**Two better than one: Analysis on static and dynamic Fault Localization as a hybrid approach**

**Nadhratunnaim Nasarudin**

Department of Computer Science

University of York

August 2021

Table of Contents

[Abstract 2](#_Toc79944381)

[1.0 Introduction 3](#_Toc79944382)

[1.1 Motivation 3](#_Toc79944383)

[1.2 Hypothesis and Problem Statement 3](#_Toc79944384)

[1.2 Objectives 3](#_Toc79944385)

[2.0 Background and related work 3](#_Toc79944386)

[2.1 Fundamental definitions and terminology: fault, error, bug, test 3](#_Toc79944387)

[2.2 Fault localization 3](#_Toc79944388)

[2.2.1 Dynamic Approach 3](#_Toc79944389)

[2.2.2 Spectrum Based Fault Localization (SBFL) 3](#_Toc79944390)

[2.2.3 Mutation Based Fault Localization (MBFL) 3](#_Toc79944391)

[2.2.4 Dynamic slicing 3](#_Toc79944392)

[2.2.5 Stack trace 3](#_Toc79944393)

[2.2.6 Predicate switching 3](#_Toc79944394)

[2.2.7 Information retrieval- based (IR) 3](#_Toc79944395)

[2.1.8 History-based fault localization 3](#_Toc79944396)

[2.3 Intersection of SBFL and IR 3](#_Toc79944397)

[3.0 Methods and Approaches 5](#_Toc79944398)

[3.1 Requirements 6](#_Toc79944399)

[3.2 Design of Technical Approach 7](#_Toc79944400)

[3.3 Tool Architecture 9](#_Toc79944401)

[3.4 Example 10](#_Toc79944402)

[4.0 Experiments, Results and Analysis 10](#_Toc79944403)

[4.1 Experimental set-up 10](#_Toc79944404)

[4.2 Analysis and Results 11](#_Toc79944405)

[5.0 Conclusion and Future works 11](#_Toc79944406)

[5.1 Summary 11](#_Toc79944407)

[5.2 Future work and outlook 11](#_Toc79944408)

[6.0 Limitation 11](#_Toc79944409)

[References 11](#_Toc79944410)

# Abstract

Software debugging is a main activity in the software development process. It is used extensively by software developers to localize faults, find sources of errors, and enhance software 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, however combining two approaches are not always means good results. 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 analysis can assist developer in better fault localization as it includes different sources 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, Darioush, & Ernst, 2014) where about 416 bug reports are selected with their appropriate assignees. The experimental results demonstrate the efficiency of our architecture in comparison with the state-of-the-art technique in assisting developers to fault localization.

# 1.0 Introduction

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

## Motivation

## Hypothesis, Problem Statement and Research Methods

RQ1

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

RQ2

Does the quality of the bug report improve the result effectiveness/accuracy?

RQ3

What is the limitation of this hybrid technique?

## 1.3 Objectives

# 2.0 Background and related work

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

## 2.2 Fault localization

### 2.2.1 Dynamic Approach

### 2.2.2 Spectrum Based Fault Localization (SBFL)

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), (Lucia et al, 2014), (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). Recently, Xie et al. conducted a theoretical analysis and found that several families of suspiciousness score formulas outperform other families (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).

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). 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).

### 2.2.3 Mutation Based Fault Localization (MBFL)

### 2.2.4 Dynamic slicing

### 2.2.5 Stack trace

### 2.2.6 Predicate switching

### 2.2.7 Information retrieval- based (IR)

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, 2012), (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 (four decades ago by Salton et al, 1975), followed by many other IR techniques, including SUM and LDA, 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. Sisman and Kak proposed a version his- tory 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). Around the same time, 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).

### 2.1.8 History-based fault localization

## 2.3 Intersection of SBFL and IR

There three general approaches to bug localization that has been studies in the past decade which are dynamic bug localization, static bug localization and mining software repositories (Kim et al, 2013).

* Dynamic bug localization is where an approach that dynamically locating bugs using technique that includes using test suite execution and program’s runtime behavior to discover bugs, data monitoring, breakpoints and so on (Abreu et al, 2009). Example of this dynamic approach including SBFL, dynamic slicing, mutation fault localization, predicate switching and learning to rank technique (Jones et al, 2002), (Liblit et al, 2005), (Cleve et al, 2005), (Zeller et al, 2011).
* Static bug localization is an approach that statically locating bugs using different types of analyses such as bug model, reports and source code without any actual run of the program (Saha et al, 2013) (Hovemeyer et al, 2004) (Lukins, 2009). Example of this technique are stack traces, information retrieval and slicing technique.
* Mining software repositories use several defect predictions approaches to explore the rich information that resides in software repositories (Kim et al, 2013). These techniques typically use metrics such as code churn, past fixes, and change-proneness to predict defects (Ohlsson et al, 1996), (Kim et al, 2008), (Moser et al, 2008), (Hassan et al, 2009), (Lee et al, 2011).

