**Empirical Analysis on Fault Localization technique as a hybrid approach**

**Nadhratunnaim Nasarudin**

Department of Computer Science

University of York

September 2021

# Abstract

Debugging is a main activity in the development process. It is used extensively by developers to localize faults, find sources of errors, and enhance quality and performance in general (Elsaka, 2017) . Determining the source of the error or also known as fault localization is the hardest aspect of debugging (Myers, 2004) because it requires analyzing hundreds of lines to identify the error causing subset. This critical area has contributed a significant amount of study in improving Fault Localization technique in assisting developer to allocate fault. Nevertheless, fault using fault localization technique are not always correct, thus by combining two approaches of Fault Localization from different analysis might overcome this issue. While many research has shown that combining more than one approaches can assist developer better in allocation of fault since different aspect from different source are included, combining two approaches are not always means good results. We must agree that there are no “one-size-fits-all solutions”. This is the purpose of my thesis where we only recommend to hybrid it only in specific situation rather than combining it in all situations. We have combined two fault localization technique which are Information Retrieval (IR) Based Technique and Spectrum Based Fault Localization (SBFL) Technique, and we believe that the combination of dynamic and static fault localization analysis can assist developer in better fault localization as it includes different sources. This is because different information sources in analyzing can help developer in allocating fault better than relying on single or limited sources. To evaluate our architecture, we have used a real-world java program, which is Defects4j projects by (Just, et al., 2014) where about 416 bug reports are selected with their appropriate assignees. The experimental results demonstrate the efficiency of our architecture in comparison with both SBFL and IR technique alone in assisting developers to fault localization.

Table of Contents

[Abstract 1](#_Toc83057568)

[1.0 Introduction 2](#_Toc83057569)

[1.1 Objective and Motivation 2](#_Toc83057570)

[1.2 Hypothesis, Problem Statement and Research Methods 5](#_Toc83057571)

[1.3 Summary and structure of the thesis 9](#_Toc83057572)

[2.0 Background and related work 10](#_Toc83057573)

[2.1 Fundamental definitions and terminology: error, fault, bug, test 10](#_Toc83057574)

[2.2 Fault localization 12](#_Toc83057575)

[2.2.1 Spectrum Based Fault Localization (SBFL) 12](#_Toc83057576)

[2.2.2 Dynamic slicing 14](#_Toc83057577)

[2.2.3 Mutation Based Fault Localization (MBFL) 15](#_Toc83057578)

[2.2.4 Predicate switching 16](#_Toc83057579)

[2.2.5 Stack trace 16](#_Toc83057580)

[2.2.6 Information retrieval- based (IR) 17](#_Toc83057581)

[2.1.7 History-based fault localization 18](#_Toc83057582)

[2.3 Hybrid Techniques and related works 20](#_Toc83057583)

[2.3.1 Intersection of SBFL and IR 20](#_Toc83057584)

[3.0 Experiment on individual techniques 22](#_Toc83057585)

[3.1 Experimental set-up 23](#_Toc83057586)

[3.1.1 Datasets 23](#_Toc83057587)

[3.2 SBFL technique experiment 24](#_Toc83057588)

[3.2.1 Program spectra 24](#_Toc83057589)

[3.2.2 Procedure 25](#_Toc83057590)

[3.2.3 Results 26](#_Toc83057591)

[3.3 IR technique experiment 27](#_Toc83057592)

[3.3.1 Bug Report Requirement 27](#_Toc83057593)

[3.3.2 Source code extraction 30](#_Toc83057594)

[3.3.3 Procedure 30](#_Toc83057595)

[3.3.4 Results 31](#_Toc83057596)

[4.0 Hybrid/combination technique experiment 32](#_Toc83057597)

[4.1. Procedure 33](#_Toc83057598)

[4.2 Results and Analysis 35](#_Toc83057599)

[4.3 Limitation 39](#_Toc83057600)

[5.0 Conclusion and Future works 40](#_Toc83057601)

[5.1 Summary 40](#_Toc83057602)

[5.2 Future work and outlook 40](#_Toc83057603)

[Bibliography 41](#_Toc83057604)

# 1.0 Introduction

This chapter is a brief introduction on the motivation of this research, as well as research’s motivation, objectives, hypothesis and problem statement.

## Objective and Motivation

In this subsection, I will briefly explain on my thesis research objective and motivation but before that I will explain first the definition of debugging and its general ideas. Debugging is defined as “determining what runtime faults led to a runtime failure, determining what software errors were responsible for those runtime faults, and modifying the code to prevent the runtime faults from occurring” (fitzgerald, et al., 2008).

There has been much research into novel debugging techniques that support various debugging tasks (Ko, et al., 2004), such as reading the code, executing the code, modifying the code, navigating the code, finding and replacing syntactic terms (e.g. keywords), inspecting code, locating a bug or fault, fixing a fault, tracing how faults propagate through the code (i.e., propagation tracing), understand how programs manipulate the stack (i.e., stack tracing), switching source files, stepping through code line-by- line, asking others, searching the internet, reading help files or forums, rewriting and searching for cues (Zayour, et al., 2016), (Layman, 2009), (Ko, et al., 2006).

Improvements are being made to produce effective and efficient debugging tools since decade ago by (Weiser, 1981), (Korel, et al., 1988), (Cleve, et al., 2000), (Jones, et al., 2002), (Renieres, et al., 2003), (Liblit, et al., 2005) , (Liu, et al., 2005), (Wong, et al., 2009), (Srivastav et al, 2010) , (Parnin, et al., 2011), (Shu, et al., 2016), (Sohn, et al., 2017), however debugging is still considered to be an expensive, time consuming activity. My research is focusing on the debugging task of fault localization and attempts to identify improvements to fault localization tools.

Latest research by (Zou, et al., 2019) on comparison between all fault localization techniques shows that SBFL technique is the most effective technique in fault localization while IR technique are one of the fastest techniques in fault localization. Fault localization is the specific task of identifying the faulty core of a program in which components are responsible for an error. Alternatively, fault localization can be thought of as reporting which portions of a system should be modified in order to correct an error (Groce, 2005).

In this thesis, I have combined two approach which presented in (Poshyvanyk, et al., 2007)and (Abreu, et al., 2007) that applies Information-Retrieval Based Fault Localization (IR) and Spectrum Based Fault Localization (SBFL) respectively. Since both techniques are using different information sources for analysis fault localization, it is interesting to know how much these techniques are correlated to each other. In fault localization research, it seems to be more promising to find new information sources than optimizing existing information sources and recent studies also confirm that integrating more information sources significantly outperforms any standalone techniques in the SBFL family (Zou, et al., 2019).

The main objective of my research is to observe the relationship between SBFL and IR when used for fault localization. In other word, to investigate how the techniques can be combined, and what results can be produced through the combination. The motivation for combining these two techniques is two-fold:

1. When IR topics from bug report provides inconclusive results to localize fault in source code, e.g., when there are less similarity in topic to map between documents, similarity coefficient result might not provide accurate fault location, SBFL may provide more precise information.
2. When there is imprecise information in SBFL analysis results, e.g., when suspiciousness scores are not in top 10 when Top N analysis are being done, these may be supplemented with text similarity analysis from IR.

Second objectives of my research are to extend existing methods by integrate SBFL and IR to support fault localization. I propose a method for using SBFL and IR in combination in particular situation where the root cause is needed to identified first. Third objective is to evaluate the integration of SBFL and IR via experiments to demonstrate the efficacy of using these techniques together for fault localization.

## Hypothesis, Problem Statement and Research Methods

In this subsection, I will briefly explain on my thesis research hypothesis, problem statement and research methods. Fault localization considered to be one of the most expensive (Jones, et al., 2005) , (Wong, et al., 2010) , (Srivastav et al, 2010), (Sun, et al., 2016), tedious and time consuming activities in the debugging activity (Wong, et al., 2010) , (Agarwal, et al., 2014). Example of Fault localization technique are such as Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization, Dynamic Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization (Zou, et al., 2019) .

Now, I will briefly explain about SBFL and IR technique that related to my research, as well as the issues that has raised concern. In SBFL technique, location of the fault will be determined after suspiciousness result are generated. The suspiciousness value for each line of code is calculated based on the pass and failed formula calculation. The idea was, the suspiciousness value with the highest number, holds the higher chances of probability that the line of code contains fault or in other word, the fault location. This contribution of idea has been a state-of-the-art technique for quite some time and massive research has evolved since.

However, the suspiciousness value that generated from SBFL technique for fault localization is not always accurate. Top N has been used to measure the accuracy of the result on many faults localization study (Rao, et al., 2011), (Sisman, et al., 2012), (Zhou, et al., 2012) , (Saha, et al., 2013), and most research and study agree that the accuracy of the technique is consider accurate when the location of fault is in between top 1, top 5 and 10 result generated.

Research made by (Zou, et al., 2019) where they perform an experiment to compare between all standalone technique in fault localization family such as Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization, Dynamic Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization, found that SBFL is the most effective fault localization technique. SBFL manage to localize about 44% and 43% faults of in the top 10 result where Ochiai and DStar have the best performance on all metrics respectively.

