

Methods for Enhancing Software Fault Localization

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR
OF PHILOSOPHY OF THE UNIVERSITY OF SZEGED

By:
Qusay Idrees Sarhan Alsarhan

Supervisor:
Árpád Beszédes, PhD, associate professor



Doctoral School of Informatics
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged
Szeged, Hungary, 2023

Acknowledgments

First of all, I would like to praise and thank God for everything. Then, I would like to thank my supervisor, Dr. Árpád Beszédes, for guiding me and directing my PhD study. I will always be thankful to him and will never forget the great moments we have spent together during our journey. Also, I would like to thank my colleagues and co-authors for their endless support. Of course, I would like to thank Edit Szűcs for correcting this thesis from a linguistic point of view. Last, but not least, I would like to thank my wife, children, parents, sister, and brothers for their constant love and support throughout my study. Without their support, my PhD study would not have been possible.

Finally, I would like to thank the Department of Software Engineering of the University of Szeged for providing me with a productive study environment. The Stipendium Hungaricum Scholarship program funded my PhD study, so I am very appreciative of their support.

My PhD research was also supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory and by project TKP2021-NVA-09 implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Qusay Idrees Sarhan Alsarhan, 2023

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions of Thesis	2
1.3	Structure of Thesis	2
2	Spectrum-based Fault Localization (SBFL)	4
2.1	Software Testing	4
2.2	Testing Principles	4
2.3	Testing Types	5
2.3.1	White-box/Black-box Testing	5
2.3.2	Static/Dynamic Testing	5
2.4	Errors, Defects, and Failures	6
2.5	Testing vs. Debugging	6
2.6	Software Fault Localization	7
2.6.1	Manual Fault Localization	7
2.6.2	Automatic Fault Localization	8
2.7	Fault Localization Using SBFL	9
2.7.1	SBFL Process	9
2.7.2	SBFL Code Example	10
2.7.3	SBFL Formulas	11
2.7.4	SBFL Scores and Ranks	12
2.8	Experimental Design and Evaluation	12
2.8.1	Evaluation Metrics	12
2.8.2	Subject Programs	15
2.8.3	Granularity of Data Collection	17
2.8.4	Evaluation Baselines	17
2.8.5	Division by Zero Problem	18
2.8.6	Ranking of Multiple Bugs	18
2.8.7	Threats to Validity	18
3	Systematic Survey of SBFL Challenges	20
3.1	Introduction	20
3.2	Related Works	21
3.3	Research Methodology	22

3.3.1	Identification of Research Objective	22
3.3.2	Search Strategy	22
3.3.3	Paper Selection	23
3.4	Results	25
3.4.1	Statistical Analysis:	25
3.4.2	Coverage Types:	26
3.4.3	Elements Tie:	27
3.4.4	Division by Zero	28
3.4.5	Negative Suspiciousness Scores	29
3.4.6	Source of Bugs	29
3.4.7	Single and Multiple Bugs	30
3.4.8	Ranked Elements	31
3.4.9	Limitations of SBFL Tools	35
3.4.10	Bugs Due to Missing Code	36
3.4.11	Simulation of SBFL	37
3.4.12	Test Flakiness	37
3.4.13	Seeded and Real Bugs	38
3.4.14	Spectra Formulas Selection	39
3.4.15	No Interactivity	39
3.4.16	Top-N Ranking	41
3.4.17	Results Visualization	41
3.4.18	No Contextual Information	44
3.5	Contributions	45
4	Tie-Breaking Method for SBFL	47
4.1	Introduction	47
4.2	Related Works	48
4.3	Evaluation	49
4.3.1	Subject Programs	49
4.3.2	Evaluation Baselines	49
4.4	Tie Statistics	50
4.5	Call Frequency-based Tie-Breaking	52
4.5.1	Frequency-based Tie Reduction	52
4.5.2	Reduction of the Critical Ties	56
4.6	Experimental Results and Discussion	59
4.6.1	Achieved Improvements in Average Ranks	59
4.6.2	Achieved Improvements in Top-N Categories	60
4.7	Contributions	62
5	Emphasizing SBFL Formulas With Importance Weights	63
5.1	Introduction	63
5.2	Related Works	64
5.3	Non-Contextual Importance Weight	65
5.3.1	The Proposed Approach	65
5.3.2	A Motivating Example From Defects4J	65

5.3.3	Evaluation	67
5.3.4	Experimental Results and Discussion	68
5.4	Contextual Importance Weight	72
5.4.1	The Proposed Approach	72
5.4.2	A Motivating Example From Defects4J	72
5.4.3	Evaluation	73
5.4.4	Experimental Results and Discussion	73
5.5	Non-contextual vs. Contextual Importance Weights	76
5.6	Contributions	78
6	New Formulas for SBFL	79
6.1	Introduction	79
6.2	Related Works	81
6.2.1	Introducing New SBFL Formulas	82
6.2.2	Modifying Existing SBFL Formulas	82
6.2.3	Combining Existing SBFL Formulas	82
6.2.4	Adding New Information to Existing SBFL Formulas	83
6.2.5	Generating New SBFL Formulas by Meta-heuristic Search	83
6.2.6	Machine Learning	83
6.2.7	Systematic Formula Generation	84
6.3	Manually Crafted New Formulas	84
6.3.1	The Proposed SBFL Formula	84
6.3.2	A Motivating Example From Defects4J	85
6.3.3	Evaluation	86
6.3.4	Experimental Results and Discussion	87
6.4	Systematic Search for New Formulas: Preliminary Study	90
6.4.1	Systematic Analysis	90
6.4.2	Formula Template for the Experiments	90
6.4.3	Evaluation	92
6.4.4	Experimental Results and Discussion	92
6.5	Systematic Search for New Formulas: Extended Study	94
6.5.1	Formula Templates for the Experiments	94
6.5.2	Evaluation	96
6.5.3	Experimental Results and Discussion	98
6.6	Contributions	101
7	SBFL Supporting Tools	102
7.1	Introduction	102
7.2	Related Works	103
7.3	CharmFL Tool	105
7.3.1	CharmFL's User Interface	105
7.3.2	CharmFL's Architecture	106
7.3.3	How to Use CharmFL	108
7.4	Software Fault Localization as a Service (SFLaaS) Tool	110
7.4.1	SFLaaS's User Interface	110

7.4.2 SFLaaS's Architecture	112
7.4.3 How to Use SFLaaS	114
7.5 Contributions	115
8 Conclusions and Future Work	116
8.1 Conclusions and Future Work	116
8.2 Potential Impacts of the Thesis	120
Bibliography	121
Appendix A: Summary	140
Appendix B: Összegzés	145

List of Figures

2.1	Debugging process	7
2.2	SBFL process	10
2.3	SBFL example: code and test cases	11
3.1	The outcome of the paper selection process	24
3.2	Challenges and issues of SBFL	25
3.3	Fault propagation	30
3.4	Running example – unoptimized code	32
3.5	Running example – optimized code	32
3.6	Suspicious program regions	34
3.7	Static vs. interactive SBFL	40
3.8	Different visualization schemes.	43
3.9	Connections between thesis chapters and main SBFL issues	46
4.1	Size distribution of critical ties	52
4.2	SBFL example: code and test cases	53
4.3	Call frequency example: call tree and call stacks	54
4.4	The proposed tie-breaking process	55
4.5	Tie-reduction distribution of critical ties	57
5.1	The non-contextual based importance weight approach	66
6.1	Systematic search for new SBFL formulas	97
7.1	CharmFL GUI	105
7.2	CharmFL ranking list output	106
7.3	Highlighted statements based on suspicion scores	107
7.4	CharmFL advanced options	107
7.5	Running example – program code and its tests	109
7.6	SFLaaS GUI	111
7.7	Highlighted statements based on suspicion scores	112
7.8	Architecture of SFLaaS	113
7.9	Technical details of SFLaaS	114

List of Tables

1.1	Mapping of PhD thesis points, chapters, and publications	2
2.1	SBFL example: spectra information	11
2.2	SBFL example: scores and ranks	12
2.3	Tied elements ranking approaches	14
2.4	Top-N categories example	15
2.5	Details of Defects4J 1.5	16
2.6	Details of Defects4J 2.0	16
2.7	Used SBFL formulas	17
3.1	Literature sources used to search relevant studies	23
3.2	Running example – spectra and four counters before optimization . . .	33
3.3	Running example – spectra and four counters after optimization	33
3.4	Proposed solutions to address the Top-N issue	42
3.5	Traditional output of SBFL	42
4.1	Number of ties: total and average per bug	50
4.2	Number of critical ties: total and average per bug	51
4.3	Improvement possibilities of critical ties	51
4.4	Example frequency-matrix	54
4.5	Ranks Before and After using the tie-breaking strategy	56
4.6	Statistics of tie-reduction (in percentage)	57
4.7	Changes in the number of critical ties after reduction	58
4.8	Achieving the minimum ranks	58
4.9	Average rank of faulty elements before and after tie-breaking	59
4.10	Comparison of average ranks before and after tie-breaking	60
4.11	Top-N categories	61
4.12	Top-N moves	62
5.1	Motivating example’s basic statistics	67
5.2	Motivating example – scores and ranks	68
5.3	Comparison of average ranks	69
5.4	Top-N categories	70
5.5	Enabling improvements	70
5.6	Top-N moves	71

5.7	Motivating example's basic statistics	72
5.8	Motivating example – scores and ranks	73
5.9	Average ranks comparison	74
5.10	Top-N categories	75
5.11	Enabling improvements	75
5.12	Comparison of average ranks	76
5.13	Top-N categories	77
5.14	Enabling improvements	78
6.1	Motivating example's basic statistics	86
6.2	Motivating example – scores and ranks	87
6.3	Average ranks comparison	87
6.4	Top-N categories	88
6.5	Variants of formulas.	91
6.6	Average ranks of the generated SBFL formulas	92
6.7	Top-N categories	93
6.8	Formulas covered by new templates (*: only rank-equivalents are covered)	96
6.9	Average ranks (the best values for a particular row are shown in bold)	99
6.10	Top-N categories	100
7.1	Comparison among Python fault localization tools	105
7.2	Framework usage commands	108
7.3	Method hit spectra (with four basic statistics)	108
7.4	Tarantula suspiciousness scores	109
A.1	Mapping of PhD thesis points, chapters, and publications	144
A.1	PhD t�zispontok, fejezetek �s publik�ci�k kapcsolata	149

Abbreviations

BSN Basic Statistical Numbers

DDG Dynamic Dependence Graph

GP Genetic Programming

GUI Graphical User Interface

IDE Integrated Development Environment

iFL Interactive Fault Localization

MECO Metrics Combination

MFL Multiple Fault Localization

ML Machine Learning

RQ Research Question

SBFL Spectrum-based Fault Localization

SBI Statistical Bug Isolation

SGF Systematically Generated Formula

Chapter 1

Introduction

1.1 Motivation

Software ¹ products cover many aspects of our everyday life as they are used in different application domains, such as communication, healthcare, military, and transportation. Thus, our modern life cannot be imagined without software. The extensive demand and use of different software products in our day-to-day activities have significantly increased their functionality, size, and complexity. As a result, the number and types of software faults ² have also increased [97]. Software faults not only lead to financial losses but also loss of lives. Therefore, faults should be fixed as soon as they are found. Finding the locations of faults in software systems has historically been a manual task that has been known to be tedious, expensive, and time-consuming, particularly for large-scale software systems [44]. Besides, manual fault localization depends on the developer's experience to find and prioritize code elements that are likely to be faulty.

Developers spend almost half or more of their time on finding faults alone [128]. Therefore, there is a serious need for automatic fault localization techniques to help developers effectively find the locations of faults in software systems with minimal human intervention [28]. Researchers and developers have proposed and implemented different types of such techniques. However, Spectrum-based Fault Localization (SBFL) is considered amongst the most prominent techniques in this respect due to its efficiency and effectiveness [47], lightweight, language-agnostic [108], easy-to-use [136], and relatively low overhead in execution time [117] characteristics.

In SBFL, the probability of each program element (e.g., statement, function ³, or class) being faulty is calculated based on the results of executing test cases and their corresponding code coverage information [99]. Currently, SBFL is not yet widely adopted in the industry [43, 175] as it poses several issues and its performance is affected by several influential factors [63, 159]. Therefore, addressing SBFL issues can lead to improving its effectiveness and making it widely used.

¹In this PhD thesis, the terms 'software' and 'program' are used interchangeably.

²In this PhD thesis, the terms 'defect', 'fault', and 'bug' are used interchangeably.

³In this PhD thesis, the terms 'function' and 'method' are used interchangeably.

1.2 Contributions of Thesis

In this PhD ⁴ thesis, the aim was to enhance SBFL by introducing new methods and enhancing previous approaches by addressing some of SBFL’s main issues and challenges. This is achieved by conducting a systematic literature survey to identify several issues and challenges in SBFL that are still not addressed, and then I started to tackle them one by one by conducting several lab experiments. As a result, several articles on the topic have been published or accepted for publication in well-known venues (conferences and journals). The scientific results I achieved and report in this thesis are grouped into several thesis points, as presented in Table 1.1.

Table 1.1: *Mapping of PhD thesis points, chapters, and publications*

No.	PhD Thesis Points	Chapters	Publications
I.	Systematic Survey of SBFL Challenges	Chapter 3	[122]
II.	Tie-Breaking Method for SBFL	Chapter 4	[60]
III.	Emphasizing SBFL Formulas with Importance Weights	Chapter 5	[119], [121]
IV.	New Formulas for SBFL	Chapter 6	[120], [123], [124]
V.	Supporting Tools for SBFL	Chapter 7	[127], [134], [125], [126]

1.3 Structure of Thesis

The structure of this thesis also follows the grouping of thesis points, hence it consists of several chapters, as follows:

Chapter 2 introduces an overview of software testing and fault localization. Then, it introduces SBFL, its concepts, how it works, and the terms that are needed to understand the contents of this thesis.

Chapter 3 presents the results of a systematic literature survey on the challenges of SBFL where different issues and challenges that affect the effectiveness of SBFL and thus prevent it from being widely used were presented and discussed. Also, different potential solutions to improve/enhance SBFL were given. This systematic survey study was the basis for the experimental contributions presented in the subsequent chapters.

Chapter 4 introduces the ties in SBFL and a novel proposed approach to address this issue. Often, SBFL formulas produce the same suspicion score for more than one code element. Thus, ties emerge between the code elements. To solve this issue, a method based on method call frequency in failed test cases to break ties is proposed and discussed. The idea is that if a method appears in many different calling contexts during a failing test case, it will be more suspicious and thus gets a higher rank position compared to other methods with the same scores.

⁴The type of this PhD thesis is a thesis by publication (also called a paper-based thesis).

Chapter 5 introduces importance weights to improve SBFL by addressing the issue of unbalanced test suites where the number of passed tests is much higher than the number of failed tests. This is achieved by emphasizing the factor of failing tests in SBFL formulas by giving more importance to code elements that are executed by more failed tests and appear in more failing method call contexts compared to other elements. Thus, such elements get higher ranks than others and get examined first by software developers.

Chapter 6 introduces a new manually crafted SBFL formula based on the “guess” or “intuitive” method. The new formula breaks ties between the elements that share the same suspicion score by emphasizing the high number of failing test cases and the low number of passing ones for a particular code element. This chapter also introduces a systematic search method to generate new SBFL formulas instead of the heuristic and ad-hoc approaches. This is achieved by examining existing formulas, defining formula structure templates, generating formulas from the defined templates, and finally comparing them to each other. All the new formulas, which are not reported in the literature, outperform many well-known existing ones.

Chapter 7 introduces two software fault localization tools that employ SBFL, namely “CharmFL” and “SFLaaS”, for Python developers. The tools are designed with a lot of useful features to help Python programmers debug their code. Through several lab settings, the usefulness of both tools has been assessed. In addition to being simple to use, the tools have been shown to be helpful for identifying faults in various programs.

Chapter 8 gives our conclusions and future work plan. Finally, this thesis is briefly summarized in English and Hungarian in Appendices A and B, respectively.

Chapter 2

Spectrum-based Fault Localization (SBFL)

2.1 Software Testing

Software products are an integral part of our lives as they cover a wide range of application domains, such as healthcare, military, industry, etc. The majority of people have encountered software that did not function as expected. Incorrectly functioning software can cause a variety of issues, such as financial loss, lost productivity, damage to a company's brand, and even physical harm or death. Therefore, software testing is necessary for evaluating the software's quality and lowering the possibility of software failures.

2.2 Testing Principles

The main principles of software testing can be listed as follows:

- Testing is done to increase trust in the program's quality level.
- The appropriate resources (e.g., programs, codes, requirements, testers, etc.) should be available before starting testing.
- Before carrying out any testing, all tests should be well planned and meet the user requirements because random testing, in which you simply try something and see if it works, is less likely to find bugs.
- Since the space of potential test cases is typically too large to be thoroughly covered, exhaustive testing is not practicable.
- Test with little code parts first, then move on to larger ones.
- As you write code, test it because debugging becomes more difficult and time-consuming when testing is left until the very end.

2.3 Testing Types

In software engineering, testing can be classified into several main types [168], as follows:

2.3.1 White-box/Black-box Testing

It refers to the tester's knowledge and understanding of the internal workings and details of the software under test.

- **White-box testing:** Here, the developer focuses on the code and its implementation details by examining every written programming statement. As the code is visible and accessible to developers during the testing process this type of testing is called a white-box. The goal of white-box testing is to know how the output is achieved exactly. This is performed mainly by testing the control flow and data flow of the program under test.
- **Black-box testing:** Here, the developer ignores the code and its implementation details and focuses instead on what is the output only. This is performed mainly by testing the functional (e.g., unit testing, integration testing, and system testing) and non-functional (e.g., performance testing, usability testing, and compatibility testing) requirements of the program under test. As the code is not visible and accessible to developers during the testing process this type of testing is called a black-box.

2.3.2 Static/Dynamic Testing

It refers to the state of the software under test; whether it is running or not.

- **Static testing:** Here, a code is tested without running it and it is performed in the early stages of software development to prevent defects.
- **Dynamic testing:** Here, a code is tested while it is running and it is performed at the later stages of software development to find and fix defects.

It is worth mentioning that white-box testing can be static or dynamic. For example, testing data flow can be static when the declaration, usage, and deletion of variables are examined without executing the code. Or, it can be dynamic when the variables and data flow are examined with the execution of the code. Black-box testing can also be static or dynamic. For example, reviewing the requirement, specification, or design documents and looking for errors is static. Or, running a code with various inputs and checking the outputs is dynamic.

2.4 Errors, Defects, and Failures

An error (mistake) made by a person in a program's design, development, or operation that results in a program failure (unexpected or wrong output) is known as a defect, bug, or fault in the field of software engineering [34]. Generally speaking, bugs are classified into two main types: logical and illogical. Logical bugs appear due to a mistake in the workflow of the software. For example, passing parameters in incorrect order could cause a logical error. Logical bugs cannot be detected by the compiler; thus finding and locating them requires fault localization techniques to be used. Illogical bugs appear due to syntax or type errors. For example, a single missing bracket could cause a syntax error. Error messages are always generated by the compiler when it finds illogical bugs. In most cases, an error message provides abundantly clear information about the source of the bug, such as what caused the bug, the line number, the impacted code elements, and a justification. These messages typically include sufficient details regarding the bug and the recommended course of action to fix it. In other words, they can be found and located easily.

2.5 Testing vs. Debugging

Bugs are so prone to be found in programs because they are written by humans and humans make mistakes due to many reasons (e.g., when they work longer hours, work under pressure, or when do not map properly between a program and its requirements) and it is very normal. Bugs have the potential to cause severe financial losses and perhaps fatalities. Therefore, a program should be tested to ensure it is bug-free. The common way to test a program is to execute it against several test inputs. The output for each input test case is then observed by developers as input-output relationships. For each test case, the program is successfully tested if it generates the required output.

However, if one of the test cases' output differs from what was expected, the program is incorrect and contains bugs. In such cases, testing just identifies the existence of bugs but does not provide any information regarding their types, their locations, or how they should be fixed. Testing, in other words, exposes bugs' effects (or symptoms), not their root causes. The developer must next go through a debugging process (i.e., fault localization) to locate the bugs' root causes and resolve them. Figure 2.1 shows the debugging process.

When an error is detected in a program, the developer begins by identifying the error that caused the program's failure. For example, the developer can try to reproduce the error to ensure it is legit. Then, the developer needs to go through the code to pinpoint exactly where the error is. After identifying the location of the error, it is crucial to understand the error (e.g., by checking around it) and know its potential impact on other parts of the code to fix it properly. Finally, regression testing is performed to ensure that the program works as expected after any code fixes/changes.

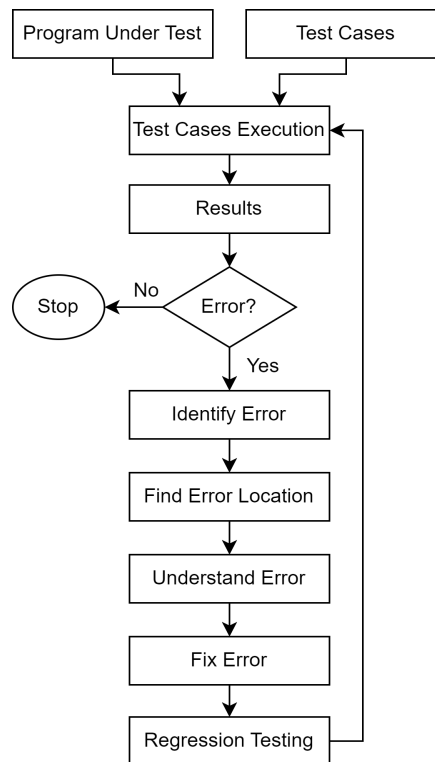


Figure 2.1: *Debugging process*

2.6 Software Fault Localization

2.6.1 Manual Fault Localization

Fault localization's purpose is to locate faults; historically, this has been done manually and has been found to be laborious, time-consuming, and excessively expensive. Additionally, manually locating faults heavily depends on the expertise, judgment, and intuition of the software developer to recognize and put more emphasis on code elements that are more likely to be buggy. The main manual techniques are summarized below [155]:

- **Logging-based Fault Localization:** The most common way of manual fault localization is to use “Print” debugging. “Print” debugging is the print statements, such as “print()” in Python or “System.out.print()” in Java, that developers insert in their source code to print out or save program state details (such as the values of variables) to the console or to a log file to get a better sense of what is happening while the program is executing. Developers look to the program log (i.e., printed runtime information or saved log files) when unexpected program behavior is discovered to identify the root cause of the problem. Thus, these printed statements can help find bugs, especially when a variable has an unexpected value.

- **Assertions-based Fault Localization:** Assertions are checks added to a program code to test if certain assumptions remain true while developing or executing it. However, the program code is considered buggy if any of the added assertions prove to be false.
- **Breakpoints-based Fault Localization:** With the use of breakpoints, a program may be paused when it reaches a specific point in its execution so that the developer can review the program status at that time. When a breakpoint is hit, the developer has the option of changing a variable value or continuing the program's execution to track the development of a bug. It is possible to set up data breakpoints such that they activate whenever the value of certain variable changes, for example. Such breakpoints are called conditional breakpoints. In other words, only when a specified condition is satisfied, do conditional breakpoints pause program execution.

Manual debugging is an easy technique to locate faults if the program scale is small; but it is a time-consuming and tedious process if the developer is dealing with a large-scale program that has several modules, functions, and variables. If the developers find themselves locating faults in large-scale programs, then it is time to switch to an automatic debugging way as follows.

2.6.2 Automatic Fault Localization

Due to the limitations of manual fault localization approaches, there is a growing interest in developing techniques that can locate faults in program code automatically with the least amount of human participation. There have been several fault localization strategies developed and each is unique in the sort of data it uses, the program elements it concentrates on, and its success. Although each technique tries to address the problem of fault localization from a different viewpoint, it frequently has both advantages and disadvantages when compared to other techniques. The main automatic techniques are summarized below [155]:

- **Slicing-based Fault Localization:** A technique to generate a reduced part of a program for testing particular test conditions or cases based on program data and control dependence information; it was proposed first in 1979 by Mark Weiser [148]. It generates a group of program statements in the program (called a slice) that may influence a value at a specific place of interest in the program (called a slicing criterion). Thus, it helps developers to narrow the search space while finding program faults. Consider a test case that fails due to an invalid value of a variable at a statement as an example. In this situation, the faulty program element must be in the slice that is associated with that variable-statement pair. Developers will be able to concentrate their search on that slice rather than having to look through the full program to find the bug. Bogdan Korel and Janusz Laski [78] proposed dynamic slicing to reduce static slice size. Dynamic slices contain only executed statements during the test

cases. Even though the amount of code that should be examined is decreased by dynamic slicing, the quantity of the code that is left is still too much to be examined.

- **Model-based Fault Localization:** When using this technique, it is expected that each program under test has a correct model that is available and accessible. Models can act as the programs' oracles. Finding bugs involves comparing a model's behavior to the program's actual observed behavior. Models can be generated either from clean actual programs or by developers before developing the actual programs.
- **Coverage-based Fault Localization:** Recently, many fault localization techniques employing the relationship between reasons (faults in our case) and results (execution failures in our case) have appeared. Among these, the most common fault localization techniques for locating suspicious code that causes execution errors are SBFL techniques. As the aim of this thesis is to enhance SBFL, the following sections elaborate on this technique and how it can be used for locating faults in programs.

2.7 Fault Localization Using SBFL

2.7.1 SBFL Process

SBFL is a popular dynamic program analysis technique that uses code coverage information, often known as program spectra, and is collected via executing tests against various program elements along with tests' results, to locate faults [34]. When a test case is executed, code coverage shows which program elements were executed and which ones were not, whereas there are two possible test results: passed or failed. Tests fail when a program's output is unexpected during its execution, whereas they pass when the output is expected. After that, a ranking formula uses this information to determine the buggy likelihood of elements (e.g., statements, functions, or classes). The main steps of the SBFL process are shown in Figure 2.2.

In 1987, the concept of program spectra was introduced [27]. However, research on the year 2000 (also called Y2K) problem to identify bugs in formatting and storing dates before and after the year 2000 utilized program spectra for fault localization [116]. "Tarantula" was one of the first approaches in this respect, proposed in 2002 [67], that used a formula to compute the suspicion scores for program elements in SBFL [34]. After that, other additional formulas were introduced, many of which had a biological basis. Examples of these formulas include "Ochiai" [102] and "Binary" [22].

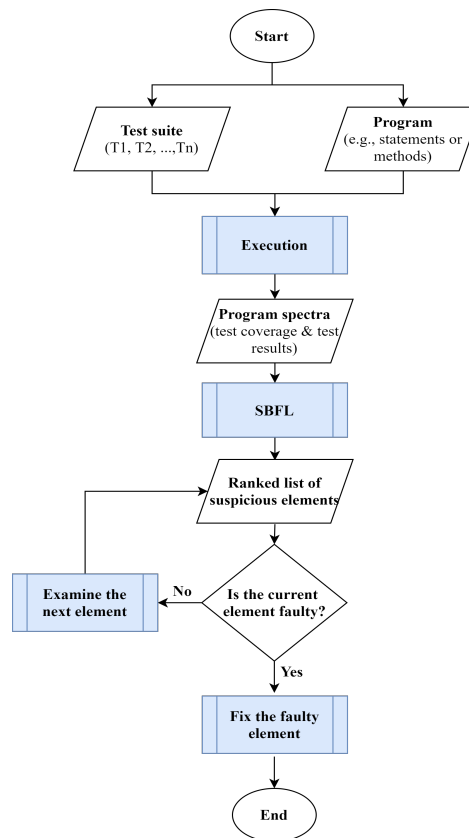


Figure 2.2: SBFL process

2.7.2 SBFL Code Example

The work of SBFL is simple, it only needs the information of executing several tests on a program's elements to locate faults. Suppose a Python function called *mid()* accepts three values as input and outputs the median of the three values. The *mid()* function, a widely used code example in fault localization research [73], consists of 12 statements S_i ($1 \leq i \leq 12$) and 6 tests T_j ($1 \leq j \leq 6$), as shown in Figure 2.3. In statement 7, there is a fault (the valid statement is $m = x$).

Then, the function was tested with all the tests, and the spectra (the execution information of statements in failed and passed tests) were recorded, as presented in Table 2.1. A 1 in the cell corresponding to the statement S_i and the test case T_j indicates that the test case T_j executed the statement S_i , otherwise it is 0. Additionally, a 1 in the row labeled "Results" denotes a failed test, whereas a 0 denotes a passed test. In SBFL, a program element (e.g., a statement in our example) that is executed in more failed test cases would be more likely to be faulty.

<pre> 1: def mid(x, y, z): 2: m = z 3: if y<z: 4: if x<y: 5: m = y 6: elif x<z: 7: m = y 8: else: 9: if x>y: 10: m = y 11: elif x>z: 12: m = x 12: return m </pre>	<pre> import mid_function def test_T1(): assert mid_function.mid(3, 3, 5) == 3 def test_T2(): assert mid_function.mid(1, 2, 3) == 2 def test_T3(): assert mid_function.mid(3, 2, 1) == 2 def test_T4(): assert mid_function.mid(5, 5, 5) == 5 def test_T5(): assert mid_function.mid(5, 4, 3) == 4 def test_T6(): assert mid_function.mid(2, 1, 3) == 2 </pre>
--	--

Figure 2.3: SBFL example: code and test cases

Table 2.1: SBFL example: spectra information

Statement	T1	T2	T3	T4	T5	T6	ef	ep	nf	np
1	1	1	1	1	1	1	1	5	0	0
2	1	1	1	1	1	1	1	5	0	0
3	1	1	1	1	1	1	1	5	0	0
4	1	1	0	0	0	1	1	2	0	3
5	0	1	0	0	0	0	0	1	1	4
6	1	0	0	0	0	1	1	1	0	4
7	1	0	0	0	0	1	1	1	0	4
8	0	0	1	1	1	0	0	3	1	2
9	0	0	1	0	1	0	0	2	1	3
10	0	0	0	1	0	0	0	1	1	4
12	1	1	1	1	1	1	1	5	0	0
Results	0	0	0	0	0	1				

2.7.3 SBFL Formulas

An SBFL formula is applied to the spectra information to calculate each program element's suspicion score. Table 2.7 presents several existing SBFL formulas. The following four statistical numbers, computed from the program spectra, are frequently used to express a formula:

- ef : the number of failed tests that executed (e) a program element.
- ep : the number of passed tests that executed (e) a program element.
- nf : the number of failed tests that did not execute (n) a program element.
- np : the number of passed tests that did not execute (n) a program element.

2.7.4 SBFL Scores and Ranks

After calculating the suspicion score for each program element, a statement in our example, the statements are ranked based on the scores after they get sorted in ascending order from the most suspicious to the least suspicious to be examined by developers. Table 2.2 presents this information. For example, the “Tarantula” formula scores both statements 6 and 7 greater than others; thus they are more suspicious and should be examined before others. While it gives statement 4 the third greatest score, and so on. However, no score was given to statement 11 because it has no spectra information (i.e., no test has executed it).

Table 2.2: *SBFL example: scores and ranks*

	Tarantula score	Rank	Ochiai score	Rank	Overlab score	Rank	Wong II score	Rank	Goodman score	Rank
1	0.5	4	0.41	4	0	1	-4	8	-0.43	4
2	0.5	4	0.41	4	0	1	-4	8	-0.43	4
3	0.5	4	0.41	4	0	1	-4	8	-0.43	4
4	0.71	3	0.58	3	0	1	-1	3	0	3
5	0	8	0	8	0	1	-1	3	-1	8
6	0.83	1	0.71	1	0	1	0	1	0.33	1
7	0.83	1	0.71	1	0	1	0	1	0.33	1
8	0	8	0	8	0	1	-3	7	-1	8
9	0	8	0	8	0	1	-2	6	-1	8
10	0	8	0	8	0	1	-1	3	-1	8
12	0.5	4	0.41	4	0	1	-4	8	-0.43	4

2.8 Experimental Design and Evaluation

This section presents the considerations necessary for designing and evaluating the SBFL experiments of this thesis.

2.8.1 Evaluation Metrics

For evaluating SBFL’s effectiveness, we use evaluation metrics that have also been used by other researchers in the literature for this purpose [18, 65, 165], as follows.

Achieved Improvements in Average Ranks

An SBFL formula assigns a suspicion score to each program element. Then, all elements are ranked based on the scores from the most suspicious to the least to be examined by the developer.

The obtained ranks can be used to evaluate the effectiveness of an SBFL approach. To illustrate this, let us consider that we have a buggy program with 1000 program elements (e.g., statements). Then, two SBFL approaches, A and B, were applied to

that buggy program. Approach A put the buggy element at the 500th position in the ranking list. While approach B put the buggy element at the 50th position in the ranking list. The *Exam* metric (also called *Expense* or *Average Rank Percentage* [101]), Equation 2.1, which measures the percentage of program elements that the programmer needs to examine to find the bug, can be used to evaluate the effectiveness of both SBFL approaches. The *Exam* value for approach A will be 50% and for approach B will be 5%. As the *Exam* value of approach B is lower than the *Exam* value of approach A, approach B is considered better because the programmer will examine a lower number of elements to find the buggy element compared to approach A.

$$\text{Exam} = \left(\frac{E}{N} \right) \cdot 100\% \quad (2.1)$$

Where E is the position of the faulty element in the ranking list and N is the total number of statements in the ranking list.

Often, different elements are assigned the same suspicion score and this is prevalent in software fault localization [60]. To rank such elements that are “score tied” to each other (also called tied elements), many approaches [156] were proposed, as follows:

- Minimum (MIN) Rank: It considers the top position of the set of elements having the same suspicion score as the rank of each element in that corresponding set (it is also called the optimistic or best case).
- Maximum (MAX) Rank: It considers the bottom position of the set of elements having the same suspicion score as the rank of each element in that corresponding set (it is also called the pessimistic or worst case).
- Average (MID) Rank: It considers the medium position of the set of elements having the same suspicion score as the rank of each element in that corresponding set (it is also called the average case). The average rank is calculated using Equation 2.2.

$$\text{MID} = S + \left(\frac{E - 1}{2} \right) \quad (2.2)$$

where S represents the starting position of a tie and E represents its size.

Tied element ranking approaches are particularly crucial for assessing the efficacy of SBFL formulas in terms of in which location they put the faulty element in the produced ranking list. Table 2.3 presents an example of the aforementioned different ranking approaches using the scores calculated in section 2.7.4.

From Table 2.3, it can be noted that the ranking approaches must be applied after the suspicion scores get sorted in ascending order. As the MID ranking approach

Table 2.3: *Tied elements ranking approaches*

Statement	Sorted Tarantula Score	MIN Rank	MAX Rank	MID Rank
6	0.83	1	2	1.5
7	0.83	1	2	1.5
4	0.71	3	3	3
1	0.5	4	7	5.5
2	0.5	4	7	5.5
3	0.5	4	7	5.5
12	0.5	4	7	5.5
5	0	8	11	9.5
8	0	8	11	9.5
9	0	8	11	9.5
10	0	8	11	9.5

considers the average position of all possible positions, compared to the other two approaches, it is more commonly used for evaluation. In this thesis, we use the MID ranking approach in Equation 2.2 to analyze SBFL efficiency in general.