Dynamic bug localization approach is time consuming and expensive to apply since it collects data from the system's execution traces while static bug localization approach involves low computational cost and minimal external dependencies. Static bug localization is easier to apply since they do not require the use of a working subject software framework and can be used at any point of the software creation or maintenance process. Static techniques, unlike most dynamic techniques, do not require one or more test cases to cause the bug (Lukins, 2009).

Nevertheless, the co-change file recommendation techniques require at least one change file be specified at the beginning. In the context of fixing bugs, the developer should at least know one buggy file before adopting any file recommendation tools to identify other co change files. Similarly, most of the state-of-the-art bug localization approaches described above implicitly assume that buggy source files are known in the first place. However, locating the first buggy file is not always an easy task (Kim et al, 2013)

Recently, there are many research that using hybrid technique to allocate bugs has been done such as research by Xuan and Monperrus (2014) and research by Le et al (2016) where they combine SBFL and learning-based technique. Sohn and Yoo (2017) where they combine SBFL, Learning-based, Genetic Programming and Linear rank. Research by Li and Zhang (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 (2020) 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.

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)

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.

There are 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 & Yoo, 2017) (Li & Zhang, 2017) (Le, Lo, Le Goues, & Grunske, 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, 2015) Multi-Modal Feature Location: 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 several multi-modal feature location techniques proposed in the literature (Poshyvanyk et al, 2007), (Liu et al, 2007), (Dit et al, 2013).

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution - LSI by (Marcus et al, 2003) and a spectrum- based solution - Tarantula by (Jones et al, 2005), and rank 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 by (Marcus et al, 2003) if they are not executed in a failing execution trace (Liu et al, 2007).

Later, Dit et al. used HITS, a popular algorithm that ranks the importance of nodes in a graph, to filter program elements returned by SITIR (Dit et al, 2013). Several variants are described in their paper and the best performing ones are IRLSIDynbinWMHITS(h,bin)bottom and IRLSIDynbinWMHITS(h,freq)bottom. I refer to these two as DITA and DITB, respectively. They have showed that these variants outperform SITIR, though they have never been compared with PROMESIR.

Topic Modelling are very useful for document clustering, organizing large blocks of textual data, information retrieval from unstructured text and feature selection. It is a tool designed to automatically extract topics from a collection of text documents (Anthes, 2010), (Blei and Laffer-ty, 2009), (Steyvers and Griffiths, 2007). Initially, topic models were developed as a way of indexing, searching, clustering, and structuring vast collection of unstructured and unlabeled documents automatically. Within the topic modeling structure, documents can be represented by the topics within them and then the whole collection can be indexed and ordered in terms of this discovered semantic structure (Chen et al, 2015).

Recent research on the implementation of information system subject models offers initial proof that topics in software systems are correlated with the with the defect-proneness of source files (Chen et al, 2016). Research by (Andrzejewski et al, 2007) proposed a latent topic model to locate predicate bugs based on cooperative bug isolation framework. They performed statistical debugging with the use of Delta LDA, a variant of LDA. Statistical debugging is the task of identifying a problematic piece of code, given a log of the execution of the code. Delta LDA is able to model two types of topics: usage topics and bug topics. Bug topics are those topics that are only found in the logs of failed executions. Hence, the authors were able to identify the pieces of code that likely caused the bugs

While (Song and Lu, 2014) apply the latent topic model to perform statistical debugging for real-world performance issues. Their study showed that end users often use comparison-based methods to observe and report performance problems, making statistical debugging a promising choice for performance diagnosis. The investigation of different design points of statistical debugging shows that branch predicates, with the help of two types of statistical models, are especially helpful for performance diagnosis. It points out useful failure predictors for 19 out of 20 real-world performance problems. Furthermore, the investigation shows that statistical debugging can also work for production-run performance diagnosis with sampling support, incurring less than 10% overhead in the evaluation.

Wang et al, (2018) proposed a localization technique for bug patterns based on the topic model for bugs in complex program loops. In order to compare the feasibility of the suggested solution to Dstar, they performed experiments involving 28 program variants contained bugs in the Siemens program suite. First, they collect program execution traces executed by instrument program based on test suites. Then, they preprocess the collecting data, learning bug topic, and ranking those bug patterns. Finally, they evaluated the effectiveness of their approach comparing with Dstar. The results revealed that their approach can achieve high accuracy for the bugs in program loops.

# 3.0 Technical and Approaches

This chapter will explain about the requirements of my research approach, the design of my research’s technical approach, research’s tools and a brief illustration of how the approach works on a small example. This chapter consist of four subsection which are 3.1. Requirement, 3.2. Design of Technical Approaches, 3.3. Tools Architecture and 3.4. Example of experiments.