In the study made by (Le, et al., 2015) , while running their experiment, 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 and this approach has been used also in other and previous research. To evaluate the SBFL result reliability, this indicator has also been used in my study to the suspiciousness list generated. Now, the problem arises when the 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 and developers tend to resort to a different debugging strategy if they do not find the fault in the first positions of a suspiciousness list (De Souza, et al., 2017). The question would be “How can 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?”.

Traditional approach by manually finding the fault line by line is tedious and time consuming, probably can be done if it involves small programs or simple one, but “how if it involves a huge program or a real-world program and to make it worse the complex one?”. (Xuan et al, 2014) found that to further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL technique.

Here is when my research come into sense where the combination of more than one fault localization technique in enhancing fault localization performance. The hypothesis of my research is the combination of more than one fault localization technique increase the result accuracy and effectiveness in locating fault. Another question arise will be “which technique should I choose to make sure optimum result?”.

Research by Rao et al, on comparative study between IR model for fault localization, they found that sophisticated models like Latent Dirichlet Allocation Model (LDA), Latent Semantic Analysis Model (LSA), and Cluster Based Document Model (CBDM) do not outperform simpler models like Unigram Model (UM) or Vector Space Model (VSM) for IR based bug localization in large software systems, where in general almost 50% of the total fault of relevant file at the rank of 10 are managed to be retrieved (Rao, et al., 2011). In fault localization research, VSM has been shown to outperform many other IR approaches (Wang, et al., 2014) (Rao, et al., 2011) while Ochiai is the best in performance compared to other technique (Zou et al, 2019) in SBFL. This are the reason why this research choose Ochiai technique for SBFL and VSM technique for IR, because of their outstanding performance other than its availability.

IR-based technique assist people in sorting through vast amounts of data or information as quickly and efficiently as possible (Lillis, et al., 2016), where it measures the textual similarity between the bug report and the source files. This approach takes 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 (Wong, et al., 2016).

IR- based fault localization techniques do not require program execution information, such as passed and failed test cases (Zhou et al, 2012) where genuinely I think this type of technique may add value and give advantage when IR and SBFL technique are being combined since additional information are best gathered between different type of source to avoid redundancy and confusion.

This combination of the most effective technique which is SBFL technique and the fastest technique which is IR technique in fault localization might produce a promising result. In my research, the first experiment made was to observe how SBFL and IR complement each other, meaning that the combination is observed from both SBFL and IR technique perspective, and when is the best time to combine it.

In IR, due to the nature of language use, the terms that constitute a topic are often semantically related (Blei, et al., 2003), (Chen, et al., 2015) , 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 accuracy of fault localization in SBFL.

The situation where a developer unable to allocate fault is not new and happens in most fault localization technique since there are no “one-size-fits-all solutions” in addressing fault localization. Same as SBFL, IR technique also face same situation where at some point the topic generated from the result unable to assist developer in allocating fault where the technique unable to map the similarity between documents.

Since different kind source of information might help developer in fault localization, hence SBFL are being applied. Optimization of information from the test cases execution for example, the top 10 of suspiciousness value might contain the location of the faults. Another hypothesis is where IR result does not assist developer in allocating the fault, SBFL might come in handy in proving fault localization.

Below are the research questions that my thesis is trying to answer.

RQ1: What is the relationship between SBFL and IR technique in fault localization?

RQ2: How effective is hybrid technique as compared to running SBFL and IR techniques alone?

To find the answer to research questions above, series of experiment will be done to observe what caused the results for SBFL and IR are not accurate or not listed on the top 10 ranking and another experiment to combine the techniques. This is my research method where will be explained further in chapter 3.

## 1.3 Summary and structure of the thesis

Chapter 2 briefly explains the fundamental definitions of basic term 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 technique will be highlighted.

In Chapter 3, is where the individual experiment will be explained in detail. It will include the requirements of the research approach, research’s technical approach and the tools that has been used to run the experiment. SBFL and IR Technique will be run separately to observe their behavior. Ochiai and VSM algorithm will be used to perform SBFL and IR technique respectively. Experiment result will also be discussed at the end of the chapter.

In Chapter 4, a hybrid or combination of both techniques will be run as the second experiments. Experiment results and analysis will be discussed, including limitation of this research will also be highlighted.

Chapter 5 is about research conclusion and future works that possibly be continued after this research.

# 2.0 Background and related work

This chapter is a background and related work of this research, and this includes the intersection of SBFL and IR technique in fault localization.

## 2.1 Fundamental definitions and terminology: error, fault, bug, test

To avoid confusion in further discussion, I will explain first the fundamental definitions and terminology of error, fault, bug and test. According to (De Souza, et al., 2017), error is a tricky term, which is also sometimes used to refer to as a fault, failure, or mistake. Usually, the term is often used to indicate an incorrect state during the program’s execution. (McCall, et al., 2014) and (Munson, et al., 2006) define error as the nature of the problem in the source code that causes the compilation or execution to fail. In other word, error is caused by a mistake made that led to unsuccessful execution or code compilation (Krawiec, 2018).

(Hristova, et al., 2003) did significant research in understanding common error that occur among novice programmer and divide it to three category which are Syntax errors, Semantic errors and Logic error. *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 interprets certain instructions. *Logic errors* are general errors that often occur which may cause unintended result, and sometimes even without failed execution.

On the other hand, bug can be defined as a program behavior that deviates from its specification (Allen, 2002). (Ko, et al., 2005) define bug as a combination of one or more errors in the code (e.g., software errors), which may produce errors in execution (e.g., runtime faults), which in turn may produce failures in program behavior (e.g., runtime failures). While fault is a manifestation of an error in software (Munson, et al., 2006) and are also called bug or defect (De Souza, et al., 2017). As this is a very clear definition, I’m taking this definition for my thesis writing and to avoid confusion I will use the term “fault” instead of “bug”.

Test or testing is the main source of information for debugging and 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” (IEEE, 2008). Testing is performed to make sure that intended function in the program that are tested are working as expected and at the same time the testing requirements are used as a guarantee that the code is widely tested, and most of the program elements are executed (De Souza, et al., 2017).

## 2.2 Fault localization

Fault localization considered to be one of the most expensive (Jones, et al., 2005), (Wong, et al., 2010), (Srivastav et al, 2010), (Sun, et al., 2016), tedious and time consuming activities in the debugging activity (Wong, et al., 2010) , (Agarwal, et al., 2014). Initially, fault localization was performed manually where developers observe failed test cases and then search the source code for faults. This includes practice such as inserting print statements and breakpoints, checking the stack trace, and verifying failing test cases (De Souza, et al., 2017). Since then, many research has been done to improves fault localization task by created techniques with different approaches.

There are numerous approaches to fault localization in debugging, including Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization, Dynamic Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization (Askarunisa et al, 2012) , (Sun, et al., 2016), (Zou, et al., 2019).

### 2.2.1 Spectrum Based Fault Localization (SBFL)

SBFL has received a lot of attention due to its simplicity and effectiveness (Xie, et al., 2013). SBFL is a technique that use a ranking metric to calculate statements’ suspiciousness (De Souza, et al., 2017). Suspiciousness is when a statements more likely to contain faults or have faulty elements (Alipour et al, 2012). The suspiciousness values are calculated according to the frequency of the statements in passing and failing test cases.

Various Spectrum based bug localization (SBFL) approaches have been proposed in the literature (Jones, et al., 2005), (Abreu et al, 2009), (Lucia, et al., 2010), (Liblit, et al., 2005), (Liu, et al., 2005), (Artzi et al, 2010), (Zeller et al, 2002), (Cleve, et al., 2005), (Lucia, et al., 2014). These approaches analyze a program spectrum which is a record of program elements that are executed in failed and successful executions and generate a ranked list of program elements. Many of these approaches propose various formulas that can be used to compute the suspiciousness of a program element given the number of times it appears in failing and successful executions (Le, et al., 2015).

Jones and Harrold proposed Tarantula that uses a suspiciousness score formula to rank program elements (Jones, et al., 2005). Later, Abreu et al. proposed another suspiciousness formula called Ochiai (Abreu et al, 2009), which outperforms Tarantula. Then, Lucia et al. investigated 40 different association measures and highlighted that some of them including Klosgen and Information Gain are promising for spectrum-based bug localization (Lucia, et al., 2010), (Lucia, et al., 2014). A study conducted by, Xie et al. on theoretical analysis and found that several families of suspiciousness score formulas outperform other families (Xie, et al., 2013). Subsequently, Xie et al. theoretically compared the performance of the formulas produced by genetic programming and identified the best performing ones (Xie, et al., 2013).

Next, Yoo proposed to use genetic programming to generate new suspiciousness score formulas that can perform better than many human designed formulas (Yoo et al, 2012).

Most recently, Xuan and Monperrus combined 25 different suspiciousness score formulas into a composite formula using their proposed algorithm named MULTRIC, which performs its task by making use of an off- the-shelf learning-to-rank algorithm named RankBoost (Xuan, et al., 2014). MULTRIC has been shown to outperform the best performing formulas studied by (Xie, et al., 2013) and the best performing formula constructed by genetic programming (Yoo et al, 2012), (Xie, et al., 2013).