It is important to note that relying solely on the MID ranking approach as a metric for measuring SBFL efficiency has several disadvantages:

- The information on the effectiveness of any SBFL technique may be distorted by outlier average ranks.
- It provides no information regarding the rank values' distribution or how they change before and after using a suggested SBFL technique.

Therefore, there is a more significant evaluation metric than the MID ranking approach. This metric measures the improvements that an SBFL technique can achieve in the “Top-N” ranks. Thus, as presented below, the evaluation benefits are more clear.

Achieved Improvements in Top-N Categories

Based on many studies including [77] and [158], it is considered acceptable by developers to examine the top ten ranked program elements recommended by an SBFL technique. Therefore, these rank positions (also known as “Top-N” categories) can also be used to assess SBFL’s effectiveness, as follows:

- **Top-1:** It refers to how many faulty program elements in the ranking list are ranked first.
- **Top-3:** It refers to how many faulty program elements have a rank less than or equal to three.
- **Top-5:** It refers to how many faulty program elements have a rank less than or equal to five.

- **Top-10:** It refers to how many faulty program elements have a rank less than or equal to ten.
- **Other:** It refers to how many faulty program elements have a rank greater than ten.

Suppose two approaches A and B were applied to a dataset of 297 bugs and they have been compared to each other based on the “Top-N” categories as presented in Table 2.4. For each approach, it presents the number of bugs in the Top-N categories (cumulative), their percentages for the used dataset, and the difference between them. It can be noted that approach B is more effective than approach A as it puts fewer bugs in the “Other” category (this kind of improvement is also known as *enabling improvement* [18]) and more bugs in any Top-N category.

Table 2.4: Top-N categories example

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Approach A	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
Approach B	59	19.9	124	41.8	148	49.8	178	59.9	119	40.1
Diff.	11	22.9	13	11.7	11	8.0	11	6.6	-11	-8.5

We also used a special non-accumulating form of Top-N categories that counts the cases where the bug fell in non-overlapping intervals of [1], (1, 3], (3, 5], (5, 10], or (10, ...]. These categories illustrate how often a bug is moved into a better (e.g., from (5, 10] to (1, 3]) or worse (e.g., from [1] to (1, 3]) category when using an SBFL technique. To put it another way, how many times do the bugs move to a higher-ranking category and how many times do they move to a lower-ranking category? As a result, an SBFL approach that improves Top-N categories by moving a large number of bugs to higher-ranked categories performs better.

2.8.2 Subject Programs

Evaluating SBFL techniques requires a suitable faults dataset to be used. In all our experiments, we used Defects4J; the popular bug dataset for Java programs [58]. It is a collection of non-trivial real single and multiple faults which enables reproducible experimental research in software fault localization [71, 89]. Defects4J version 1.5¹ was used in most of our studies; it has 438 faults extracted from 6 open-source Java programs. However, some faults were excluded due to technical limitations. Consequently, 411 bugs were present in the final dataset that was used. Table 2.5 presents the main details of Defects4J version 1.5.

¹<https://github.com/rjust/defects4j/tree/v1.5.0>

Table 2.5: *Details of Defects4J 1.5*

Project	Number of bugs	Size (KLOC)	Number of tests	Number of methods
Chart	25	96	2.2k	5.2k
Closure	168	91	7.9k	8.4k
Lang	61	22	2.3k	2.4k
Math	104	84	4.4k	6.4k
Mockito	27	11	1.3k	1.4k
Time	26	28	4.0k	3.6k
Total	411	332	22.1k	27.4k

However, Defects4J version 2.0² was also used in some of our studies, where 17 open-source Java programs have 835 real single and multiple faults. We excluded some faults in this study due to instrumentation errors or unreliable test results. Thus, a total of 782 faults were included in our final dataset. Table 2.6 presents the main details of Defects4J version 2.0.

Table 2.6: *Details of Defects4J 2.0*

Project	Number of bugs	Size (KLOC)	Number of tests	Number of methods
Chart	25	96	2.2k	5.2k
Cli	39	4	0.1k	0.3k
Closure	171	91	7.9k	8.4k
Codec	17	10	0.4k	0.5k
Collections	1	46	15.3k	4.3k
Compress	36	11	0.4k	1.5k
Csv	16	1	0.2k	0.1k
Gson	15	12	0.9k	1.0k
JacksonCore	25	31	0.4k	1.8k
JacksonDatabind	101	4	1.6k	6.9k
JacksonXml	5	6	0.1k	0.5k
Jsoup	90	14	0.5k	1.4k
JXPath	21	21	0.3k	1.7k
Lang	60	22	2.3k	2.4k
Math	104	84	4.4k	6.4k
Mockito	30	11	1.3k	1.4k
Time	26	28	4.0k	3.6k
Total	782	492	42.3k	47.4k

²<https://github.com/rjust/defects4j>

2.8.3 Granularity of Data Collection

Method-level granularity was used as the basic element for localizing faults. It has several advantages over the commonly used level of granularity, statement-level granularity [130], as follows:

- It offers more contextual details regarding the element under examination.
- It reduces the number of tied program elements.
- It is more scalable to the execution of large-scale programs.
- It is preferable for the users according to several studies [11, 183].

However, there is no theoretical barrier preventing further research into lower granularity levels.

2.8.4 Evaluation Baselines

Several well-known SBFL formulas [1, 2, 67, 99, 135, 142, 153], which are presented in Table 2.7, were employed as the benchmarks to assess and contrast our proposed solutions with. Also, the selected formulas were widely used in other research on software fault localization.

Table 2.7: *Used SBFL formulas*

No.	Name	Formula	No.	Name	Formula
1	Tarantula	$\frac{ef}{\frac{ef+nf}{ef+nf} + \frac{ep}{ep+np}}$	11	Cohen	$\frac{2*(ef*np)-2*(nf*ep)}{(ef+ep)*(ep+np)+(nf+np)*(ef+nf)}$
2	Ochiai	$\frac{ef}{\sqrt{(ef+nf)*(ef+ep)}}$	12	Confidence	$\frac{ef}{ef+nf} - \frac{ep}{ep+np}$
3	Jaccard	$\frac{ef}{ef+nf+ep}$	13	GP13	$ef * (1 + \frac{1}{2*ep+ef})$
4	Barinel = SBI	$\frac{ef}{ef+ep}$	14	Wong I	ef
5	SorensenDice	$\frac{2*ef}{2*ef+nf+ep}$	15	Wong II	$ef - ep$
6	DStar	$\frac{ef*ef}{ep+nf}$	16	Overlab	$\frac{ef}{\min(ef,nf,ep)}$
7	Dice	$\frac{2*ef}{ef+nf+ep}$	17	Goodman	$\frac{2*ef-nf-ep}{2*ef+nf+ep}$
8	Interest	$\frac{ef}{(ef+nf)*(ef+ep)}$			
9	Baroni	$\frac{\sqrt{ef*np+ef}}{\sqrt{ef*np+ef+nf+ep}}$			
10	Kulczynski1	$\frac{ef}{nf+ep}$			

2.8.5 Division by Zero Problem

A program element is given the 0.0 suspicion score when the denominator of a formula is 0 for that particular element to avoid the division by zero exception.

2.8.6 Ranking of Multiple Bugs

The Defects4J dataset contains single and multiple buggy versions of programs. If a program version has multiple bugs, we take into account the highest rank among the bugs.

2.8.7 Threats to Validity

Every survey or experiment faces some validity threats. To avoid or reduce the impact of such threats, the following steps were taken into consideration:

Survey Validity Threats

- Finding related papers: We cannot guarantee that every publication that is relevant to our study was found. To reduce the effect of this threat, we used different synonyms to define a search string that then was used to search for relevant publications in different literature sources. Even so, there can still be some missing related publications. The risk of missing them was reduced by using the snowballing search approach.
- Paper selection criteria: There is always a risk of bias when choosing relevant papers. The publications were thus only included or removed from this study when the authors agreed.
- Study reproducibility: This threat concerns whether other researchers can conduct a similar study and come to the same conclusions. By detailing the steps of how the survey study was conducted, this threat can be addressed. Therefore, Section 3.3 describes all the steps that were employed to conduct the study in great detail.

Experiment Validity Threats

- Selection of evaluation metrics: We used well-known and widely-used evaluation metrics to ensure the validity of our experimental results. This is very important as it makes our results reliable and comparable to other researchers' results.
- Correctness of implementation: To ensure that the implementations of our experiments have no issues and their outputs are correct, code reviews were done. We have also tested our proposed approaches numerous times to make sure they work as intended.

- Selection of subject programs: In our experiments, we selected the programs of the Defects4J dataset as our subjects. Thus, we cannot ensure that the same results can be obtained by using other different programs. However, as Defects4J programs are representative and include bugs of various types and complexity levels, we think this threat is minimal. In addition, Defects4J is often used in various fault localization studies.
- Exclusion of faults: We had to exclude a small number of faults from the Defects4J dataset owing to technical restrictions. The issue is whether or not our findings will be repeatable by other researchers using the same dataset. We feel that this threat is extremely minimal because our results were unaffected by the excluded faults. Besides, the used dataset had almost a homogeneous distribution of such faults.
- Selection of SBFL formulas: To assess the efficacy of our approaches, we conducted experiments using a selection of popular SBFL formulas, which account for a tiny portion of the published formulas in the literature. We cannot, however, guarantee that the same results would be obtained by utilizing other formulas. To reduce the impact of this problem, we applied formulas that were used in other research works as well.
- Granularity level: In our experiments, we verified the proposed solutions/concepts on the granularity of functions. However, in certain applications, such as automated program repair, statement granularity is required. It is unclear at present if the findings in this thesis are generalizable to statements as well.

Chapter 3

Systematic Survey of SBFL Challenges

3.1 Introduction

SBFL techniques are not yet widely adopted in the industry. The rationale behind this is that they pose several issues and their performance is affected by several influential factors. For example, the characteristics of bugs, target programs, test suites, and supporting tools make their effectiveness differ dramatically from one case to another.

In the literature, there are massive studies on SBFL covering its formulas, performance, and applications. However, no dedicated survey points out comprehensively the challenges and issues of SBFL. Thus, it is crucial to present and categorize various SBFL challenges and issues to offer a comprehensive survey on the topic.

The main contributions in this chapter can be summarized as follows:

1. Conducting a systematic literature survey study on the challenges of SBFL.
2. Identifying and presenting 18 SBFL challenges and issues.
3. The study also raises awareness of the works being achieved to address the identified challenges and issues and suggests some potential solutions to help those working on this topic and those interested in making contributions to it.

The study begins with the formulation of a Research Question (RQ) that addresses several aspects of the considered topic. It then identifies the related papers that should be read to answer the defined RQ. Finally, it discusses potential research opportunities in the field. To accomplish the aforementioned goals, relevant papers were collected and thoroughly analyzed in a systematic manner.

It is worth mentioning that we started our thesis with this important systematic survey study on the topic. As a result, this survey study was the basis for the experimental contributions presented in the subsequent chapters as will be seen later.

3.2 Related Works

SBFL has been an important and active research field for decades. However, a survey study on the issues and challenges in this research field was lacking. A few general survey studies on software fault localization have been found in the literature as the most relevant publications. In this section, these studies are presented briefly.

The authors in [168] provided an overview of coverage-based testing and compared 17 coverage-based testing tools including a tool called “eXVantage” which is developed by the authors. The comparison was based on several factors but focused more on coverage measurement. Then, they discussed various features (e.g., test case generation, test report customization, and automation) that should make tools more useful and practical. Also, they briefly mentioned that some tools have scalability issues, which makes them only suitable for small-scale software systems. Many others provide fine testing granularity, but the performance overhead prevents them from being useful for testing. However, the study helps developers pick the right tool that suits their requirements and development environment.

In [132], the authors presented evidence that the empirical evaluation of the accuracy of coverage-based fault locators depends on many factors. They summarized the problems that they encountered during their empirical evaluation of the accuracy of fault locators and classified them into two main categories: threats to validity and threats to value. Then, each category presents its own set of issues and their consequences on accuracy, including fault injection, instrumentation, multiple faults, and unrealistic assumptions.

In [6], the authors briefly provided a review of the previous studies on software fault-localization in a table in terms of techniques, evaluation methods, and the datasets used. However, their results are very abstract and no details have been provided nor have issues and challenges been discussed.

In [155], the authors surveyed the fault localization techniques from 1977 to 2014. They classified the techniques into eight categories: program slicing, spectrum-based, statistics, program state, Machine Learning (ML), data mining, model-based debugging, and additional techniques. They also listed popular subject programs used to study the effectiveness of different fault localization techniques. They also discussed fault localization tools developed by the presented studies. Additionally, they presented some research challenges with fault localization techniques such as fault interference, programs with multiple faults, and granularity level selection.

In [34], the authors surveyed the state-of-the-art of SBFL research including the proposed techniques, the type and number of faults they address, the types of spectra they use, the programs they utilize in their validation, and their use in industrial settings. Also, they highlighted some challenges (e.g., tied program elements, faults introduced by missing code, and coincidental correctness) of SBFL that have to be tackled to improve its effectiveness to be used in real development settings.

In [43], the authors briefly discussed two issues, granularity levels and program elements having the same suspiciousness, based on what the authors encountered in their collaboration with the industry. They highlighted that many different gran-

ularity levels can be employed to generate a spectrum, but there is no guide for practitioners to help them select the right spectrum granularity they require. Also, they discussed ties within rankings due to having program elements with the same suspiciousness. They concluded that this issue needs more attention and suggested the proposal of new strategies for tie-breaking.

In [175, 176], the authors presented the issue of Multiple Fault Localization (MFL) of software systems in the software fault localization domain. They identified three prominent MFL debugging approaches, i.e., one-bug-at-a-time debugging approach, parallel debugging approach, and multiple-bug-at-a-time debugging approach. Also, they presented some challenges with the identified approaches and provided some directions for future works.

All the survey studies mentioned earlier were general surveys that did not focus in detail on the issues and challenges of SBFL. Some of them briefly highlighted a very limited number of issues. However, most of them were not conducted systematically. In contrast, our study provides a thorough and systematic survey based on a detailed research methodology to examine different issues and challenges of SBFL alongside possible solutions or research gaps for further investigations. As a result, our systematic survey study extends the aforementioned studies by identifying, categorizing, and discussing 18 important issues comprehensively.

3.3 Research Methodology

The systematic process followed in this survey study is based on the guidelines provided by [110] and [75]. It consists of several stages as presented in the following subsections.

3.3.1 Identification of Research Objective

The objective of this survey study is to answer the following RQ:

“What are the challenges and issues posed by SBFL?”

Answering the aforementioned question is achieved by providing a comprehensive survey by reviewing the publications on the topic. Thus, helping software developers and researchers to better understand SBFL and contribute to its development and research.

3.3.2 Search Strategy

Literature Sources

Five well-known online literature sources that index publications of software engineering and computer science were used. Table 3.1 lists these sources as well as links to their websites.

Table 3.1: *Literature sources used to search relevant studies*

Source	Link
IEEE Xplore	http://ieeexplore.ieee.org
Elsevier ScienceDirect	http://sciencedirect.com
ACM Digital Library	http://portal.acm.org
Scopus	http://scopus.com
SpringerLink	http://springerlink.com

Search String

The following search string was used to find the relevant publications from the literature sources:

(“spectrum” OR “statistical” OR “coverage”) AND (“fault”) AND (“localization”)

In the defined search string, the Boolean operators were employed to link all the selected terms with each other [19]. The “OR” operator was used to link synonyms or related terms and the “AND” operator was used to link the major terms.

3.3.3 Paper Selection

Paper Inclusion and Exclusion Criteria

Several criteria for including and excluding papers (based on the titles, abstracts, and full-text readings) were considered to decide whether a publication is relevant to our study or not, as follows:

Inclusion criteria:

- Publications related directly to the topic of this survey study. This is ensured by reading the title of each obtained paper. When the title reading was not enough, the abstract or full-text reading has also been applied. It is worth mentioning that in full-text reading/filtering, we eliminated those papers that do not talk about issues or we could not use them to identify issues and challenges.
- Papers published online from 2002-2021.

Exclusion criteria:

- Publications that are not available in English.
- Duplicated publications.

Snowballing

In this survey study, the snowballing technique [151] was also used to reduce the risk of missing some relevant papers. The newly found papers are then subjected to the paper selection process recursively.

Figure 3.1 shows the paper selection process and its outcome at each stage. In addition, all the papers obtained after applying the paper selection process are listed below:

[168], [132], [6], [155], [34], [43], [176], [175], [5], [54], [55], [1],[69], [129], [18], [4], [142], [183], [11], [36], [156], [165], [174], [92], [141], [98], [99], [106], [127], [67], [25], [76], [49], [61], [20], [143], [23], [118], [10], [41], [38], [139], [84], [79], [104], [112], [85], [9], [17], [71], [48], [150], [62], [169], [73], [13], [46], [59], [90], [16], [77], [158], [144], [149], [8], [47], [21], [140], [152], [74], [109], [178], [12], [88], [173], [154], [81], [172], [50], [100], [180], [115], [147], [146], [32], [96], [91], [26], [138], [14], [167], [163], [31], [80], [103], [94], [170], [70], [86], [171], [87], [72], [24], [162], [160], [57], [35], [82], [30], [137], [83], [56], [164], [133], [42], [145], [107], [15], [64], [33], [45], [39], [68], [37], [157], [117], [182], [179], [177], [93]

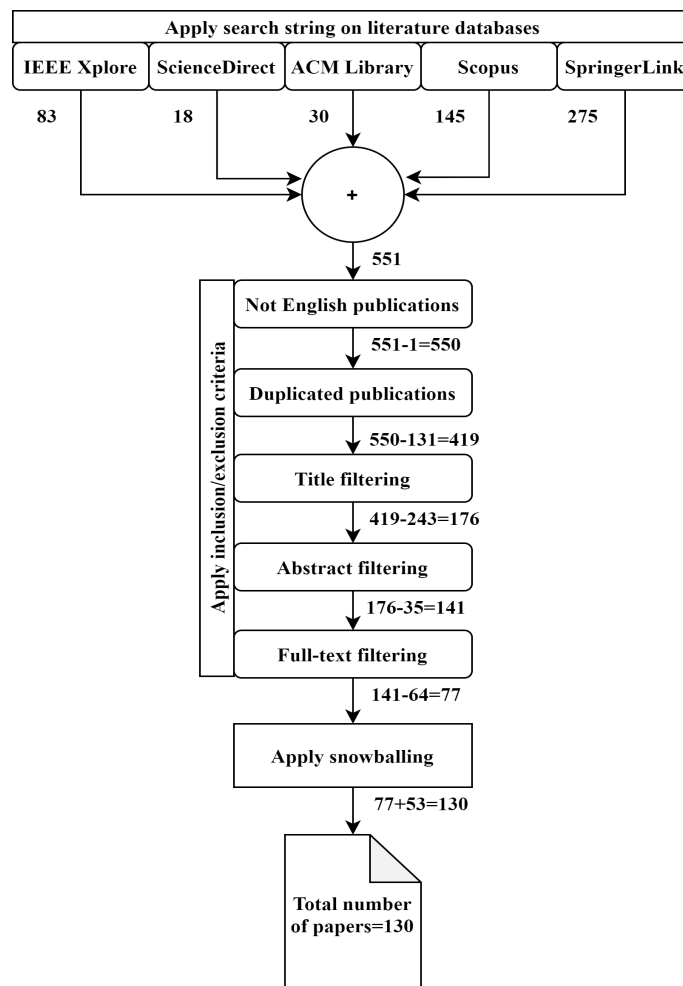


Figure 3.1: The outcome of the paper selection process

3.4 Results

To answer the identified RQ of this survey study, all the related publications were extensively read and analyzed. Thus, several challenges and issues posed by SBFL have been identified alongside many directions, as shown in Figure 3.2, classified into several categories, and then discussed, as follows.

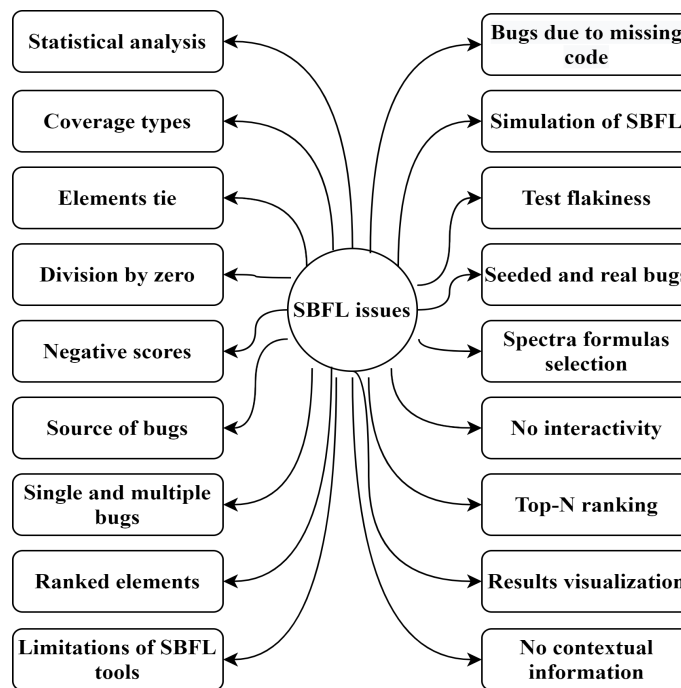


Figure 3.2: Challenges and issues of SBFL

3.4.1 Statistical Analysis:

In SBFL, statistical analysis is used to correlate program elements with failures [5], where similarity formulas from the statistics and data mining domains are used to measure the likelihood of a program element being faulty. The issue here is that software testers and researchers are not statisticians. Worse yet, most of them do not have access to statisticians or cannot afford to send their data to one. As a result, they often select SBFL formulas without statistical justifications. Another issue is that they evaluate their contributions using statistics to demonstrate that their contributions are significantly better than the state-of-the-art. In other words, they use statistics to demonstrate that a proposed new technique locates faults “significantly” better than the state-of-the-art. As they are not statisticians and do not have statisticians readily available, this may lead to incorrect statistical analysis and conclusions about the importance of their SBFL results [54].

To solve these issues, more studies are required to evaluate if some SBFL formulas are statistically significantly better than others. Besides, statistical tools are needed

to help developers evaluate their results. For example, the authors in [54, 55] presented the first such tool called “MeansTest”. The tool automates some aspects of the statistical analysis of results by checking whether the statistical methods used and the results obtained are both plausible. It examines the data under consideration for several properties including normality and distribution. Then, it uses that information to determine which statistical method to use to obtain better results. The tool has been applied to the works presented in the papers at the 6th International Conference on Software Testing, Verification, and Validation (ICST’13). Six papers were discovered to have potentially misstated the significance of their findings because of the selection of inappropriate statistical techniques.

3.4.2 Coverage Types:

Since the granularity of fault localization is determined by the granularity of code coverage, the selection of which coverage type to use for SBFL is crucial, as each coverage type influences the performance of SBFL techniques in one way or another [155]. Program coverage elements can be divided into several common types, as follows:

- **Statement coverage:** Different lines of code can be considered for statement coverage. Thus, the issue is which line of code can be considered the most suitable choice. In [1] for example, all the lines of code in the target program are considered for statement coverage. While in [69], lines of code that are preprocessor directives, variable declarations, and function declarations have not been considered for statement coverage. The number and type of lines of code considered for statement coverage may have a notable impact on the performance (in terms of the average ranks [129]) of any SBFL formula based on the location of the buggy line. Therefore, comprehensive experimental studies have to be conducted to distinguish between different types of lines of code and to analyze their impact on fault localization performance. For example, an interesting investigation could be giving an importance score to each line of code in the target program. Importance scores could be computed via the influence of a specific line of code on the behavior of the target program. However, statement coverage is one of the most used coverage types as it often provides the exact locations of faults [18].
- **Branch coverage:** Here, each one of the possible branches from each decision point is considered for branch coverage. The issue in this type of coverage is that a fault in the condition of an if-then-else may lead to the execution of the else branch in all failed test cases. Thus, ranking the statements in this branch higher than the faulty condition, which is also executed by passing test cases [4, 132, 164].
- **Block coverage:** Here, several program statements are considered for block coverage [142]. Block size is determined by the compiler and it depends on the

program size and structure. The standard size of a block is 5-7 statements [97]. Using statement coverage may result in ties of scores between the statements within the same block of a program. While this issue is reduced in the block-based spectra coverage. However, the issue here is which types of statements can be considered as a single block.

- **Function coverage:** Function (or method)-level granularity can also be employed as a program spectra or coverage type. Compared to statement-level granularity, it has several advantages [11, 183] including providing global contextual information and understandability [145], scalability, reduced tied program elements, and it was used as the basic program element for fault localization research [18]. However, the number of statements in some functions is huge sometimes. Thus, it would not be easy to locate a faulty statement in such functions. Because of the aforementioned advantages, we selected this coverage type as the basic program element for our experimental studies.
- **Data flow coverage:** This is about how variables are defined and then used in a target program. Also, it concerns the relationships among them. Data flow coverage provides more details than the standard coverage types but it requires more execution and memory overheads during test cases execution [117].

3.4.3 Elements Tie:

In SBFL, program elements are ranked in order of their suspiciousness from the most suspicious to the least. To decide whether an element is faulty or not, developers examine each element starting from the top of the ranking list. To help developers discover the faulty element early in the examination process and with minimal effort, the faulty element should be put in the highest place in the ranking list. However, ranking only based on the suspiciousness scores computed by SBFL formulas causes an issue called elements tie [36].

Elements tie means having a similar suspiciousness score for more than one program element in the target program [43]. Tied elements are usually ranked based on three approaches [156] (see Section 2.8.1).

Ties among program elements can be divided into two types [165], as follows:

- **Non-critical ties:** This type refers to the case where only non-faulty elements are tied together for the same position in the ranking list. Here, if the tied elements have a higher suspiciousness score than the actual faulty element, then every element will be examined before finding the faulty element. On the other hand, if the tied elements have a lower suspiciousness score than the actual faulty element, then the faulty element will be examined before the tied elements. Thus, there is no need to continue examining the ranking list. In both cases, the internal order in which the tied elements are examined does not affect the performance of fault localization in terms of the number of elements that must be examined before finding the faulty element.

- **Critical ties:** This type refers to the case where a faulty element is tied with other non-faulty elements. In this type, the internal order of examination affects SBFL's performance. It is worth mentioning that critical ties are not a rare case in fault localization. Besides, a significant portion of the elements in the program under consideration might be critically tied. Therefore, there is a need for tie-breaking strategies to address this problem.

It is quite frequent that ties include faulty elements and it is not limited to any particular SBFL technique or target program. Such elements are tied for the same position in the ranking list. Also, it indicates that the used technique cannot distinguish between the tied elements in terms of their likelihood of being faulty. Thus, no guidance is provided to developers on what to examine first [72]. In addition, the greater the number of ties involving faulty elements, the more difficult it is to predict at what point the faulty element will be found during the examination process. Therefore, tie-breakers are required to address this problem [174].

We handled the ties problem in the ranking list of program elements by proposing a novel tie-breaker that uses information extracted from method calls (see Chapter 4).

3.4.4 Division by Zero

There is always a possibility of the denominators of some SBFL formulas having zero. As a result, error messages are produced. For example, when the formula "Overlab" is applied to the information presented in Table 2.1; the error message "Division by Zero" is printed for each program statement. Therefore, we considered the value zero as a score for each statement as can be noted from Table 2.2. To overcome this issue, several possible solutions have been proposed in the literature as follows:

- Considering zero as a result. The value zero is assigned to each program element in which its denominator is zero [69, 92, 141].
- Adding a small fixed constant such as 10^{-6} to the denominator [98, 99].
- Adding a larger value such as the number of tests plus 1 to the denominator. Such a value is larger than any value which can be returned with a non-zero denominator [99, 141].

However, the aforementioned solutions may introduce undesired issues as well. For example, more program elements will have the same suspiciousness score in the ranking list, forming new ties. Often, scores generated using these solutions are not considered by the researchers in the literature. Simply, they are removed from the ranking list and thus not displayed to the developer. However, more studies are required here to answer what is the rate of program elements having the same score using these solutions and whether a faulty element could be within these elements or not.

3.4.5 Negative Suspiciousness Scores

In SBFL, most of the formulas used to compute suspiciousness scores of program elements produce positive scores. However, few formulas (e.g., “Wong II” and “Goodman”) produce both positive and negative scores. This may cause an issue when a weighting method is applied to the generated scores for some valid reasons. For example, the final score of each element in the whole program or a group of elements can be multiplied by a weighting value to determine which element is more important than others based on a reason, such as which one contributes mostly to the behavior of the program, which one appears more in failed test cases, which one appears less in passed test cases, etc. Therefore, applying a weighting method to the negative scores produced by such formulas will change the original rank order of the scored elements. In other words, the rank order of the suspicious elements after applying a weighting method will be different from the rank order of the same elements before applying a weighting method.

To illustrate this, consider the scores produced by the Wong II formula in Table 2.2. It can be noted that statements 4, 5, and 10 are assigned with the same score (i.e., -1) and the same rank order (i.e., 3rd); but we would like to consider statement 4 as the most suspicious element because it has been executed by a failed test case while the two other statements were not. So, we decided to apply a simple weighting method that multiplies the score of statement 4 by the weighting value 0.9 (more suspicious) and the scores of statements 5 and 10 by the weighting value 0.1 (less suspicious). The results of applying our weighting method will decrease the score of statement 4 and thus put it in the worst position in the ranking list (i.e., 5th rank instead of 3rd); while it does the opposite with both statements 5 and 10. A possible solution to this issue is to apply the weighting method to each score generated by such formulas; then the absolute of each score has to be taken before ranking the scores.

3.4.6 Source of Bugs

In the software development process, it is common to break the code of a program into several source code files. For example, putting the functions in one file and the classes using these functions into another. This practice is useful for structuring source code files and for reusing existing code. However, it also has its drawbacks in the context of software fault localization. In Figure 3.3 for example, File B includes two functions, *LessThanFunction()* and *GreaterThanFunction()*, with a bug in statement 6 of the first function, it should be $m = x$ instead of $m = y$. These two functions have been imported into File A. Thus, File B propagated its bug to File A. As a result, File A will also have a bug in statement 4. When File A is tested using the statement granularity/coverage level, it will show that it has a bug in statement 4. The developer will then examine statement 4 to find out that it calls a function from File B. The issue here is that he/she will not be able to know which statement in the called function caused the bug to be fixed.

In the literature, there is a lack of experimental studies that try to distinguish

between propagated/imported bugs and not propagated bugs. Therefore, it would be very useful to study this issue in many directions, such as deciding if a bug is imported or not, specifying where it is imported from, how to locate it in its original place, and measuring its impact on the whole fault localization performance and process.

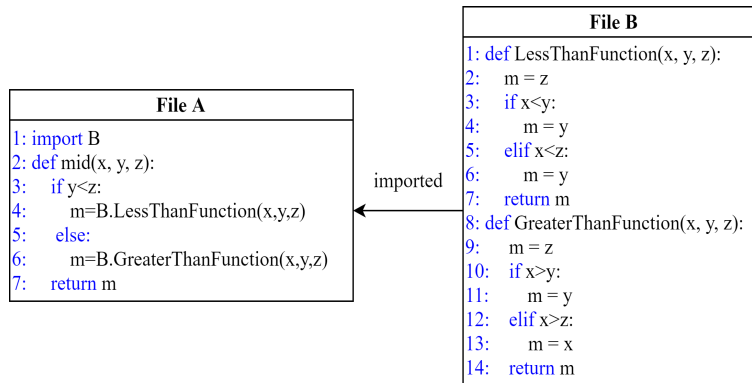


Figure 3.3: Fault propagation

3.4.7 Single and Multiple Bugs

In general, program failures are caused either by a single bug or multiple bugs [175, 176]. A single-bug problem is where all the failures of test cases are caused by just one bug. In other words, whenever a test case fails, the same buggy element should have been executed in that test case. On the other hand, a multi-bug problem is where the failures of test cases are caused by more than one bug. Sometimes, a bug could be in a preprocessor directive or an initialization element that is used at multiple places in the target program. This issue shows that the target program has multiple bugs. Another issue here is that as the bug is in a statement (e.g., initialization statement) that is executed by all passed and failed test cases, that buggy statement is mostly not to be ranked high; making it difficult to be identified [4].

To address this issue, further studies are required to know whether a program has multiple different bugs or a single bug element that is used at multiple places. In the case of the latter, it would be useful to specify the location of the first appearance of the bug and consider it in the fault localization process while ignoring the other places where it has been used. It is worth mentioning that many SBFL techniques are designed for programs with a single bug only. Therefore, it would be interesting to study the impact of multiple bugs on the performance of SBFL. A good starting point on this is what has been performed in [160], where an empirical investigation on multiple-fault versions from different open-source programs has been conducted to study the negative impact of multiple faults on SBFL and to explore the fundamental causes of this negative impact. Also, it has been found that some SBFL formulas are more robust to multiple faults and showed the best performance among all others.

In general, pure SBFL is not always sufficient for effective fault localization in multi-fault programs [37, 42]. Other ways to address the issue of multiple bugs in a program are to design novel suspiciousness formulas as in [133] or to divide the failed test cases into different clusters. The test cases in a cluster fail due to the same bug. In other words, each test cluster represents a different bug. Then, the failed test cases in each cluster combined with all passed test cases are used to localize only a single fault as in [39, 68, 157].

3.4.8 Ranked Elements

The Ranked List of Elements is Huge

Mostly, a large number of program elements are included in the ranking list generated by SBFL techniques [4, 106]. This is not preferable for the following main reasons:

- The more ranked elements, the more ties are produced as many program elements exhibit the same execution patterns.
- It may increase the number of elements having a suspiciousness score of 0 due to the issue of division by zero.
- A huge number of elements that are unrelated to suspects of a bug get considered in the ranking list.

Possible ways to address this issue are either combining the ranking with other suspiciousness factors derived from the testing and program elements contexts, such as using program slicing or reducing the length of the target programs via applying code optimization and transformation techniques. To illustrate this, consider a Java function called *match()* which takes two inputs *s* and *w*, and returns back whether the sentence *s* contains the word *w* or not. The function code is written in two ways, an unoptimized version of the code with a bug in statement 9 and an optimized version of the code with the same bug in statement 8, as shown in Figures 3.4 and 3.5. The unoptimized code of the function has 15 statements which all will be included in the ranking list; while the optimized code of the same function has only 10 statements to be included in the ranking list.

Table 3.2 presents the spectra and test case information of all the statements alongside their suspiciousness scores before optimizing the code, and Table 3.3 presents the same information but after applying code optimization. It can be noted that code optimization reduces the number of ranked statements. We can see that it eliminates some ties completely and reduces some others. Additionally, no statement has been scored with the value 0.