## 3.1 Requirements

In this sub section, I will explain on the requirements for my approach in combining both SBFL and IR technique. To validate my research architecture, I have embarked a series of experiments and it involves a machine with M1 chip (2020) with 8 GB memory and 256GB storage capacity. Both SBFL and IR technique are run independently, meaning that all result from the experiment of both techniques is collected and analyzed separately.

Since both techniques are using different type of sources to execute, the result produced will also be different in term of the granularity level. For IR Technique, the result generated will indicate a word that represent the file location or feature location of the fault (e.g., File level or method level). The file that contains source code will be model using topic modeling and word vector for each file or document will be calculated using cosine similarity to measure 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.

While for SBFL Technique, since this 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.

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 technique are needed.

Usually, people use feature location 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).

## 3.2 Design of Technical Approach

This sub section I will explain on my technical approach in running my experiment. Figure 3.2 below is my technical framework of my experiment. Experiment started with the execution of IR process and SBFL process. 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 steps 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.

Diagram

Description automatically generated

Figure 3.2: Technical Design

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 measures the similarity between two vectors based on the cosine angle between them (Usino et al, 2019).

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.

## 3.3 Tool Architecture

This sub section I will explain about the tools I produced to implement that architecture in execution of technical design in Figure 3.2.

*Defects4J*

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.

*Python*

Python programming are used to run these experiments from extracting the text files, running the topic model, running the SBFL(ochiai) program, until ranking and compare it. A remover function written in python are used to remove punctuation and numbers. While a simple Lexical Analyzer program written also in python are used to separates lexemes by white space and special characters. Python3 is the latest version of Python programming language.

*Information Retrieval (IR) process*

IR technique use bug report information to localize fault, 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.

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 ad 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.

*Spectrum Based Fault Localization (SBFL)*

SBFL is the most effective technique in fault localization family where Ochiai and DStar have the best performance on all metrics (You et al, 2019). In contrast with IR technique, SBFL technique require the source code coverage (matrix and spectra) of the test case to ensure that the suspiciousness score to be generated. SBFL is the most effective fault localization family and Ochiai is the best in performance compared to other technique (Zou et al, 2019). Hence, I’m using Ochiai for my experiment.

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).

Unlike other technique in SBFL, Ochiai is more effective for object-oriented programs thus, most SBFL-based repair tools use Ochiai(Motwani et al, 2020). Since all programs in Defects4j are Java based that are object-oriented, so I decided to use Ochiai for our experiments.

The code coverage includes class and method that taken from the program code. All requirement resources can be downloaded from [[1]](#footnote-1)Defects4j site. These techniques produce a ranked list of program elements based on the suspiciousness score.

*Topic Modelling*

Topic Modelling are very useful for document clustering, organizing large blocks of textual data, information retrieval from unstructured text and feature selection. It is a tool designed to automatically extract topics from a collection of text documents (Anthes, 2010), (Blei and Lafferty, 2009), (Steyvers and Griffiths, 2007).

Initially, topic models were developed as a way of indexing, searching, clustering, and structuring vast collection of unstructured and unlabeled documents automatically. Within the topic modeling structure, documents can be represented by the topics within them and then the whole collection can be indexed and ordered in terms of this discovered semantic structure (Chen et al, 2015).

Latent Dirichlet Allocation (LDA) appears to be at present one of the commonly used topic models, developed by (Blei et al in 2003). However, only if the representation of texts that it yields enables algorithmic similarity computations in finding document similarity, such as Cosine Similarity or Vector Space Model (VSM) (Zhong et al, 2020).

After applying topic modelling to the documents, Cosine similarity are computed between all possible pairs of document vectors to find the similarity between them. 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.

## 3.4 Example of experiment

# 4.0 Experiment, Results and Analysis

This chapter will explain on the experiment, 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 with condition that has been explain in section 3.0.

## 4.1 Experimental set-up

## 4.2 Results and Analysis

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

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

3)

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

# 6.0 Limitation

# Bibliography

Myers, G. (2004). *The Art of Software Testing, Second Edition.* New York: Wiley.

Elsaka, E. (2017). *Fault Localization Using Hybrid Static/Dynamic Analysis.* University of Maryland, US: Elsevier.

Sohn, J., & Yoo, S. (2017). *Fluccs: using code and change metrics to im- prove fault localization.* in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM.

Li, X., & Zhang, L. (2017). *Transforming programs and tests in tandem for fault localization.* Proceedings of the ACM on Programming Languages, vol. 1.

Le, T. B., Lo, D., Le Goues, C., & Grunske, L. (2016). *A learning-to- rank based fault localization approach using likely invariants.* in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM,.

Just, R., Darioush, J., & Ernst, M. D. (2014). *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).

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