### 2.2.2 Dynamic slicing

Slicing and SBFL have a similar mechanism where they need to trace the execution of test cases, once. The main difference between these two techniques is their efficiency where SBFL needs to trace all the test cases while slicing only needs to trace failed test cases. Dynamic slicing is an enhance technique that developed from static slicing. A slicing criterion is a set of variables at a program location where they might be variables that have unexpected or undesired values. A program slice is a subset of program elements that potentially affect the slicing criterion (Xu et al, 2005). Program slicing was introduced as a debugging tool to reduce a program to a minimal form while still maintaining a given behavior (Weiser, 1981) and static slicing only uses the source code and accounts for all possible executions of the program.

While a dynamic slicing focuses on one execution for a specific input (Korel, et al., 1988). The key difference between dynamic slicing and static slicing is that dynamic slicing only includes executed statements for the specific input, but static slicing includes possibly executed statements for all potential inputs. Since dynamic slices are significantly smaller, they are more suitable and effective for program debugging compared to static slices (Zhang et al, 2007).

### 2.2.3 Mutation Based Fault Localization (MBFL)

MBFL is the slowest technique compared to other fault localization technique. This technique not only needs a lot of time to execute it, it also needs a huge capacity of hardware and software to run it. MBFL technique uses information from mutation analysis (Jia et al, 2011), rather than from regular program execution, as inputs to its ranking metric or risk evaluation formula. While SBFL techniques consider whether a statement is executed or not, MBFL techniques consider whether the execution of a statement affects the result of a test by injecting mutants. A mutant typically changes one expression or statement by replacing one operand or expression with another (Pearson, et al., 2017). If a program statement affects failed tests more frequently and affects passed tests more rarely, it is more suspicious.

(Nica et al, 2010) proposed a technique to reduce bug candidates by using constraint-based debugging. First, statements that do not violate the constraints and that explain the failing test cases are deemed bug candidates. Second, the technique generates mutants for each bug candidate. Mutants that make the failing test cases pass are used to suggest possible faulty sites. (Moon, et al., 2014) proposed a technique that uses mutation to modify faulty and correct statements. The rationale is that, if a mutant inserted in a faulty statement reduces the amount of failing test cases, then the faulty statement is more likely to be faulty.

Conversely, a mutant inserted in a correct statement which generates more failing test cases is less likely to be faulty. (Hong et al, 2015) proposed a similar approach for multilingual programs. Mutation testing is also used to seed faults for experiments, and to suggest fixes for program repair (Weimer, 2006) (Debroy et al, 2014). (Ali, et al., 2009) used mutation testing to generate faults and shown that these faults are similar to real faults.

### 2.2.4 Predicate switching

Predicate switching (Zhang, et al., 2006) is a fault localization technique designed for faults related to control flow. A predicate, or conditional expression, controls the execution of different branches. If a failed test case can be changed to a passed test case by modifying the evaluated result of a predicate, the predicate is called a critical predicate and may be the root cause of the fault.

The technique first traces the execution of the failed test and identifies all instances of branch predicates. Then it repeatedly re-runs the test, forcibly switching the outcome of a different predicate each time. If switching a predicate produces the correct output, the predicate is potentially the cause of the fault and is called a critical predicate.

Predicate switching is similar to MBFL techniques, as they both apply mutations and examine the change of the execution results. However, predicate switching mutates the control flow rather than the program itself. For example, if a conditional expression has been evaluated multiple times during the program execution, predicate switching inverses one evaluation at a time instead of all evaluations. Furthermore, previous work (Pearson, et al., 2017), (Li, et al., 2017) also does not include predicate switching as an MBFL approach.

### 2.2.5 Stack trace

Stack trace fault localization is a technique that needs to execute the test cases at least once before it be able to track the fault. A stack trace is the list of active stack frames during execution of a program. Each stack frame corresponds to a function call that has not yet returned. Stack traces are useful information sources for developers during debugging tasks. When the system crashes, the stack trace indicates the currently active function calls and the point where the crash occurred (Zou, et al., 2019). Stack trace analysis usually works on crash faults and less efficient for other type of faults.

Bug reports often contains stack trace information; however, existing approaches often treat this information as a plain text. (Wong, et al., 2014) propose to use segmentation and stack-trace analysis to improve the performance of bug localization. (Moreno, et al., 2014) proposed a technique named Lobster where it uses the stack trace that recorded in the bug report to compute the similarity with the code elements and the source code of the programs.

(Davies et al, 2013)

### 2.2.6 Information retrieval- based (IR)

IR-based technique assist people in sorting through vast amounts of data or information as quickly and efficiently as possible (Lillis, et al., 2016) where it measures the textual similarity between the bug report and the source files. IR was initially used to index text and search for documents (Baeza-Yates et al, 2008). Recent studies (Saha, et al., 2013), have applied information retrieval techniques to fault localization. This approach takes as input a bug report and source code file, rather than a set of test cases, then generate a list of relevant source code files as output (Wong, et al., 2016). IR- based fault localization techniques do not require program execution information, such as passed and failed test cases (Zhou, et al., 2012).

IR techniques aim to identify the elements of the software system that need to be modified to correct a bug. Such techniques do not attempt to identify every element of the software system that must be fixed. Instead, they aim to identify a starting point from which correction of the bug can be undertaken. Typical IR approaches locate fault files by comparing the bug reports with the source files (Zhong, et al., 2020).

Numerous IR fault localization approaches that employ IR techniques to calculate the similarity between a bug report and a program element (e.g., a method or a source code file) that has been proposed (Rao, et al., 2011), (Lukins, et al., 2010), (Le, et al., 2013), (Sisman, et al., 2012), (Zhou, et al., 2012), (Saha, et al., 2013), (Wang, et al., 2014), (Ye , et al., 2014).

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization (Lukins, et al., 2010). Then, Rao and Kak evaluated the utility of many standard IR techniques for bug localization including Vector Space Model (VSM) and Smoothed Unigram Model (SUM) (Rao, et al., 2011). In the IR community, historically, VSM is proposed very early about four decades ago (Salton, et al., 1975) , followed by many other IR techniques, including SUM and LDA (Blei, et al., 2003), which address the limitations of VSM.

More recently, several approaches which considers information aside from text in bug reports to better locate bugs were proposed. Zhou et al. proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs (Zhou, et al., 2012). Next, Saha et al. proposed an approach that considers the structure of source code files and bug reports and employs structured retrieval for bug localization, and it performs better than BugLocator (Saha, et al., 2013).

### 2.1.7 History-based fault localization

History-based fault localization is a technique that only needs to examine the development history of the codes (Zou, et al., 2019) where are usually used for fault prediction. This technique ranks the elements in a program by their likelihood to be defective. Generally, fault prediction and fault localization are considered as different issues, moreover fault prediction runs before any failure has been discovered (Kim, et al., 2007).

(Dallmeier, et al., 2007) introduce a tools named iBUGS where it collects all past successes and failures of a project to leverages it’s history by automatically extract benchmarks for bug localization tools. These benchmarks are useful for both static and dynamic bug localization tools. Sisman and Kak proposed a version history aware bug 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 bugs (Sisman, et al., 2012).

Wen et al, propose an approach named “Locus” to locate bugs from software changes, and evaluate it on six large open-source projects. (Wen, et al., 2016). While (Wang, et al., 2016) proposes an approach to prioritizing test cases based on historical data where the priorities of test cases are initialized based on requirement priorities and then are calculated dynamically according to historical data in regression testing. (Le, et al., 2016) propose a technique that utilizes the information of bug fixes across projects in the development history to effectively guide and drive a program repair process.

Wen et al, proposed a Historical SBFL where it is a combination of two technique from different family. The approach record the version histories on how bugs are introduced to software projects and this information reflects the root cause of bugs directly where at the same time, the evolution histories of code can also assist to differentiate those suspicious code entities ranked in tie by SBFL (Wen, et al., 2018).

## 2.3 Hybrid Techniques and related works

Over the decade , there are many researches has been done that applying a solution to combine or hybrid techniques in fault localization. Either the combination within same family (e.g. SBFL with SBFL) or between different family (e.g. Stack trace with IR). However, since this research are focusing on SBFL and IR technique, further explanation will be made on this topic.

### 2.3.1 Intersection of SBFL and IR

Both SBFL and IR techniques ultimately generate a ranked list of program elements that likely contain a bug; however, they only consider one source of information either bug reports or program spectra, which is not optimal (Hoang, et al., 2019). Different techniques in SBFL family may contain strongly correlated information on real-world projects. To further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL family (Zou, et al., 2019). In this subsection I will explain the intersection research on SBFL and IR technique, and also the combination of varies fault localization technique with each other.

There is so much research that has been done in recent decade regarding Fault localization and it seems to be more promising to find new information sources than optimizing existing information sources (Zou, et al., 2019). Recent studies by (Sohn, et al., 2017) (Li, et al., 2017) (Le, et al., 2016) also confirm that integrating more information sources significantly outperforms any techniques in the SBFL family and the combined techniques significantly outperform any standalone technique. Le et al, build an oracle that could predict whether the output of a fault localization tool either reliable or not before developer proceed to localize fault using the combination of SBFL and machine learning (Le, et al., 2013).