However, it would be interesting to study code optimization and its impact on the performance of SBFL in many directions. Many code optimization techniques reduce the length of programs without changing their outputs. Thus, the effects of these techniques have to be investigated experimentally and their feasibility has to

```

1: boolean match(String s, String w)
  {
2:   boolean result=false;
3:   int count=0;
4:   int i=0;
5:   while(i!=s.length())
  {
6:     if(s.charAt(i)==w.charAt(count))
  {
7:       count++;
8:       if(count==w.length())
  {
9:         result=false;
10:        break;
  }
  }
  else
11:  if(count!=0)
  {
12:    i--;
13:    count=0;
  }
14:  i=i+1;
  }
15:  return result;
  }

```

Figure 3.4: *Running example – unoptimized code*

```

1: boolean match(String s, String w)
  {
2:   boolean result=false;
3:   int wordLen=w.length();
4:   int diffLen=s.length()-wordLen;
5:   for(int i=0;i<=diffLen;i++)
  {
6:     String str = s.substring(i,wordLen+i);
7:     if(w.equals(str))
  {
8:       result=false;
9:       break;
  }
  }
10:  return result;
  }

```

Figure 3.5: *Running example – optimized code*

be reported with evidence. A possible solution to the issue of the suspicious elements are not related logically is to group them into different logically related categories to at least understand why these elements were considered suspicious. Software module clustering could be employed in this respect as a potential solution to this issue. More studies are required to evaluate the usage of other potential factors and to measure their impacts on SBFL's performance. Here, we list out some factors that we believe will have a positive impact on the ranking effectiveness, as follows:

Table 3.2: *Running example – spectra and four counters before optimization*

	T1	T2	T3	ef	ep	nf	np	Scores
1	1	1	1	1	2	0	0	0.5
2	1	1	1	1	2	0	0	0.5
3	1	1	1	1	2	0	0	0.5
4	1	1	1	1	2	0	0	0.5
5	1	1	1	1	2	0	0	0.5
6	1	1	1	1	2	0	0	0.5
7	1	1	0	1	1	0	1	0.67
8	1	1	0	1	1	0	1	0.67
9	1	0	0	1	0	0	2	1
10	1	0	0	1	0	0	2	1
11	0	1	1	0	2	1	0	0
12	0	1	0	0	1	1	1	0
13	0	1	0	0	1	1	1	0
14	1	1	1	1	2	0	0	0.5
15	1	1	1	1	2	0	0	0.5
R	1	0	0					

Table 3.3: *Running example – spectra and four counters after optimization*

	T1	T2	T3	ef	ep	nf	np	Scores
1	1	1	1	1	2	0	0	0.5
2	1	1	1	1	2	0	0	0.5
3	1	1	1	1	2	0	0	0.5
4	1	1	1	1	2	0	0	0.5
5	1	1	1	1	2	0	0	0.5
6	1	1	1	1	2	0	0	0.5
7	1	1	1	1	2	0	0	0.5
8	1	0	0	1	0	0	2	1
9	1	0	0	1	0	0	2	1
10	1	1	1	1	2	0	0	0.5
R	1	0	0					

- The sequence, number, and coverage of executing failed test cases.
- The importance of each failed test case in the used test suit.
- The importance of each element in the target program. For example, the statements that directly have an impact on the program's output should be given more importance than others.
- Using various software metrics (e.g., the complexity of functions, relationships, elements types, etc.) to group the elements sharing similar metrics into different categories and then relate them to the faulty element.
- All the elements near the faulty element may be given more importance than

others when being ranked.

- Using the union of dynamic slices of failed test cases to reduce the number of elements included in the ranking list.

The Ranked List of Elements is Practically Arbitrary

In SBFL, the ranked list of program elements is formed as follows: you can get a statement from function $a()$, then another one from function $b()$, and so forth. As a result, the ranking list suggested by SBFL is not followed linearly by developers [106] because they have trouble understanding the context of the bug since they are only given each bug location in isolation. Instead, they examine the statements that were ranked high in the ranking list and then look for the location of the actual fault in the surrounding function, class, or file. This suggests that pointing developers towards good starting points with SBFL is more important than only improving the ranking of program elements in the ranking list. Thus, many researchers worked on this aspect of the problem too [107, 127].

In [107], the authors proposed a technique that reports the most suspicious program regions instead of a single program element that is likely to be faulty. In other words, each faulty element is reported together with its context. This is useful because the contexts can assist developers in identifying and comprehending the infection flow of each faulty program element. This is performed by extracting the execution traces of each program element in different failed and passing runs. Then, a final execution sequence for each element is formed as a graph that represents the faulty element and its context. Figure 3.6 shows program elements and their execution contexts within suspicious regions.

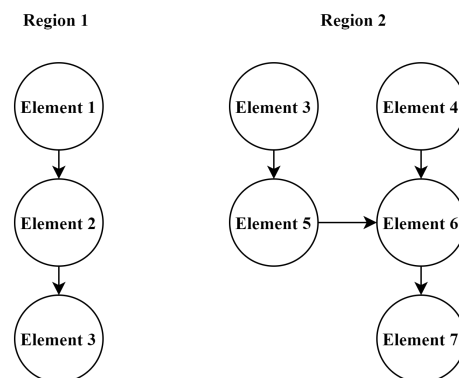


Figure 3.6: *Suspicious program regions*

To address this problem, we also suggested a hierarchical ranked list of elements (see Chapter 7). This is accomplished by placing all of each function's statements under the name of the corresponding function, followed by all of each class's functions under the name of the corresponding class. The classes are then sorted according to their suspiciousness scores, followed by the functions, and lastly the statements.

By using this approach, the user can get more helpful information about the faulty elements across different levels of granularity.

3.4.9 Limitations of SBFL Tools

SBFL techniques require suitable tools to automatically collect spectra data and testing information from the target programs [155]. However, the currently available tools [20, 23, 25, 49, 61, 67, 76, 118, 143] suffer from some limitations [10], as follows:

- Mostly, they only collect abstract and trivial testing information, such as whether a program element is executed by a specific test case.
- Some of them collect more and different types of information (e.g., control flow and data flow) that may be time-consuming, not well scalable for large-scale target programs, and unusable in practice.
- Most of them are developed for programs written in Java or C/C++ programming languages. This is because these languages have been used widely in the past decades compared to other languages. Another possible reason is that the choice of programming languages represents the target industries of each company. For example, companies providing tools for embedded and real-time software vendors focus more on supporting C/C++ [168]. Tools for helping Python developers in their debugging process have not been proposed by the researchers previously. Therefore, tools that target programs written in Python, which is considered one of the most popular programming languages, are extremely required to be proposed and developed.
- They have the issue of inaccuracies in their results. The inaccuracy of a tool's recorded coverage data can lead to various problems. For example, false trust in the result may be introduced by a code element that is falsely reported as covered in a tool and not covered in another tool. Therefore, to guide how to avoid the inaccuracies of the tools, further studies are needed. This can then help testers to determine the degree of risk of measurement inaccuracies on the performance of fault localization [41].
- Proposing and developing tools or plug-ins for specific IDEs is considered a practical limitation of usage as not all developers use the same Integrated Development Environment (IDE) and many developers use more than one IDE. Developers do the debugging during/within the development phase itself but this is not always true and it is not a preferred practice. Therefore, developing standalone software tools that do not depend on a specific IDE is a good option in this respect. Perhaps the best option is to have some generic tool that can be invoked from the command line or to use some APIs and then develop different plugins for various development environments that are calling this generic tool.

In order to make SBFL tools more useful and practical, they should be developed with some important features [168], as follows:

- A user-friendly graphical interface is a crucial feature for users nowadays as such interfaces act as the gates into using software systems interactively and efficiently [53]. Thus, a proposed fault localization tool should also be run in a Graphical User Interface (GUI) mode besides a command line mode to meet the requirements of different users. For example, developers usually like to use the GUI mode but the integrators usually like the command line mode.
- The results generated from a tool should be stored in various file formats according to the user's needs (e.g., CSV, XLS, or JSON). As a result, the results will be useful for further processing or even for other testing tools.
- A tool should provide control to the user to change the settings and configurations of its functionality, such as where to store the results, which task should be automated, which results should be displayed first, etc.

We contributed to solving some of the aforementioned issues by providing two supporting tools for SBFL, namely “CharmFl” and “SFLaaS” (see Chapter 7). These tools target Python developers and provide them with many useful features to help them debug their programs.

3.4.10 Bugs Due to Missing Code

Generally, software bugs appear due to wrongly written code (e.g., using a wrong variable instead of another one or using a wrong arithmetic operator instead of another one) or due to missing code (e.g., missing an element that performs a specifically required operation or missing a required conditional element) [34]. In some open-source projects, it has been found that missing code faults form the majority of the total faults in these projects [38].

Locating a bug that is introduced by a missing code is a challenging task in SBFL. This is because the code responsible for the bug is not in the program and SBFL is designed to locate a faulty element, the execution of which triggers failure [24]. However, a missing code will have an impact on some other elements in the target program. For example, some elements pose undesired behavior, get executed before other program elements, or get executed where they should not be. This issue could be addressed by analyzing the undesired behavior or the unexpected execution of the elements impacted by a missing code. Such elements could be identified by their high suspiciousness scores. Thus, the high scores of some elements may indicate that some elements in their neighborhood (i.e., preceding or succeeding elements) are missing [156, 162]. However, more work is needed to propose techniques to address the issue of bugs caused by missing code.

3.4.11 Simulation of SBFL

Implementing and using SBFL requires target programs, test cases, and different types of coverage data. Providing these requirements is challenging for many reasons, as follows:

- Executing test cases on the collected target programs requires that all the programs be provided with proper execution environments. Some programs depend on external libraries to be executed properly. Many others require some configuration settings to be set.
- There is a lack of tools that extract various spectra data from the target programs.

Therefore, advanced SBFL simulation tools are very useful to be proposed and implemented to support researchers in this respect [131]. They should be able to simulate various program structures and behaviors, relationships among elements, different coverage types and test cases, different numbers and types of faults, and calculate suspicion scores using various ranking formulas. Such tools can be used to validate new ideas or concepts before starting the actual and concrete experiment and development.

3.4.12 Test Flakiness

SBFL depends on the results of executing several test cases. Sometimes, a test case may pose an issue called “test flakiness”, which refers to a test case with a non-deterministic result. In other words, sometimes it passes and sometimes it fails on the same code depending on unknown circumstances [139]. This issue negatively impacts the effectiveness of SBFL techniques as it provides misleading signals during the fault localization process [84]. It has been found that the flakiness of individual test cases influences fault localization scores and ranks, and that some SBFL formulas (e.g., “Tarantula”) are more sensitive to this issue than others (e.g., “Ochiai” and “DStar”).

The dominant approach when addressing this issue is to detect and then remove all the identified flaky test cases from the test cases’ execution. However, it has been found that the number of flaky test cases is sometimes so high that removing them is not considered a practical solution [79]. Therefore, proposing new approaches which give good performance even with the existence of flaky test cases is preferable. Flaky test cases can be detected in many ways, as follows:

- Re-run a test case several times after it has failed. If some re-runs pass, then the test case is considered a flaky one. One issue here is how many times a failed test case has to be re-run. Different studies used different numbers. For example, in [104], each test case has been re-run 10 times. In [112], 30 times. In [139], 100 times. In [85], 4000 times. In [9], 10000 times, and even with this huge number of re-runs, the authors interestingly found that

some of the previously identified flaky tests were still not detected. The re-run approach suffers from several issues [17] such as: (a) flaky test cases are non-deterministic. Therefore, there is no guarantee that re-running a flaky test case will change its outcome; (b) there is no guidance for how many times a failed test case has to be re-run to maximize the likelihood of considering it flaky; (c) the performance overhead of re-runs scales with the number of failed tests.

- Monitor only the coverage of the most recent code changes rather than the entire target program and mark as flaky any newly failed test case that did not execute any of the changes without re-running and with minimal runtime overhead. In other words, a test case is considered flaky if it passes in the previous version of the code but fails in the current version [17].

3.4.13 Seeded and Real Bugs

Artificial faults (also called seeded faults) are made when a researcher places a fault in a program source code to intentionally break its functionality. This is performed with the hope that the SBFL techniques under study will be able to identify the location of the seeded fault in the modified source code.

Seeded faults are often used to replicate real fault behavior, especially when the real faults cannot be reproduced due to many reasons including technical ones, or because they are not available for programs written in certain programming languages. Also, they can be used to solve the issue of unbalanced test suits in real fault datasets, such as Defects4J [71] for Java programs, BugsJS [48] for JavaScript programs, and BugsInPy [150] for Python programs, where the passed test cases are much more common than the failed test cases. It is worth mentioning that seeded faults are widely used in multiple fault localization studies. However, the issues with these faults are, as follows:

- They may be picked arbitrarily.
- There is a potential for bias in the selection of the faults.
- They may not be representative of real industry faults.

To overcome these issues, it is recommended to use real faults, such as the faults presented in Defects4J and BugsInPy datasets, or to seed faults in well-known and complex software systems and provide all the created faulty versions publicly online, which legitimizes the experimental results by reducing bias and enhancing result generalization [176].

As mentioned before in Chapter 2, in this thesis we used real faults from well-known faulty programs to avoid some of the aforementioned issues. Also, it is worth mentioning that we addressed the issue of unbalanced test suites where the number of passed tests is much higher than the number of failed tests and many SBFL formulas treat passing and failing tests equally by proposing to use *the importance weight* to emphasize the factor of failing tests in SBFL formulas to enhance the overall effectiveness (see Chapter 5).

3.4.14 Spectra Formulas Selection

There are many SBFL formulas proposed in the literature. However, still, there is a lack of guidance on how to select the right formula for a specific purpose. In [62], SBFL formulas were divided into three groups based on how the formulas of each group are affected by the number of failed test cases. It has been found that some formulas (e.g., “Ochiai” and “Tarantula”) are more sensitive to the number of failed test cases than others. In [169], several formulas generated by genetic algorithms have been evaluated, and it has been found that the “Genetic Programming (GP)13” formula is one of the best-performing formulas of its kind. In [139], it has been found that some SBFL formulas (e.g., “Tarantula”) are more sensitive to the issue of test flakiness than others.

However, many other aspects are not yet evaluated, for example, which formula is more sensitive to the tie issue or which formula performs better with a specific type of fault. In Chapter 4, we analyzed several well-known formulas and measured their outputs regarding both critical and non-critical ties. We also introduced many new formulas that can enhance SBFL’s effectiveness by breaking ties among program elements or by putting more faulty elements in higher-ranked Top-N categories (see Chapter 6).

It is worth mentioning that multiple formulas can be combined into a single new formula. The resulting formula is called a hybrid formula; which combines the advantages of other existing formulas that have been used in the combination. As a result, a hybrid formula should outperform other existing formulas as in [73]. To produce an effective hybrid formula, more experimental studies are required to be conducted to understand the behavior and characteristics of each existing formula, as each has its strengths and weaknesses at the same time. Thus, providing a detailed guideline with experimental evidence to help researchers select the right formulas for the combination will help a lot in this respect. Also, the computed suspiciousness is different for every formula according to its peculiarity for the same target program. Thus, it would be interesting to investigate the relationship between the used formula and the target program. This may lead to the introduction of some improvements in the combination process. All the aforementioned issues are possible avenues worthy of further exploration.

3.4.15 No Interactivity

Often, SBFL techniques compute the suspiciousness scores of program elements without involving the user. In other words, only the statistical analysis of program spectra is used for this purpose. Thus, the user’s previous knowledge about the program under test is not utilized to improve the fault localization performance [13]. This issue can be addressed by involving user interactivity. Involving the user and considering their feedback on the suspicious elements and their ranks can help to re-rank them, thus improving the fault localization process.

Figure 3.7, which is adapted from [46], shows the difference between the static SBFL (i.e., without user interactivity) and the interactive SBFL (i.e., with user inter-

activity).

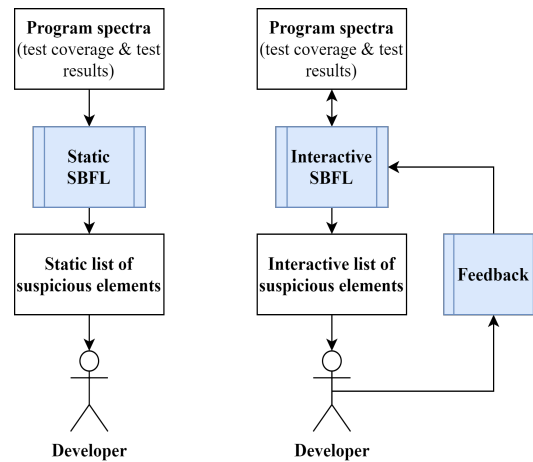


Figure 3.7: *Static vs. interactive SBFL*

In [59], the authors proposed and implemented an approach called “Interactive Fault Localization (iFL)” to support user interactivity in the SBFL process. Their approach allows the user to interact with the output of the SBFL process based on their understanding of the system elements and their contexts by considering the following three feedback actions: (a) the user decides that a proposed suspicious element is faulty. Thus, the SBFL process will stop as the faulty element is found; (b) the user decides that a proposed suspicious element and its context are not faulty. Thus, it can be given low importance and then moved lower in the ranking list; (c) the user decides that a proposed suspicious element is not faulty but its context is suspicious. Thus, it can be given high importance and then moved higher in the ranking list.

In [46, 49], the authors also proposed an interactive fault localization approach that leverages simple user feedback. The user can interact with their approach by labeling a suggested suspicious element as faulty or not. Following that, the proposed approach utilizes such simple user feedback and re-orders the rest of the suspicious program elements based on that, intending to put truly faulty elements higher in the ranking list.

In [90], the authors proposed an approach called “Enlighten” which is similar to the previous approach except that it uses dynamic program slicing to form a Dynamic Dependence Graph (DDG) for every failed test in the test suite. In the DDG, nodes represent occurrences of statements in the program, whereas edges represent dynamic (data or control) dependencies between these statements. This information will then be used to create queries for the user to interact with. Each query consists of a method invocation, together with its input and output values, which the user can mark as correct or not. This approach is also iterative and in each iteration, it updates the debugging data and the ranking list based on the user feedback until the fault is found.

In [16], the authors proposed an interactive approach to use user feedback about the correctness of a set of statements to estimate the number of coincidentally correct test cases (those that execute faulty statements but do not cause failures).

Despite the attempts to propose and improve interactive fault localization approaches, many issues are still not addressed comprehensively in the literature. For example, more studies are required to investigate the effectiveness of different proposed approaches and the comparison among them. Performing user studies to evaluate the usability of the tools implemented in this context is also required. It would be interesting to investigate cases when developers or users make the wrong estimation and give incorrect feedback due to mistakes or not being quite familiar with the faulty program as they are not the actual developers of it. This could be addressed by proposing new methods to allow users to roll back their feedback if they made mistakes. Enabling users to provide multiple feedback at the same time rather than one by one following the recommended list, especially in scenarios where multiple bugs exist is also recommended.

3.4.16 Top-N Ranking

Due to the nature of SBFL, a faulty element cannot always be ranked at higher-ranked Top-N categories (i.e., within the acceptable top 10 ranks [18, 77, 158]). This issue is the biggest obstacle to the usefulness of SBFL in practice [144]. It is worth mentioning that many SBFL studies published in the literature specifically addressed this crucial issue compared to the other issues. Therefore, we will list them in Table 3.4 with a brief description of each proposed solution.

It is worth mentioning that all the solutions proposed in this thesis (see Chapters 4, 5, and 6) address this issue and enhance the effectiveness of SBFL by putting more buggy program elements at higher-ranked Top-N categories.

3.4.17 Results Visualization

During testing a program, software developers gather a large amount of testing data. These data can be used for the following two main purposes [67]:

- To identify failures and to help developers locate the faults causing these failures.
- To identify program elements that were not executed by the used test suite. As a result, more test cases can be added to cover these elements.

SBFL uses such data to compute the suspiciousness of program elements under test and often displays them in a table of many fields as in Table 3.5. This form of output helps the users know which program elements are suspicious, their locations in the source files, their suspiciousness scores, and their ranks.

However, there are two main issues with this approach of displaying the results of SBFL, as follows:

Table 3.4: *Proposed solutions to address the Top-N issue*

Solution	Description	Reference
Removing non-faulty elements	Improving fault absolute ranking for SBFL if some non-faulty elements ranked higher were excluded from the ranking list of a target program based on the failed test cases.	[144, 146]
Categorizing program elements	The ranking list of SBFL can be improved if program elements get categorized into “suspicious group” and “unsuspicious group”. Under such categorization, we only need to calculate the risks for suspicious statements, and simply assign the risks of unsuspecting statements as the lowest value.	[163]
Using program slicing	Deleting program elements that have no dependence on faulty elements to improve the precision of locating faults.	[70, 87, 96, 115, 138, 149, 172]
Introducing new ranking formulas	Proposing new risk evaluation formulas that outperform the existing ones.	[8, 50, 86, 91, 100, 152, 154, 167, 170, 171, 173, 177]
Combining existing ranking formulas	Combining multiple formulas into a single formula. The resulting formula is called a hybrid formula that has the advantages of the formulas used in the combination.	[12, 73, 93]
Optimizing test cases	Optimization methods can maximize SBFL’s performance using a minimum (e.g., by removing redundant test cases) or a balanced number of test cases used by SBFL formulas.	[26, 32, 74, 80, 88, 94, 109, 147, 180]
Weighting and prioritizing test cases	The performance of SBFL can be improved by differentiating the importance of different test cases. In other words, not all test cases have the same importance (e.g., some test cases are more important than others).	[15, 33, 45, 64, 179]
Mitigating the impact of coincidental correctness	Coincidentally correct test cases execute faulty program elements but do not cause failures. Such test cases reduce the effectiveness of SBFL. Therefore, removing or reducing such cases can improve the SBFL.	[14]
Increasing failed test cases	Some SBFL formulas may become less accurate if there are very few failed test cases. Therefore, cloning the whole set of failed test cases or adding some more to enlarge them can improve their performance.	[31, 81, 103, 178]

Table 3.5: *Traditional output of SBFL*

Element	Source file	Line number	Score	Rank
1	Processing.py	100	0.98	1
2	Processing.py	150	0.98	1
3	Processing.py	200	0.5	2
4	Processing.py	500	0.3	3

- The huge amount of displayed results is not attractive and difficult to interpret when large-scale programs and test suites are used.
- It causes developers to focus their attention locally rather than providing a global view of the target program. Therefore, there is a need for different approaches that provide users with a global view of the program under test, while still giving access to the local view. This can be achieved by visualizing the whole source code of the program in which each program element is colored according to its state (i.e., executed or not) in the passed and failed test cases.

To address the aforementioned issues, two main visualization approaches for the results of SBFL have been proposed in the literature, as follows:

- The discrete coloring scheme. In this simple scheme, if a program element is only executed by failed test cases, then its color will be red. If a program

element is only executed by passed test cases, then its color will be green. If a program element is executed by both the passed and failed test cases, its color will be yellow. The problem with this approach is that it is not considered very informative because the majority of program elements are in yellow, and the developer is not provided with helpful hints about the location of faults. It is worth mentioning that the red, green, and yellow colors were selected because they are convenient for viewing [67].

- The continuous coloring scheme. This scheme uses colors and brightness to denote how program elements participate in the passed and failed test cases. It colors the elements according to their suspiciousness scores, from higher (red) to middle (yellow) to lower (green) scores. Thus, an element's color can range from red to yellow to green. Then, it presents different brightness levels according to the frequency at which an element is executed by the test cases. Elements more frequently executed are the brightest ones. If a greater proportion of failed test cases execute an element, the element turns red (i.e., highly suspicious as being faulty). The element appears greener (i.e., not likely to be faulty) if a greater proportion of passed test cases execute it. Elements are colored in yellow (i.e., not suspicion nor completely safe) when they are executed by nearly equal percentages of passed and failed test cases. The visualization based on this scheme can be displayed to the user in many forms as shown in Figure 3.8: (a) coloring program elements in the source code itself [23, 61, 67, 118]; (b) visualizing the results as a “Sunburst” [20, 21, 47]; (c) visualizing the results as a “Treemap” [20, 47]. (d) visualizing the results as a “Bubble Hierarchy” [47].

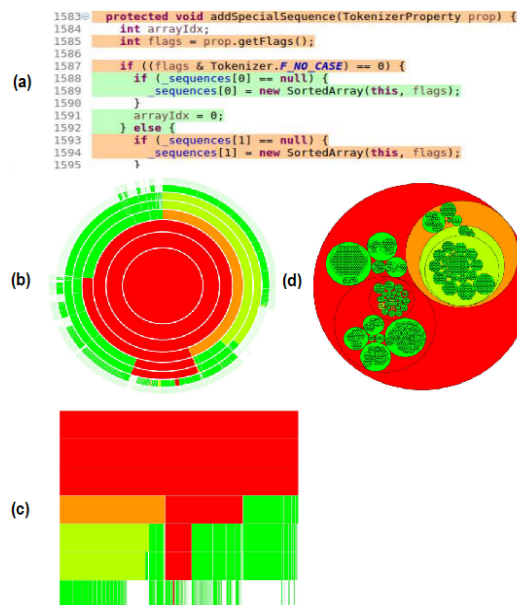


Figure 3.8: Different visualization schemes.

We addressed the issue of displaying SBFL results by providing a tool called “CharmFl” (see Chapter 7) that offers a layered approach to examine the ranking list produced by an SBFL technique. Instead of displaying program elements without showing any relationship among them in terms of which element belongs to which higher granularity level, our approach allows the user to examine the elements at different layers or granularity levels (i.e., classes, functions, and statements). Thus, the ranking list becomes more understandable to the user.

However, more studies are required to propose new approaches or to improve the usability and effectiveness of the existing approaches in many directions. For example, providing a zoomable user interface that lets the user view the results at various abstraction levels is essential, especially for large-scale software systems. Also, providing users with interactive visualization filtering options is an interesting area to be investigated.

3.4.18 No Contextual Information

In SBFL, the ranking is performed only based on the suspiciousness score of each program element. An element with a high score will get positioned at the beginning of the ranking list and vice versa. Thus, SBFL cannot distinguish between program elements that exhibit the same execution patterns. The reason behind this issue is that SBFL techniques leverage hit spectra (i.e., whether an element is executed or not) only as the abstraction for program executions without considering any other useful contextual information [57]. In other words, they represent a program’s behavior as an abstract hit spectra model that cannot capture the semantics of each program element individually [4].

Recently, the authors in [140] addressed this issue by using method call frequency. The frequency of the investigated methods occurring in call stack instances during the execution of failed test cases is used to modify the standard SBFL formulas. The basic idea is that if a method is called multiple times in a failed test case, it is more likely to be faulty than others. Thus, the ef of each formula was changed to the frequency ef . Their experimental results showed that adding this new information to the existing formulas can lead to improvements in the effectiveness of SBFL. However, this approach can only be applied to the formulas that have the ef numerator. Also, it is considered heavy, as it requires tracing the execution of each method call, as caller or callee, in the failed test cases.

In [56, 182], the authors also utilized the relations of software methods. Particularly, they investigated the fault influence propagation implied in method calls. The basic idea is that a caller method often calls several callee methods with complex logical controls, making the complexity of the caller method usually higher than the callee methods. According to the complexity degree, fault influence may often propagate from the callee method to the caller method. Also, the callee’s influence is statistically the most crucial factor, and this influence can be utilized to improve the suspiciousness estimation. From the caller’s perspective, the caller’s suspiciousness evaluation often contains multiple callees’ behaviors and influences. Also, propagat-

ing redundant fault influence reduces the accuracy of the suspiciousness computation. Therefore, the authors extended the basic intuition of SBFL (i.e., a program element executed in more failed test cases is more likely to be faulty) with a hypothesis that the method linked with more and higher suspicious methods is more likely to be the root cause. Based on such intuitions, a heuristic approach called “Fault Centrality” was proposed in this paper to capture the local faulty suspiciousness influence of the callee method on the caller for boosting SBFL.

Method call sequence mining with a slide-window method has been used in [82] to boost the performance of SBFL. The authors achieved this by splitting each method call sequence into different sub-sequences. Then, they computed the hit spectra for each sub-sequence. After that, they took the maximum suspicion score of the sub-sequences that contain the target method as its final score. In [30, 35, 83, 137], the method call sequences have also been employed to highlight the methods that are more often related to other methods in the failed executions of test cases.

We addressed this issue by utilizing contextual information extracted from method call frequency to break ties among program elements and thus enhance the effectiveness of SBFL (see Chapter 4). Also, we used contextual information-based importance weights to improve the SBFL ranks by giving more importance to code elements that are executed by more failed tests and appear in more failing call contexts compared to other elements (see Chapter 5). However, many such studies and other contextual information can be considered to improve the effectiveness of SBFL.

3.5 Contributions

In this chapter, the following points summarize my main contributions to the topic of thesis point I. The results of this chapter were published in [122].

- Providing a theoretical background on the topic of SBFL and its main concepts.
- Conducting a systematic survey study that discussed the papers related to SBFL and the challenges and issues preventing it from being widely used. The results of the systematic survey showed that SBFL still poses many problems that have not yet been addressed despite their importance to the effectiveness of SBFL.
- Categorizing the identified challenges and issues into 18 categories. Addressing SBFL challenges can enhance the performance of SBFL in many directions, as will be seen in the subsequent chapters.

Finally, Figure 3.9 shows the connections between the main issues of SBFL in Chapter 3 and the subsequent Chapters (4, 5, 6, and 7) that address them.

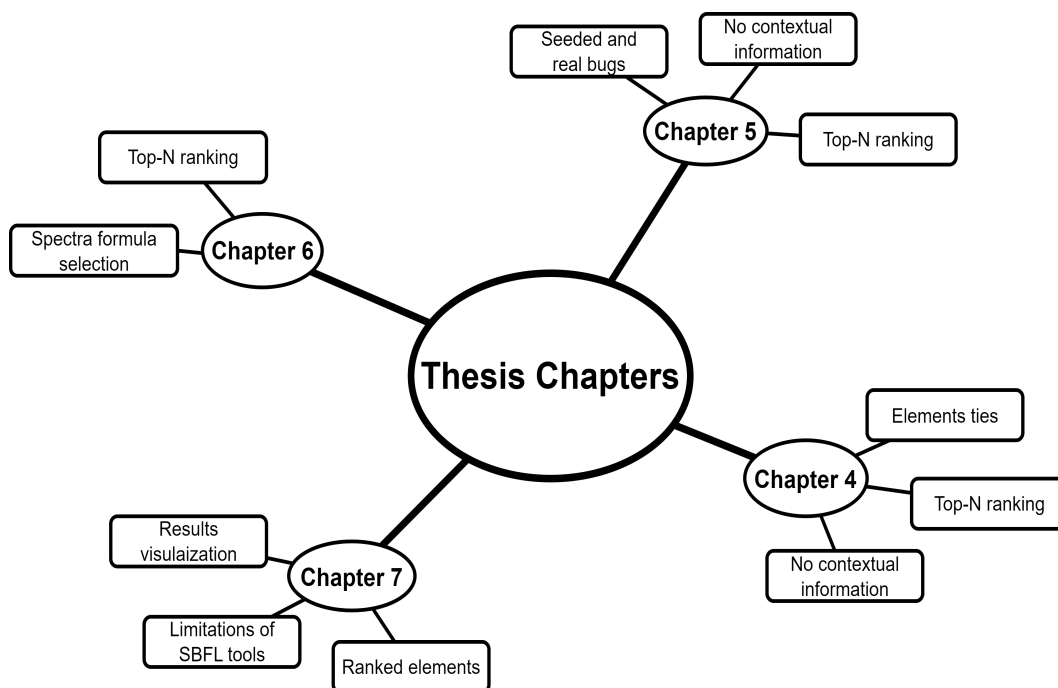


Figure 3.9: Connections between thesis chapters and main SBFL issues

Chapter 4

Tie-Breaking Method for SBFL

4.1 Introduction

In SBFL, program elements (e.g., statements, methods, or classes) are ranked in order of their suspiciousness from most suspicious to least. To decide whether an element is faulty or not, programmers examine each element starting from the top of the ranking list. To help developers discover the faulty element early in the examination process and with minimal effort, the faulty element should be put near the highest place in the ranking list. However, ranking based only on suspiciousness scores inevitably involves a problem called *rank ties* [122]. When different code elements are tied this means that they have the same suspiciousness scores, so they are indistinguishable from each other in this respect. If the faulty element falls within a tie (this is called a *critical tie*) then the overall performance of the SBFL method will be reduced.

Probably none of the known SBFL formulas are guaranteed to produce different scores for all the program elements, hence ties inevitably emerge between the code elements. In fact, as we shall see in this study, ties in SBFL are prevalent regardless of the underlying formula. In this study, we propose a tie-breaking strategy to improve the performance of SBFL by utilizing contextual information extracted from method call chains (our strategy is at method-level granularity, meaning that the basic program element considered for fault localization is a method). Method call chains are the call sequences of methods in the call stack during their executions. Both call chains and call stack traces can provide valuable context to the fault being traced. For example, a method may fail if called from one place and performs successfully when called from another.

The proposed strategy is based on how often a method has been called, directly or indirectly, during the execution of failed test cases *in different contexts*. However, here we do not count all occurrences of a method call but only those that occur in unique call contexts. Thus, repeating sequences of method calls due to, e.g., loops are not considered. The intuition is that if a method is present in many different calling contexts during a failing test case, it will be more suspicious and get a higher rank position compared to other methods with the same scores. The strategy can be

applied to any underlying SBFL formula, and, as we will see, it can favorably break the occurring ranks in the ties in many cases.

We empirically evaluated the approach using 411 real faults from the Defects4J dataset and five well-known SBFL formulas. The obtained results indicate that for all the selected formulas, the call frequency-based tie-breaking strategy can improve localization effectiveness in many ways. For example, it completely eliminated 72–73% of the critical ties over the full dataset. In other cases, it reduced their sizes significantly. Ranks of buggy elements improved by two positions on average. The approach achieved positive movement of bug ranks in most Top-3/Top-5/Top-10 rank categories and in particular, the number of cases where the faulty method became the top-ranked element increased by 23–30%.

The main contributions in this chapter can be summarized as follows:

1. Analysis of rank tie prevalence in the benchmark programs.
2. A new tie-breaking algorithm that successfully breaks critical ties in many cases.
3. The analysis of the impact of tie-breaking on the overall SBFL's effectiveness.

In terms of the concrete research goals, we defined the following RQs for this study:

RQ1 How prevalent are rank ties when applying a selection of different SBFL formulas? In particular:

- How common are rank ties in the Defects4J benchmark and what are their sizes?
- What would be the theoretically achievable maximum improvement if all critical ties were broken?

RQ2 What level of tie-breaking can we achieve using the call frequency-based strategy?

RQ3 What is the overall effect of the proposed tie-breaking on SBFL's effectiveness in terms of global rank improvement?

4.2 Related Works

Many fault localization techniques, in addition to the ones used in this study, have been proposed and discussed in the literature [34, 132, 168] and various empirical studies [62, 183] performed to compare the effectiveness of various techniques. However, systematic research work on the problem of addressing ties in the context of fault localization is still modest. The most related publications are presented in this section.

