# **Table of content thesis**:

# Abstract

Fault localization

This research is focusing on one of the critical process of debugging task which is fault localization. This research attempts to improve fault localization tools by combining two technique of fault localization.

Our plans include attempting to improve fault localization by providing a database to record historical data that is not currently recorded by Tarantula, and as well as current data. Those data include date, time, code line number, and the suspiciousness (a prediction of the likelihood that the code has a bug) of code line. Our hypothesis is that visualizing the changes of data may help in identifying patterns of fault behavior, as well as to assist developers in locating faults more effectively and efficiently.

In this study, we have combined two fault localization technique which are Information Retrieval (IR) Based Technique and Spectrum Based Fault Localization (SBFL) Technique.

Since techniques in different families use different information sources, it is interesting to know how much these techniques are correlated to each other (Zou et al, 2019).

To evaluate our architecture, we have use a real world java program, which is Defects4j projects (Just et al,204). more than “” bug reports with their appropriate assignees. The experimental results demonstrate the efficiency of our architecture in comparison with the state-of-the-art methods in assisting developers to 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 [25], [26], [27] also confirm that integrating more information sources significantly outperforms any techniques in the SBFL family and the combined techniques significantly outperform any standalone technique.

# In (Zou et al, 2019) paper, they found that 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.

# 1.0 Introduction

## Motivation

## 1.2 Hypothesis and Problem Statement

## 1.2 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)

#### 2.2.2.1 Tarantula

#### 2.2.2.2 Ochiai

### 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)

Information retrieval based static 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. Many bug fixes do not relate directly to the bug itself, but rather are the result of the impact of the bug on the other elements of the software system. These additional fixes are considered to be a concern of impact analysis.

#### 2.2.7.1 Latent Dirichlet Allocation (LDA)

#### 2.2.7.2 Latent Semantic Analysis (LSA) or Latent Semantic Indexing (LSI)

### 2.1.8 History-based fault localization

## 2.3 Intersection of SBFL and IR

spend most of its time on SBFL and LDA, and probably less on the other parts.

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 prediction 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 techniques ultimately generate a ranked list of program elements that likely contain a bug, 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)

# 3.0 Methods and Approaches

This chapter will explain about the requirements of my research approach, the design of my research approach 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. Research design, 3.3. Tools Infrastructure and 3.4. Example of experiments.

## 3.1 Requirements

In this sub section, I will explain on the requirements for my approach before running the experimentation phase.

*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) process*

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). 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 (Motwani et al, 2020).

## 3.2 Design of Technical Approach

This sub section I will explain on my Research Design. Figure 3.2 below is my Research 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.

Diagram

Description automatically generated

Figure 3.2: Research Framework

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.

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

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). Ranking and text preproceesing

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. Expanded the datasets of Defects4j is also other contributions made by (Almulla et al, 2017), (Gay et al, 2016) and (Colanzi et al, 2018).

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 programs written also in python are used to separates lexemes by white space and special characters. Python3 is the latest version of Python programming language.

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. 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).

Ochiai is more effective for object-oriented programs. Thus, most SBFL-based repair tools use Ochiai, and so does our study (Motwani et al, 2020).

SBFL is the most effective fault localization family. Ochiai and DStar have the best performance on all metrics(You et al, 2019).

## 3.4 Example

Section 3.2 is about your Technical Approach. So call it Section 3.2 Technical Approach (or Design of Technical Approach)

Section 3.2 is about the Design of your technical approach

Your Research Method could be something to talk about in the Introduction chapter.

E.g., your hypothesis, the fact that this is an empirical thesis, that you test your hypothesis by experiments.

3.3 Integrated technique

Section 3.3 - I think there’s more you can say.

Section 3.3 - talk about Ochiai, and about how you integrate the different components (even if you integrate them only loosely)

I would call Section 3.3 Tool Architecture

Section 3.3 is about the tools you produced to implement that architecture.

# 4.0 Experiments, Results and Analysis

Chapter 4 - about the purpose of the experiments.

## 4.1 Experimental set-up

Section 4.1 should be the experimental set-up (data sets you are using, any preprocessing to the datasets, tool configuration, etc – describe it so that someone can reproduce what you’ve done).

Then you will probably have separate sections for each experiment.

## 4.2 Analysis and Results

Then the final section can be the analysis/results section.

# 5.0 Conclusion and Future works

## 5.1 Summary

## 5.2 Future work and outlook

# 6.0 Limitation

# References

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