Another hybrid technique to allocate bugs has been done such as research by (Xuan, et al., 2014) and research by (Le, et al., 2016) where they combine SBFL and learning-based technique. (Sohn, et al., 2017) where they combine SBFL, Learning based, Genetic Programming and Linear rank. Research by (Li, et al., 2017) on combining Learning-based and MBFL, (Wen, et al., 2018) that are using SBFL and History-based to allocate bugs. While (Jiang, et al., 2019) did a systematical empirical study on the combination of SBFL and Statistical Debugging techniques, (Cui, et al., 2020) propose an approach by combining SBFL and MBFL to improves localization accuracy.

Research made by (Le, et al., 2015) on Multi-modal feature location takes as input a feature description and a program spectra, and finds program elements that implement the corresponding feature. There are also several multi-modal feature location techniques proposed in the literature by (Poshyvanyk, et al., 2007), (Liu et al, 2007), (Dit, et al., 2013) with varies combination of fault localization technique.

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution, LSI technique by (Marcus et al, 2003) and a SBFL techniqueby (Jones et al, 2005). They later rank the program elements based on their corresponding weighted sums (Poshyvanyk et al, 2007). Then, Liu et al. proposed an approach named SITIR which filters program elements returned by an IR-based feature location solution , LSI technique by (Marcus et al, 2003) if they are not executed in a failing execution trace (Liu et al, 2007). Later, Dit et al. proposed a LDA technique and Genetic Algorithm (GA) (Dit, et al., 2013).

Since techniques in different families use different information sources, it is interesting to know how much these techniques are correlated to each other and in (Zou, et al., 2019) paper, they found that different techniques in SBFL family may contain strongly correlated information on real-world projects. The research suggest that to further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL family. This research is focusing on one of the critical processes of debugging task which is fault localization, and this research attempts to improve fault localization tools by combining two technique of fault localization from different family.

# 3.0 Experiment on individual techniques

In this sub section, I will explain on the individual experiment for both SBFL and IR technique which are taken separately, also meaning that all result from the experiment of both techniques is collected and analyzed separately.

## 3.1 Experimental set-up

Experiments are usually referred as research-in-the-small, since they are concerned with a limited scope and most often are run in a laboratory setting (Wohlin et al, 2003). The purpose of my experiments is to observe the relationship between these two techniques in fault location. This experiment is to compare and understand “when does the SBFL and IR technique are accurate or helpful” and “when does it is not?”. “Does combining technique helpful when individual technique is not?”.

Since both techniques are using different type of sources to execute, the result produced will also be different in term of the granularity level. In the study made by (Le, et al., 2015), while running their experiment, 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 and this are the rules that are also been applied in our experiment as an indicator either the suspiciousness list generated reliable or not. Hence that was also been applied to our experiment where we assume that both SBFL and IR results are reliable if the location of the fault are among the top 10 most high value ranking. If after ranking top 10 most high rank value unable to allocate fault, then the hybrid of both techniques are needed

### 3.1.1 Datasets

Datasets for all experiment in this thesis will be using Defects4j program for to ensure comparability and uniformity between experiments. Defects4J is 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 (Just, et al., 2014), (Gay, et al., 2020). The defects4j database contains of 357 bugs from 5 programs initially, and since then has grows into 835 bugs from 17 programs (Version 2.0.0). However, for my experiment, I’m using 6 programs with 416 bugs from defects4j due to time constrains.

Defects4j has been used as supporting resources for professionals in both software testing and debugging study (Gay, et al., 2020) where it can be used as a benchmarks to evaluate the effectiveness of automated test generation and corresponding fitness function (Rueda et al, 2016) (Shamsiri et al, 2015), automated program repair (Martinez et al, 2016) (Motwani et al, 2020), and fault localization (Pearson, et al., 2017) research.

## 3.2 SBFL technique experiment

SBFL is the most effective technique in fault localization family where Ochiai have the best performance on all metrics (You et al, 2019) (Zou, et al., 2019). According to Vancsics et al, Ochiai obtained the best results (Vancsics et al, 2020) where it is more effective in locating individual bugs (Oo et al, 2020) and manage to handle multiple faults (Xiaobo et al, 2018).

Unlike other technique in SBFL, Ochiai is more effective for object-oriented programs thus, most SBFL-based repair tools also use Ochiai (Motwani et al, 2020). Since all programs in Defects4j are Java based that are object-oriented, and based on all these finding, other than its availability, Ochiai has been chosen for this individual SBFL experiment.

### 3.2.1 Program spectra

A program spectrum is a measurement of run-time behavior, such as code coverage (Harrold, et al., 2000). Collofello et al, proposed that program spectra be used for fault localization (Collofello, et al., 1986), where comparing program spectra on passed and failed test cases enable ranking of program elements. 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 (Zou, et al., 2019).

Ochiai is an enhanced version of SBFL technique that assigns a suspiciousness to each statement in the program based on the number of passed and failed test cases in a test suite that executed that statement (Abreu et al, 2007). 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 (Jones et al, 2007). Figure 3.1 below is the formula that has been proposed by (Abreu, et al., 2007):

A picture containing text

Description automatically generated

Figure 3.1: Ochiai SBFL formula

### 3.2.2 Procedure

SBFL technique are using probability algorithm between pass and fail execution of the test cases, so the result of this technique is in suspiciousness value are represent as; the highest value is 1 and the lowest value is 0. The line number and file name or method name of the fault also being showed together with the suspiciousness values. The suspiciousness values will be rank according to this value which at the same time it also contains the line of code number that are likely containing faults. Before running SBFL, program spectra or code coverage that includes class and method are taken from the program code. All requirement resources such as such as Time, Mockito, Math, lang, Closure and Chart program that are used for this experiment can be downloaded directly from [[1]](#footnote-1)Defects4j site.

### 3.2.3 Results

Result generated from Time program from Defects4j where SBFL score are inconclusive, mostly because of fault that involving fault sequence pattern (Pan, et al., 2009) or (SQ) which are relates to additions or removals of operations in a field setting sequence or in a sequence of method calls to an object. Usually there will be three or four changes in an operating sequence, and this pattern also includes the case when there is addition or removal of method calls in a short construct body. According to Vancsics et al, faults that require modifications of SQ type are less likely to be successfully localized at very high rank positions by the algorithms in SBFL techniques (Vancsics, et al., 2020) .

|  |  |  |
| --- | --- | --- |
| Bug ID | Bug title and link | Result from SBFL ranking |
| 11 | NPE in DateTimeZoneBuilder | org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1460,1.0  org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1464,1.0  org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1465,1.0  org.joda.time.tz.DateTimeZoneBuilder$Recurrence#800,1.0  org.joda.time.tz.DateTimeZoneBuilder$Recurrence#804,1.0  org.joda.time.PeriodType#674,1.0  org.joda.time.PeriodType#675,1.0  org.joda.time.base.BasePeriod#329,1.0  org.joda.time.Period#704,1.0  org.joda.time.Period#713,1.0 |
| <https://github.com/JodaOrg/joda-time/issues/18> |
| 12 | Check Calendar.ERA in LocalDate.fromCalendarFields | org.joda.time.field.PreciseDurationDateTimeField#129,0.7071067811865475  org.joda.time.LocalDateTime#2016,0.47140452079103173  org.joda.time.LocalDateTime#199,0.4082482904638631  org.joda.time.LocalDateTime#200,0.4082482904638631  org.joda.time.LocalDateTime#612,0.4082482904638631  org.joda.time.LocalDateTime#615,0.4082482904638631  org.joda.time.LocalDate#210,0.4082482904638631  org.joda.time.LocalDate#211,0.4082482904638631  org.joda.time.LocalDate#1805,0.4082482904638631  org.joda.time.LocalDateTime#196,0.35355339059327373 |
| <https://github.com/JodaOrg/joda-time/issues/8> |
| 17 | Bug on withLaterOffsetAtOverlap method | org.joda.time.DateTimeZone#1167,0.5773502691896258  org.joda.time.DateTimeZone#1168,0.5773502691896258  org.joda.time.DateTimeZone#1169,0.5773502691896258  org.joda.time.DateTimeZone#1174,0.5773502691896258  org.joda.time.DateTimeZone#1175,0.5773502691896258  org.joda.time.DateTime#665,0.5773502691896258  org.joda.time.DateTime#666,0.5773502691896258  org.joda.time.DateTime#685,0.5773502691896258  org.joda.time.DateTime#686,0.5773502691896258  org.joda.time.DateTime#331,0.23570226039551587 |
| <https://sourceforge.net/p/joda-time/bugs/141> |
| 25 | Date Time Zone get Off set From Local error during DST transition | org.joda.time.tz.DateTimeZoneBuilder$OfYear#552,0.5222329678670935  org.joda.time.DateTime#160,0.20100756305184242  org.joda.time.DateTime#161,0.20100756305184242  org.joda.time.DateTime#285,0.19245008972987526  org.joda.time.DateTime#287,0.19245008972987526  org.joda.time.tz.CachedDateTimeZone$Info#210,0.16744367165578428  org.joda.time.chrono.BasicGJChronology#190,0.14691063206231753  org.joda.time.tz.CachedDateTimeZone#170,0.14484136487558028  org.joda.time.tz.CachedDateTimeZone#171,0.14484136487558028  org.joda.time.tz.CachedDateTimeZone#172,0.14484136487558028 |
| <https://sourceforge.net/p/joda-time/bugs/90/> |

Table 3.1: Result from Ochiai, one of SBFL Technique

## 3.3 IR technique experiment

IR technique use bug report information to localize fault in source code files, unlike SBFL techniques, IR techniques do not require program coverage information, but their generated ranking is based solely on source code files (Wong, et al., 2016). Before conducting the experiments, source code of the Defects4j program such as Time, Mockito, Math, lang, Closure and Chart are required. Since all the bugs that selected in this research are categorized as “fixed” bug so, the failing program version or the program version before bug repair are needed. This source code is necessary to be used in Information Retrieval technique.