The authors in [174] proposed a tie-breaking strategy that first sorts program statements based on their suspiciousness and then breaks ties by sorting statements based on applying a confidence metric. The metric is intended to assess the degree of

certainty in a given suspiciousness value. For example, when two or more statements are assigned the same level of suspicion, the suspiciousness assigned to the statements with a higher level of certainty is more reliable. As a result, the corresponding statements are more likely to be faulty.

In [165], the authors presented the most systematic analysis of the problem associated with critical ties (ties with faulty statements) where four tie-breaking strategies were considered and evaluated via experimental case studies. Their results indicated that some of the strategies can reduce ties without having an adverse impact on fault localization effectiveness. Besides, they proposed some other tie-breaking techniques to be studied and evaluated in the future, such as a slicing-based approach for breaking ties.

In [36], the authors proposed a grouping-based strategy that employs another influential factor alongside the statements' suspiciousness. This strategy groups program statements based on the number of failed tests that execute each statement and then sorts the groups that contain statements that have been executed by more failed tests. Afterward, it ranks the statements within each group by their suspiciousness to generate the final ranking list. Thus, the statements are examined firstly based on their group order and secondly based on their suspiciousness. Their results show that ranking based on several factors can improve SBFL's effectiveness. Thus, the grouping-based strategy could be effective in tie-breaking as well.

In [83], the authors employed the idea of utilizing method calls to improve the performance of SBFL. In their proposed approach, they combined method calls and their sequences with program slicing to extract spectra patterns from different contexts that can be used to effectively locate faults compared to only using the standard SBFL formulas.

It can be noted that utilizing method calls to improve the performance of SBFL is not new. However, using method call frequency for tie-breaking is a novel approach that has not been investigated by other researchers previously.

4.3 Evaluation

4.3.1 Subject Programs

Here, we used the single and multiple faulty programs (i.e., 411 faults) of the dataset Defects4J 1.5 (see Table 2.5).

4.3.2 Evaluation Baselines

In this study, five standard SBFL formulas "Confidence", "DStar", "GP13", "Ochiai", and "Tarantula" which are presented in Table 2.7, were used as the baselines to evaluate and compare our proposed method against. The reasons behind this are: (a) there is no other proposed tie-breaking approach that works on the method-level granularity as our method does; (b) our goal was to use contextual information

from program executions only to break ties and not as the underlying SBFL formula for all program elements (this was done in [140]). The authors in [165] used two confidence formulas to break ties and data-dependency among program statements as well, but these approaches are not directly comparable to ours.

4.4 Tie Statistics

In this section, we analyze the existence of rank ties in a set of benchmark programs. Then, we present the properties of ties we obtained before applying our tie-breaking strategy and to which extent they can be reduced.

As mentioned earlier, there is no guarantee that SBFL formulas produce unique suspiciousness scores for all the elements of a program under test. As a result, many elements may share the same scores and get tied with each other. Here, we present brief yet informative statistics on the number of ties that the selected five SBFL formulas produce when applied to the Defects4J dataset (see Table 4.1). It can be noted that all the selected SBFL formulas produce ties across all the target programs. This may indicate different things: (a) ties are not rare in fault localization; (b) ties can be formed regardless of which subject program is under consideration; (c) different SBFL formulas are affected.

Table 4.1: *Number of ties: total and average per bug*

Project	Confidence		DStar		GP13		Ochiai		Tarantula	
	#	avg	#	avg	#	avg	#	avg	#	avg
Chart	3656	146.24	508	20.32	512	20.48	506	20.24	490	19.60
Closure	86181	512.98	19069	113.51	19017	113.2	19043	113.35	19109	113.74
Lang	3430	56.23	185	3.03	188	3.08	187	3.07	187	3.07
Math	12856	123.62	844	8.12	846	8.13	854	8.21	851	8.18
Mockito	3381	125.22	779	28.85	780	28.89	779	28.85	789	29.22
Time	5776	222.15	589	22.65	571	21.96	597	22.96	609	23.42
All	115280	280.49	21974	53.46	21914	53.32	21966	53.45	22035	53.61

Table 4.2 presents the number of critical ties. An interesting observation is that the number of ties is not related to program size. For example, smaller programs may have more critical ties than larger programs as in the case of the “Lang” program (22 KLOC) having more critical ties compared to the “Chart” program (96 KLOC). The average number of critical ties per bug is an important indicator, as it means in essence the probability that a buggy element will be tied (assuming a single-bug scenario).

Table 4.2: *Number of critical ties: total and average per bug*

Project	Confidence		DStar		GP13		Ochiai		Tarantula	
	#	avg	#	avg	#	avg	#	avg	#	avg
Chart	14	0.56	17	0.68	14	0.56	15	0.6	16	0.64
Closure	102	0.61	101	0.60	102	0.61	101	0.60	100	0.60
Lang	20	0.33	22	0.36	20	0.33	21	0.34	22	0.36
Math	65	0.62	69	0.66	65	0.62	65	0.62	65	0.62
Mockito	13	0.48	13	0.48	13	0.48	13	0.48	13	0.48
Time	9	0.35	9	0.35	9	0.35	10	0.38	10	0.38
All	223	0.54	231	0.56	223	0.54	225	0.55	226	0.55

We can conclude that more than half of the bugs (54–56%) are within critical ties, i.e., in most cases, there is at least one method whose suspiciousness score is the same as the score of the faulty method.

The sizes of ties is another important factor when considering the potential improvements by tie-breaking. This can be investigated by looking at the differences between the MIN (best case) and the MID (average case) approaches described in Section 2.8.1. Consider Table 4.3, which shows the number of critical ties for which MIN and MID values are different in columns 2 and 3 (essentially, the critical tie numbers as shown above), and also the sum of the corresponding rank differences (column 4), and its average per critical tie (column 5). Put differently, the double of the average difference is the average critical tie size in the benchmark, which is around 7 methods. The difference between the different formulas is not notable.

It also follows that, ideally, the best improvement we could achieve using a tie-breaking technique is these averages. From Table 4.3, we can see in how many cases, based on critical tie numbers and average tie sizes, there is some possible improvement, so we can use these numbers as a baseline for evaluating our tie-breaking approach in subsequent sections.

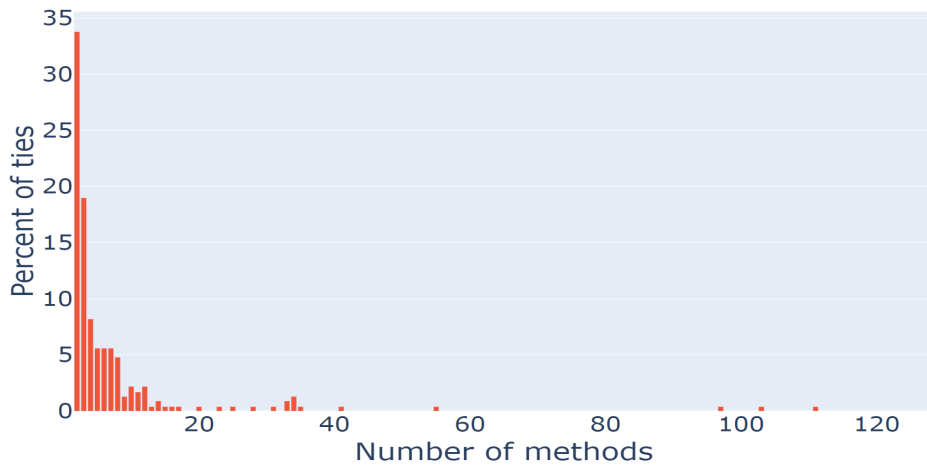
Table 4.3: *Improvement possibilities of critical ties*

	MIN != MID (count)	MIN != MID (%)	Diff.	Avg. diff.
Confidence	223	54.3	758.5	3.40
DStar	231	56.2	795.0	3.44
GP13	223	54.3	799.5	3.59
Ochiai	225	54.7	784.0	3.48
Tarantula	226	55.0	831.0	3.68

We examined the distribution of the critical tie sizes as well, which is shown in Figure 4.1. The X-axis represents the number of methods involved in critical ties and the Y-axis represents the percentage of method groups that have the same tie size. As expected, most ties are relatively small (2–4 elements), 67% of the critical

ties contain 5 or fewer methods, and sizes above 15 are rare (the average is 7.8, the median is 3, and the maximum is 128). Interestingly, there are some outlier cases where the tie sizes are very big, which is the explanation for the relatively large average number.

Figure 4.1: *Size distribution of critical ties*



Answer to RQ1: Overall, it can be said that ties and critical ties are very common (for the bugs in our benchmark). Each of the examined SBFL formulas created critical ties for more than half of the bugs, and on average, the ranks could potentially be improved by around 3.5 positions by eliminating the ties.

4.5 Call Frequency-based Tie-Breaking

In this section, we present the concepts of our proposed tie-breaking strategy and how it works. Then, we present its effectiveness in reducing critical ties when applied to our bug benchmark.

4.5.1 Frequency-based Tie Reduction

In Chapter 2, we introduced the basic concepts of hit-based SBFL. One disadvantage of this approach is that it does not take into account the frequency of executing the program elements, in our case methods, (also known as count-based SBFL). There have been studies that used counts [51, 52], but recent results [3] have shown that these are unable to improve the efficiency of SBFL.

The authors in [140] proposed a technique to replace the simple count-based approach that proved to enhance hit-based spectra while eliminating the problems of naive counts. It is based on replacing the value of ef in the SBFL formulas with

the frequency of different call contexts in the call stack for failing tests, i.e., the “frequency-based ef”. The basic intuition is that if a method participates in many different calling contexts (both as a caller and as a callee), it will be more suspicious. In other words, the frequency of methods occurring in the unique call stacks belonging to failing test cases can effectively indicate the location of the bugs. In this study, we will employ this concept for the purpose of tie-breaking.

To illustrate the basic concept of frequency-based tie reduction, first, we define the *frequency-based SBFL* matrix that replaces the traditional hit-based one. In the new matrix, each element will get an integer instead of $\{0, 1\}$ indicating the number of occurrences of a particular code element in the unique call stacks (effectively, the different contexts) when executing the given test cases.

To illustrate this, assume a simple Java program, which is adapted from [140], that has four main methods (a , b , f , and g) and its four test cases ($t1$, $t2$, $t3$, and $t4$) as shown in Figure 4.2. The program has a bug in the method g .

Figure 4.2: SBFL example: code and test cases

<pre> public class Example{ private int _x = 0; private int _s = 0; public int x() {return _x;} public void a(int i){ _s = 0; if (i==0) return; if (i<0) for (int y=0;y<=4;y++) f(i); else g(i); } public void b(int i){ _s = 1; if (i==0) return; if (i<0) a(Math.abs(i)); else for (int y=0;y<=1;y++) g(i); } private void f(int i){ _x -= i; } private void g(int i){ //should be _x += i; _x += (i+_s); } } </pre>	<pre> public class ExampleTest { @Test public void t1() { Example tester = new Example(); tester.a(-1); tester.a(1); tester.b(1); // failed -> 8 assertEquals(9, tester.x()); } @Test public void t2() { Example tester = new Example(); tester.a(1); tester.b(1); // failed -> 3 assertEquals(4, tester.x()); } @Test public void t3() { Example tester = new Example(); tester.a(1); tester.b(0); assertEquals(1, tester.x()); } @Test public void t4() { Example tester = new Example(); tester.a(-1); tester.a(1); tester.b(-1); assertEquals(7, tester.x()); } } </pre>
A - Program Code	B - Test Cases

To obtain the call frequency matrix, we first build the call tree of each test case during the execution and then we count the call frequency of each method without considering the repetition. Figure 4.3, which is adapted from [140], shows for example the frequency-based spectrum of all methods after executing the test $t1$. It can be

seen that the call stacks of $t1$ are (a, f) , (a, g) , and (b, g) , so the frequency of method a , for example, will be 2 for the test $t1$ as it appeared twice in the call stacks.

Figure 4.3: Call frequency example: call tree and call stacks

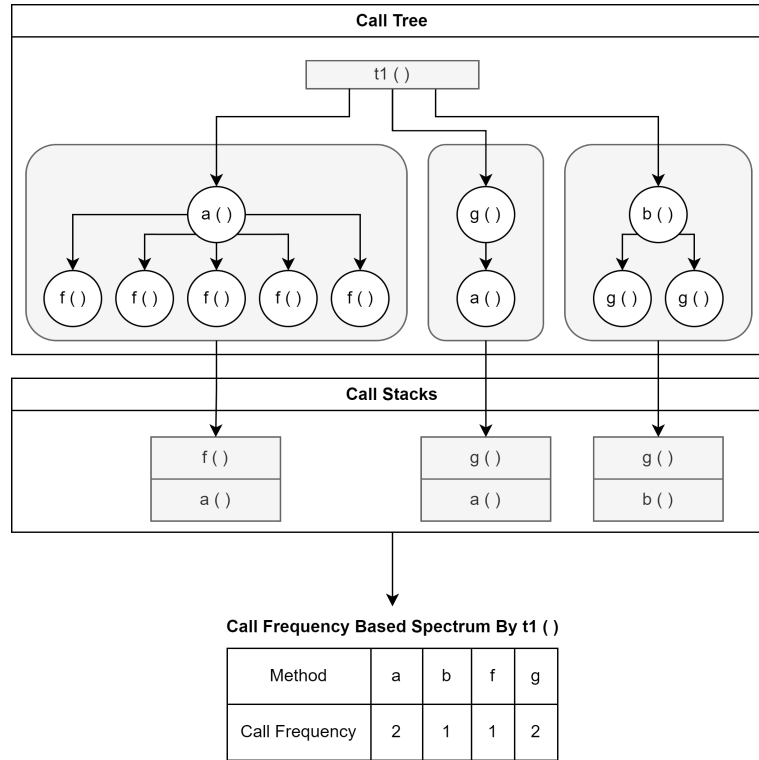


Table 4.4 shows the complete frequency-based matrix for the example after executing all the tests.

Table 4.4: Example frequency-matrix

	a	b	f	g	Results
t1	2	1	1	2	Failed
t2	1	1	0	2	Failed
t3	1	1	0	1	Passed
t4	3	1	1	2	Passed
ϕ	3	2	1	4	

In the next step, we define our metric to be used as a discriminating factor for tie-breaking. The ϕ corresponds to the “frequency-based ef ” and is calculated by summing the corresponding frequency-based values in the matrix for the failing test cases (see Equation 4.1). For example, the ϕ value for method a is 3 by adding the call frequency values of it in the failed tests $t1$ and $t2$. The values for our example are shown in the last row of Table 4.4.

$$\phi(m) = \sum_{t \in \text{failed test}} c_{m,t} \quad (4.1)$$

$m \in \text{methods}$, $c_{m,t} \in \text{frequency-matrix}$

Figure 4.4 shows our tie-breaking process, which can be seen as a two-stage process. In the first stage, we compute the suspiciousness scores of program methods and their ranks by applying different SBFL formulas to the program spectra (test coverage and test results). The output of this stage is an initial ranking list of program methods including critical and non-critical ties. In the second stage, we trace the execution of program methods to obtain the ϕ , i.e., frequency-based ef . This will then be used as a tie-breaker after re-arranging the order of the critically tied methods in the initial ranking list based on the value of ϕ for each method. The output of this stage is a final ranking list, where many critical ties are either eliminated or their sizes were reduced.

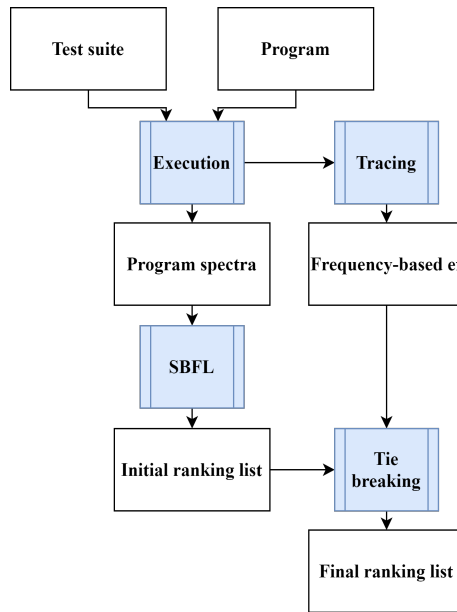


Figure 4.4: *The proposed tie-breaking process*

Our proposed tie-breaking method uses the obtained ϕ call frequency values to break the methods sharing the same score, by putting the methods with higher ϕ higher in the rank. Thus, the most suspicious one will be the method that was called in more different call stacks from failing test cases. The rationale behind using ϕ rather than other contexts (such as the context of method calls in passing test cases) is the intuition that a method is more likely to contain a fault when executed by more failing test cases than passing ones, while non-suspicious when mostly executed by passing tests. However, other different contexts could be considered in the future to investigate their impacts on breaking ties.

The ranks without (columns B) and with tie-breaking (columns A) with our approach for the example are presented in Table 4.5. Ties marked in gray were eliminated with the use of call frequency. As a result, we were able to differentiate between the faulty method g and the other suspicious ones using all of the SBFL formulas (the faulty method got the highest rank in all cases).

Table 4.5: Ranks *Before* and *After* using the tie-breaking strategy

Method	ϕ	Confidence		DStar		GP13		Ochiai		Tarantula	
		B	A	B	A	B	A	B	A	B	A
a	3	2.5	2	2	2	2	2	2	2	2.5	2
b	2	2.5	3	2	3	2	3	2	3	2.5	3
f	1	2.5	4	4	4	4	4	4	4	2.5	4
g	4	2.5	1	2	1	2	1	2	1	2.5	1

4.5.2 Reduction of the Critical Ties

The *Tie-Reduction* metric, which was defined in [165], measures how much a critical tie can be reduced/broken in terms of size. Here, size simply means the number of code elements sharing the same score value, and obviously, the minimum tie size is 2. The goal of any tie-breaking strategy is to reduce the size of the tie or completely eliminate it (when the resulting size is 1). We modified the original definition of this metric to better reflect the actual gain in terms of what portion of the elements in a tie can be eliminated (see Equation 4.2).

$$\text{Tie-Reduction} = \left(1 - \frac{\text{size}_{\text{after}} - 1}{\text{size}_{\text{before}} - 1} \right) \cdot 100\% \quad (4.2)$$

Here, $\text{size}_{\text{after}}$ is the size of a critical tie after applying a tie-breaking strategy and $\text{size}_{\text{before}}$ is the size of a critical tie before applying a tie-breaking strategy. In an ideal case, the critical tie is completely eliminated, in which case the value of the tie-reduction is 100%. If no reduction can be obtained, this metric will give 0%. In all other cases, it will show the percentage of the removed elements that share the same score value as the faulty element.

In Figure 4.5, we visualized the amount of critical tie-reduction on our benchmark using the Tie-Reduction metric. Each dot represents one bug in the dataset and the violin plot offers a more general picture of the distribution of the data points. It can be seen from the shape of the plots that in several cases, the reduction was not possible but the majority of the ties were completely eliminated. Similar to the number and size of critical ties, there was no significant difference in this aspect. Regardless of what SBFL formula was used, we obtained very similar results.

Figure 4.5: Tie-reduction distribution of critical ties

Table 4.6 shows some important statistical values for this dataset: mean, median, and quartile 1 (the value in the middle between the smallest and the median points). Since the median is 100%, we can state that the critical ties are eliminated by our method in more than half of the cases (72–73%, as detailed below), and the reduction is between 83.9–91.5% for three-quarters of the bugs, while the average rate of reduction is greater than 80% in all cases.

Table 4.6: Statistics of tie-reduction (in percentage)

	Confidence	DStar	GP13	Ochiai	Tarantula
Mean	81.8	80.5	81.8	81.5	81.8
Median	100.0	100.0	100.0	100.0	100.0
<i>Q1</i>	91.5	83.9	91.5	90.4	88.9

Table 4.7 presents the number of remaining critical ties for each program and SBFL formula after applying the proposed tie-breaking method. The difference to the previous values (shown in Table 4.2) is also included. For example, 15 bugs of “Chart” were in critical ties with the “Ochiai” formula, but after applying the call frequency-based tie-breaking method 11 critical ties are eliminated, which is 73.3% of the initial ties. Overall, we achieved 72–73% improvement in the number of critical ties over the full dataset, the best case being “Mockito” with over 84.6% and the worst result was 54.5% on “Lang” using “Tarantula” and “DStar” formulas.

Table 4.7: *Changes in the number of critical ties after reduction*

		Chart	Closure	Lang	Math	Mockito	Time	All
Confidence	after	4	24	9	18	2	2	59
	diff. (#)	10	78	11	47	11	7	164
	diff. (%)	71.4	76.5	55.0	72.3	84.6	77.8	73.5
DStar	after	6	24	10	21	2	2	65
	diff. (#)	11	77	12	48	11	7	166
	diff. (%)	64.7	76.2	54.5	69.6	84.6	77.8	71.9
GP13	after	4	25	9	18	2	2	60
	diff. (#)	10	77	11	47	11	7	163
	diff. (%)	71.4	75.5	55.0	72.3	84.6	77.8	73.1
Ochiai	after	4	24	9	19	2	2	60
	diff. (#)	11	77	12	46	11	8	165
	diff. (%)	73.3	76.2	57.1	70.8	84.6	80.0	73.3
Tarantula	after	5	24	10	19	2	2	62
	diff. (#)	11	76	12	46	11	8	164
	diff. (%)	68.8	76.0	54.5	70.8	84.6	80.0	72.6

The sizes of the critical ties determine the level of achievable improvement after applying a tie-breaking approach. However, it is also important in which direction the faulty element moved in the new ranking after tie-breaking. Using the terminology from the previous section, moving from the MID position towards MIN means improvement. In the previous section, in Table 4.3, we presented the maximum potential improvement that sets a theoretical constraint on SBFL’s effectiveness after tie-reduction. Table 4.8 presents what we actually achieved using our proposed method (the meaning of the data is the same as in Table 4.3).

Table 4.8: *Achieving the minimum ranks*

	MIN != MID (count)	MIN != MID (%)	Diff.	Avg. diff.
Confidence	59	14.4	53.5	0.9
DStar	65	15.8	61.5	0.9
GP13	60	14.6	54.0	0.9
Ochiai	60	14.6	55.5	0.9
Tarantula	62	15.1	94.0	1.5

We examined whether our method was able to reduce the number of cases where the MIN (best case) and the MID (average case) approaches give different results. If there were no such cases that would mean that the obtained new ranking after tie-breaking would always be the best possible, the MIN case. Table 4.8 shows that, after applying our approach, only around 15% of the bugs contained critical ties (column 3), compared to around 55% before tie-breaking.

Comparing this with the result of Table 4.3, we find that in more than 160 cases we managed to achieve the ideal result with our method where the original formula was not able to do so. It means that for nearly three-quarters of bugs in critical ties (72–73%), the non-optimal result was improved to optimal.

If we compare the sum of the rank differences (column 4) and their averages (column 5) in Tables 4.3 and 4.8, it can be seen that our approach was able to reduce the sum significantly by 89–93%, and the average by 59–74%. Put differently, the overall rank positions from the ideal case improved from around 3.5 to 1 in the cases when we achieved optimal results, which essentially means a rank improvement between 2.2 and 2.5.

Answer to RQ2: By using the call frequency-based tie-breaking strategy, we achieved a significant reduction in both the size and the number of critical ties in our benchmark. In 72–73% of the cases, the ties were completely eliminated, the average reduction rate being more than 80%. In nearly three-quarters of the cases (72–73%), the faulty element got the highest rank among the tie-broken code elements, and here it improved its position by 59–74% on average.

4.6 Experimental Results and Discussion

4.6.1 Achieved Improvements in Average Ranks

Table 4.9 presents the average ranks before (column 2) and after (column 3) applying our tie-breaking strategy and it shows the difference between the average ranks before and after tie-reduction (column 4). If the difference is negative then this means that we could achieve improvement with our proposed strategy.

Table 4.9: Average rank of faulty elements before and after tie-breaking

	Before	After	Diff.
Confidence	55.16	53.11	-2.05
DStar	46.86	44.79	-2.07
GP13	68.79	66.68	-2.11
Ochiai	46.95	44.81	-2.14
Tarantula	50.39	48.33	-2.06

We can see that our strategy achieved improvements with all the selected SBFL formulas: the average rank reduced by more than 2 in all cases, which corresponds to 3.1–4.1% with respect to the total number of elements. Note that this average is similar to what we got for RQ2, but it is not the same because for RQ2 we examined only the cases when we achieved the optimal result, while in this section we are interested in the global results.

We also examined how many times our tie-breaking strategy changed the rank of bugs (in positive and negative directions) and what was the impact of the changes. Table 4.10 presents the possible changes in several categories, as follows (B means before, A means after applying tie-breaking):

- The faulty method moved to the top of the critical tie (column: best), when $B^{\text{MIN}} = A^{\text{MID}}$ (this is the case that we discussed using Tables 4.3 and 4.8)

- It has moved up in the rankings (column: better), when $B^{\text{MID}} > A^{\text{MID}}$ and $B^{\text{MIN}} < A^{\text{MID}}$
- It remained in the same position (column: same), when $B^{\text{MID}} = A^{\text{MID}}$
- We worsened the result (column: worse), when $B^{\text{MID}} < A^{\text{MID}}$ and $B^{\text{MAX}} > A^{\text{MID}}$
- It slipped back to the worst place (column: worst), when $B^{\text{MAX}} = A^{\text{MID}}$

In addition, column “improve” represents improvements in rank modifications (i.e., best+better), while “disimprove” is worse+worst. The table also includes the average differences in rank positions for the given categories.

The results indicate that in about 3–4 times more cases we achieved improvement than disimprovement of the ranking results. Moreover, the improvement differences are much higher than the disimprovement differences (compare, for example, the *better* cases of around -7 to *worse* cases of around 2). Another interesting insight is that in the case of *best*, the difference is relatively small as the size of the ties broken in this category was small as well (they contained 3–4 methods). Looking at the overall result, the average rate of improvement ranged from -3.73 to -3.86, while the disimprovement was only between 1.34 and 1.54 rank positions on average.

Table 4.10: Comparison of average ranks before and after tie-breaking

		Best	Better	Same	Worse	Worst	Improve	Disimprove
Confidence	count	85	51	50	17	20	136	37
	avg. diff.	-1.71	-7.13	0	2.32	0.55	-3.74	1.36
DStar	count	85	53	53	18	22	138	40
	avg. diff.	-1.71	-7.32	0	2.31	0.55	-3.86	1.34
Gp13	count	85	51	50	17	20	136	37
	avg. diff.	-1.71	-7.39	0	2.32	0.55	-3.84	1.36
Ochiai	count	86	52	50	16	21	138	37
	avg. diff.	-1.70	-7.42	0	2.38	0.62	-3.86	1.38
Tarantula	count	83	58	47	17	21	141	38
	avg. diff.	-1.46	-6.97	0	2.68	0.62	-3.73	1.54

The overall rank position improvement might seem modest, but we must consider the fact that the improvement can be achieved only by rearranging the positions in the critical ties. Thus, the sizes of the critical ties serve as a hard constraint (as discussed in the previous section).

4.6.2 Achieved Improvements in Top-N Categories

Table 4.11 presents the number of bugs belonging to the corresponding Top-N categories (cumulative) with their percentages, for the whole dataset, before and after

applying our tie-breaking strategy, as well as the differences between them. A decrease in the number of cases of the “Other” category and an increase in any Top-N means improvement.

Table 4.11: Top-N categories

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Confidence	75	18.2	169	41.1	203	49.4	246	59.9	165	40.1
After tie-breaking	92	22.4	180	43.8	214	52.1	252	61.3	159	38.7
Diff.	17	22.7	11	6.5	11	5.4	6	2.4	-6	-3.6
DStar	65	15.8	172	41.8	210	51.1	249	60.6	162	39.4
After tie-breaking	84	20.5	186	45.4	222	54.1	257	62.7	153	37.3
Diff.	19	29.2	14	8.1	12	5.7	8	3.2	-8	-4.9
GP13	75	18.2	169	41.1	203	49.4	245	59.6	166	40.4
After tie-breaking	92	22.4	179	43.6	212	51.6	250	60.8	161	39.2
Diff.	17	22.7	10	5.9	9	4.4	5	2.0	-5	-3.0
Ochiai	68	16.5	173	42.1	210	51.1	250	60.8	161	39.2
After tie-breaking	87	21.2	186	45.3	222	54.0	257	62.5	154	37.5
Diff.	19	27.9	13	7.5	12	5.7	7	2.8	-7	-4.3
Tarantula	65	15.8	166	40.4	203	49.4	244	59.4	167	40.6
After tie-breaking	83	20.2	177	43.1	212	51.6	251	61.1	160	38.9
Diff.	18	27.7	11	6.6	9	4.4	7	2.9	-7	-4.2

It can be clearly seen that our proposed tie-breaking strategy achieves improvements in all categories by moving many bugs to higher-ranked categories. On the lower end of the scale (“Other” category with rank > 10), 5–8 bugs were moved into one of the Top-N categories. This is important as it brings a “new hope” that a bug could be found by the user with the proposed strategy while it was not very probable without it. We can see a quite large number of improvements in higher categories as well, around 18 bugs moved to Top-1, for instance. Note that the percentages of bugs in each category before and after applying the strategy were calculated with respect to the total number of bugs in the dataset. While the difference percentage was calculated with respect to the number of bugs before applying the strategy.

To better understand the actual changes between the different Top-N categories we can use the non-accumulating variant of these categories. This shows whether there has been a beneficial change in the rank category. These moves between the Top-N categories are presented in Table 4.12. The sign \times indicates the number of changes in the negative direction (worsening result), and \checkmark marks improvement. For example, there were a total of 2 bugs with a rank greater than 1 but less than or equal to 3 before reduction by “Tarantula”, but our method resulted in a rank value greater than 3 (this is a negative result). In contrast, our method gave a rank of 1 for the faulty method 15 times which was previously greater than 1 but smaller than 3 (using “Tarantula”).

These numbers clearly show that the improvement was dominant: degradation by the proposed method was observable only for 2–3 bugs in the dataset, while we observed positive changes for 36–44 bugs.

Table 4.12: Top-N moves

	[1]	(1, 3]	(1, 3]	(3, 5]	(3, 5]	(5, 10]	(5, 10]	Other	✗	✓
	↓	↓	↓	↓	↓	↓	↓	↓		
	✗	✗	✓	✗	✓	✗	✓	✓		
Confidence	0	1	13	1	8	0	11	6	2	38
DStar	0	1	15	1	10	0	11	8	2	44
GP13	0	1	13	1	8	0	10	5	2	36
Ochiai	0	1	15	1	9	1	11	8	3	43
Tarantula	0	2	15	1	11	0	8	7	3	41

Answer to RQ3: The efficiency of all investigated SBFL formulas could be improved by using the proposed tie-breaking strategy: the average improvement of rank values in the benchmark was about *two positions*, and we observed improvement about 3–4 times more frequently than disimprovement, such improvements being much higher as well. Considering the Top-N categories, notable improvements could be observed: all Top-N categories showed positive results (improvements in 36–44 cases), and at the same time, in only a few (2–3) cases did Top-N categories worsen. We were able to increase the number of cases where the faulty method became the top-ranked element by 23–30%.

4.7 Contributions

In this chapter, the following points summarize my main contributions to the topic of thesis point II. The results of this chapter were published in [60].

- Providing the idea of using tie-breaking to improve the effectiveness of SBFL.
- Providing a thorough background on the problem of ties in SBFL.
- Gathering and discussing the related papers.
- Developing a tie-breaking method based on method call frequency to enhance the performance of SBFL.
- Evaluating and discussing the experimental results of the proposed tie-breaking method.

Chapter 5

Emphasizing SBFL Formulas With Importance Weights

5.1 Introduction

In this chapter, we are enhancing SBFL by presenting an approach that gives more importance to program elements that are executed by more failed test cases compared to other elements. The intuition is the following. A typical SBFL matrix is unbalanced in the sense that there are many more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly. We propose to emphasize the factor of the failing tests in the formulas, which is achieved by introducing a multiplication factor to SBFL formulas. This factor is called the *importance weight*. This importance weight can be used without contextual information (see Section 5.3) and is given as the ratio of executed failing tests for a program element with respect to all failing tests. Or, it can be used with contextual information (see Section 5.4) and is given as the ratio of covering failing tests over all failing tests combined with the so-called method call frequency in these tests. Thus, we multiply each element's suspicion score obtained by an SBFL formula by this importance weight. In other words, a program element will be more suspicious if it is affected by a larger portion of the failing tests. The proposed approach can be applied to SBFL formulas without modifying their structures.

The experimental results of our study show that our approach achieved a better performance in terms of average ranking and Top-N categories compared to the underlying SBFL formulas.

The following are the study's main contributions:

1. A new approach that successfully improves the performance of SBFL in many cases is proposed.
2. The impact of the new approach on the overall effectiveness of SBFL is discussed.

And, our concrete RQs are:

- **RQ1:** What level of average ranks improvements can we achieve using the proposed approach?
- **RQ2:** What is the impact of the proposed approach on SBFL's effectiveness across the Top-N categories?

5.2 Related Works

This section briefly presents the most relevant works of improving SBFL by targeting its formulas.

Forming new SBFL formulas is one of the ways of improving SBFL. Here, the researchers attempt to introduce new formulas that outperform the existing ones. For example, the authors in [153] proposed a new formula called "DStar". The proposed formula has been compared with several widely used formulas and it showed good performance. SBFL formulas can also be automatically generated by using GP. The authors in [8] used GP to automatically design SBFL formulas directly from the program spectra. The authors were able to produce 30 different formulas. Their results concluded that the GP is a good approach for producing effective formulas.

Modifying existing SBFL formulas also leads to improvements. The authors in [171] also modified three well-known formulas based on the idea that some failed tests may provide more information than others. Therefore, for the three used formulas, different weights for failed tests were assigned and then applied with multi-coverage spectra.

A different approach is to combine existing SBFL formulas with each other. The authors in [8] proposed a method for generating a new formula tailored to a certain program by combining 40 different formulas. The proposed method extracts information from the program using mutation testing and then combines multiple formulas based on the gathered information using different voting systems to generate a new formula. The results of the experiments show that the formula generated by their method is better than several existing ones. It is worth mentioning that researchers tried to merge numerous formulas to create new ones. Because the benefits of several existing formulas have been merged, the new formula is known as a hybrid formula. The performance of a hybrid formula should be superior to that of existing formulas, as shown in [73, 105].

Another interesting way is to add new information to existing SBFL formulas. The authors in [140] utilized the method call frequency of the subject programs during the execution of failed tests to add new contextual information to the standard formulas. Here, the frequency ef was substituted for the ef in each formula. The results of their study demonstrated that employing new information from method calls into the underlying formulas can improve SBFL's effectiveness.

All the studies mentioned in this section improved the performance of SBFL formulas in different ways. Our proposed approach also improves SBFL's performance by giving more importance to program elements that are executed by more failed

test cases (when it is used without contextual information) and appeared in different method calls contexts in failing tests (when it is used with contextual information). The advantages of our proposed approach over others are: (a) it does not modify the existing SBFL formulas. Thus, it could be applied to a wide range of formulas to enhance their effectiveness. This is very important as it makes the proposed approach more applicable than other approaches; (b) it solves the issue of an unbalanced SBFL matrix in the sense that there are many more passing tests than failing ones, and many formulas treat passing and failing tests similarly.

