantifiable definition," *Advances in Engineering Software,* 2006.

[166] E. Allen, "Bug Patterns in Java," pp. 7-24, 2002.

[167] A. J. Ko and B. A. Myers, "Aframeworkandmethodologyforstudyingthe causes of software errors in programming systems," *Journal of Visual Languages and Computing,* p. 41–84, 2005.

[168] X.-Y. Zhang, D. Towey, T. Y. Chen, Z. Zheng and K.-Y. Cai, "Using Partition Information to prioritize test cases for Fault Localization," *Annual International Computers, Software, & Applications Conference,* pp. 121-126, 2015.

[169] M. D. Hoffman, D. M. Blei and F. Bach, "Online Learning for Latent Dirichlet Allocation," 2010.

[170] A. Koyuncu, T. F. Bissyande, D. Kim, K. Liu, J. Klein, M. Monperrus and Y. Le Traon, "D&C: A Divide-and-Conquer Approach to IR-based Bug Localization," 2019.

[171] K. Pan, S. Kim and E. J. Whitehead, "Toward an understanding of bug fix patterns," vol. 14, no. 3, pp. 286-315, Jun 2009.

[172] T. Denmat, M. Ducasse´ and O. Ridoux, "Data Mining and Cross-checking of Execution Traces," 2005.

[173] C. Tantithamthavorn, A. Ihara, H. Hata and K. Matsumoto, "Impact Analysis of Granularity Levels on Feature Location Technique," pp. 135-149, 2014.

[174] S. R. Ramya, S. S. Iyengar, L. M. Patnaik, D. Sejal and R. K. Venugopal, "DRDLC: Discovering Relevant Documents Using Latent Dirichlet Allocation and Cosine Similarity," *ICNCC 2018,* pp. 87-91, 14-16 December 2018.

[175] W. Usino, A. S. Prabuwono, K. H. S. Allehaibi, A. Bramantoro, A. Hasniaty and W. Amaldi, "Document Similarity Detection using K-Means and Cosine Distance," *(IJACSA) International Journal of Advanced Computer Science and Applications,* vol. 10, no. 2, pp. 165-170, 2019.

[176] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu and L. Yi, "An empirical investigation of the relationship between spectradifferences and regression faults," *Software Testing Verification and Reliability,* vol. 10, no. 3, pp. 171-194, 2000.

[177] C. Wohlin, M. Höst and K. Henningsson, "Empirical Research Methods in Web and Software Engineering," 2003.

[178] G. Myers, "The Art of Software Testing, Second Edition", New York: Wiley, 2004, p. 234.

[179] E. Elsaka, "Fault Localization Using Hybrid Static/Dynamic Analysis", University of Maryland, US: Elsevier, 2017.