### 3.3.1 Bug Report Requirement

Research made by Biggers et al, 2014 found that the exclusion of comments and literals from the source of source code lowers the accuracy of the end results since bug report might contains natural language context. Later in their research they grouped the sources of source code text into three categories which are Identifier, comments and string literals and combination from the three sources of group are highly recommended to generate more accurate results. Below is the definition of the three source of source code.

**Identifier** 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 (Abebe et al, 2009) (Biggers et al, 2014).

**Comments** generally are used either to map requirements to code or to describe the code (Vinz and Etzkorn, 2006).

**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 are also included into string literals (Biggers et al, 2014).

However, for this research, I only use Identifier and Comments sources since string literals information are not available except the copyright information where typically such data are not indexed, as they add no information about program purpose or behavior hence it has been removed.

The basic source of knowledge for developers to understand a fault is a bug report where, the summary gives a concise overview of the issue (Kim et al, 2013). That is why bug report is important so that developers manage to map the issues raised in bug report to the fault location in the source code.

Insufficient information generated from bug report may obstructing fault localization process. A complete and good bug report is a combination of bug report’s title and its description (Dit et al. 2011). However, Zimmerman et al, 2010 added that a quality bug report are the one that includes an codes attachment or code snippets.

To investigate this further, (Saha, et al., 2013) found that the important of program constructs such as class names and method names to present in bug reports where this might be effectively used to improve fault localization. While (Tantithamthavorn et al, 2018) found that the best results from their experiment has to do with the similar textual characteristics between bug reports and source code. They also conclude that increasing the number of topics has little impact on the performance. However their experiment on the bug report representation in Eclipse program, that contains title only without description are sufficient in achieving best performance while for Mozilla system need both title and description showing that length of the documents does not matter as long as the information in it is good enough to use to localize fault.

One of criteria included in (Tantithamthavorn et al, 2014) research for bug report information is they select only already-fixed issue reports which labelled as “fixed” and exclude issue reports where they could not establish a link to the source code entities. Based on literature evidence, for this research I decided that a bug report that will included in the experiments should have both title and descriptions, or at least contain class name or method name if no description included and best to contains codes attachment or code snippets. However, for bug report that does not have complete information, the documents will also being executed using IR technique to compare and confirm that the incomplete or insufficient information of bug report might affect the similarity of documents results.

### 3.3.2 Source code extraction

IR process started with extract information from each source code, and this includes information as comments and identifiers. Before writing the semantic information to the document collection, source code needs to be preprocessed first and the steps 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”, and typically using the Porter stemmer algorithm (Porter, 1980). Normalizing is 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”. Same goes to the programming language keywords such as “if”, “while” are also removed.

Splitting is done by removing all punctuation, numbers including characters related to the syntax of the programming language such as “&&”, “->”. However, unlike common coding style convention, splitting process for this research does not include splitting the word where we retain the original (unsplit) tokens for example such as “AgeCalculator”, “PeriodType”, “LocalDate”. The main idea behind 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 (Poshyvanyk, et al., 2007) plus. This preprocessing step should be the same with bug report extraction process where all information of the bug report such as title, descriptions, codes attachment or code snippets.

### 3.3.3 Procedure

The result generated will indicate a word that represent the file location or feature location of the fault. The file that contains source code will be modeled and word vector for each file or document will be calculated using cosine similarity, an algorithm of VSM to measure the file or document similarity with the bug report document. The value from vector of words calculated will be rank from highest to less similarity according to this value. Figure 3.2 below shows the cosine similarity algorithm that are being used in this experiment.

Text

Description automatically generated

Figure 3.2: Cosine similarity from VSM technique formula

Usually, IR fault localization are used when there’s a need to understand and modify especially an unfamiliar codebase. When a developer unfamiliar with the large code base of the software system and does not know where to begin. Lacking sufficient documentation on the system and the ability to ask the code’s original authors for help, the only option they sees is to manually search for the code relevant to her task which might consume a lot of time (Dit et al, 2011).

### 3.3.4 Results

Results below are generated from experiment Time datasets with cosine similarity algorithm. This result shows that IR provides inconclusive results, where the location of the faults are not appeared on top 10 list ranking and this situation may be supplemented with SBFL technique

|  |  |  |  |
| --- | --- | --- | --- |
| Bug ID | Bug title and link | Bug report word ranking | Result from querying |
| 3 | addDays(0) changes value of MutableDateTime | ('mdt', 1.9473341662692296)  ('time', 1.7372209362026192)  ('addday chang', 1.6696119815669257)  ('date', 1.620n3977120317812) | org/joda/time/DateMidnight ('chang', 2.2441453522012753)  org/joda/time/DateTime ('chang', 1.9363017283483024)  org/joda/time/LocalDateTime ('chang', 1.3982032619707028)  org/joda/time/LocalDate ('chang', 1.172406268885782) |
| <https://github.com/JodaOrg/joda-time/issues/77> |
| 8 | DateTimeZone.forOffsetHoursMinutes cannot handle negative offset < 1 hour | ('hour', 1.5919433450687475)  ('check', 1.2981079385385732) | org/joda/time/Hours ('hour', 9.314329059167445) |
| org/joda/time/chrono/LimitChronology ('checklimit', 1.602752754346091)  org/joda/time/DateTimeZone ('check', 1.4325035391141914) |
| <https://github.com/JodaOrg/joda-time/issues/42> |
| 20 | Errors creating/parsing dates with specific time zones. | ('datetime', 3.641999116961016)  ('timezon', 2.101361256154582)  ('privat', 2.074149471816985)  ('time', 2.0091871899962355)  ('string', 1.9288023102335368)  ('formatt', 1.7463670100681619)  ('import', 1.641335352771036)  ('fail', 1.434680714253186)  ('system', 1.4137638725168353) | org/joda/time/DateTime ('datetime', 13.727878745958867)  org/joda/time/DateTimeZone ('timezon', 3.4270805577489556) |
| <https://sourceforge.net/p/joda-time/bugs/126> |
| 23 | Incorrect mapping of the MET time zone | ('compat', 1.7269922099144388)  ('zone', 1.473779344125367)  ('state', 1.3651327378367069) | org/joda/time/ReadableDuration ('compat', 1.705714075981504)  org/joda/time/ReadableInstant ('compat', 1.6275891420372903)  org/joda/time/DateTimeZone ('compat', 1.1637095190347406) |
| <https://sourceforge.net/p/joda-time/bugs/112/> |

Table 3.2: Result from Cosine similarity, one of SVM in IR Technique

# 4.0 Hybrid/combination technique experiment

To validate my research architecture, I have embarked a series of experiments and both SBFL and IR technique are run independently in section 3.0 before combining it in section 4.0.

## 4.1. Procedure

Rao et al believe that combining IR fault localization tools with dynamic fault localization could significantly improve the state-of-the-art in fault localization (Rao, et al., 2011). Figure 4.1 below is my technical framework of my experiment. Experiment started with the execution of IR process and SBFL process.

Diagram

Description automatically generated

Figure 4.1: Technical Design of hybrid approach

Each source element's preprocessed data should be saved as a separate document in the document collection. Each document is saved in .txt files and represents one class that contains several methods of the source code, as the experiments in this article are at the method and class level of granularity.

Each document is modeled using topic modelling and from the model generated, word similarity is computed using Cosine Similarity or also known as cosine distance technique. Cosine Similarity measure is computed for unique terms in the documents (Ramya et al, 2018). Cosine Similarity are computed between all possible pairs of document vectors to measures the similarity between two vectors based on the cosine angle between them (Usino et al, 2019). For each document computed, it will produce a score ordering of the documents based on the degree to which they contained the same distribution of topics.

Bug report extraction that mimic the document extractor’s preprocessing steps also use cosine similarity technique to find word similarity in bug report documents. Both results from source code and bug report extraction now are in the same format and this will ease the query process. Query will be manually done to generate ranking result of word similarity to know the location of fault.