5.3 Non-Contextual Importance Weight

5.3.1 The Proposed Approach

Figure 5.1 shows our proposed approach. Using the selected SBFL formulas on the program spectra, we calculate the suspicion scores of program methods. The output is the initial suspicion scores of methods. Then, we multiply each initial score of each method by its importance weight which is computed via $ef/(ef + nf)$ of each corresponding method. This will improve the initial ranking list by emphasizing the methods that are executed by more failing tests and lowering the rank of the methods that are executed by fewer failing tests. As a result, a final improved ranking list is produced. This approach does not require any additional information from a program and the execution of its tests other than the basic statistics (i.e., ef and nf) calculated from program spectra. Thus, it is efficient and generalizable.

5.3.2 A Motivating Example From Defects4J

To show how our proposed approach works and how it achieves improvements, several bugs from the used Defects4J dataset were carefully examined. Bug 6 from the “Chart” project was one of the more interesting cases we looked into¹. Thus, we will illustrate using the basic statistics extracted from the spectra of 26 methods (M1-M26), including the faulty method M21, as presented in Table 5.1.

The “Tarantula” formula was applied to the extracted execution information to compute the suspicion score of each method as presented in Table 5.2. It can be seen that the underlying “Tarantula” formula cannot put the faulty method M21 (it is ranked 13 based on Equation 2.2) near the top of the ranking list suggested by the formula. The reason is that “Tarantula” assigned higher scores to the other 11 methods (i.e., M6, M7, M15-M17, M19, and M22-M26) that have been executed by a lower number of failed test cases (i.e., one failed test). As a result, these methods got higher ranks in the ranking list and will be examined before the actual faulty method M21.

In our example, the faulty method M21 was executed by two failed test cases. As the faulty method M21 was executed by more failed test cases compared to the

¹<http://program-repair.org/defects4j-dissection/#!/bug/Chart/6>

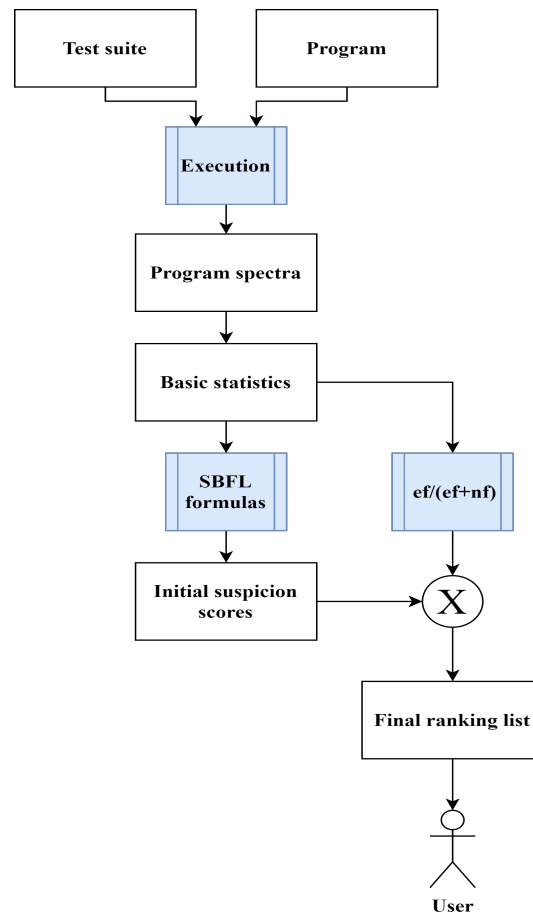


Figure 5.1: *The non-contextual based importance weight approach*

other 11 methods, it should be the most suspicious method and it should get more importance weight than the other 11 methods. To achieve this, we multiply the initial suspicion score of each method by the *importance weight* $ef/(ef + nf)$ of each one.

This will emphasize the methods that are executed by more failing tests and lower the rank of the methods that are executed by fewer failing tests. The initial “Tarantula” suspicion score for the faulty method M21 is 0.895 and its importance weight will be $2/(2+0)=1.0$. Thus, the final score of M21 will be the same which is $0.895 \times 1.0 = 0.895$. However, the initial scores of all the other 11 methods will be reduced and the ranks will be lower. For example, the final score of the method M6 will be $0.952 \times 0.5 = 0.476$ and its final rank will be 16.5 instead of 6.5. It can be noted that the faulty method M21 has the second highest suspicion score and thus ranked nearest to the top in the ranking list after applying our proposed approach (denoted with * as presented in Table 5.2).

Table 5.1: *Motivating example’s basic statistics*

	ef	ep	nf	np
M1	1	121	1	1759
M2	1	121	1	1759
M3	1	147	1	1733
M4	1	221	1	1659
M5	2	683	0	1197
M6	1	47	1	1833
M7	1	54	1	1826
M8	2	522	0	1358
M9	2	522	0	1358
M10	2	522	0	1358
M11	2	237	0	1643
M12	2	240	0	1640
M13	2	429	0	1451
M14	2	259	0	1621
M15	1	47	1	1833
M16	1	78	1	1802
M17	1	78	1	1802
M18	2	429	0	1451
M19	1	21	1	1859
M20	2	25	0	1855
M21	2	221	0	1659
M22	1	0	1	1880
M23	1	65	1	1815
M24	1	65	1	1815
M25	1	0	1	1880
M26	1	0	1	1880

5.3.3 Evaluation

Subject Programs

Here, we used the single faulty programs (i.e., 302 faults) of the dataset Defects4J 1.5 (see Table 2.5). However, 5 faults were excluded due to instrumentation issues. Thus, the final used dataset contained a total of 297 faults.

Evaluation Baselines

In this study, 11 widely-studied SBFL formulas, the formulas “Tarantula”, “Ochiai”, “Jaccard”, “Barinel”, “SorensenDic”, “DStar”, “Dice”, “Interest”, “Baroni”, “Kulczynski”, and “Cohen”, which are presented in Table 2.7, were used as benchmarks against our proposed approach. It is worth mentioning that the approaches mentioned in Section 5.2 are not directly comparable to ours as they applied modifications to the SBFL formulas. Our proposed approach can be applied to an SBFL formula without modifying the formula itself.

Table 5.2: *Motivating example – scores and ranks*

	Tarantula score	Tarantula rank	Tarantula* score	Tarantula* rank
M1	0.886	16.5	0.443	23.5
M2	0.886	16.5	0.443	23.5
M3	0.865	19	0.432	25
M4	0.810	22	0.405	26
M5	0.734	26	0.734	11
M6	0.952	6.5	0.476	16.5
M7	0.946	8	0.473	18
M8	0.783	24	0.783	9
M9	0.783	24	0.783	9
M10	0.783	24	0.783	9
M11	0.888	14	0.888	3
M12	0.887	15	0.887	4
M13	0.814	20.5	0.814	6.5
M14	0.879	18	0.879	5
M15	0.952	6.5	0.476	16.5
M16	0.923	11.5	0.462	21.5
M17	0.923	11.5	0.462	21.5
M18	0.814	20.5	0.814	6.5
M19	0.978	5	0.489	15
M20	0.987	4	0.987	1
M21	0.895	13	0.895	2
M22	1.000	2	0.500	13
M23	0.935	9.5	0.468	19.5
M24	0.935	9.5	0.468	19.5
M25	1.000	2	0.500	13
M26	1.000	2	0.500	13

5.3.4 Experimental Results and Discussion

This section presents and discusses the overall impact of the proposed approach on SBFL’s effectiveness.

Achieved Improvements in Average Ranks

Table 5.3 presents the average ranks before (column 2) and after (column 3) applying our proposed approach and also the difference between the average ranks (column 4) in both cases. If the difference is negative, it indicates that our proposed approach has the potential to improve.

With all of the selected SBFL formulas, we can observe that our proposed approach improved the average rank; reduced by more than 3 overall, which corresponds to 0.26–2.26% of the total number of methods in the used dataset. It can be noted that the formulas “Cohen” and “Tarantula” reduced the average ranks quite a lot compared to other formulas. Considering the formulas that have lower average ranks after applying our proposed approach, “Tarantula” and “Ochiai” are the best ones, respectively. This is good, considering the fact that the improvement is achieved only by using an importance emphasis from the basic statistics (i.e., ef and

Table 5.3: Comparison of average ranks

	Before	After	Diff.
Tarantula	83.05	76.94	-6.11
Ochiai	79.25	77.99	-1.26
Jaccard	82.08	79.05	-3.03
Barinel	83.05	79.24	-3.81
SorensenDice	82.08	79.01	-3.07
DStar	238.01	237.34	-0.67
Dice	82.08	79.05	-3.03
Interest	83.05	79.24	-3.81
Baroni	85.21	80.62	-4.59
Kulczynski1	240.98	238.01	-2.97
Cohen	86.56	79.84	-6.72

nf) rather than other additional information.

Answer to RQ1: Our proposed approach enhanced all the SBFL formulas. The improvement of average ranks by our approach in the used benchmark was about 3 positions overall. In a few cases, the improvement was even more than 6 positions. In terms of average ranks, our approach reduced more positions. This indicates that using an importance weight could have a positive impact and enhance the SBFL results. Also, it encourages us to investigate other forms of importance weights in the future and measure their impacts on the effectiveness of SBFL.

Achieved Improvements in Top-N Categories

Table 5.4 presents the number of bugs in the Top-N categories (cumulative) as well as their percentages for the entire dataset, before and after applying our proposed approach (denoted with *), as well as the differences between them. Here, improvement is defined as a decrease in the number of cases in the “Other” category and an increase in any of the Top-N categories.

It is evident that by relocating many bugs to higher-ranked categories, our proposed approach improved all Top-N categories. Also, 4–11 bugs were moved from the “Other” category (with a rank > 10) into one of the Top-N categories. This is significant since it raises the possibility of finding a bug with our approach while it was not very probable without it. Table 5.5 presents the summary of the enabling improvements achieved by our proposed approach.

It can be noted that each formula after applying our proposed approach achieves enabling improvements for at least 1% of the total number of single faults in the used dataset. In these cases, the basic SBFL formulas ranked the faulty method in the “Other” category, but our approach managed to bring it forward into the Top-10

Table 5.4: Top-N categories

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Tarantula	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
Tarantula*	59	19.9	125	42.1	148	49.8	178	59.9	119	40.1
Diff.	11	22.9	14	12.6	11	8.0	11	6.6	-11	-8.5
Ochiai	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
Ochiai*	57	19.2	122	41.1	147	49.5	176	59.3	121	40.7
Diff.	5	9.6	4	13.6	4	2.8	5	2.9	-5	-4.0
Jaccard	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
Jaccard*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.0	5	3.6	4	2.4	-4	-3.1
Barinel	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
Barinel*	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
Diff.	4	8.2	7	6.3	6	4.4	4	2.4	-4	-3.1
SorensenDic	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
SorensenDic*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.0	5	3.6	4	2.4	-4	-3.1
DStar	51	17.2	103	34.7	127	42.8	151	50.8	146	49.2
DStar*	53	17.8	105	35.4	129	43.4	155	52.2	142	47.8
Diff.	2	3.9	2	1.9	2	1.6	4	2.6	-4	-2.7
Dice	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
Dice*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.1	5	3.6	4	2.4	-4	-3.1
Interest	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
Interest*	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
Diff.	4	8.3	7	6.3	6	4.4	4	2.4	-4	-3.1
Baroni	48	16.2	111	37.4	132	44.4	166	55.9	131	44.1
Baroni*	59	19.9	123	41.4	147	49.5	174	58.6	123	41.4
Diff.	11	22.9	12	10.8	15	11.4	8	4.8	-8	-6.1
Kulczynski1	46	15.5	96	32.3	123	41.4	147	49.5	150	50.5
Kulczynski1*	51	17.2	103	34.7	127	42.8	151	50.8	146	49.2
Diff.	5	10.9	7	7.3	4	3.3	4	2.7	-4	-2.7
Cohen	49	16.5	113	38.0	138	46.5	168	56.6	129	43.4
Cohen*	55	18.5	121	40.7	143	48.1	172	57.9	125	42.1
Diff.	6	12.2	8	7.1	5	3.6	4	2.4	-4	-3.1

Table 5.5: Enabling improvements

	Rank > 10 (%) before our approach	Enab. improv. (%)
Tarantula vs. Tarantula*	130 (43.8%)	11 (3.7%)
Ochiai vs. Ochiai*	126 (42.4%)	5 (1.7%)
Jaccard vs. Jaccard*	129 (43.4%)	4 (1.3%)
Barinel vs. Barinel*	130 (43.8%)	4 (1.3%)
SorensenDic vs. SorensenDic*	129 (43.4%)	4 (1.3%)
DStar vs. DStar*	146 (49.2%)	4 (1.3%)
Dice vs. Dice*	129 (43.4%)	4 (1.3%)
Interest vs. Interest*	130 (43.8%)	4 (1.3%)
Baroni vs. Baroni*	131 (44.1%)	8 (2.7%)
Kulczynski1 vs. Kulczynski1*	150 (50.5%)	4 (1.3%)
Cohen vs. Cohen*	129 (43.4%)	4 (1.3%)

(or better) categories. Note that the formulas “Tarantula*”, “Ochiai*”, and “Baroni*” are the best in this aspect. Overall, each formula based on our proposed approach was able to achieve enabling improvements in the possible cases.

Higher categories have seen improvements as well, with 2–11 bugs moved to Top-1, for example. It should be noted that the percentages of bugs in each category before and after using the proposed method were computed based on the number of single faults in Defect4J, while the difference percentages were computed based on the number of single faults before applying our proposed approach.

We may utilize the non-accumulating variation of Top-N categories to better comprehend the actual changes across the different Top-N categories. This demonstrates whether the rank category has changed for the better. Table 5.6 presents these moves between the Top-N categories. The number of negative changes (worsening result) is indicated by the sign **X**. For example, suppose there were 10 bugs with a rank of more than 3 but less than or equal to 5 before we used our approach, but our approach produced a rank value larger than 5 (this is considered a negative result). The number of positive changes is indicated by the sign **✓** (improving result). For example, if our approach resulted in a rank value less than 3 (this is considered a positive result).

These numbers clearly show that improvement was dominant and degradation by our proposed approach was not observed in the used dataset, while we observed positive changes for 9–30 bugs.

Table 5.6: *Top-N moves*

	[1]	(1,3]	(1,3]	(3,5]	(3,5]	(5,10]	(5,10]	Other	X	✓
	↓	↓	↓	↓	↓	↓	↓	↓		
	X	X	✓	X	✓	X	✓	✓		
Tarantula vs. Tarantula*	0	0	7	0	6	0	6	11	0	30
Ochiai vs. Ochiai*	0	0	4	0	4	0	3	5	0	15
Jaccard vs. Jaccard*	0	0	5	0	4	0	4	4	0	17
Barinel vs. Barinel*	0	0	4	0	4	0	4	4	0	16
SorensenDic vs. SorensenDic*	0	0	5	0	4	0	4	4	0	17
DStar vs. DStar*	0	0	2	0	2	0	1	4	0	9
Dice vs. Dice*	0	0	5	0	4	0	4	4	0	17
Interest vs. Interest*	0	0	4	0	4	0	4	4	0	16
Baroni vs. Baroni*	0	0	7	0	3	0	10	8	0	28
Kulczynski1 vs. Kulczynski1*	0	0	4	0	5	0	3	4	0	16
Cohen vs. Cohen*	0	0	5	0	4	0	4	4	0	17

Answer to RQ2: Overall, it can be said that there were noticeable improvements in terms of the Top-N categories with positive results (improvements in 9–30 cases). Also, we were successful in increasing the number of cases in which the faulty method was ranked first by 4–23%. Another interesting finding is that in some cases we were able to achieve more than 3% enabling improvement by moving 4–11 bugs from the “Other” category into one of the higher-ranked categories. These cases are now more likely to be discovered and then fixed than before.

5.4 Contextual Importance Weight

5.4.1 The Proposed Approach

The results of using non-contextual importance weight to enhance the effectiveness of SBFL encouraged us to continue the topic. Thus, we used contextual information, method call frequency in failed test cases (see Chapter 4), as an additional factor to be combined with the non-contextual importance weight. Using the selected SBFL formulas on the program spectra, we calculate the suspicion scores of program methods. The output is the initial suspicion scores of methods. Then, we multiply each initial score of each method by its importance weight, which is computed via Equation 5.1.

$$\text{Contextual Importance Weight} = \left(\frac{\text{ef} * \phi}{\text{ef} + \text{nf}} \right) \quad (5.1)$$

5.4.2 A Motivating Example From Defects4J

To show how our proposed approach works and how it achieves improvements, several bugs from the used Defects4J dataset were carefully examined. Bug 5 from the “Time” project was one of the more interesting cases we looked into². Thus, we will illustrate using the basic statistics extracted from the spectra of 6 methods (M1-M6), including the faulty method M2, as presented in Table 5.7.

Table 5.7: *Motivating example’s basic statistics*

	ef	ep	nf	np	ϕ
M1 ..org.joda.time.Period.withYears()	3	9	0	3999	15
M2 ..org.joda.time.Period.normalizedStandard()	3	18	0	3990	98
M3 ..org.joda.time.PeriodType.yearWeekDay()	1	3	2	4005	5
M4 ..org.joda.time.PeriodType.yearDay()	1	3	2	4005	4
M5 ..org.joda.time.PeriodType.months()	3	7	0	4001	4
M6 ..org.joda.time.PeriodType.forFields()	1	8	2	4000	80

The “Jaccard” formula was applied to the extracted execution information to compute the suspicion score of each method as presented in Table 5.8.

It can be seen that the underlying “Jaccard” formula cannot put the faulty method M2 (it is ranked 5 based on Equation 2.2) near the top of the ranking list suggested by the formula. The reason is that “Jaccard” assigned higher scores to the other 4 methods (i.e., M1 and M3-M5). As a result, these methods got higher ranks in the ranking list and will be examined before the actual faulty method M2.

In our example, the faulty method M2 was executed by three failed test cases and has appeared in 98 different call contexts, more than any other method. Method

²<http://program-repair.org/defects4j-dissection/#!/bug/Time/5>

Table 5.8: *Motivating example – scores and ranks*

	Jaccard score	Jaccard rank	Jaccard* score	Jaccard* rank
M1	0.250	2	3.750	2
M2	0.143	5	14.014	1
M3	0.167	3.5	0.278	5
M4	0.167	3.5	0.222	6
M5	0.300	1	1.200	4
M6	0.091	6	2.424	3

M2 should be the most suspicious method and it should get more importance weight than the other 5 methods. To achieve this, we multiply the initial suspicion score of each method by the Equation 5.1 of each one.

This will emphasize the methods that are executed by more failing tests and lower the rank of the methods that are executed by fewer failing tests. The initial “Jaccard” suspicion score for the faulty method M2 is 0.143 and its importance weight will be $(3 \cdot 98) / (3 + 0) = 98.0$. Thus, the final score of M2 will be $0.143 \cdot 98.0 = 14.014$. Thus, the faulty method M2 has the highest suspicion score and is thus top-ranked in the ranking list after applying our proposed approach (denoted with * as presented in Table 5.8).

5.4.3 Evaluation

Subject Programs

Here, we used the single and multiple faulty programs (i.e., 411 faults) of the dataset Defects4J 1.5 (see Table 2.5).

Evaluation Baselines

In this study, 8 widely-studied SBFL formulas, the formulas “Jaccard”, “Barinel”, “SorensenDic”, “DStar”, “Dice”, “Interest”, “Kulczynski1”, and “Cohen”, which presented in Table 2.7, were used as benchmarks against our proposed approach. It is worth mentioning that Vancsics et al’s approach proposed in [140] is comparable to ours; thus, we will compare our results to it too.

5.4.4 Experimental Results and Discussion

This section presents and discusses the overall impact of the proposed approach on SBFL’s effectiveness.

Achieved Improvements in Average Ranks

Table 5.9 presents the average ranks before and after using our proposed approach (denoted with **) and Vancsics et al’s approach (denoted with *), as well as the

difference between them. If the difference is negative, it indicates that the used approach has the potential to improve.

Table 5.9: Average ranks comparison

			Diff.	Diff.
Jaccard = 38.51	Jaccard* = 23.58	Jaccard** = 21.83	Jaccard - Jaccard* = -14.93	Jaccard - Jaccard** = -16.68
Barinel = 38.5	Barinel* = 23.66	Barinel** = 21.7	Barinel - Barinel* = -14.84	Barinel - Barinel** = -16.8
SorensenDic = 38.51	SorensenDic* = 23.77	SorensenDic** = 21.96	SorensenDic - SorensenDic* = -14.74	SorensenDic - SorensenDic** = -16.55
DStar = 149.03	DStar* = 150.59	DStar** = 136.67	DStar - DStar* = 1.56	DStar - DStar** = -12.36
Dice = 38.51	Dice* = 23.58	Dice** = 21.83	Dice - Dice* = -14.93	Dice - Dice** = -16.68
Interest = 38.5	Interest* = 23.66	Interest** = 21.7	Interest - Interest* = -14.84	Interest - Interest** = -16.8
Kulczynski1 = 153.34	Kulczynski1* = 138.26	Kulczynski1** = 136.66	Kulczynski1 - Kulczynski1* = -15.08	Kulczynski1 - Kulczynski1** = -16.68
Cohen = 38.54	Cohen* = 20.76	Cohen** = 17.87	Cohen - Cohen* = -17.78	Cohen - Cohen** = -20.67

We can see that our proposed approach achieved improvements with all of the selected SBFL formulas: the average rank got reduced by about 17 overall, which corresponds to 8–54% with respect to the total number of methods in the used dataset. It can be noted that the “Cohen**” formula reduced the average rank more than the others. Considering the formulas that have the lower average ranks after applying our proposed approach, “Cohen**”, “Barinel**”, and “Interest**” are the best ones, respectively.

Vancsics et al’s approach also achieved improvements in the average ranks of all the selected formulas except in the case of the “DStar*” formula, where disimprovement was observed. However, the average rank reduced by this approach was about 13 overall. The difference is 4 positions between the two approaches. In other words, our approach outperformed Vancsics et al’s approach by 4 positions in terms of reducing the average rank.

Answer to RQ1: Our proposed approach enhanced all the SBFL formulas compared to Vancsics et al’s approach. The improvement of average ranks by our approach in the used benchmark was about 17 positions overall while in Vancsics et al’s approach was about 13. In terms of average ranks, our approach reduced more positions. This indicates that using an importance weight could have a positive impact and enhance the SBFL results. Also, it encourages us to investigate other forms of importance weights in the future and measure their impacts on the effectiveness of SBFL.

Achieved Improvements in Top-N Categories

Table 5.10 presents the number of bugs in the Top-N categories for each approach. Here, improvement is defined as a decrease in the number of cases in the “Other” category and an increase in any of the Top-N categories.

It is evident that by relocating many bugs to higher-ranked categories, our proposed approach and Vancsics et al’s approach improved all Top-N categories. However, our approach placed more bugs (i.e., 19–25 bugs) into one of the Top-N categories from the “Other” category (with rank > 10) compared to Vancsics et al’s approach (i.e., 16–21 bugs). This is significant since it raises the possibility of finding a bug with our approach while it was not very probable without it. Table 5.11 presents the enabling improvements achieved by each approach.

Table 5.10: Top-N categories

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Jaccard	66	16.1	168	40.9	203	49.4	250	60.8	161	39.2
Jaccard*	77	18.7	185	45.0	223	54.6	269	65.5	142	34.5
Jaccard**	78	19.0	192	46.7	226	55.0	271	65.9	140	34.1
Barinel	65	15.8	165	40.1	201	48.9	248	60.3	163	39.7
Barinel*	74	18.0	179	43.6	222	54.0	269	65.5	142	34.5
Barinel**	79	19.2	194	47.2	227	55.2	273	66.4	138	33.6
SorensenDic	66	16.1	168	40.9	203	49.4	250	60.8	161	39.2
SorensenDic*	77	18.7	183	44.5	221	53.8	266	64.7	145	35.3
SorensenDic**	81	19.7	189	46.0	226	55.0	270	65.7	141	34.3
DStar	70	17.0	168	40.9	193	47.0	232	56.4	179	43.6
DStar*	71	17.3	185	45.0	178	43.3	222	54.0	189	46.0
DStar**	78	19.0	192	46.7	212	51.6	251	61.1	160	38.9
Dice	66	16.1	156	38.0	203	49.4	250	60.8	161	39.2
Dice*	77	18.7	145	35.3	223	54.6	269	65.5	142	34.5
Dice**	78	19.0	174	42.3	226	55.0	271	65.9	140	34.1
Interest	65	15.8	165	40.1	201	48.9	248	60.3	163	39.7
Interest*	74	18.0	179	43.6	222	54.0	269	65.5	142	34.5
Interest**	79	19.2	194	47.2	227	55.2	273	66.4	138	33.6
Kulczynski1	66	16.1	151	36.7	188	45.7	230	56.0	181	44.0
Kulczynski1*	76	18.5	166	40.4	211	51.3	248	60.3	163	39.7
Kulczynski1**	81	19.7	175	42.6	213	51.8	251	61.1	160	38.9
Cohen	66	16.1	168	40.9	203	49.4	250	60.8	161	39.2
Cohen*	77	18.7	186	45.3	224	54.5	270	65.7	141	34.3
Cohen**	80	19.5	191	46.5	231	56.2	272	66.2	139	33.8

Table 5.11: Enabling improvements

	Rank > 10 (%)	Enab. impr. (%)	Enab. impr. (%)
Jaccard vs. Jaccard* vs. Jaccard**	161 (39.2%)	19 (4.6%)	21 (5.1%)
Barinel vs. Barinel* vs. Barinel**	163 (39.7%)	21 (5.1%)	25 (6.0%)
SorensenDic vs. SorensenDic* vs. SorensenDic**	161 (39.2%)	16 (3.9%)	20 (4.9%)
DStar vs. DStar* vs. DStar**	179 (43.6%)	10 (2.4%)	19 (4.6%)
Dice vs. Dice* vs. Dice**	161 (39.2%)	19 (4.6%)	21 (5.1%)
Interest vs. Interest* vs. Interest**	163 (39.7%)	21 (5.1%)	25 (6.0%)
Kulczynski1 vs. Kulczynski1* vs. Kulczynski1**	181 (44.0%)	18 (4.4%)	21 (5.1%)
Cohen vs. Cohen* vs. Cohen**	161 (39.2%)	20 (4.9%)	22 (5.4%)

It can be noted that each new formula achieves enabling improvements, the average enabling improvements was about 5% of the total number of faults in the used dataset by our approach. In these cases, the basic SBFL formulas ranked the faulty method in the “Other” category, but our proposed approach managed to bring it forward into the Top-10 (or better) categories. Note that the formulas “Barinel**”, “Interest**”, and “Cohen**” are the best in this aspect. Overall, each formula based on our proposed approach was able to achieve enabling improvements in the possible cases. It can be noted that the improvement of Vancsics et al’s approach was about 4% of the total number of faults in the used dataset with the formulas “Barinel*”, “Interest*”, and “Cohen*” as the best ones. Here, this improvement seems modest

considering the fact that only an importance weight was used. Other, more complex weights may yield much more improvement which will be investigated in the future. Higher categories have significant improvements as well, with roughly 8–15 bugs moving to Top-1 by our approach compared to Vancsics et al’s approach with 1–11 bugs, for example. Here also, our approach outperformed Vancsics et al’s approach by moving more bugs to the Top-1 category.

Answer to RQ2: We were able to raise the number of cases where the faulty method was ranked first by 11–23%. While Vancsics et al’s approach moved a lower number of bugs to the Top-1 category. Another interesting finding is that our approach achieved more enabling improvement compared to Vancsics et al’s approach by moving 19–25 bugs from the “Other” category into one of the higher-ranked categories. These cases are now more likely to be discovered and then fixed than before.

5.5 Non-contextual vs. Contextual Importance Weights

In this section, we will compare non-contextual and contextual importance weights on the effectiveness of SBFL. For this purpose, we will use the single and multiple faulty programs (i.e., 411 faults) of the dataset Defects4J 1.5 (see Table 2.5). In terms of the achieved improvements in average ranks, Table 5.12 presents the average ranks using non-contextual importance weight (column 2) and using contextual importance weight (column 3). It can be seen that using both types of importance weights can improve the effectiveness of SBFL. However, employing contextual information (i.e., method call frequency) into the importance weight gives more positive results (i.e., it reduces the average ranks more than using the non-contextual importance weight).

Table 5.12: Comparison of average ranks

	Non-contextual importance weight (+)	Contextual importance weight (++)
Jaccard	34.21	21.83
Barinel	34.49	21.7
SorensenDice	34.18	21.96
DStar	148.49	136.67
Dice	34.21	21.83
Interest	34.49	21.7
Kulczynski1	149.03	136.66
Cohen	35.06	17.87

While in terms of the achieved improvements in the Top-N Categories, Table 5.13 presents the number of bugs in the Top-N categories for each approach. Here, im-

provement is defined as a decrease in the number of cases in the “Other” category and an increase in any of the Top-N categories. It is evident that by relocating many bugs to higher-ranked categories, using the contextual importance weight (denoted with ++) is better than using the non-contextual importance weight (denoted with +) as it improved all Top-N categories. Also, it placed more bugs (i.e., 16–19 bugs) into one of the Top-N categories from the “Other” category (with a rank > 10). This is significant since it raises the possibility of finding a bug by using contextual importance weight while it was not very probable without it.

Table 5.14 presents the enabling improvements achieved by each approach. It can be noted that each formula after applying the contextual importance weight approach achieves enabling improvements for at least 4% of the total number of faults in the used dataset. In these cases, using the non-contextual importance weight ranked the faulty method in the “Other” category, but using the contextual importance weight approach managed to bring it forward into the Top-10 (or better) categories. Overall, each formula based on the contextual importance weight approach was able to achieve enabling improvements in the possible cases.

Table 5.13: *Top-N categories*

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Jaccard+	70	17.0	171	41.6	206	50.1	252	61.3	159	38.7
Jaccard++	78	19.0	192	46.7	226	55.0	271	65.9	140	34.1
Barinel+	68	16.5	172	41.8	209	50.9	254	61.8	157	38.2
Barinel++	79	19.2	194	47.2	227	55.2	273	66.4	138	33.6
SorensenDic+	70	17.0	171	41.6	206	50.1	254	61.8	157	38.2
SorensenDic++	81	19.7	189	46.0	226	55.0	270	65.7	141	34.3
DStar+	73	17.8	156	38.0	192	46.7	234	57.0	177	43.1
DStar++	78	19.0	192	46.7	212	51.6	251	61.1	160	38.9
Dice+	70	17.0	171	41.6	206	50.1	252	61.3	159	38.7
Dice++	78	19.0	174	42.3	226	55.0	271	65.9	140	34.1
Interest+	68	16.5	172	41.8	209	50.9	254	61.8	157	38.2
Interest++	79	19.2	194	47.2	227	55.2	273	66.4	138	33.6
Kulczynski1+	70	17.0	156	38.0	193	47.0	232	56.4	179	43.6
Kulczynski1++	81	19.7	175	42.6	213	51.8	251	61.1	160	38.9
Cohen+	70	17.0	171	41.6	206	50.1	254	61.8	157	38.2
Cohen++	80	19.5	191	46.5	231	56.2	272	66.2	139	33.8

The results mentioned in this chapter show that both non-contextual and contextual importance weights can improve the effectiveness of SBFL. However, the positive impact of using the contextual importance weight is more obvious. This encourages us to try other types of contextual information (other than the method call frequency in failed test cases) and other forms of importance weights in the future.

Table 5.14: *Enabling improvements*

	Rank > 10 (%) of non-contextual importance weight	Enab. improv. (%)
Jaccard+ vs. Jaccard++	159 (38.7%)	19 (4.6%)
Barinel+ vs. Barinel++	157 (38.2%)	19 (4.6%)
SorensenDic+ vs. SorensenDic++	157 (38.2%)	16 (3.9%)
DStar+ vs. DStar++	177 (43.1%)	17 (4.1%)
Dice+ vs. Dice++	159 (38.7%)	19 (4.6%)
Interest+ vs. Interest++	157 (38.2%)	19 (4.6%)
Kulczynski1+ vs. Kulczynski1++	179 (43.6%)	19 (4.6%)
Cohen+ vs. Cohen++	157 (38.2%)	18 (4.4%)

5.6 Contributions

In this chapter, the following points summarize my main contributions to the topic of thesis point III. The results of this chapter were published in [119] and [121].

- Providing the idea of using importance weights to improve the effectiveness of SBFL.
- Gathering and discussing the related papers.
- Developing two methods based on importance weights. The first method improves SBFL without using any contextual information and the second method improves SBFL by using contextual information.
- Evaluating and discussing the experimental results of the proposed methods of importance weights.

Chapter 6

New Formulas for SBFL

6.1 Introduction

The basic elements in SBFL are the risk evaluation formulas, which calculate a suspiciousness score for each program element based on test coverage and test case outcome information. This score can be used in debugging to identify the faulty element more efficiently. One of the main challenges in SBFL is how to introduce new formulas to enhance SBFL’s performance by putting faulty elements at the beginning of the ranking list produced by SBFL as much as possible [122]. Thus, they can be examined and found efficiently.

In this chapter, we present two approaches for finding new SBFL formulas. In the first approach (Section 6.3), we introduce a new formula based on the “guess” or “intuitive” method. Our new SBFL ranking formula enhances a base formula by ranking code elements slightly higher than others that are executed by more failed tests and fewer passing ones. Its novelty is that it breaks ties between the elements that share the same suspicion score of the base formula.

The new SBFL formula addresses the issue of ties by emphasizing the high number of failing test cases and the low number of passing ones for a particular code element. This way, typical situations of ties can be handled very simply. Our approach is to add a small enhancement component to the base formula, which slightly modifies the resulting value, only sufficiently to produce different suspicion values, hence effectively breaking the ties.

Experiments were conducted on six single-fault programs of the Defects4J dataset to evaluate the effectiveness of the proposed formula. The results show that our new formula, when compared to three widely-studied SBFL formulas, achieved a better performance in terms of average ranking. It also achieved positive results in all of the Top-N categories.

This study’s main contributions are as follows:

1. A new SBFL formula that improves the performance of SBFL in many cases, which is a good candidate for tie-breaking in combination with other formulas as well.

2. The analysis of the impact of the new SBFL formula on the overall SBFL's effectiveness is discussed.

While the RQs are as follows:

- **RQ1:** What level of average rank improvement can we achieve using the proposed SBFL formula?
- **RQ2:** What is the overall effect of the proposed formula on SBFL's effectiveness in terms of Top-N categories?

In the second approach (Section 6.4), we introduce new SBFL formulas based on systematic search. The majority of the published formulas have been manually crafted, and some of the well-known examples include "Tarantula" [67], "Ochiai" [1], and "DStar" [153]. All these are based on some intuition and/or previous results from other domains. Researchers also experimented with combining existing formulas, adding external information, or generating new formulas by meta-heuristic search or artificial intelligence; and still, the number of possible formulas is infinite. However, no systematic search for new formulas was reported in the literature. In this study, we do so by examining existing formulas, defining formula structure templates, generating formulas automatically (including already proposed ones), and comparing them to each other.