However there are some situation that might lead to inability to allocate fault accurately. This is because, IR techniques aim to identify a starting point from which correction of the bug can be undertaken. For example, when there is limited relevancy of information in bug reports though all precaution and attributes to a good quality of bug report has been considered into account and still provides us with inconclusive results, this situation may be supplemented with SBFL technique where the suspiciousness score will be look at.

On the other hand, SBFL technique that use source code coverage are executed using Ochiai technique in order to generate suspiciousness result. The suspiciousness results are calculated according to the frequency of the statements in passing and failing test cases. 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 (Jones et al, 2007). However, in some cases SBFL suspiciousness score are inconclusive and fault location cannot be determine, here is the example of situation when the results from IR technique come into consideration to allocate fault. SBFL Output includes program’s Method name, Statement line number, and Suspiciousness score. The result of suspiciousness score will be sorted, and the highest suspiciousness value are likely is the location of the fault. The highest score for suspiciousness value is 1 and the lowest score is 0.

Since both techniques use different information sources, instead of using one information sources or technique, using both in allocating fault localization can be count as using optimal information that has been provided. This research design that combining SBFL technique and IR technique can be used as a tie breaker when one of the techniques results unable to allocate fault accurately.

The purpose of embarking experiment in this thesis have several purpose, which are:

* to confirm a theory
* to explore a relationship
* to evaluate the accuracy of a model
* to validate a measure

## 4.2 Results and Analysis

This chapter will explain on the results and analysis that has been produced from running the experiment. As explained before, the purpose of the experiment is to find the relationship and compatibility between technique with each other in situation where the condition that has been explain in section 3.0.

Fine-grained data collection or statement level data grain contributes 69.9% better fault localization result than the coarse-grained data collection or method level grain (Jiang, et al., 2019). As we all know, SBFL technique is an execution of test cases at a statement level. Compared to IR technique that are using method level or coarse-grain data, SBFL result contribute to a better fault localization result. Taking this argument as one of important point that has been highlighted in the previous research finding, contributes to the idea in merging SBFL technique with IR technique.

The result from the experiment showed that the two combined techniques, based on different analysis methods and data, are not only expressing different results in trying to identify fault location, but that these results are complementary. This is proved by the fact that the results obtained with the combined techniques betters any one of them used independently. The result shows that the different and independent data from both techniques that combined complement to each other as it produced a new expression of data from both perspectives which is dynamic and static analysis. Table. 4.1 below shows Situation , where IR provides inconclusive results, these may be supplemented with SBFL.

|  |  |  |  |
| --- | --- | --- | --- |
| Bug ID | Bug title and link | Bug report word ranking | Result from querying IR and SBFL |
| 3 | addDays(0) changes value of MutableDateTime | ('mdt', 1.9473341662692296)  ('time', 1.7372209362026192)  ('addday chang', 1.6696119815669257)  ('date', 1.620n3977120317812) | org/joda/time/DateMidnight ('chang', 2.2441453522012753)  org/joda/time/DateTime ('chang', 1.9363017283483024)  org/joda/time/LocalDateTime ('chang', 1.3982032619707028)  org/joda/time/LocalDate ('chang', 1.172406268885782) |
| <https://github.com/JodaOrg/joda-time/issues/77> | **Suspiciousness score**  org.joda.time.MutableDateTime#785, 0.5976143046671968  org.joda.time.MutableDateTime#786, 0.5976143046671968  org.joda.time.DateTimeZone#931, 0.3952847075210474  org.joda.time.chrono.BasicYearDateTimeField#60, 0.31622776601683794  org.joda.time.chrono.ZonedChronology$ZonedDurationField#351, 0.28867513459481287  org.joda.time.chrono.ZonedChronology$ZonedDurationField#352, 0.28867513459481287  org.joda.time.chrono.ZonedChronology$ZonedDurationField#354, 0.28867513459481287  org.joda.time.chrono.ZonedChronology$ZonedDurationField#357, 0.28867513459481287  org.joda.time.MutableDateTime#351, 0.2817180849095055  org.joda.time.MutableDateTime#353, 0.2817180849095055 |
| 8 | DateTimeZone.forOffsetHoursMinutes cannot handle negative offset < 1 hour | ('hour', 1.5919433450687475)  ('check', 1.2981079385385732) | org/joda/time/Hours ('hour', 9.314329059167445) |
| org/joda/time/chrono/LimitChronology ('checklimit', 1.602752754346091)  org/joda/time/DateTimeZone ('check', 1.4325035391141914) |
| <https://github.com/JodaOrg/joda-time/issues/42> | org.joda.time.DateTimeZone#280,1.0  org.joda.time.DateTimeZone#218,0.7071067811865475  org.joda.time.DateTimeZone#274,0.7071067811865475  org.joda.time.DateTimeZone#660,0.2773500981126146  org.joda.time.DateTimeZone#661,0.2773500981126146  org.joda.time.DateTimeZone#383,0.22360679774997896  org.joda.time.DateTimeZone#384,0.22360679774997896  org.joda.time.DateTimeZone#385,0.22360679774997896  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneOffset#2308,0.2182178902359924  org.joda.time.DateTimeZone#216,0.18569533817705186 |
| 20 | Errors creating/parsing dates with specific time zones. | ('datetime', 3.641999116961016)  ('timezon', 2.101361256154582)  ('privat', 2.074149471816985)  ('time', 2.0091871899962355)  ('string', 1.9288023102335368)  ('formatt', 1.7463670100681619)  ('import', 1.641335352771036)  ('fail', 1.434680714253186)  ('system', 1.4137638725168353) | org/joda/time/DateTime ('datetime', 13.727878745958867)  org/joda/time/DateTimeZone ('timezon', 3.4270805577489556) |
| <https://sourceforge.net/p/joda-time/bugs/126> | org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2519,0.35355339059327373  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2520,0.35355339059327373  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2537,0.35355339059327373  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2541,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2542,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2543,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2544,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2545,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder$TimeZoneId#2547,0.3333333333333333  org.joda.time.format.DateTimeFormatterBuilder#1030,0.31622776601683794 |
| 23 | Incorrect mapping of the MET time zone | ('compat', 1.7269922099144388)  ('zone', 1.473779344125367)  ('state', 1.3651327378367069) | org/joda/time/ReadableDuration ('compat', 1.705714075981504)  org/joda/time/ReadableInstant ('compat', 1.6275891420372903)  org/joda/time/DateTimeZone ('compat', 1.1637095190347406) |
| <https://sourceforge.net/p/joda-time/bugs/112/> | org.joda.time.DateTimeZone#314,0.7071067811865475  org.joda.time.DateTimeZone#320,0.3779644730092272  org.joda.time.DateTimeZone#302,0.35355339059327373  org.joda.time.DateTimeZone#305,0.35355339059327373  org.joda.time.DateTimeZone#306,0.35355339059327373  org.joda.time.DateTimeZone#311,0.35355339059327373  org.joda.time.DateTimeZone#312,0.35355339059327373  org.joda.time.DateTimeZone#313,0.35355339059327373  org.joda.time.DateTimeZone#316,0.35355339059327373  org.joda.time.DateTimeZone#319,0.35355339059327373 |

Table 4.1: Situation 1

While table 4.2 below shows situation 2, where SBFL suspiciousness scores are inconclusive, text similarity analysis from IR may provide more precise information.

|  |  |  |  |
| --- | --- | --- | --- |
| Bug ID | Bug title and link | Bug report word ranking | Result from querying IR and SBFL |
| 11 | NPE in DateTimeZoneBuilder | ('datetimezonebuild', 1.8274135304495818)  ('initi', 1.7282142926068145)  ('verbos', 1.6279007926737712)  ('generat', 1.6189370230008977)  ('max', 1.459332273868403) | org/joda/time/src/ZoneInfoCompiler (‘datetimezonebuild', 1.9680649673549422)  org/joda/time/tz/DateTimeZoneBuilder ('datetimezonebuild', 1.7454393189395059) |
| <https://github.com/JodaOrg/joda-time/issues/18> | org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1460,1.0  org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1464,1.0  org.joda.time.tz.DateTimeZoneBuilder$PrecalculatedZone#1465,1.0  org.joda.time.tz.DateTimeZoneBuilder$Recurrence#800,1.0  org.joda.time.tz.DateTimeZoneBuilder$Recurrence#804,1.0  org.joda.time.PeriodType#674,1.0  org.joda.time.PeriodType#675,1.0  org.joda.time.base.BasePeriod#329,1.0  org.joda.time.Period#704,1.0  org.joda.time.Period#713,1.0 |
| 12 | Check Calendar.ERA in LocalDate.fromCalendarFields | ('locald', 1.2295856917294476) | org/joda/time/LocalDateTime ('localdatetim', 11.33617790725211)  org/joda/time/LocalDate ('locald', 9.824642959879595) |
| <https://github.com/JodaOrg/joda-time/issues/8> | org.joda.time.field.PreciseDurationDateTimeField#129,0.7071067811865475  org.joda.time.LocalDateTime#2016,0.47140452079103173  org.joda.time.LocalDateTime#199,0.4082482904638631  org.joda.time.LocalDateTime#200,0.4082482904638631  org.joda.time.LocalDateTime#612,0.4082482904638631  org.joda.time.LocalDateTime#615,0.4082482904638631  org.joda.time.LocalDate#210,0.4082482904638631  org.joda.time.LocalDate#211,0.4082482904638631  org.joda.time.LocalDate#1805,0.4082482904638631  org.joda.time.LocalDateTime#196,0.35355339059327373 |
| 17 | Bug on withLaterOffsetAtOverlap method | ('timezon', 1.5124464189959461)  ('test', 1.4594589019968027)  ('dtch', 1.419821233868467)  ('paulo', 1.1533423758680443) | org/joda/time/DateTimeZone ('datetimezon', 6.042247655987804)  org/joda/time/DateTimeZone ('timezon', 3.4270805577489556) |
| <https://sourceforge.net/p/joda-time/bugs/141> | org.joda.time.DateTimeZone#1167,0.5773502691896258  org.joda.time.DateTimeZone#1168,0.5773502691896258  org.joda.time.DateTimeZone#1169,0.5773502691896258  org.joda.time.DateTimeZone#1174,0.5773502691896258  org.joda.time.DateTimeZone#1175,0.5773502691896258  org.joda.time.DateTime#665,0.5773502691896258  org.joda.time.DateTime#666,0.5773502691896258  org.joda.time.DateTime#685,0.5773502691896258  org.joda.time.DateTime#686,0.5773502691896258  org.joda.time.DateTime#331,0.23570226039551587 |
| 25 | Date Time Zone get Off set From Local error during DST transition | ('datetim', 2.4433119814801203)  ('date time zone', 2.1484630587228293)  ('hemispher', 1.9468334417628317)  ('result', 1.6727109333067252)  ('dst', 1.4836718181574595)  ('arg', 1.4593921363651106)  ('ambigu', 1.2981945779491384)  ('datetim', 2.1447961658933563)  ('datetimezon getoffsetfromloc', 1.7595260729384758)  ('arg', 1.459333019313418)  ('america', 1.3928396462047579) | org/joda/time/DateTime ('datetim', 13.727878745958867)  org/joda/time/DateTimeZone ('datetimezon', 6.042247655987804) |
| <https://sourceforge.net/p/joda-time/bugs/90/> | org.joda.time.tz.DateTimeZoneBuilder$OfYear#552,0.5222329678670935  org.joda.time.DateTime#160,0.20100756305184242  org.joda.time.DateTime#161,0.20100756305184242  org.joda.time.DateTime#285,0.19245008972987526  org.joda.time.DateTime#287,0.19245008972987526  org.joda.time.tz.CachedDateTimeZone$Info#210,0.16744367165578428  org.joda.time.chrono.BasicGJChronology#190,0.14691063206231753  org.joda.time.tz.CachedDateTimeZone#170,0.14484136487558028  org.joda.time.tz.CachedDateTimeZone#171,0.14484136487558028  org.joda.time.tz.CachedDateTimeZone#172,0.14484136487558028 |

Table 4.2: Situation 2

## 4.3 Limitation

Due to time constraints, and the experiment are being done manually, all other program in Defects4j that are not mentioned in this thesis do not have a chance to be executed with this technique.

# 5.0 Conclusion and Future works

## 5.1 Summary

From the experiment taken, the result shows that the different and independent data from both techniques that has been combined does complement to each other as it produced a new expression of data from both perspectives which is dynamic and static analysis of fault localization technique. This also conclude that the proposed hybrid technique is reliable where it does assist the location of fault. However, if the bug report does not meet the quality/requirement that has highlighted in section 3 the result does not assist developer in allocating fault.

The main contributions of this thesis can be identified as the followings:

1) Observe relationship between SBFL and IR technique and its behavior

2) Proposed a novel hybrid technique from different fault localization family, to combine IR based technique with a SBFL technique for fault localization.

3) Applied the proposed hybrid technique in a new case study from defects4j dataset.

The case studies showed that the two combined techniques, based on different analysis methods and data, are not only expressing different judgments in trying to identify features, but that these judgments are complementary. This is proved by the fact that the results obtained with the combined techniques betters any one of them used independently.

## 5.2 Future work and outlook

Finding solutions for area that not being covered for example;

* When IR and SBFL both not helpful what should we do in improving this?
* Try on larger datasets or different program language
* Automate the technique for faster performance

# Bibliography

**Myers G.J.** The Art of Software Testing, Second Edition [Book]. - New York : Wiley, 2004. - p. 234.

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

**Sohn J and Yoo S** Fluccs: using code and change metrics to im- prove fault localization [Book]. - [s.l.] : in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2017. - pp. 273–283.

**Li X and Zhang L** Transforming programs and tests in tandem for fault localization [Book]. - [s.l.] : Proceedings of the ACM on Programming Languages, vol. 1, 2017. - p. 92.

**Le T B [et al.]** A learning-to- rank based fault localization approach using likely invariants [Book]. - [s.l.] : in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM,, 2016. - pp. 177-188.

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

**fitzgerald Sue [et al.]** Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers [Journal] // Computer Science Education. - 2008. - Issue 2: Debugging by Novice Programmers : Vol. 18. - pp. 93-116.

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

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

**Layman M** Information Needs of Developers for Program Comprehension During Software Maintenance Tasks [Journal] // PhD Thesis. - North Carolina State University : [s.n.], 2009.

**Ko J A [et al.]** An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. [Journal] // IEEE Transactions on Software Engineering. - 2006. - 12 : Vol. 32. - pp. 971-987.

**Korel B and Laski J** Dynamic program slicing [Journal] // Information Processing Letters. - 1988. - 3 : Vol. 29. - pp. 155-163.

**Jones A J, Harrold J M and Stasko T J** Visualization of test information to assist fault localization [Journal] // ICSE. - 24th International Conference on Software Engineering : ACM, 2002. - pp. 467-477.

**Liblit B [et al.]** Scalable statistical bug isolation [Journal] // PLDI. - New York, NY, USA : ACM, 2005. - pp. 15-26.

**Liu C [et al.]** Sober: Statistical model- based bug localization. [Journal] // SIGSOFT, Softw. Eng. Notes. - 2005. - 5 : Vol. 30. - pp. 286-295.

**Parnin C and Orso A** Are automated debugging techniques actually helping programmers? [Journal] // In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA ’11. - New York, NY, USA : ACM, 2011. - pp. 199-209.

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

**Shu T [et al.]** Fault localization based on statement frequency [Journal] // Information Sciences. - 2016. - pp. 43-56.

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

**Weiser M** Program slicing [Journal] // In Proceedings of the 5th International Conference on Software Engineering. - San Diego, California, USA : [s.n.], 9-12 March, 1981. - pp. 439–449.

**Wong E W and Qi Y** Bp neural network-based effective fault localization [Journal] // International Journal of Software Engineering and Knowledge Engineering. - 2009. - 4 : Vol. 19. - pp. 573– 597.

**Cleve H and Zeller A** Finding failure causes through automated testing [Journal] // Proceedings of the Fourth International Workshop on Automated Debugging. - Munich, Germany : [s.n.], 28-30th August, 2000.

**Lukins Stacy K, Kraft Nicholas A and Etzkorn H Letha** Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation [Journal] // 15th Working Conference on Reverse Engineering. - [s.l.] : IEEE, 2008. - pp. 155-164.

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

**Zou Daming [et al.]** An Empirical Study of Fault Localization Families and Their Combinations [Journal]. - 2019.

**Jones J A and Harrold M J** Empirical evaluation of the tarantula automatic fault-localization technique. [Journal] // International Conference on Automated Software Engineering (ASE 2005). - Long Beach, CA, USA : IEEE/ACM , 2005. - pp. 273–282.

**Sun C and Huang T** Using mutation in fault localization [Journal] // Computer Science Information Technology. - Institute of Software, Chinese Academy of Sciences, Beijing, China : [s.n.], 2016. - pp. 59–66.

**Agarwal Pragya and Agrawal Arun Prakash** Fault-localization techniques for software systems: a literature review [Journal] // ACM SIGSOFT Software Engineering Notes. - 2014. - 5 : Vol. 39. - pp. 1–5.

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

**Zhou Jian, Zhang Hongyu and Lo David** Where Should the Bugs Be Fixed? [Journal] // ICSE. - Zurich, Switzerland : IEEE, 2012. - pp. 14-24.

**Saha Ripon K [et al.]** Improving Bug Localization using Structured Information Retrieval [Journal] // ASE. - Palo Alto, USA : IEEE, 2013. - pp. 345-355.

**Le Tien-Duy B, Lo David and Thung Ferdian** Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness [Journal] // Empirical Software Engineering. - 2015. - 5 : Vol. 20. - pp. 1237-1274.

**De Souza Higor A [et al.]** Contextualizing spectrum-based fault localization [Journal] // ScienceDirect. - 2017. - pp. 245–261.

**Kim Sunghun [et al.]** Predicting Faults from Cached History [Journal] // IEEE. - [s.l.] : ICSE, 2007.

**Lillis David and Scanlon Mark** On the Benefits of Information Retrieval and Information Extraction Techniques Applied to Digital Forensics [Journal]. - 2016.