In this study, we propose to systematically search for SBFL formulas, based on formula templates. First, we examine existing formulas reported in the literature to define formula templates that describe the structure of the formula. Then, we systematically generate all possible formulas for these templates and examine them. To eliminate redundancy in the work, we sort out *useless* (constant or duplicate) formulas, determine *equivalent* (that produce the same rankings), *inverse* (that produce exactly the opposite rankings), and *mostly equivalent* ones (that only differ from each other in special cases).

To illustrate the concept, we performed a preliminary search with a simple formula template. We used the generated formulas on the Defects4J dataset to compute rankings and determine their effectiveness. While most of our preliminary expectations about the performance of the formulas are supported by the results, some numbers indicated that handling special cases can have a strong influence on the effectiveness of similar formulas.

The main contributions of this study are as follows:

1. The idea of finding new SBFL formulas via a systematic search based on formula templates.
2. A formula template based on existing formulas and an evaluation of formulas derived from it.

While the RQ) is as follows:

- **RQ1:** Does using a systematic search approach for generating new SBFL formulas deserves study and investigation?

Previously, we proposed an approach based on systematic search – in contrast to ad-hoc, intuitive, search-based, or ML methods – for introducing new formulas for SBFL [123]. But, we were able to check only a limited number of generated formulas and were not able to find new better formulas compared to other top existing ones. This is not surprising since the template we used was very simple.

In this study, we extend our previous work by using more formula templates, and we apply them to the Defects4J dataset to determine their effectiveness. The systematic search for formulas means that we generate the formulas that conform to a predefined formula template, and we enumerate all possible ones. Then, the formula candidates are evaluated on a benchmark suite for fault localization effectiveness.

Our experimental results show that the systematic approach for finding new SBFL formulas is successful. We were able to find two new formulas that are better than all of the generated formulas presented in our preliminary study [123], and most of the formulas from related literature as well. The two new formulas achieved better performance in terms of average ranking compared to others. Also, they gained positive improvements in the Top-N categories, as will be seen later.

The main contributions of this study are as follows:

1. A large number of SBFL formulas were systematically generated and evaluated using two formula templates.
2. Two new SBFL formulas that are generated using a systematic approach based on formula templates are proposed.
3. The analysis of the impact of the new SBFL formulas on the overall SBFL' effectiveness is discussed.

Note, that our approach significantly differs from heuristic approaches including search-based and ML-based methods. We are checking all possibilities systematically, and that way we can ensure that no option is left out in the search space.

While the RQs are as follows:

- **RQ1:** Can systematic search lead to new formulas that could outperform the existing ones?
- **RQ2:** What level of average rank improvements can we achieve using the systematically generated formulas?
- **RQ3:** What is the overall effect of the systematically generated formulas on SBFL's effectiveness in terms of Top-N?

6.2 Related Works

There are many approaches proposed in the literature to enhance the performance of SBFL. One enhancing approach is to improve SBFL formulas to more accurately guide and pinpoint faults in the fault localization process. This is achieved by introducing

new SBFL formulas, modifying currently used SBFL formulas, or combining existing ones. The most important efforts that aim to improve SBFL by targeting its formulas are briefly presented in this section.

6.2.1 Introducing New SBFL Formulas

The first approach is to design new SBFL formulas based on intuition, past experience, or by reusing results from other disciplines. For example, “Ochiai” [102] and “Binary” [22] came from the fields of biological research. The authors in [153] and in [2] proposed new SBFL formulas called “DStar” and “Barinel”, respectively, based on intuition. Each proposed formula has been compared with several widely used formulas and it showed good performance compared to others. The authors in [7] proposed a new formula called Metrics Combination (MECO) which effectively finds errors without the need for prior knowledge of program structure or semantics. Their idea is that several metrics (e.g., failed execution flag, assumption proportion) can be extracted from the target program spectra and combined to propose a new formula. Many studies, such as [2] and [99] claimed (on a theoretical level) that an optimal SBFL formula exists. However, this is not necessarily true in practice. The fact that faulty programs in practice might not adhere to the same theoretical assumptions is one potential explanation for the discrepancy between the theoretical and empirical results of SBFL formulas, as discussed in [95, 166]. As shown in [170], there is no optimal formula for all types of faults.

6.2.2 Modifying Existing SBFL Formulas

The second approach is formula modification. The authors in [91] improved the performance of the “Tarantula” formula by modifying some parts of it to amplify its scores. However, the improved Tarantula does not always make improvements in the ranking. Also, the authors did not evaluate the improved Tarantula using well-known evaluation metrics. The authors in [171] modified three well-known SBFL formulas based on the idea that some failed test cases may provide more testing information than other failed test cases. Therefore, for the three used formulas, different weights for failed test cases were assigned and then applied with multi-coverage spectra.

6.2.3 Combining Existing SBFL Formulas

Another way is to combine existing formulas. The authors in [12] proposed a new SBFL formula by combining 40 different formulas using different voting systems. The proposed method extracts information from the program using mutation testing and then combines multiple formulas based on the gathered information using different voting systems to generate a new formula. The results of experiments have shown that the formula generated by their method is better than several existing ones. Multiple formulas also can be combined into a single new one. The resulting formula is a

hybrid formula; which combines the advantages of the formulas that have been used in the combination as in [73].

6.2.4 Adding New Information to Existing SBFL Formulas

Involving new information in existing SBFL formulas can also lead to improvements. For example, the authors in [140] utilized the method call frequency during the execution of failed tests to add new contextual information to existing formulas. Thus, the ef of each formula was changed to the frequency ef . The experiments improved SBFL's effectiveness. However, this approach can only be applied to the formulas that have the ef numerator. Also, it is considered heavy as it requires tracing the execution of each method call, as caller or callee, in the failed test cases.

6.2.5 Generating New SBFL Formulas by Meta-heuristic Search

The authors in [8] used GP to evolve new formulas from a hybrid dataset (i.e., from different benchmark datasets). They were able to produce several new formulas that outperformed many existing ones. However, this approach poses several issues: (a) it is not systematic, thus it does not guarantee that even a simple formula is examined; (b) the results of applying GP may vary greatly from one run to another as it depends on the initial selection of the population; (c) a couple of parameters (e.g., population size, number of generations, mutation rate, etc.) must be set by humans, thus the probability of finding the optimal solution is not too high; (d) generating formulas based on this approach has a disadvantage in that existing datasets do not cover all possible types of bugs. Thus, a generated formula may fail to locate a bug that is not included in the used datasets.

A final critique of this approach is that the generated formulas are often difficult to comprehend and non-intuitive, including complex computations and magic constants, such as the following formulas:

$$\text{HDGPCR02} = (\sqrt[5]{\sqrt[4]{np}} + 3ef + np) - \left(\frac{ef}{ep} - 2ef\right)(ef + np + \frac{np}{ep})$$

$$\text{HDGPCR20} = \sqrt[5]{\sqrt{ep}\left(\frac{np}{ep}\right)} - \sqrt[3]{nf} - \frac{np}{ef} + (nf - ef - \sqrt{ep}(np))$$

$$\text{HDGPCR23} = \frac{\sqrt{ep} + \frac{np}{ef}}{ep + ef + nf} - (nf)\left(\frac{ef}{ep}\right) + \left(\frac{\sqrt[3]{nf}}{np}\right)(nf)$$

6.2.6 Machine Learning

ML has also been used for fault localization to learn scores from spectra [161, 181], or to use likely invariant diffs and suspiciousness scores as features to learn the ranks [11]. But, such approaches do not produce a resulting formula that can be reused and are typically specific to a particular subject system. Also, the search space cannot be meaningfully controlled, and the decisions made by ML and parts of the search space explored will remain black-box.

6.2.7 Systematic Formula Generation

To overcome the aforementioned problems assigned with meta-heuristic search or ML to find new formulas, in this study, we follow a completely different direction to automatically generate formulas, and explore the formulas in a *systematic* manner. Of course, there is an infinite number of formulas that can be generated based on the spectrum metrics, which makes it impossible to exhaustively examine all of them. The idea we proposed in our previous study [123] was to limit the search space using formula templates and explore a particular class of formulas *exhaustively*.

Also, using systematic search (in contrast to ad-hoc, intuitive, or heuristic methods) for introducing new formulas is a novel approach that has not been investigated previously. We are doing this by defining formula templates (based on existing formulas) and instantiating them in all possible ways.

Compared to heuristic search and machine learning approaches, our approach is complementary and it cannot replace them because they can cover a much larger search space but not completely; while our approach can cover a smaller search space with each formula template but completely.

This approach also has limitations and may lead to problems. First, the more generic a template is, the more combinatorial explosions we will face and the more formula instances we will get. Second, as the authors in [99, 162] have shown, in theory, several formulas can produce the same rankings. We can utilize this result when a large number of new formulas are generated: it is enough to (and we practically have to) choose a single representative of the equivalent ones. Our goal is not to show formula equivalences, but to find completely new formulas.

6.3 Manually Crafted New Formulas

6.3.1 The Proposed SBFL Formula

All the aforementioned studies improved the performance of SBFL formulas in different ways. Our proposed approach improves SBFL's performance by introducing a new SBFL formula. The main advantage of our proposed formula over others is that it ranks program elements that are executed in more failed test cases and fewer passed test cases higher than other elements, and at the same time it effectively breaks ties in many cases. Our proposed formula is a sum of two parts: a base component and a tie-breaking enhancement part. At present, we use the simplest possible SBFL formula ef for the base part, but this can be replaced in theory by any other existing formula. The second component serves the purpose of modifying the base part by a slight amount, thus breaking ties with higher probability, and giving higher scores to elements with more failing tests and/or fewer passing tests:

$$\text{New Formula} = ef + \left(\frac{ef - nf}{ef + nf + ep} \right) \quad (6.1)$$

The intuition behind the effect of the modification part is that the resulting value of the formula will be dominated by ef because typically only a small value between $[0 - 1]$ will be added to or removed from it. Since all four basic counters are positive, $ef + nf$ is bigger than their difference, and ep is typically much bigger than zero, the result of this modification component will be probably closer to 0 than 1. Element scores are often tied because they share the same ef and nf numbers, so the tie will be broken by ep , which is more likely different in the two cases. Furthermore, if the two elements differ in ef and nf , and since $ef + nf$ is constant, the element for which $ef - nf$ is bigger (more failing tests that cover the element) will be ranked higher. This will also help in the situation where ep is the same with the two elements.

6.3.2 A Motivating Example From Defects4J

To show how our proposed approach works and how it achieves improvements, several bugs from the used Defects4J dataset were carefully examined. Bug 6 from the “Chart” project was one of the more interesting cases we looked into¹. Thus, we will illustrate using the basic statistics extracted from the spectra of 26 methods (M1-M26), including the faulty method M21, as presented in Table 6.1.

The “Tarantula” formula was applied to the extracted execution information to compute the suspicion score of each method as presented in Table 6.2. It can be seen that the “Tarantula” formula cannot put the faulty method M21 near the top of the ranking list suggested by the formula (it is ranked 13 based on Equation 2.2). The reason is that “Tarantula” assigned higher scores to the other 11 methods (i.e., M6, M7, M15-M17, M19, and M22-M26) that have been executed by a lower number of failed test cases (i.e., one failed test). As a result, these methods got higher ranks in the ranking list and will be examined before the actual faulty method M21.

In our example, the faulty method M21 was executed by two failed test cases. As method M21 was executed by more failed test cases compared to the other 11 methods, it should be the most suspicious method and it should get a higher rank than the other 11 methods. After applying our proposed formula, the faulty method M21 has the second highest suspicion score and thus is ranked nearest to the top of the list.

This example clearly shows that the proposed formula works. It can be observed that the obtained scores are only slight modifications of the respective ef values. It is intuitive that code elements that have two failing tests rather than one should be more suspicious. However, from the 11 elements having $ef = 2$, the ones that have fewer passing tests will be ranked higher (smaller ep). This makes element M20 first and M21, the faulty one, second in the ranked list.

¹<http://program-repair.org/defects4j-dissection/#!/bug/Chart/6>

Table 6.1: *Motivating example’s basic statistics*

	ef	ep	nf	np
M1 ..chart.util.SerialUtilities.readShape()	1	121	1	1759
M2 ..chart.util.SerialUtilities.writeShape()	1	121	1	1759
M3 ..chart.util.SerialUtilities.class\$()	1	147	1	1733
M4 ..chart.util.ObjectUtilities.<clinit>()	1	221	1	1659
M5 ..chart.util.ObjectUtilities.equal()	2	683	0	1197
M6 ..chart.util.HashUtilities.hashCode(I)	1	47	1	1833
M7 ..chart.util.HashUtilities.hashCode(II)	1	54	1	1826
M8 ..chart.util.AbstractObjectList.<init>()	2	522	0	1358
M9 ..chart.util.AbstractObjectList.<init>(I)	2	522	0	1358
M10 ..chart.util.AbstractObjectList.<init>(II)	2	522	0	1358
M11 ..chart.util.AbstractObjectList.get()	2	237	0	1643
M12 ..chart.util.AbstractObjectList.set()	2	240	0	1640
M13 ..chart.util.AbstractObjectList.size()	2	429	0	1451
M14 ..chart.util.AbstractObjectList.equals()	2	259	0	1621
M15 ..chart.util.AbstractObjectList.hashCode()	1	47	1	1833
M16 ..chart.util.AbstractObjectList.writeObject()	1	78	1	1802
M17 ..chart.util.AbstractObjectList.readObject()	1	78	1	1802
M18 ..chart.util.ShapeList.<init>()	2	429	0	1451
M19 ..chart.util.ShapeList.getShape()	1	21	1	1859
M20 ..chart.util.ShapeList.setShape()	2	25	0	1855
M21 ..chart.util.ShapeList.equals()	2	221	0	1659
M22 ..chart.util.ShapeList.hashCode()	1	0	1	1880
M23 ..chart.util.ShapeList.writeObject()	1	65	1	1815
M24 ..chart.util.ShapeList.readObject()	1	65	1	1815
M25 ..chart.util.junit.ShapeListTests.testEquals()	1	0	1	1880
M26 ..chart.util.junit.ShapeListTests.testSerialization()	1	0	1	1880

6.3.3 Evaluation

Subject Programs

Here, we used the single faulty programs (i.e., 302 faults) of the Defects4J 1.5 dataset (see Table 2.5). However, 5 faults were excluded due to instrumentation issues. Thus, the final dataset used contained a total of 297 faults.

Evaluation Baselines

In this chapter, 3 widely-studied SBFL formulas, the formulas “Tarantula”, “Ochiai”, and “Barinel” which are presented in Table 2.7, were used as benchmarks against our proposed new formula.

Table 6.2: *Motivating example – scores and ranks*

	Tarantula score	Tarantula rank	New Formula score	New Formula rank
M1	0.886	16.5	1.000	19
M2	0.886	16.5	1.000	19
M3	0.865	19	1.000	19
M4	0.810	22	1.000	19
M5	0.734	26	2.003	11
M6	0.952	6.5	1.000	19
M7	0.946	8	1.000	19
M8	0.783	24	2.004	9
M9	0.783	24	2.004	9
M10	0.783	24	2.004	9
M11	0.888	14	2.008	3
M12	0.887	15	2.008	4
M13	0.814	20.5	2.005	6.5
M14	0.879	18	2.008	5
M15	0.952	6.5	1.000	19
M16	0.923	11.5	1.000	19
M17	0.923	11.5	1.000	19
M18	0.814	20.5	2.005	6.5
M19	0.978	5	1.000	19
M20	0.987	4	2.074	1
M21	0.895	13	2.009	2
M22	1.000	2	1.000	19
M23	0.935	9.5	1.000	19
M24	0.935	9.5	1.000	19
M25	1.000	2	1.000	19
M26	1.000	2	1.000	19

6.3.4 Experimental Results and Discussion

Achieved Improvements in Average Ranks

Table 6.3 presents the average ranks of each SBFL formula compared to our proposed formula and it shows the difference between the average ranks too. If the difference is negative, it indicates that our proposed formula is better.

Table 6.3: *Average ranks comparison*

	Average rank
Tarantula	83.05
New Formula	72.01
Diff.	-11.04
Ochiai	79.25
New Formula	72.01
Diff.	-7.24
Barinel	83.05
New Formula	72.01
Diff.	-11.04

We can see that our proposed formula achieved improvements, providing lower average ranks, compared to all the selected SBFL formulas: the average rank was reduced by about 10 positions overall, which corresponds to 2.4–3.7% with respect to the total number of program elements (i.e., methods), demonstrating that our proposed formula can provide significant improvements.

Answer to RQ1: The effectiveness of SBFL could be improved by using the proposed formula: the average improvement of rank positions in the used benchmark was about 10 positions overall. This indicates that the proposed formula could have a positive impact and enhances the results.

Achieved Improvements in Top-N Categories

Table 6.4 presents the number of bugs in the Top-N categories (cumulative) as well as their percentages for the entire dataset, of the baseline formulas and our proposed one, as well as the differences between them. There has been an improvement if there are fewer bugs in the “Other” category and more bugs in any Top-N category.

Table 6.4: *Top-N categories*

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Tarantula	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
New Formula	59	19.9	124	41.8	148	49.8	178	59.9	119	40.1
Diff.	11	22.9	13	11.7	11	8.0	11	6.6	-11	-8.5
Ochiai	52	17.5	118	39.7	143	48.1	171	57.6	126	42.4
New Formula	59	19.9	124	41.8	148	49.8	178	59.9	119	40.1
Diff.	7	13.5	6	5.0	5	3.5	7	4.1	-7	-5.6
Barinel	48	16.2	111	37.4	137	46.1	167	56.2	130	43.8
New Formula	59	19.9	124	41.8	148	49.8	178	59.9	119	40.1
Diff.	11	22.9	13	11.7	11	8.0	11	6.6	-11	-8.5

It is clear that by relocating many bugs to higher categories, our new formula improves all Top-N categories. 7–11 bugs were moved from the “Other” category with a rank > 10 into one of the higher Top-N categories. This is important as it gives a “new hope” that a bug will be discovered with our proposed formula while without it, it was not very likely. A significant number of improvements are also visible in higher categories; for example, about 10 bugs were located in the Top-1 category. Note that the percentages of bugs in each category for each formula were computed based on the number of faults in Defect4J. While the difference percentages were computed based on the number of faults before applying our proposed formula.

Answer to RQ2: Every Top-N category showed successful outcomes. Additionally, we were able to raise the proportion of instances in which the faulty method was the highest-ranked element by 13–23%. Another interesting finding is that in some cases, we were able to achieve 11% enabling improvement by moving 7–11 bugs from the “Other” category into one of the higher-ranked categories. Such cases are now more likely to be discovered than before.

6.4 Systematic Search for New Formulas: Preliminary Study

6.4.1 Systematic Analysis

Several researchers have been trying to find new and better formulas, and numerous formulas have been proposed in the literature in the past decades, but no research is known to us that carried out a systematic search to find new possible formulas. On the one hand, this is understandable since the number of possible formulas is infinite.

On the other hand, by examining the reported SBFL formulas, we found that groups of them have similar structures, so they could be built using patterns, in other words, *formula templates*. Using a single formula template, the number of formulas that can be generated is finite, thus, the formulas can be systematically produced and examined. Utilizing this property, we examined the set of already reported formulas and defined a single linear/reciprocal formula template for this study that covers several existing formulas. In this template, Basic Statistical Numbers (BSN) will denote the basic statistical numbers, i.e., $BSN = \{ef, ep, nf, np\}$. The following template can literally yield in, for example, the “Barinel”, “Jaccard”, “Kulczynski1”, and “Wong (I-II)” formulas, but also covers the ranking of e.g. “Statistical Bug Isolation (SBI)” [101].

$$\frac{\sum_{t \in BSN \cup \{1\}} n_t t}{\sum_{t \in BSN \cup \{1\}} d_t t} = \frac{n_{ef}ef + n_{ep}ep + n_{nf}nf + n_{np}np + n_1}{d_{ef}ef + d_{ep}ep + d_{nf}nf + d_{np}np + d_1},$$

where $\{n, d\}_{\{ef, ep, nf, np, 1\}} \in \{-1, 0, 1\}$ are the coefficients in the numerator and denominator. This formula yields 29,040 valid and usable formulas (division by zero and constant formulas could be excluded, and $\frac{a}{b} = \frac{-a}{-b}$ are the same). Among these formulas, there are several groups, within which all the formulas have the same ranking effect; the most obvious examples are $\frac{a}{b} \equiv \frac{a+b}{b} \equiv \frac{a-b}{b} \equiv \frac{a+1}{b} \equiv \frac{a-1}{b}$.

6.4.2 Formula Template for the Experiments

In this study, we illustrate the method using a simplified version of the first formula template:

$$\frac{\sum_{t \in \{ef, ep\}} n_t t}{\sum_{t \in \{ef, ep\}} d_t t} = \frac{n_{ef}ef + n_{ep}ep}{d_{ef}ef + d_{ep}ep},$$

where $\{n, d\}_{\{ef, ep\}} \in \{-1, 0, 1\}$ are the coefficients in the numerator and denominator, and $0ef + 0ep$ is treated as constant 1. This template yields 81 formulas. 32 of these formulas are mathematical duplicates ($\frac{-a}{a} = \frac{a}{a}$), 9 of the rest are constants (in forms $\frac{1}{1}$, $\frac{a}{a}$ and $\frac{-a}{a}$). These can be excluded from the examination. The remaining 40 formulas are 20 *normal* and *inverse* pairs ($\frac{a}{b}$ and $\frac{-a}{b}$), that will produce exactly the reverse rankings.

It is theoretically possible that both formulas of a pair produce good results for different subsets of bugs, thus, we cannot leave out either of them from the measurement. However, a normal and its inverse formula have similar attributes and relations to other formulas (Note, that it is arbitrary which element of a pair is treated as normal or inverse).

Formulas in the form $\frac{a}{1}$ are similar in their rankings with their $\frac{1}{-a}$ versions (the latter are denoted by the orange background in Table 6.5). Differences are due to program elements for which the denominator of the used formula equals zero. This gives 8 formulas (plus 8 inverses).

The remaining 12 formulas have 4 denominators, 3 formulas sharing each one. The 3 formulas with the same denominator are equivalent to each other regarding the rankings they produce (their computed suspicion scores differ only by a constant from each other), thus, any two of them can be eliminated from the measurements. Furthermore, the 4 remaining formulas (and their inverses, denoted by green background) are also mostly equivalent to each other as they produce similar rankings except for elements with zero denominators.

Thus, we have 8 + 4 formulas and their 12 inverses. The 24 formulas to be measured are shown in Table 6.5, which includes well-known formulas “Barinel” (F10), “Wong I” (F2), and “Wong II” (F16). Formulas in the same row are equivalent to each other under the condition in the column header. Formulas F13-F24 are the inverses of formulas F1-F12. Note that if we had no program elements for which ep or ef is zero or $ep = ef$, then the formulas in the same row would produce the same rankings, thus, it would be enough to measure only 5 different formulas and their 5 inverses.

Table 6.5: Variants of formulas.

conditions \rightarrow	$ep \neq 0$	$ef \neq 0$	$ep + ef \neq 0$	$ep \neq ef$
F1 = ep	F5 = $\frac{1}{-ep}$			
F2 = ef		F7 = $\frac{1}{-ef}$		
F3 = $ep + ef$			F9 = $\frac{1}{-ep-ef}$	
F4 = $ep - ef$				F11 = $\frac{1}{-ep+ef}$
	F6 = $\frac{ef}{ep}$	F8 = $\frac{-ep}{ef}$	F10 = $\frac{ef}{ep+ef}$	F12 = $\frac{ef}{ep-ef}$
F13 = $-ep$	F17 = $\frac{1}{ep}$			
F14 = $-ef$		F19 = $\frac{1}{ef}$		
F15 = $-ep - ef$			F21 = $\frac{1}{ep+ef}$	
F16 = $-ep + ef$				F23 = $\frac{1}{ep-ef}$
	F18 = $\frac{-ef}{ep}$	F20 = $\frac{ep}{ef}$	F22 = $\frac{-ef}{ep+ef}$	F24 = $\frac{-ef}{ep-ef}$

In the analysis above, we examined the formulas to find equivalences and similarities. Finding equivalences was not our goal, but it reduced the number of formulas to be evaluated. However, for a larger number of automatically generated formulas, this analysis should be performed automatically as well. This issue seems to be a non-trivial mathematical problem that we are planning to investigate in the future.

6.4.3 Evaluation

In this study, we compare all the generated SBFL formulas, presented in Table 6.5, on Defects4J 1.5 (see Table 2.5) to each other to measure the effectiveness of each generated formula.

6.4.4 Experimental Results and Discussion

Achieved Improvements in Average Ranks

Table 6.6 presents the average rank of each generated SBFL formula. It can be noticed that F10 (“Barinel”) performs the best, and formulas F2 (“Wong I”), F6, and F19 also produce much better average ranks than all other formulas. It is worth mentioning that according to [99], F6 is equivalent to F10 in ranking. However, the difference in results shows that $div/0$ conditions ($ef \neq 0$ and $ef + nf \neq 0$) make a difference in practice even for theoretically equivalent formulas. Also, the result of F19 is a bit surprising at first glance, as it is similar to F14, which is the inverse of F2 (“Wong I”). However, F19 fails to handle program elements that do not fail ($ef = 0$). We assign a 0 suspicion score to these elements, thus ranking them lowest in the list, while F2 (“Wong I”) also ranks them lowest due to their natural 0 score. This shows how handling special cases, like $div/0$, can influence the performance of otherwise similar formulas.

Table 6.6: Average ranks of the generated SBFL formulas

Name	Average rank	Name	Average rank
F1	1956.0	F13	4129.55
F2 (Wong I)	156.61	F14	6052.76
F3	1751.38	F15	4381.29
F4	2198.84	F16 (Wong II)	3833.54
F5	2963.85	F17	3119.95
F6	216.16	F18	5899.8
F7	6012.12	F19	205.77
F8	5655.43	F20	497.87
F9	3193.56	F21	2940.67
F10 (Barinel)	38.5	F22	6160.15
F11	2674.05	F23	3373.45
F12	614.03	F24	5393.14

Achieved Improvements in Top-N Categories

Table 6.7 presents the number of bugs in the Top-N categories and their percentages for the dataset.

This evaluation also shows that F10 (“Barinel”), F6, and F12 are the best three formulas from this set. Surprisingly, at least at first glance, the similar F8 performs very badly. The reason is again the handling of program elements with $ef = 0$ value,

Table 6.7: Top-N categories

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
F1	0	0	0	0	0	0	0	0	411	100
F2	10	2	57	14	72	18	110	27	301	73
F3	0	0	0	0	0	0	0	0	411	100
F4	0	0	0	0	0	0	0	0	411	100
F5	0	0	0	0	0	0	0	0	411	100
F6	64	16	142	35	180	44	223	54	188	46
F7	0	0	0	0	0	0	0	0	411	100
F8	0	0	0	0	0	0	0	0	411	100
F9	0	0	0	0	0	0	0	0	411	100
F10	65	16	165	40	201	49	248	60	163	40
F11	15	4	31	8	35	9	38	9	373	91
F12	57	14	124	30	158	38	199	48	212	52
F13	0	0	0	0	0	0	0	0	411	100
F14	0	0	0	0	0	0	0	0	411	100
F15	0	0	0	0	0	0	0	0	411	100
F16	17	4	34	8	36	9	39	9	372	91
F17	0	0	0	0	0	0	0	0	411	100
F18	0	0	0	0	0	0	0	0	411	100
F19	6	0	55	13	71	17	107	26	304	74
F20	13	3	37	9	51	12	74	18	337	82
F21	0	0	0	0	0	0	0	0	411	100
F22	0	0	0	0	0	0	0	0	411	100
F23	0	0	0	0	0	0	0	0	411	100
F24	16	4	32	8	34	8	38	9	373	91

to which elements we assign the 0 scores. As the other scores are negative (at most 0), these not failing elements will be ranked at the top of the list. Other interesting results are those of F12 and F24. These are inverses of each other, yet F24 also ranks 4% of the buggy elements in the Top-1 category. We think the main reason for this (besides the zero denominators, i. e. when $ep = ef$) is that when $ef > ep$ (i. e. more failing tests cover the element than passing ones), F12 turns the score into negative, ranking these elements low, while F24 will result in a positive score (negative numerator divided by the negative denominator), ranking them in the top of the list. The F12 and F24 pair is practical proof that inverse rankings can perform well on different spectra, thus both of them need to be examined.

Answer to RQ1: This study presented a systematic search approach based on a formula template for new SBFL formulas. The results of our preliminary research indicate that the proposed approach deserves further investigation. In this preliminary study, we did not find an automatically generated formula that is not published in the literature but outperforms existing ones (though F6 was a strong candidate and it outperformed “Wong II”). However, since the template we used was very simple, this is not surprising. We hope that by extending the template to other forms, we will be able to identify new formulas which outperform existing ones.

6.5 Systematic Search for New Formulas: Extended Study

In the mentioned preliminary study, we found formulas that outperformed some existing ones but failed to achieve significant improvement over the most successful existing techniques.

In this extended study, we introduce our experimental results with extended formula templates compared to [123]. Our goal is to systematically examine a broader set of formulas and to find out how different the generated formulas will be in terms of their ranking ability. We also outline the possible extensions for future work, primarily in terms of more advanced formula templates. The goal is to be able to cover some already published formulas (as a sanity check that the method is meaningful), and potentially discover new, better ones as we managed to do in this study.

6.5.1 Formula Templates for the Experiments

The basic idea of systematic formula generation is that we combine the four spectrum metrics ef , ep , nf , and np using mathematical operations in all possible combinations. While at first, this seems straightforward, systematic enumeration of all possible SBFL formulas is not simple. Since, in general, the number of possible formulas is infinite, we must limit the types of formulas to a meaningful subclass. For example, all four values can have a fixed exponent at most. But even using basic mathematical operations only, like addition, multiplication, fraction, and only linear combinations of the elements we will face a combinatorial explosion.

Further issues are that many generated formulas will contain elements that can be mathematically simplified or rewritten and that they will still sometimes produce syntactically different, but semantically *equivalent* formulas to each other.

Furthermore, even if two formulas are not mathematically equivalent, they can be *rank-equivalent*. This means that they will produce the same ranking lists; despite the score values being different, in this case, there is a monotonic transformation between the values [162].

At the same time, many of the previously published, manually crafted formulas can fit a relatively simple structure (as opposed to GP-generated ones). Hence, our goal in this research is an *exhaustive exploration of all possible formulas that conform to a specific formula template*. Our goal is to define templates that have the following properties:

- They cover as many existing formulas as possible (this means these are probably useful structures).
- Are combinatorically feasible.
- Can be competitive with manually crafted formulas.

We build on our previous work, where we proposed a systematic search to evaluate SBFL formulas and possibly find new ones [123]. We reported the evaluation of 24 formulas generated by a simple template (see Table 6.5).

The template in Equation 6.2 covers previously reported formulas “Barinel” (= “Braun” = “Coef” = “SBI” = F10) and “Wong I” (= F2) and “Wong II” (= F16) [101].

$$\frac{\sum_{t \in \{ef, ep\}} n_t t}{\sum_{t \in \{ef, ep\}} d_t t} = \frac{n_{ef} ef + n_{ep} ep}{d_{ef} ef + d_{ep} ep}, \quad (6.2)$$

where $n_{ef}, n_{ep}, d_{ef}, d_{ep} \in \{-1, 0, 1\}$.

In this study, we define more elaborate formula templates. The template in Equation 6.3 extends the template in Equation 6.2 with a single spectrum metric (ef , ep , nf , np) by applying a simple arithmetic operation (+, −, ·, /) between them.

$$\begin{aligned} & \frac{\sum_{t \in \{ef, ep\}} n_t t}{\sum_{t \in \{ef, ep\}} d_t t} \otimes \{ef, ep, nf, np\} = \\ & \frac{n_{ef} ef + n_{ep} ep}{d_{ef} ef + d_{ep} ep} \otimes \{ef, ep, nf, np\}, \end{aligned} \quad (6.3)$$

where $n_{\{ef, ep\}}, d_{\{ef, ep\}} \in \{-1, 0, 1\}$ and $\otimes \in \{+, -, \cdot, /\}$.

Equation 6.2 resulted in 81 literal templates, 24 of which remained after sorting out constant and equivalent ones. Equation 6.3 results in $4 \cdot 4 \cdot 81 = 1,296$ formulas literally. However, based on the 24 different non-trivial formula instances of paper [123], there remain only $4 \cdot 4 \cdot 24 = 384$ candidates. Plus $4 \cdot 2 = 8$, the four basic mathematical operations for the two additional spectrum metrics nf and np , which are added because the 24 formula instances from [123] do not include constant-equivalent ones. For example, $\frac{ef}{ef}$ was not examined in [123], but $\frac{ef}{ef} \cdot nf = nf$ is a valid formula instance. At least 80 formula instances generated from this new template have already been covered in [123], and there are at least two pairs of newly generated formulas that are equivalent to each other.

The next formula template we examined is shown in Equation 6.4:

$$\begin{aligned} & \frac{\sum_{t \in \{ef, ep\}} n_{1,t} t}{\sum_{t \in \{ef, ep\}} d_{1,t} t} \otimes \frac{\sum_{t \in \{ef, ep\}} n_{2,t} t}{\sum_{t \in \{ef, ep\}} d_{2,t} t} = \\ & \frac{n_{1,ef} ef + n_{1,ep} ep}{d_{1,ef} ef + d_{1,ep} ep} \otimes \frac{n_{2,ef} ef + n_{2,ep} ep}{d_{2,ef} ef + d_{2,ep} ep}, \end{aligned} \quad (6.4)$$

where $n_{\{1,2\},\{ef, ep\}}, d_{\{1,2\},\{ef, ep\}} \in \{-1, 0, 1\}$ and $\otimes \in \{+, -, \cdot, /\}$.

In this template, we have put together two instances of the formulas generated by the template in Equation 6.2 using a basic arithmetic operation. This would

result in $81 \cdot 81 \cdot 4 = 26,244$ formula instances literally. However, counting on the non-equivalent formulas defined in [123], we can restrict our measurements to $24 \cdot 24 \cdot 4 = 2,304$ literally generated formulas. Furthermore, addition and multiplication are commutative operations, technically halving the resulting formula instances, while addition-subtraction and multiplication-division are inverses that can also yield the same formula when applied on different operands (e.g. $F1 + F2 = F2 - F13$). Subtraction and division can result in identical (constant) formulas, and some combinations can also yield formulas already reported in our previous work [123]. Thus, by rough calculation, at most 400 different formulas exist, but these formulas may contain equivalent ones too.

The two presented formula templates also overlap, generating literally equivalent formulas. However, we did not make a thorough equivalence or ranking equivalence analysis of the resulting formulas for either of the formula templates (the equivalence proofs of different SBFL formulas have been investigated in [162] via a theoretical comparison approach). Instead, we utilized previous equivalence calculations and ran the measurements for all the resulting formulas (including the equivalent ones) and sanity-checked the results based on the discovered equivalences (i.e., checked manually by random sampling if two formulas that should be equivalent really produce the same results).