**Wong Eric W [et al.]** A Survey on Software Fault Localization [Journal] // IEEE. - 2016.

**Blei David M, Ng Andrew Y and Jordan Michael I** Latent Dirichlet Allocation [Journal] // Journal of Machine Learning Research. - 2003. - pp. 993-1022.

**Chen Tse-Hsun, Thomas Stephan W and Hassan Ahmed E** A Survey on the Use of Topic Models when Mining Software Repositories [Journal] // Empirical Software Engineering. - 2015.

**McCall Davin and Kölling Michael** Meaningful Categorisation of Novice Programmer Errors [Journal]. - Madrid, Spain : IEEE, 22-25 Oct, 2014.

**Munson John C, Nikora Allen P and Sherif Joseph S** Software faults: A quantifiable definition [Journal] // Advances in Engineering Software. - [s.l.] : Elsevier, 2006.

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

**Hristova Maria [et al.]** Identifying and correcting Java programming errors for introductory computer science students [Journal] // SIGCSE. - Nevada, USA : ACM, 19-23 Febuary, 2003.

**Allen Eric** Bug Patterns in Java [Journal]. - 2002. - pp. 7-24.

**Ko Andrew J and Myers Brad A** Aframeworkandmethodologyforstudyingthe causes of software errors in programming systems [Journal] // Journal of Visual Languages and Computing. - Forbes Ave., Pittsburgh, PA 15213, USA : Elsevier, 2005. - pp. 41–84.

**Wong E W and Debroy V** Software fault localization [Journal] // Encyclopedia of Software Engineering. - [s.l.] : Taylor & Francis, 2010. - pp. 1147–1156.

**Lucia [et al.]** Comprehensive Evaluation of Association Measures for Fault Localization [Journal] // International Conference on Software Maintenance. - Timisoara, Romania : IEEE, 2010.

**Lucia [et al.]** Extended comprehensive study of association measures for fault localization [Journal] // Journal of Software: Evolution and Process. - [s.l.] : John Wiley & Sons, Ltd, 2014. - pp. 172–219.

**Cleve Holger and Zeller Andreas** Locating Causes of Program Failures [Journal] // ICSE. - St. Louis, Missouri, USA : ACM, 15-21 May, 2005.

**Xie Xiaoyuan, Chen Tsong Yueh and Kuo Fei-Ching** A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization [Journal]. - [s.l.] : Trans. Softw. Eng. Methodol, 2013. - 4 : Vol. 22. - p. 40.

**Xuan Jifeng and Monperrus Martin** Learning to Combine Multiple Ranking Metrics for Fault Localization [Journal] // International Conference on Software Maintenance and Evolution. - [s.l.] : IEEE, 2014. - pp. 191-200.

**Pearson Spencer [et al.]** Evaluating and improving fault localization [Journal]. - 2017.

**Moon Seokhyeon [et al.]** Ask the Mutants: Mutating Faulty Programs for Fault Localization [Journal] // IEEE. - [s.l.] : International Conference on Software Testing, Verification and Validation, 2014. - pp. 153-162.

**Ali Shaimaa [et al.]** Evaluating the Accuracy of Fault Localization Techniques [Journal] // IEEE. - [s.l.] : International Conference on Automated Software engineering, 2009. - pp. 76-87.

**Wong Chu-Pan [et al.]** Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis [Journal]. - 2014.

**Moreno Laura [et al.]** On The Use of Stack Traces to Improve Text Retrieval-based Bug localization [Journal] // International Conference on Software Maintenance and Evolution. - [s.l.] : IEEE, 2014. - pp. 151-160.

**Zhang Xiangyu, Gupta Neelam and Gupta Rajiv** Locating Faults Through Automated Predicate Switching [Journal] // ICSE. - Shanghai, China : ACM, 20-28 May, 2006.

**Wen Ming, Wu Rongxin and Cheung Shing-Chi** Locus: Locating Bugs from Software Changes [Journal] // ASE'16. - Singapore : ACM, 3-7 September, 2016. - pp. 262-273.

**Zhang Xiao-Yi [et al.]** Using Partition Information to prioritize test cases for Fault Localization [Journal] // Annual International Computers, Software, & Applications Conference. - [s.l.] : IEEE, 2015. - pp. 121-126.

**Zhong Hao and Mei Hong** Learning a graph-based classifier for fault localization [Journal] // Science China - Information Sciences. - [s.l.] : Springer, 2020.

**Lukins Stacy K, Kraft Nicholas A and Etzkorn Letha H** Bug localization using latent Dirichlet allocation [Journal] // Information and Software Technology. - [s.l.] : Elsevier, 2010. - pp. 972-990.

**Le Tien-Duy B and Lo David** Will Fault Localization Work For These Failures ? An Automated Approach to Predict Effectiveness of Fault Localization Tools [Journal] // International Conference on Software Maintenance. - [s.l.] : IEEE, 2013. - pp. 310-319.

**Wang Shaowei and Lo David** Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization [Journal] // ICPC. - Hyderabad, India : ACM, 2-3 June, 2014. - pp. 53-63.

**Ye Xin, Bunescu Razvan and Liu Chang** Learning to Rank Relevant Files for Bug Reports using Domain Knowledge [Journal] // FSE. - Hong Kong, China : ACM, 16-21 November, 2014. - pp. 689-699.

**Salton G, Wong A and Yang C S** A vector space model for automatic indexing [Journal]. - [s.l.] : ACM, November, 1975. - 11 : Vol. 18. - pp. 613-620.

**Sisman Bunyamin and Kak Avinash C** Incorporating Version Histories in Information Retrieval Based Bug Localization [Journal]. - Zurich, Switzerland : IEEE, 2012.

**Dallmeier Valentin and Zimmermann Thomas** Extraction of Bug Localization Benchmarks from History [Journal] // ASE. - Atlanta, Georgia,USA : ACM, 4-9 November, 2007.

**Wang Shaowei and Lo David** AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization [Journal] // Journal of Software Evolution and Process. - [s.l.] : Wiley Online Library, 10 October, 2016. - pp. 921-942.

**Wen Ming [et al.]** Historical Spectrum based Fault Localization [Journal]. - [s.l.] : IEEE, 2018.

**Hoang Thong [et al.]** Network-Clustered Multi-Modal Bug Localization [Journal] // IEEE Transactions on Software Engineering. - [s.l.] : IEEE, October, 2019. - 10 : Vol. 45.

**Jiang Jiajun [et al.]** Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study [Journal] // International Conference on Automated Software Engineering (ASE). - [s.l.] : IEEE/ACM, 2019. - pp. 502-514.

**Cui Zhanqi [et al.]** Improving Software Fault Localization by Combining Spectrum and Mutation [Journal]. - [s.l.] : IEEE, 2020. - Vol. 8.

**Poshyvanyk Denys and Marcus Andrian** Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code [Journal]. - 2007.

**Dit Bogdan [et al.]** Configuring Topic Models for Software Engineering Tasks in TraceLab [Journal]. - 2013.

**IEEE** IEEE Standard for Software and System Test Documentation [Journal]. - [s.l.] : IEEE Standard 829-2008, 2008.

**IEEE** IEEE Standard Glossary of Software Engineering Terminology [Journal]. - [s.l.] : IEEE Standard 610.12, 1990.

**Hoffman Matthew D, Blei David M and Bach Francis** Online Learning for Latent Dirichlet Allocation [Journal]. - 2010.

**Koyuncu Anil [et al.]** D&C: A Divide-and-Conquer Approach to IR-based Bug Localization [Journal]. - [s.l.] : IEEE, 2019.

**Gay Gregory and Just Rene** Defects4J as a Challenge Case for the Search-Based Software Engineering Community [Journal]. - 2020.

**Poshyvanyk Denys [et al.]** Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval [Journal] // IEEE Transactions on software engineering. - [s.l.] : IEEE, 2007. - 6 : Vol. 33. - pp. 420-432.

**Wang Shaowei, Lo David and Lawall Julia** Compositional Vector Space Models for Improved Bug Localization [Journal]. - [s.l.] : IEEE, 2014. - pp. 171-180.

**Harrold M J [et al.]** An empir- ical investigation of the relationship between spectra differences and regression faults [Journal]. - [s.l.] : Software Testing Verification and Reliability, 2000. - 3 : Vol. 10. - pp. 171-194.

**Collofello J S and Cousins L** Towards automatic software fault location through decision-to-decision path analysis [Journal] // National Computer Conference. - [s.l.] : IEEE, 1986. - p. 539.

**Pan K, Kim S and Whitehead E J** Toward an understanding of bug fix patterns [Journal]. - [s.l.] : Empirical Softw. Engg, Jun, 2009. - 3 : Vol. 14. - pp. 286-315.

**Vancsics Béla, Szatmári Attila and Beszédes Árpád** Relationship between the Effectiveness of Spectrum-Based Fault Localization and Bug-Fix Types in JavaScript Programs [Journal] // SANER. - London, ON, Canada : IEEE, 2020. - pp. 308-319.

1. https://github.com/rjust/defects4j [↑](#footnote-ref-1)