The above templates cover “Hamming” (= “Lee” = “NFD”) and equivalents of “Euclid” and “Ochiai3” formulas. These formulas [66] are presented in Table 6.8.

Table 6.8: *Formulas covered by new templates (*: only rank-equivalents are covered)*

	Formulas
Hamming	$ef + np$
Euclid*	$\sqrt{ef + np}$
Ochiai3*	$\frac{ef^2}{(ef+ep+nf+np) \cdot (ef+ep)}$

Figure 6.1 shows the systematic steps followed in this extended study to search for new SBFL formulas.

6.5.2 Evaluation

Subject Programs

In this study, we performed two types of measurements. In the first, lightweight measurement, we generated all the formulas from our templates and checked their fault localization performance on Defects4J 1.5 (see Table 2.5).

We used this measurement to check the potential of all the generated formulas. Note that we did not filter out equivalent formulas, all generated ones were measured. We used the results as a sanity check of our implementation: after the measurement, we checked if some theoretically equivalent formulas produce the same

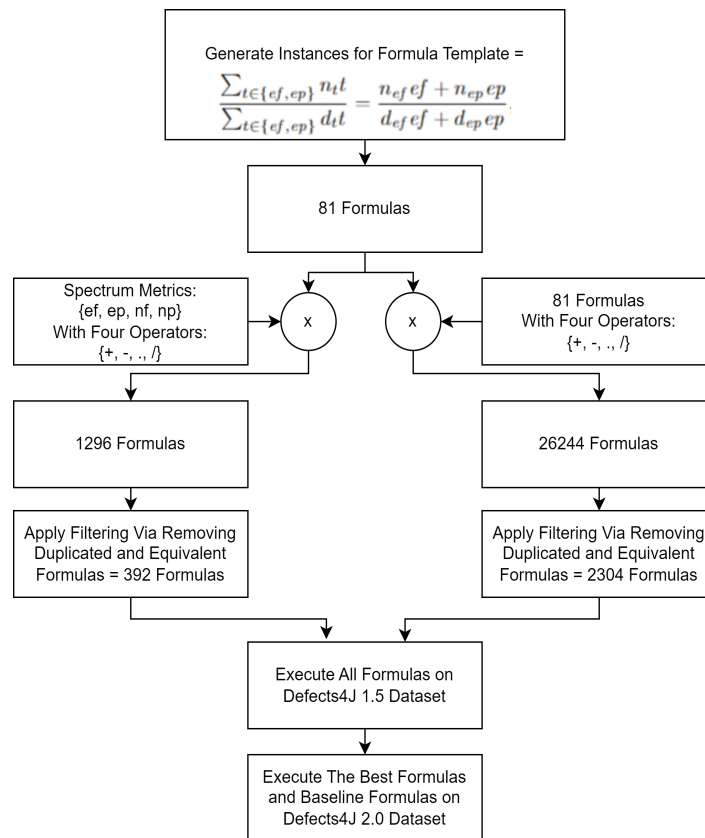


Figure 6.1: Systematic search for new SBFL formulas

numbers. The results are presented in the subsection “Generated formulas” of Section 6.5.3.

Then, based on the results of this first measurement, we re-measured the most promising well-performing formulas (together with the chosen existing baseline formulas) on Defects4J 2.0 (see Table 2.6) as it has more programs and bugs. The other subsections of Section 6.5.3 present the results of this second measurement.

Evaluation Baselines

In this chapter, 9 widely-studied SBFL formulas, the formulas “Barinel”, “Cohen”, “Dice”, “DStar”, “Jaccard”, “Kulczynski1”, “Ochiai”, “SorensenDic”, and “Tarantula”, which are presented in Table 2.7, were used as benchmarks against our new formulas.

6.5.3 Experimental Results and Discussion

Generated formulas

Our template in Equation 6.3 generates 1,296 formula instances, while Equation 6.4 yields 26,244 formula instances. Based on the equivalence calculations of the previous work [123], we generated 392 and 2,304 candidate formulas, respectively. All formulas left out are equivalent with one or more of the candidate ones, and there were several equivalent formulas within these candidate sets too. However, we did not further filter the set, but measured all of the formulas in it, performing a lightweight analysis as described in Section 6.5.2.

As we started to generate formula instances from the ones reported in [123], we could check if the resulting combined formula instance improved fault localization performance. Based on the average ranks, 25.97% of the newly generated formulas improved the ones used in the combination. However, we found 8 formulas that made absolute improvement, i.e., better average ranks than any of the formulas in [123]. By examining each Systematically Generated Formula (SGF) of these 8 formulas, there were equivalent versions of two formulas. The two new formulas are the following:

$$\text{SGF-1} = \frac{ef^2}{ef + ep} \quad (6.5)$$

$$\text{SGF-2} = ef \cdot np \quad (6.6)$$

After finding these formulas, we performed a further analysis of our full set of subject programs using these two new formulas. We compared them to our baselines, that is, to some of the state-of-the-art SBFL formulas listed in Table 2.7. This second analysis has shown that these two formulas can outperform many already published state-of-the-art formulas too (details are presented in the next sections).

One of the advantages of our new formulas is that they produce fewer ties (i.e., fewer number of program elements that share the same suspicion score [60]) compared to the existing ones used in this study. Take this example: we have two program elements with the following spectrum metrics: Element A ($ef = 1$, $ep = 0$, $nf = 3$, $np = 6$) and Element B ($ef = 2$, $ep = 0$, $nf = 2$, $np = 6$). Applying “Tarantula” will result in the same suspicion score (i.e., 1.0) for both elements; this is considered an issue for the developer to which element he/she will examine first. However, SGF-1 will result in a different score for each element; a score of 1.0 for Element A and a score of 2.0 for Element B, while SGF-2 gives a score of 6.0 to Element A and a score of 12.0 to Element B. As a result, this reduces the ties between program elements and the new formulas give greater suspicion scores to program elements that are executed by more failed test cases compared to others.

Also, notice the similarity of SGF-1 with Barinel. The difference is the square of ef in the numerator, which essentially means that our new formula puts a bigger

emphasis on the failed tests executing the code element, which is the most important constituent of SBFL.

The success of SGF-2 also seems logical, as it combines the previously mentioned important element ef with the counter of passing tests that are not executing the element in question. This measure, np is in essence the opposite of the former, and intuition dictates that the bigger this number the more suspicious the current element should be.

Answer to RQ1: Analysis shows that systematic search can lead to new SBFL formulas not present in the literature, whose performance is comparable to or can even outperform existing ones.

Achieved Improvements in Average Ranks

Table 6.9 presents the average ranks of SGF-1 and SGF-2 in equations 6.5 and 6.6 compared to our baseline formulas, for each subject system separately.

Table 6.9: Average ranks (the best values for a particular row are shown in bold)

Project	Barinel	Cohen	Dice	DStar	Jaccard	Kulczynski1	Ochiai	SorensenDice	Tarantula	SGF-1	SGF-2
Chart	15.94	9.42	9.46	9.18	9.46	609.54	8.82	9.46	15.94	8.82	34.34
Cli	16.68	16.63	16.58	15.29	16.58	19.64	15.4	16.58	16.68	15.4	14.01
Closure	97.27	98.67	98.58	87.61	98.58	98.55	88.48	98.58	97.27	88.46	84.23
Codec	28.29	26.44	28.06	28.03	28.06	28	28.06	28.06	28.29	28.06	26.85
Collections	1	1	1	1	1	2155	1	1	1	1	1
Compress	17.62	17.62	17.54	16.01	17.54	43.43	16.12	17.54	17.62	16.12	15.54
Csv	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5
Gson	19.27	19.17	19.17	19.23	19.17	18.9	19.23	19.17	19.27	19.23	19.23
JacksonCore	6.84	6.36	6.36	6.64	6.36	136.64	6.92	6.36	6.78	6.92	7.36
JacksonDatabind	59.53	59.56	59.57	59.11	59.57	234.04	59.12	59.57	59.53	59.12	61.39
JacksonXml	18.6	18.6	18.6	18.6	18.6	72.3	18.6	18.6	18.6	18.6	18.6
Jsoup	34.77	35.88	34.71	34.01	34.71	45.15	33.97	34.71	34.77	33.97	34.29
JxPath	44.24	44.81	45	54.36	45	97.02	54.07	45	44.24	54.07	74.38
Lang	5.37	4.74	4.72	4.62	4.72	147.77	4.67	4.72	5.37	4.67	4.72
Math	9.94	9.81	9.82	9.94	9.82	190.04	10	9.82	9.94	10	10.56
Mockito	32.15	31.07	30.77	28.57	30.77	30.77	28.73	30.77	32.15	28.73	32.17
Time	19.71	19.67	19.63	18.69	19.63	106.5	18.4	19.63	19.71	18.4	21.79
Total Average Rank	41.38	41.46	41.33	38.80	41.33	132.07	38.99	41.33	41.38	38.99	39.97

We can observe that some of the new formulas SGF-1 and SGF-2 can outperform almost all the selected formulas in terms of reducing the average rank or producing the same results. Interestingly, the two formulas are better performing in mutually exclusive cases. For 3 projects almost all of the formulas produced the same ranks, in 3 cases SGF-1 (together with “Ochiai”) produced the best average ranks, while in 3 cases SGF-2 was the best ranking formula in average. Regarding the median values, in 12 projects SGF-1 could produce the best value of the other formulas, while SGF-2 did the same for 7 projects and produced even better medians in 5 additional cases. In two cases, SGF-2 produced better maximum ranks than any other examined formula. Overall, “DStar” is still the best average performing formula, but as we can observe, the margin is very small, and we can see later on the detailed data it is not necessarily a clear winner.

Another interesting phenomenon is that SGF-1 produced the same average ranks as “Ochiai” for all but one case; for the “Closure” program it was 0.02 ranks better. A possible explanation for this is the following. By squaring the “Ochiai” formula, – this transformation being monotonic – it will result in the same ranking in many cases because we get SGF-1 divided by $ef + nf$. But, in most of the cases, we have only a single failing test case, which means this factor will be 1, not modifying the score and the rank of the two formulas.

Answer to RQ2: We noticed that the two new formulas can improve the performance of SBFL by reducing the ranks compared to most of the baseline formulas, while the results are very similar to some of the existing ones. The two formulas produce better results in mutually exclusive cases. The average improvement of rank positions in the used benchmark was about 2 positions overall.

Achieved Improvements in Top-N Categories

Table 6.10 presents the number of bugs in the Top-N categories and their percentages for the used dataset. It can be noted that the newly generated formulas achieve improvements in all categories by moving many bugs to higher-ranked categories compared to almost all the other formulas. For example, both new formulas move more bugs to the Top-1 category compared to almost all the other formulas and move more bugs also from the “Other” category to the higher ranks categories compared to almost all the others. In particular, SGF-2 managed to put most bugs into first place, compared to any of the baselines, and to SGF-1 too.

Table 6.10: *Top-N categories*

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Barinel	98	12.53	265	33.89	344	43.99	437	55.88	345	44.12
Cohen	104	13.30	271	34.65	344	43.99	438	56.01	344	43.99
Dice	104	13.30	271	34.65	344	43.99	438	56.01	344	43.99
DStar	103	13.17	274	35.04	352	45.01	450	57.54	332	42.46
Jaccard	104	13.30	271	34.65	344	43.99	438	56.01	344	43.99
Kulczynski1	103	13.17	251	32.10	319	40.79	407	52.05	375	47.95
Ochiai	106	13.55	276	35.29	353	45.14	450	57.54	332	42.46
SorensenDice	104	13.30	271	34.65	344	43.99	438	56.01	344	43.99
Tarantula	98	12.53	265	33.89	344	43.99	437	55.88	345	44.12
SGF-1	106	13.55	276	35.29	353	45.14	450	57.54	332	42.46
SGF-2	119	15.22	275	35.17	348	44.50	449	57.42	333	42.58

Answer to RQ3: The two new formulas showed improvements in the Top-N categories. Using SGF-1, we were able to increase the number of cases where the faulty method became the top-ranked element by 2–8%, and by using SGF-2 this rate was 13–21%. SGF-2 produced the most Top-1 elements overall. In some cases, we were able to achieve 13% enabling improvement by moving 12–43 bugs from the “Other” category into one of the higher-ranked categories by using the formula SGF-1, while by using SGF-2 this rate was 12% (enabling improvements for 11–42 bugs).

6.6 Contributions

In this chapter, the following points summarize my main contributions to the topic of thesis point IV. The results of this chapter were published in [120], [123], and [124].

- Providing the idea of introducing new formulas to improve the effectiveness of SBFL.
- Gathering and discussing the related papers.
- Developing several new formulas and comparing their effectiveness to the existing formulas.
- Evaluating and discussing the experimental results of the proposed new SBFL formulas.

Chapter 7

SBFL Supporting Tools

7.1 Introduction

Fault localization is a time-consuming task in software debugging. Several tools are available to aid developers with the fault localization process and its automation. However, they mostly target programs written in Java and C/C++ programming languages. Although these existing tools are helpful, we must not overlook the fact that Python is currently the most widely used programming language. For Python developers, there are still not enough fault localization tools with various features available to help them debug their programs.

In this chapter, we present two software fault localization tools, namely “CharmFL”¹ and “SFLaaS”², for Python developers. “CharmFL” (see Section 7.3) is a plug-in for PyCharm IDE, a popular Python development platform [113]. And “SFLaaS” (see Section 7.4) is provided in the form of software as a service.

The tools employ SBFL to help Python developers automatically analyze their programs and generate useful data at runtime that can then be used to generate a ranked list of potentially faulty program elements. Thus, the proposed tools support different code coverage types with the possibility to investigate these types in a hierarchical approach. To determine whether an element is faulty or not, developers examine each element in turn, beginning at the top of the list (the most suspicious element). Also, the tools are implemented with many other helpful and practical features to aid Python developers in debugging their programs. Several lab experiments with Python projects were conducted to assess the applicability of our tools. The results indicate that the tools are useful for locating faults in various types of programs and are easy to use.

¹<https://sed-szeged.github.io/SpectrumBasedFaultLocalization/>

²<https://sflaas.daxazi.com/>

7.2 Related Works

There are many software fault localization tools implemented and proposed in the literature. This section briefly presents them. The authors in [67] proposed a standalone software fault localization tool called “Tarantula” to help C programmers debug their programs. The tool assigns different colors to program statements based on how suspicious they are, ranging from red (most suspicious) to green (not suspicious). Besides, the tool displays varying brightness levels based on how frequently the tests execute a statement. The brightest statements are those that are most commonly executed. However, the tool does not run test cases and record their results; it takes as input a program’s source code and the results of executing a test suite on the program. Furthermore, the tool’s only supported formula is the Tarantula formula.

In [25], the authors proposed an Eclipse plug-in tool called “Crisp”, which helps developers identify the reasons for a failure that occurs due to code edits by constructing intermediate versions of a program that is being edited. For example, if a test case fails, the tool will identify parts of the program that have been changed and caused the failing test. Thus, developers can concentrate only on those affecting changes that were applied.

In [76], the authors proposed a standalone debugging tool called “Whyline” for Java programs. The tool employs both static and dynamic slicing to formulate why and why-not questions, which are then presented in a graphical and interactive way to help developers in understanding the behavior of a program under test. It also records program execution traces and the status of each used class whether it is executed or not. Using the tool also allows the user to load the execution trace of a program and select a program element at a specific point during its execution. Then he/she can click on the selected element to bring up a pop-up window containing a set of questions that include data values gathered during the execution as well as information about the properties of the selected element.

In [49], the authors proposed an Eclipse plug-in tool called “VIDA” for programs written in Java. The tool extracts the hit spectra of statements from the target programs, executes JUnit tests, and based on their results, calculates suspiciousness. It also provides a list of the ten most suspicious statements as potential breakpoints. It displays the history of breakpoints including the developers’ previous estimates of the correctness of the breakpoint candidates as well as their current suspiciousness. Moreover, it employs colors to distinguish between the developers’ estimations, ranging from red (wrong) to green (correct), and suspiciousness, ranging from black (very suspicious) to light gray (less suspicious). Additionally, it provides the users with the ability to extract static dependency graphs from their programs to assist developers with their estimations and also to help them understand the relationships among different program elements.

In [20, 61], the authors proposed a fault localization tool that adopts SBFL and it is available as a command line tool called “Zoltar” and as an Eclipse plug-in called “Gzoltar”. The tool provides a complete infrastructure to automatically instrument the source code of the programs under test in order to generate runtime data, which

is then used to return a ranked list of faulty locations. It also uses colors to mark the execution of program elements from red to green based on their suspiciousness scores. The tool only employs the “Ochiai” formula to compute suspiciousness.

In [143], the authors proposed a fault localization tool called “FLAVS” for developers using the Microsoft Visual Studio platform. The tool provides an automatic instrumentation mechanism to record program spectra information during the execution. It also provides a user with two options to either automatically or manually mark the result of each used test case; whether it is successful or not. Additionally, it monitors each test environmental factor of the running program, such as memory consumption, CPU usage, and thread numbers. For example, the developer can notice that there is something wrong when the CPU time drops to zero and never gets increased again during the running of a test case. The tool provides different levels of granularities for fault localization analysis, such as statement, predicate, and function. Using the tool allows the users to examine the correct positions in the source code files by clicking on the suspicious units, which are displayed and highlighted in different colors as well. The functionalities of “FLAVS” have been extended by the authors in [23] in another tool called “UnitFL”. The tool uses program slicing to decrease the program execution time. Besides, it provides different levels of granularities for fault localization analysis to provide different aspects of execution during the program analysis. Moreover, it shows fault-related elements with different colors based on their suspiciousness; ranging from green to red.

In [118], the authors proposed an SBFL tool called “Jaguar” for Java developers. The tool supports two advanced spectra types, which are control flow and data flow. Also, it visualizes suspicious program elements where the user can easily inspect suspicious methods, statements, or variables. Although the data flow spectra provide more information, it is not widely adopted in SBFL because of the high costs of execution. To overcome this issue, the tool utilizes a lightweight data flow spectra coverage tool called “ba-dua”. This enables the tool to be used for testing large-scale programs at affordable execution costs. The tool can be used as an Eclipse plug-in or as a command line tool.

All the previous tools target programs written in Java and C/C++ programming languages. Tools for helping Python developers in their debugging process have not been previously proposed in the literature by other researchers. However, two open-source fault localization tools for Python’s pytest testing framework are available, namely, “Fault-Localization” [40] and “PinPoint” [111]. In this chapter, we propose the “CharmFL” and “SFLaaS” tools with more features to target programs written in Python; which is considered one of the most popular programming languages nowadays. Compared to the other two tools, our proposed tools support different types of code coverage (i.e., class, method, and statement), display the fault localization results in different ways, provide a graphical user-friendly interface to examine the suspicious elements, and enable the user to smoothly examine any suspicious element via clickable links to the source code. Table 7.1 summarizes the features of our proposed tools compared to the others.

Table 7.1: Comparison among Python fault localization tools

Features	Fault-Localization	PinPoint	CharmFL	SFLaaS
Statement hit coverage	Yes	Yes	Yes	Yes
Method hit coverage	No	No	Yes	No
Class hit coverage	No	No	Yes	No
Supported SBFL formulas	1	5	4	80+
User-defined formulas	No	No	No	Yes
Shows ranking	Color-based	Value-based list	Colors and values	Colors and values
Shows suspicion scores	Yes	No	Yes	Yes
Ties ranking	No	No	Min, Max, or Average	Min, Max, or Average
GUI	No	No	Yes	Yes
Command line interface	Yes	Yes	Yes	No
Elements investigation	Flat	Flat	Hierarchy	Flat
Elements navigation	No	No	Via clickable links to each element in the source code	Via clickable links to each element in the source code
Tool type	An option for pytest framework	An option for pytest framework	A plug-in for PyCharm IDE	Software as a service
Installation	Required	Required	Required	Not required

7.3 CharmFL Tool

This section presents an overview of the tool’s user interface, architecture, and how it can be used. The tool is divided into two parts: plug-in and framework. The developer can use the plug-in for the PyCharm IDE when debugging inside PyCharm (see Section 7.3.1). Or, the developer can use the framework to integrate the functionality of “CharmFL” into other development environments (see Section 7.3.2).

7.3.1 CharmFL’s User Interface

The interface of the plug-in part of the tool is shown in Figure 7.1. After installing the plug-in in PyCharm, the developer can see a simple “CharmFL” menu with several options. To start the fault localization process, the developer has to open a Python project (which includes source code and tests) and then select the option “Run” to get the ranking list of suspicious program elements.

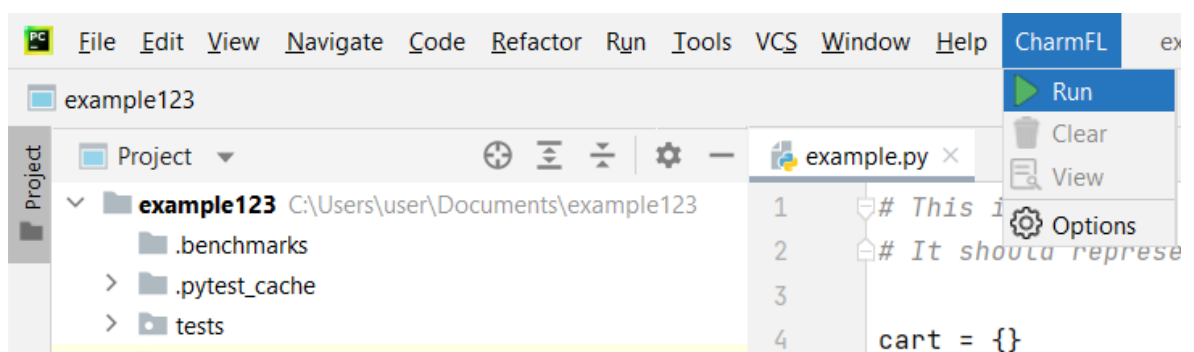
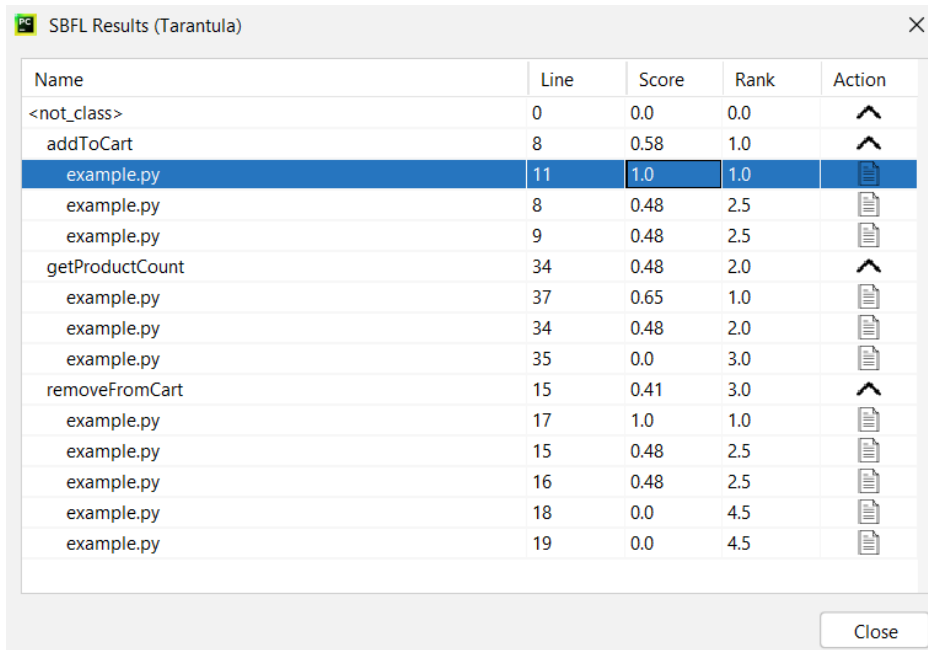
**Figure 7.1:** CharmFL GUI

Figure 7.2) shows the list of program elements hierarchically. For each program element, the developer can see the line/location number of the element in the source code file, its suspicion score, and its rank. In the “Action” column, the developer can hide/show the elements inside each level of the hierarchy (e.g., to show all the statements inside a method). Then, he/she can navigate to a specific element in the source code file by clicking on its corresponding document icon. If the results table is closed by mistake, the developer can reopen it again by clicking on the “View” option in the “CharmFL” menu.



Name	Line	Score	Rank	Action
<not_class>	0	0.0	0.0	^
addToCart	8	0.58	1.0	^
example.py	11	1.0	1.0	📄
example.py	8	0.48	2.5	📄
example.py	9	0.48	2.5	📄
getProductCount	34	0.48	2.0	^
example.py	37	0.65	1.0	📄
example.py	34	0.48	2.0	📄
example.py	35	0.0	3.0	📄
removeFromCart	15	0.41	3.0	^
example.py	17	1.0	1.0	📄
example.py	15	0.48	2.5	📄
example.py	16	0.48	2.5	📄
example.py	18	0.0	4.5	📄
example.py	19	0.0	4.5	📄

Close

Figure 7.2: CharmFL ranking list output

The program elements in the source code file will be highlighted with different colors, ranging from red (most suspicious) to green (not suspicious), based on the suspicion scores as shown in Figure 7.3.

It is worth mentioning that the developer can click on the “Options” button of the menu to show some advanced options as shown in Figure 7.4. These options allow the developer to select different formulas and tie-breaking methods. A results table will be provided for each formula the developer chooses. This is especially good for comparing the output of different formulas.

7.3.2 CharmFL’s Architecture

The framework part of the tool is responsible for gathering and processing the code coverage and test result data. It can be used as an independent tool that can be run via a command line interface or as a plug-in within other IDEs.

```

7 def addToCart(product):
8     if(product not in cart.keys()):
9         cart[str(product)] = 1
10    else:
11        cart[str(product)] = cart[str(product)] + 2_# bug -> should be 1
12
13
14 def removeFromCart(product):
15     if(product in cart.keys()):
16         if(cart[str(product)] > 1):
17             cart[str(product)] = cart[str(product)] - 1
18         elif(cart[str(product)] == 1):
19             del cart[str(product)]
20     else:
21         print("Something's fishy")

```

Figure 7.3: Highlighted statements based on suspicion scores

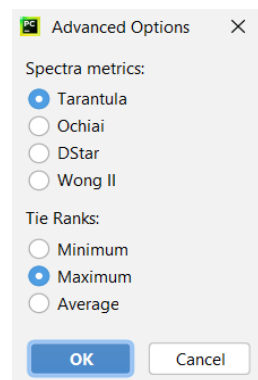


Figure 7.4: CharmFL advanced options

Code coverage measurement is required to gather the program’s spectra. To do so, the framework uses “coverage.py” [29], the popular code coverage measuring tool for Python. It shows which parts of the target program are being executed by tests and which are not. “coverage.py” only measures either statement or branch coverage levels. Our framework transforms the statement level coverage to method and class levels as shown in Figure 7.2. This is achieved by putting all the statements of each function under the corresponding function’s name and then putting all the functions of each class under the corresponding class’s name. Thus, each function will have its own set of statements, and each class its own set of functions including the statements. Afterward, the classes are sorted based on their suspiciousness scores, then the functions, and finally the statements. For example, the statement in line 37 will not be examined before the statement in line 8 because the latter belongs to a function of higher rank in the ranking list.

This hierarchical coverage feature gives additional useful information about the suspicion scores on all layers to the user. The user can examine the suspicious elements from the highest level in the hierarchy (i.e., classes) to lower levels (in the hierarchy) and repeat the steps above until he/she reaches the lowest level, which is

the statements level. This is better than only one level of granularity as the developer can exclude methods or even classes from the ranking list. As a result, debugging time is reduced.

The framework also uses the “pytest” [114] to run the test cases and get their results after gathering the coverage report. Then, the coverage and test results matrix will be built according to Jones et al. [67]. The framework calculates the suspiciousness score for each program element in the matrix based on the selected SBFL formula. The framework provides different code coverage levels, test results, a coverage matrix, and a hierarchical ranking list. Table 7.2 lists some of the main commands of how to use the framework from a command line interface.

Table 7.2: *Framework usage commands*

Command	Purpose
main.py -d <project_directory>	To start the fault localization process
main.py -c <file_name>	To get class coverage
main.py -m <file_name>	To get method coverage
main.py -s <file_name>	To get the spectra
main.py -r <rank_type>	To apply a specific tie-breaking method

7.3.3 How to Use CharmFL

A Python program of four methods and four test cases, shown in Figure 7.5, will be used to illustrate how “CharmFL” can be used for fault localization. In order to keep things simple, we will refer to the four test cases in the text as T1, T2, T3, and T4 in the order they appear in the figure.

It can be seen that the highest granularity in the example program is the method level as there are no classes. Table 7.3 presents the method-level coverage matrix and the basic statistical numbers.

Table 7.3: *Method hit spectra (with four basic statistics)*

	T1	T2	T3	T4	ef	ep	nf	np
addToCart	1	1	1	1	2	2	0	0
removeFromCart	0	1	1	0	1	1	1	1
printProductsInCart	0	0	0	0	0	0	2	2
getProductCount	1	1	1	1	2	2	0	0
Test results	0	0	1	1				

Any formula, such as “Tarantula”, can be used to generate a list of elements (e.g., methods) with scores indicating their level of suspiciousness, as presented in Table 7.4. The developer starts examining the elements with the highest scores until he/she finds the bug. It can be noted that the “addToCart” method has the highest suspiciousness score. Thus, all the statements in the “addToCart” method have to be examined first.

```

cart = {}
def addToCart(product):
    if(product not in cart.keys()):
        cart[str(product)] = 1
    else:
        cart[str(product)] = cart[str(product)] + 2 # bug -> should be 1

def removeFromCart(product):
    if(product in cart.keys()):
        if(cart[str(product)] > 1):
            cart[str(product)] = cart[str(product)] - 1
        elif(cart[str(product)] == 1):
            del cart[str(product)]
    else:
        print("Something's fishy")

def printProductsInCart():
    print("Your cart: ")
    for product_name in cart.keys():
        product_count = cart[str(product_name)]
        print("* " + str(product_name) + ": " + str(product_count))

def getProductCount(product):
    if(product not in cart.keys()):
        return 0
    else:
        return cart[str(product)]

import pytest
import example

def test_AddProduct_Once():
    example.cart.clear()
    example.addToCart("Apple")
    assert example.getProductCount("Apple") == 1

def test_RemoveProduct_WhenTheresOne():
    example.cart.clear()
    example.addToCart("Apple")
    example.removeFromCart("Apple")
    assert example.getProductCount("Apple") == 0

def test_RemoveProduct_FromMoreThanOne():
    example.cart.clear()
    example.addToCart("Apple")
    example.addToCart("Apple")
    example.removeFromCart("Apple")
    assert example.getProductCount("Apple") == 1

def test_AddProduct_MoreThanOnce():
    example.cart.clear()
    example.addToCart("Apple")
    example.addToCart("Apple")
    assert example.getProductCount("Apple") == 2

```

Figure 7.5: Running example – program code and its tests

Table 7.4: Tarantula suspiciousness scores

Method	Score
addToCart	0.58
removeFromCart	0.41
printProductsInCart	0.00
getProductCount	0.48

Developers can use the “CharmFL” tool in different scenarios when debugging their programs, as follows:

1. Executing the test suite and then using the “CharmFL” tool.
2. Using the “CharmFL” tool and then injecting breakpoints.

The developer begins each scenario at the moment when the bug was first discovered (i.e., when it was reported by a user of the software). The same example program, shown in Figure 7.5, will be used here for the demonstration; nevertheless, any other Python program can also be used. The program has a bug, in the “addToCart” method, that has been executed by two failed tests (i.e., T3 and T4). The “CharmFL” tool is considered effective if the bug appears in the Top-10 of the ranking list and debugging is considered successful if the tests pass after the bug is corrected.

In the first scenario, the developer runs the test cases using “pytest”. From the generated test report, he/she will know that there are two failed tests. In the failed tests, the method “addToCart” is called two times, and the method “removeFromCart” is called once. Thus, the developer starts examining the method “addToCart” first. Examining the method “addToCart” shows that it has three statements, including the

buggy one. The developer then starts the “CharmFL” tool and uses it to decide which statement to examine first. The tool suggests that the third statement is the most suspicious one, as shown in Figure 7.2. Here, the developer has to navigate to the element with the highest score in the ranking list and fix the bug in it. Then, the developer will once again run the tests to see that all of them now pass as the bug is corrected. In this scenario, the “CharmFL” tool assisted the developer in selecting the first statement to be examined, thus saving time during debugging.

In the second scenario, the developer runs the “CharmFL” tool and then clicks on the element with the highest score in the suspiciousness list, shown in Figure 7.2. The tool will redirect the developer to the location of the statement in the source code file. Here, he/she can insert breakpoints (to and around the clicked suspicious statement) and then start the debugging session to look into what values the variables take and what is not working out as expected. The developer then fixes the buggy statement and runs the tests to ensure all of them pass.

7.4 Software Fault Localization as a Service (SFLaaS) Tool

We also developed another tool named “SFLaaS” for locating faults in programs written in Python, a popular programming language, and is provided as a service rather than as a plugin or a command line tool that needs to be installed. Thus, our tool, which also employs SBFL, does not require any installation from the user side and can be accessed anytime and from anywhere. Our proposed tool supports different important features in fault localization, such as supporting about 80 SBFL formulas, different tie-breaking methods, showing code elements with different colors, ranging from most suspicious (red) to not suspicious (green) based on their suspicion scores, allowing the user to define his/her own formula, etc. Because of the aforementioned features, using our tool could be useful for educational purposes, such as teaching students fault localization. Also, it could help developers efficiently find the locations of different types of faults in their programs.

7.4.1 SFLaaS’s User Interface

The user interface of “SFLaaS” is shown in Figure 7.6. It can be noted that many options are provided for the user to start the software fault localization process.

The main features of “SFLaaS” are listed below:

Accessibility

Unlike plugins, command line, or standalone tools, our tool can be accessed anytime and from anywhere as it is provided as a service. Thus, the user only needs a browser and an Internet connection.

SFLaaS: Software Fault Localization as a Service

Upload your python files and submit to show the testing results.

Upload files: _____

Upload your program here

No file chosen

Upload your test cases here

No file chosen

Or write your program here: _____

<p>main.py</p> <table style="width: 100%; border-collapse: collapse;"> <tr><td style="width: 20px; border-right: 1px solid black; padding: 2px;">1</td><td style="padding: 2px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 2px;">2</td><td style="padding: 2px;"></td></tr> </table>	1		2		<p>test_main.py</p> <table style="width: 100%; border-collapse: collapse;"> <tr><td style="width: 20px; border-right: 1px solid black; padding: 2px;">1</td><td style="padding: 2px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 2px;">2</td><td style="padding: 2px;"></td></tr> </table>	1		2	
1									
2									
1									
2									

Select Ranking Method: _____

▾

Select Formula: _____

User defined formula

Results: _____

	YOUR PYTHON CODE	RESULT
1		

Figure 7.6: SFLaaS GUI

Easy upgrades

It does not require manual installation, configuration, or updates on the user's side as the service provider deals with hardware and software updates; thus removing this workload and responsibility from the user.

Code Editor

It enables the user to write the code of his/her Python program and its test cases directly into an editor provided by "SFLaaS". This is useful especially when the tool is used for educational purposes, such as teaching students fault localization.

Tie-breaking methods

It enables the user to select a tie-breaking method (e.g., Min, Max, or Average) and apply it to the elements sharing the same suspicion score in the ranking list.

Formulas selection

It enables the user to select one or more SBFL formulas (e.g., "Tarantula", "Ochiai", "Barinel", etc.). In our tool, we have implemented about 80 formulas that have been

proposed in the literature. This is especially important for researchers who would like to compare the efficiency of different SBFL formulas with each other.

User-defined formulas

It enables the user to define his/her own formula either by combining existing formulas or by introducing new formulas via combining different statistical numbers (i.e., ef , ep , nf , and np). This is crucial when comparing newly proposed formulas to the existing ones.

Highlighted code elements

When SBFL is performed, the corresponding code elements are highlighted with different colors, ranging from red (most suspicious) to green (not suspicious), based on the suspicion scores as shown in Figure 7.7.

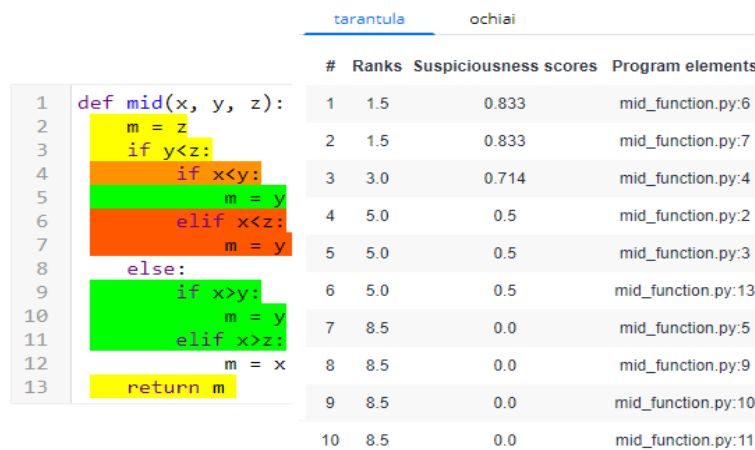


Figure 7.7: Highlighted statements based on suspicion scores

Navigation

The SBFL results in Figure 7.7 present the program elements with their positions in the source code, ranks, and scores. Clicking on an element in the SBFL results table puts the cursor at the element's location in the source code to be easily examined by the user.

7.4.2 SFLaaS's Architecture

The architecture of our tool is shown in Figure 7.8. We run the test cases on the target program using “pytest” [114] to fetch the results. To collect the program's spectra on the statements level, code coverage measurement is required. The program must

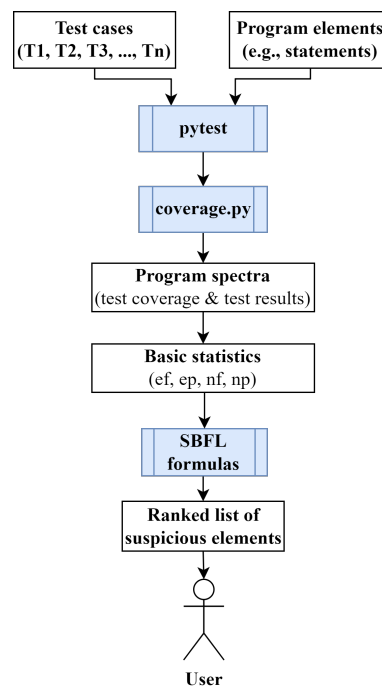


Figure 7.8: Architecture of SFLaaS

be instrumented to generate code coverage. Therefore, the famous Python coverage measuring framework, called “coverage.py” [29] has been used in our tool.

Next, the tool constructs coverage and test results from the gathered data. Then, based on the specified SBFL formulas, it scores the suspiciousness of each program element. It is worth mentioning that all the aforementioned steps are performed on the server side. Also, there are no mandatory elements (other than a Python code and its tests to be given) for using the tool. For instance, if the developer writes/uploads a piece of code, it is not mandatory to install/upload the Pytest, Coverage.py, or Python as all the necessary elements for running the tool are already installed on the server side. Figure 7.9 shows the technical details of how our tool works.

The front-end interface allows users to submit their Python programs to be debugged. Upon submission, the PHP back-end stores the program files and user settings in a MySQL database. The front-end then waits for the response. A C# application constantly listens for new program submissions. When a new submission is detected, the C# application downloads the program files and user settings from the MySQL database to the server. The C# application then runs the SBFL algorithm. After the program has completed executing, the results are stored in the database, and the files are deleted from MySQL and the server. The front-end of the application retrieves the results and displays them to the user.

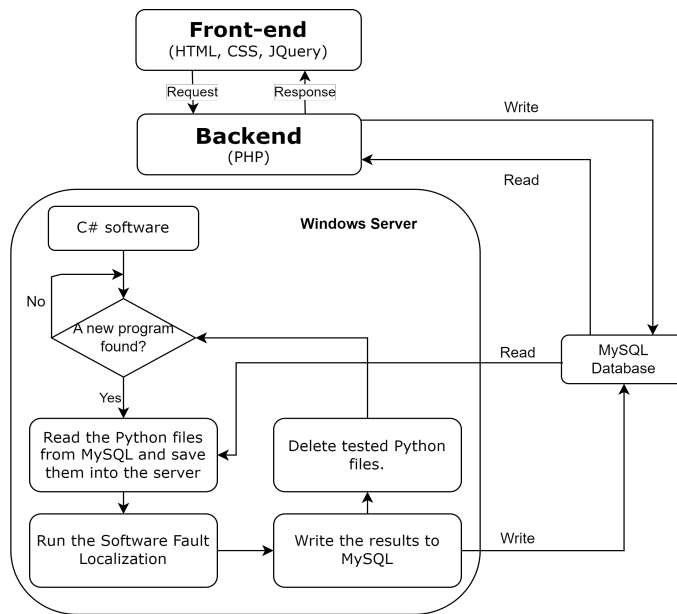


Figure 7.9: Technical details of SFLaaS

7.4.3 How to Use SFLaaS

In this section, we will describe how our tool can be used to locate faults in Python programs with an applicability scenario. The user has two options to submit his/her program and its tests to the tool: (a) the user uploads a Python program file and its related tests file using the buttons made for this purpose; (b) the user writes his/her program and its tests in a particular editing area specified for this purpose. Then, he/she starts the fault localization process by clicking on the “Submit” button as shown in Figure 7.6. The tool then provides the ranking list of suspicious statements of the uploaded program. The user clicks on the first statement in the list with the highest score, and the tool redirects the user to the statement location in the source code for investigation. If it is a bug, then the user can fix it. Then, the user re-runs the tests and notices the pass state of all the test cases. This indicates that the bug is fixed and the task terminates. If the statement, however, did not lead to the error, the user may go on to the following statement in the list based on the ranks. The user goes through the statements one by one until he/she finds the one that is causing the fault. It is worth mentioning that each uploaded program gets deleted after its execution on the server side; this is very important to ensure privacy. Only the top ten elements from the ranking list are explored by the developers because, after that, they begin to lose the desire to follow up the fault localization tools [77, 158]. Thus, a tool is considered successful, if it puts the most faulty elements in the Top-10 ranks.

We tested the tool in lab settings with researchers and students, but we do not have much practical experience with its usefulness among professional programmers. Hence, we would also like to draw the attention of the developers and research communities and invite them to test the tool to understand its benefits and provide

constructive feedback about enhancing its usability and user experience.

7.5 Contributions

In this chapter, the following points summarize my main contributions to the topic of thesis point V. The results of this chapter were published in [127], [134], [126], and [125].

- Regarding the “SFLaaS” tool, I did the following:
 - Developed the fault localization tool as a service to support SBFL for Python developers.
 - Performed the literature review of the currently available tools.
 - Prepared the use cases of the tool.
- Regarding the “CharmFL” tool, I participated in the following:
 - Developed the fault localization tool to support SBFL for Python developers.
 - Performed the literature review of the currently available tools.
 - Prepared the use cases of the tool.

Chapter 8

Conclusions and Future Work

Software products cover many aspects of our day-to-day life and our world could not be imagined without different types of software products that automate most of our activities. Therefore, developing high-quality software is crucial. However, faults are almost unavoidable in software products even with all the current advancements in software development. Locating faults in software is a difficult, time-consuming, tedious, and costly task. To overcome this issue, many fault localization techniques have been proposed in the literature. Compared to other available techniques, SBFL is considered the most prominent one. It computes the suspiciousness of each program element of being faulty based on execution information gathered from test cases, their results, and their corresponding code coverage.

However, SBFL poses many issues and challenges that prevent software developers from using it widely in the industry. In this thesis, we first tried to identify what exactly these issues and challenges are. Then, we tried to tackle many of them to improve the effectiveness of SBFL and make it more applicable in different contexts.

8.1 Conclusions and Future Work

In this section, the content of each chapter is summarized and concluded.

In **Chapter 3**, we started our thesis with an important systematic survey study on the topic. As a result, several important issues and challenges of SBFL have been identified and categorized in this survey study. In each category, the most important issues have been briefly presented with some possible ideas to address them.

In **Chapter 4**, we proposed a method to break the ties between program elements when they are ranked by an SBFL formula. Rank ties in SBFL are very common regardless of the formula employed, and by breaking these ties, improvements to localization effectiveness can be expected. We propose the use of method call contexts for breaking critical ties in SBFL. We rely on instances of call stack traces, which are useful software artifacts during runtime and can often help developers in debugging. The frequency of the occurrence of methods in different call stack instances determines the position of the code elements within the set of other methods tied together by the same suspiciousness score.

Experimental results show that the proposed tie-breaking strategy, using the Defects4J benchmark, (a) completely eliminated many critical ties with a significant reduction of others, and (b) achieved improvements in average rank positions for all investigated SBFL formulas by moving many bugs to the highest Top-N rank positions. However, there are limits to how much improvement one can expect from tie-breaking alone (we analyzed this limit and compared it to the results achieved). This means that no matter how clever a tie-breaking method is, it cannot rearrange code elements outside of the tied ranking positions. Since ties seem to be prevalent, it could be interesting further research to devise specific tie-aware approaches or modified formulas that minimize ties in the scores and/or break them automatically.

In the future, we would like to do the following:

- Measure the effectiveness of the proposed tie-breaking strategy on other levels of granularity, such as statement, branch, etc.
- Employ other SBFL formulas across a much broader range of programs in terms of numbers, types, sizes, and used programming languages, to capture the ties problem characteristics and identify what factors affect them would be interesting for further investigation.
- Employ other contextual factors beyond method call frequency to tackle the ties problem and to measure their impacts on SBFL.

In **Chapter 5**, we enhanced SBFL by proposing the use of emphasis on the failing tests that execute the program element under consideration in SBFL. We rely on the intuition that if a code element gets executed in more failed test cases compared to the other elements, it will be more suspicious, and it will be given a higher ranking. This is achieved by multiplying the initial suspicion score, computed by underlying SBFL formulas, of each program method by an importance weight that represents the rate of executing a method in failed test cases.

The main features of the proposed approach are: (a) it can be applied to a wide range of SBFL formulas without modifying a formula's structure or its concept; (b) it solves the issue of an unbalanced SBFL matrix in the sense that there are many more passing tests than failing ones, and many formulas treat passing and failing tests similarly. The experimental results of this study show that by shifting several bugs to the highest Top-N ranks, our approach improved the average ranks for all investigated formulas.

In the future, we would like to do the following:

- Assess the effectiveness of our approach at different levels of granularity, such as the statement level.
- Involve other SBFL formulas in the study to identify which formulas give the best results and categorize them according to that into groups would be interesting for further investigation.

- Employ other contextual/importance weights beyond method executions in failed test cases and determine how they affect SBFL's efficiency.

In **Chapter 6**, we proposed a new SBFL ranking formula to automatically lead developers to the locations of faults in programs. It is based on the intuition that ties often happen because of shared ef and nf values, and in this case, more failing tests (larger ef) and/or fewer passing ones (smaller ep) will determine the outcome.

Via an evaluation across 297 different single-fault programs of Defects4J, the proposed formula is shown to be more effective than all the selected SBFL formulas in this study. It approves the average rank and the Top-N categories as well.

Introducing new SBFL formulas is an interesting line of research. Sometimes we can get good results from not-so-obvious formulas or a simple combination of ef , ep , nf , and np . Therefore, we performed a more systematic approach to finding new formulas.

In this systematic search approach for new formulas, we use only the four basic statistical numbers from the spectra. For this purpose, formula templates are determined and the possible formulas are generated automatically. As a demonstration, we used a formula template to systematically generate all formulas for that template, then these were analyzed and their effectiveness was evaluated on the Defects4J dataset. Interestingly, the analysis has shown that in theory several formulas generated from the same template are equivalent to or should similarly rank elements to each other, while the handling of special cases (like division-by-zero) can significantly influence the practical performance of the formulas and thus the relations among them. In the aforementioned preliminary study, we found formulas that outperformed some existing ones but failed to achieve significant improvement over the most successful existing techniques. Thus, we extended the effort to systematically search for SBFL formulas in [123]. We defined new formula templates, which are more elaborate and can cover more existing formulas. The results of our extended formula templates show that the proposed approach led to new formulas reported in the literature and also outperformed many well-known existing ones. In particular, formula SGF-2 performed very well in all measurements, and being surprisingly simple, we think that it is very competitive against many previously advised and widely used manually crafted formulas.

This proves that the concept is valid and research on systematic SBFL formula generation is a promising direction. Compared to the GP-generated approaches or ML (Section 6.2), our approach generates readable and explainable formulas.

In the future, we would like to do the following:

- Involve other existing SBFL formulas in the evaluation.
- Try the enhancement component of the proposed formula together with other base formulas instead of only using ef .
- Investigate the effect of our formula in more detail, for instance, statistics about how many ties are broken, how many times ep helped, and so on.

- Extend the template to e.g. polynomial, exponential, and/or logarithmic to generate more elaborate formulas and maybe get even better results.
- Compare the effectiveness of the formulas generated for all the identified templates with each other.
- Combine the best formulas from different templates into a single formula that has the advantages of others.
- Expand the four spectrum metrics ef , ep , nf , and np of program spectra with other contextual information or possibly involve count-based spectra too.
- Assess how the newly generated formulas perform at granularity levels other than the method level.
- Perform a theoretical analysis of how formula equivalences can be automatically detected.
- Involve other benchmark datasets to measure how much impact they have on finding good formulas.
- Find new ways other than formula equivalences to limit the size of our search space.
- Study how handling exceptional/special cases (e.g., division-by-zero) influences the performance of SBFL formulas.
- Assess the performance trade-offs between our approach and other heuristic search and ML approaches.

In **Chapter 7**, we present “CharmFL”, an open-source fault localization tool for Python programs. The tool is developed with many interesting features that can help developers debug their programs by providing a hierarchical list of ranked program elements based on their suspiciousness scores. Also, we present “SFLaaS”, a fault localization tool for Python programs, which is provided in the form of software as a service. It is implemented with many helpful and practical characteristics to aid developers in debugging their programs. The applicability of both tools has been evaluated via lab settings. The tools have been found to be useful for locating faults in different types of programs and they are easy to use.

In the future, we would like to do the following:

- Implement interactivensness to enable the user to give his/her feedback on the suspicious elements to help re-rank them, thus improving the fault localization process of the tools.
- Add other features, such as displaying each tool’s output using different visualization techniques.
- Assess the tools with real users and in real-world scenarios would also be a valuable next step.

8.2 Potential Impacts of the Thesis

This thesis could have several potential impacts on the SBFL research community, the developers, and the industry, as follows.

Impact on the research community: For the research community, this thesis has investigated the issues and challenges of SBFL and provided a comprehensive survey study on them. Thus, other researchers will further contribute based on the performed systematic survey study and the results presented in this thesis. In this thesis, it has been found that addressing SBFL issues and challenges can lead to improving the effectiveness of SBFL. However, some problems were not addressed here due to the time limit, but others can start from where we ended. We have provided the categorization of SBFL problems and how they could be addressed too. Thus, we have established a complete understanding of how to enhance SBFL in many directions, thus motivating other researchers to empirically demonstrate that new and enhanced SBFL methods can be used to find more faults. Also, we provided the source codes and results data in our all research papers to help other researchers reuse them in their research or to be used as a benchmark, so other researchers can compare our approaches with theirs and verify their findings, therefore, boosting the work on enhancing SBFL.

Impact on the developers: There is always a desire and need for new tools to help developers and testers understand and correct the bugs in their programs. Therefore, this thesis has put substantial effort into tools that can help them automatically locate bugs in their programs. Also, our solutions are useful for guiding them to know different ways of improving the performance of software fault localization.

Impact on the industry: In addition to being interesting for research, SBFL has a wide range of industrial fault localization applications. Therefore, this thesis addressed some of the main demands of the industry. We proposed an effective tie-breaking method to reduce the number of tied program elements in the ranking list produced by SBFL. Also, we provided new SBFL tools for Python programs. These solutions can significantly reduce the time for industry practitioners, who currently do most of their fault localization tasks manually. We believe that our solutions can also open the doors for more collaborations between researchers and industry practitioners. As a result, academics and researchers can use real-world systems as their study subjects. This could also help address the issue that currently there is no method for researchers to determine whether the bugs they use in their studies are representative of the bugs that appear in the industry.

Finally, we believe that all the results of this thesis will be of great interest to software testers and researchers who would like to provide contributions to this interesting field of research, such that additional studies can be carried out to overcome the remaining issues or possible avenues can be suggested for further exploration. We hope that this thesis will be regarded as a primary source of useful and relevant information on enhancing SBFL in many directions.

References

- [1] R. Abreu, P. Zoeteweyj, and A. J. C. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–46, 2006.
- [2] R. Abreu, P. Zoeteweyj, and A. J. C. Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.
- [3] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan JC van Gemund. Exploiting count spectra for bayesian fault localization. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.
- [4] Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 409–414, 2009.
- [5] Rui Abreu, Peter Zoeteweyj, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [6] Pragya Agarwal and Arun Prakash Agrawal. Fault-localization techniques for software systems: A Literature Review. *ACM SIGSOFT Software Engineering Notes*, 39(5):1–8, 2014.
- [7] Adekunle Ajibode, Ting Shu, Kabir Said, and Zuohua Ding. A fault localization method based on metrics combination. *Mathematics*, 10(14), 2022.
- [8] Adekunle Akinjobi Ajibode, Ting Shu, and Zuohua Ding. Evolving suspiciousness metrics from hybrid data set for boosting a spectrum based fault localization. *IEEE Access*, 8:198451–198467, 2020.
- [9] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. Flakeflagger: Predicting flakiness without rerunning tests. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1572–1584, 2021.

- [10] Luciano C. Ascari, Lucilia Y. Araki, Aurora R.T. Pozo, and Silvia R. Vergilio. Exploring machine learning techniques for fault localization. In *The 10th Latin American Test Workshop*, pages 1–6, 2009.
- [11] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188, 2016.
- [12] Babak Bagheri, Mohammad Rezaalipour, and Mojtaba Vahidi-Asl. An approach to generate effective fault localization methods for programs. In *International Conference on Fundamentals of Software Engineering*, pages 244–259, 2019.
- [13] Gergo Balogh, Ferenc Horvath, and Arpad Beszedes. Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 371–374. IEEE, 2019.
- [14] Aritra Bandyopadhyay. Mitigating the effect of coincidental correctness in spectrum based fault localization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 479–482, 2012.
- [15] Aritra Bandyopadhyay and Sudipto Ghosh. Proximity based weighting of test cases to improve spectrum based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 420–423, 2011.
- [16] Aritra Bandyopadhyay and Sudipto Ghosh. Tester Feedback Driven Fault Localization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 41–50. IEEE, 2012.
- [17] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tiffany Yung, and Darko Marinov. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.
- [18] Arpad Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. Leveraging contextual information from function call chains to improve fault localization. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–479, 2020.
- [19] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

- [20] José Campos, André Ribeiro, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 378–381, 2012.
- [21] Bruno Castro, Alexandre Perez, and Rui Abreu. Pangolin: An SFL-Based Toolset for Feature Localization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1130–1133. IEEE, 2019.
- [22] A. H. Cheetham and J. E. Hazel. Binary (presence-absence) similarity. *Journal of Paleontology*, 43(5):1130–1136, 1969.
- [23] Cheng Chen and Nan Wang. UnitFL: A fault localization tool integrated with unit test. In *Proceedings of 2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pages 136–142, 2017.
- [24] Tsong Yueh Chen, Xiaoyuan Xie, Fei-Ching Kuo, and Baowen Xu. A revisit of a theoretical analysis on spectrum-based fault localization. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, pages 17–22, 2015.
- [25] Ophelia C. Chesley, Xiaoxia Ren, Barbara G. Ryder, and Frank Tip. Crisp - A fault localization tool for Java programs. In *Proceedings - International Conference on Software Engineering*, pages 775–778, 2007.
- [26] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 184–191, 2018.
- [27] J. S. Collofello and L. Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *Managing Requirements Knowledge, International Workshop on*, pages 539–550, Los Alamitos, CA, USA, 1987. IEEE Computer Society.
- [28] James S. Collofello and Scott N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.
- [29] Coverage.py tool. <https://coverage.readthedocs.io/en/coverage-5.5/>. Accessed: 01-06-2021.
- [30] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *European Conference on Object-Oriented Programming*, pages 528–550, 2005.
- [31] P. Daniel, Kwan Yong Sim, and S. Seol. Incremental spectrum cloning algorithm for optimization of spectrum-based fault localization. *Contemporary Engineering Sciences*, 7(29):1649–1655, 2014.

- [32] Patrick Daniel and K. Y. Sim. Spectrum-based fault localization tool with test case preprocessor. In *2013 IEEE Conference on Open Systems (ICOS)*, pages 162–167, 2013.
- [33] Tung Dao, Max Wang, and Na Meng. Exploring the triggering modes of spectrum-based fault localization: An industrial case. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 406–416, 2021.
- [34] Higor A. de Souza, Marcos L. Chaim, and Fabio Kon. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *arXiv*, pages 1–46, 2016.
- [35] Higor A. de Souza, Danilo Mutti, Marcos L. Chaim, and Fabio Kon. Contextualizing spectrum-based fault localization. *Information and Software Technology*, 94(C):245–261, 2018.
- [36] Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques. In *The 10th International Conference on Quality Software*, pages 13–22. IEEE, 2010.
- [37] Nicholas DiGiuseppe and James A. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering*, 20:928–967, 2015.
- [38] Joao A. Duraes and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.
- [39] Arpita Dutta, Krishna Kunal, Saksham Sahai Srivastava, Shubham Shankar, and Rajib Mall. Ftlf: A fisher’s test-based approach for fault localization. *Innovations in Systems and Software Engineering*, pages 1–25, 2021.
- [40] Fault-localization tool. <https://pypi.org/project/fault-localization/>. Accessed: 01-06-2021.
- [41] Horváth Ferenc, Gergely Tamás, Beszédes Árpád, Tengeri Dávid, Balogh Gergő, and Gyimóthy Tibor. Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal*, 27:79–123, 2019.
- [42] Debolina Ghosh and Jagannath Singh. Spectrum-based multi-fault localization using chaotic genetic algorithm. *Information and Software Technology*, 133:1–16, 2021.
- [43] Mojdeh Golagha and Alexander Pretschner. Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective. In *IEEE*

- International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 379–381, 2017.
- [44] Mojdeh Golagha, Alexander Pretschner, and Lionel C. Briand. Can We Predict the Quality of Spectrum-based Fault Localization? In *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation (ICST)*, pages 4–15, 2020.
- [45] Cheng Gong, Zheng Zheng, Wei Li, and Peng Hao. Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 470–475, 2012.
- [46] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. Interactive fault localization leveraging simple user feedback. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 67–76. IEEE, 2012.
- [47] Carlos Gouveia, Jose Campos, and Rui Abreu. Using HTML5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10. IEEE, 2013.
- [48] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.
- [49] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jia Su Sun. Interactive Fault Localization Using Test Information. *Journal of Computer Science and Technology*, 24(5):962–974, 2009.
- [50] P. Hao, Z. Zheng, Y. Gao, and Z. Zhang. Statistical fault localization in decision support system based on probability distribution criterion. In *2013 Joint IFSA World Congress and NAFIPS Annual Meeting (IFSA/NAFIPS)*, pages 878–883, 2013.
- [51] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
- [52] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90, 1998.
- [53] Hassan Bapeer Hassan and Qusay Idrees Sarhan. Performance Evaluation of Graphical User Interfaces in Java and C#. In *2020 International Conference*

- on *Computer Science and Software Engineering (CSASE)*, pages 290–295. IEEE, 2020.
- [54] Mark Hays, Jane Huffman Hayes, and Arne C. Bathke. Validation of software testing experiments: A meta-analysis of icst 2013. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 333–342, 2014.
- [55] Mark Hays, Jane Huffman Hayes, Arnold J. Stromberg, and Arne C. Bathke. Traceability challenge 2013: Statistical analysis for traceability experiments: Software verification and validation research laboratory (svvrl) of the university of kentucky. In *The 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 90–94, 2013.
- [56] Hongdou He, Jiadong Ren, Guyu Zhao, and Haitao He. Enhancing spectrum-based fault localization using fault influence propagation. *IEEE Access*, 8:18497–18513, 2020.
- [57] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49(8):1197–1224, 2019.
- [58] Thomas Hirsch and Birgit Hofer. A systematic literature review on benchmarks for evaluating debugging approaches. *Journal of Systems and Software*, 192:111423, 2022.
- [59] F Horváth, V S Lacerda, Á Beszédes, L Vidács, and T Gyimóthy. A New Interactive Fault Localization Method with Context Aware User Feedback. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 23–28, 2019.
- [60] Qusay Idrees Sarhan, Béla Vancsics, and Árpád Beszédes. Method calls frequency-based tie-breaking strategy for software fault localization. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 103–113, 2021.
- [61] Tom Janssen, Rui Abreu, and Arjan J.C. Van Gemund. Zoltar: A spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution at Runtime*, pages 23–29, 2009.
- [62] Kim Jeongho and Lee Eunseok. Empirical evaluation of existing algorithms of spectrum based fault localization. In *The International Conference on Information Networking (ICOIN)*, pages 346–351. IEEE, 2014.
- [63] Minghua Jia, Zhanqi Cui, Yiwen Wu, Ruilin Xie, and Xiulei Liu. Smfl integrating spectrum and mutation for fault localization. In *2019 6th International*

- Conference on Dependable Systems and Their Applications (DSA)*, pages 511–512, 2020.
- [64] Bo Jiang and W.K. Chan. On the integration of test adequacy, test case prioritization, and statistical fault localization. In *2010 10th International Conference on Quality Software*, pages 377–384, 2010.
- [65] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 502–514, 2019.
- [66] Lee Hua Jie. *Software Debugging Using Program Spectra*. PhD thesis, University of Melbourne, Parkville VIC 3010, Australia, 2011.
- [67] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 467–477, 2002.
- [68] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 16–26, 2007.
- [69] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.
- [70] Xiaolin Ju, Shujuan Jiang, Xiang Chen, Xingya Wang, Yanmei Zhang, and Heling Cao. Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, 90:3–17, 2014.
- [71] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis - (ISSTA)*, pages 437–440, New York, New York, USA, 2014. ACM Press.
- [72] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125, 2017.
- [73] Jeongho Kim, Jonghee Park, and Eunseok Lee. A new hybrid algorithm for software fault localization. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, pages 1–8, 2015.
- [74] Jeongho Kim, Jonghee Park, and Eunseok Lee. A new spectrum-based fault localization with the technique of test case optimization. *Journal of Information Science and Engineering*, 32(1):177–196, 2016.

- [75] B. Kitchenham and S. Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. *version 2.3., EBSE Technical Report EBSE- 2007-01, Software Engineering Group, School of Computer Science and Mathematics, Keele University, UK and Department of Computer Science, University of Durham, 2007.*
- [76] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings - International Conference on Software Engineering*, pages 301–310, 2008.
- [77] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 165–176, New York, NY, USA, 2016. Association for Computing Machinery.
- [78] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
- [79] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. Modeling and ranking flaky tests at apple. In *IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 110–119, 2020.
- [80] Tetsushi Kuma, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. Improving the accuracy of spectrum-based fault localization for automated program repair. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 376–380, 2020.
- [81] Yiğit Küçük, Tim A. D. Henderson, and Andy Podgurski. The impact of rare failures on statistical fault localization: The case of the defects4j suite. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–28, 2019.
- [82] Gulsher Laghari and Serge Demeyer. On the use of sequence mining within spectrum based fault localisation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1916–1924, 2018.
- [83] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 274–285, 2016.
- [84] Wing Lam, Krvañç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1471–1482, 2020.
- [85] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through

- test reruns in java projects. In *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–413, 2020.
- [86] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. Evaluation of measures for statistical fault localisation and an optimising scheme. In *International Conference on Fundamental Approaches to Software Engineering*, pages 115–129, 2015.
- [87] Yan Lei, Xiaoguang Mao, Ziyang Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 1–10, 2012.
- [88] Ning Li, Rui Wang, Yu li Tian, and Wei Zheng. An effective strategy to build up a balanced test suite for spectrum-based fault localization. *Mathematical Problems in Engineering*, pages 1–13, 2016.
- [89] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
- [90] Xiangyu Li, Shaowei Zhu, Marcelo D’Amorim, and Alessandro Orso. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering*, pages 82–92, New York, NY, USA, 2018. ACM.
- [91] Xu Liang, Liqiang Mao, and Ming Huang. Research on improved the tarantula spectrum fault localization algorithm. In *Proceedings of 2nd International Conference on Information Technology and Electronic Commerce*, pages 60–63, 2014.
- [92] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.
- [93] Chu-Ti Lin, Wen-Yuan Chen, and Jutarporn Intasara. A framework for improving fault localization effectiveness based on fuzzy expert system. *IEEE Access*, 9:82577–82596, 2021.
- [94] Chang Liu, Chunyan Ma, and Tao Zhang. Improving spectrum-based fault localization using quality assessment and optimization of a test suite. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 72–78, 2020.
- [95] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.
- [96] Xiaoguang Mao, Yan Lei, Ziyang Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

- [97] Abha Maru, Arpita Dutta, K. Vinod Kumar, and Durga Prasad Mohapatra. Software fault localization using BP neural network based on function and branch coverage. *Evolutionary Intelligence*, 14(1):87–104, 2021.
- [98] Lee Naish and Hua Jie Lee. Duals in spectral fault localization. In *The 22nd Australian Software Engineering Conference*, pages 51–59, 2013.
- [99] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectrum-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):1–32, 2011.
- [100] Akbar Siami Namin. Statistical fault localization based on importance sampling. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 58–63, 2015.
- [101] Neelofar. *Spectrum-based Fault Localization Using Machine Learning*. PhD thesis, University of Melbourne, Parkville, Australia, 2017.
- [102] A. Ochiai. Zoogeographical studies on the soleoid fishes found in Japan and its neighbouring regions–II. *Bulletin of the Japanese Society of Scientific Fisheries*, 22(9):526–530, 1957.
- [103] P. Daniel and Kwan Yong Sim and S. Seol. Improving spectrum-based fault-localization through spectra cloning for fail test cases beyond balanced test suite. *Contemporary Engineering Sciences*, 7(13):677–682, 2014.
- [104] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings - EEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12, 2017.
- [105] Jonghee Park, Jeongho Kim, and Eunseok Lee. Experimental Evaluation of Hybrid Algorithm in Spectrum based Fault Localization. *International conference on Software Engineering Research and Practice (SERP)*, 2014.
- [106] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 199–209, New York, NY, USA, 2011. Association for Computing Machinery.
- [107] Saeed Parsa, Somaye Arabi, and Neda Ebrahimi Koopaei. Software fault localization via mining execution graphs. In *International Conference on Computational Science and Its Applications (ICCSA)*, pages 610–623, 2011.
- [108] Alexandre Perez and Rui Abreu. Poster: A qualitative reasoning approach to spectrum-based fault localization. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 372–373, 2018.

- [109] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664, 2017.
- [110] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.
- [111] Pinpoint tool. <https://pypi.org/project/pytest-pinpoint/>. Accessed: 01-06-2021.
- [112] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. Wait, wait, no, tell me. analyzing selenium configuration effects on test flakiness. In *IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 7–13, 2019.
- [113] Pycharm ide. <https://www.jetbrains.com/pycharm/>. Accessed: 11-01-2023.
- [114] Pytest tool. <https://docs.pytest.org/en/6.2.x/>. Accessed: 01-06-2021.
- [115] Sofia Reis, Rui Abreu, and Marcelo d’Amorim. Demystifying the combination of dynamic slicing and spectrum-based fault localization. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 4760–4766, 2019.
- [116] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC ’97/FSE-5*, pages 432–449, Berlin, Heidelberg, 1997. Springer-Verlag.
- [117] Henrique L. Ribeiro, P. A. Roberto de Araujo, Marcos L. Chaim, Higor A. de Souza, and Fabio Kon. Evaluating data-flow coverage in spectrum-based fault localization. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019.
- [118] Henrique Lemos Ribeiro, Higor Amario De Souza, Roberto Paulo Andrioli De Araujo, Marcos Lordello Chaim, and Fabio Kon. Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software. In *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 404–409, 2018.
- [119] Qusay Idrees Sarhan. Enhancing spectrum based fault localization via emphasizing its formulas with importance weight. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 53–60, 2022.

- [120] Qusay Idrees Sarhan and Árpád Beszédés. Experimental evaluation of a new ranking formula for spectrum based fault localization. In *the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 276–280, 2022.
- [121] Qusay Idrees Sarhan and Árpád Beszédés. *Quality of Information and Communications Technology*, chapter Effective Spectrum Based Fault Localization Using Contextual Based Importance Weight, pages 93–107. Springer International Publishing, Cham, 2022.
- [122] Qusay Idrees Sarhan and Árpád Beszédés. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022.
- [123] Qusay Idrees Sarhan, Tamas Gergely, and Árpád Beszédés. New ranking formulas to improve spectrum based fault localization via systematic search. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 306–309, 2022.
- [124] Qusay Idrees Sarhan, Tamás Gergely, and Árpád Beszédés. Systematically generated formulas for spectrum-based fault localization. In *Accepted for publication at the 6th International Workshop on the Next Level of Test Automation (NEXTA), co-located with the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2023.
- [125] Qusay Idrees Sarhan, Hassan Bapeer Hassan, and Árpád Beszédés. Poster: Software fault localization as a service (sflaas). In *Accepted for publication at the Posters track of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2023.
- [126] Qusay Idrees Sarhan, Hassan Bapeer Hassan, and Árpád Beszédés. Sflaas: Software fault localization as a service. In *Accepted for publication at the Tool Demo track of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2023.
- [127] Qusay Idrees Sarhan, Attila Szatmari, Rajmond Toth, and Arpad Beszedes. Charmfl: A fault localization tool for python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 114–119, 2021.
- [128] Yui Sasaki, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. SBFL-Suitability: A Software Characteristic for Fault Localization. In *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 702–706, 2020.
- [129] Qiong Shi, Zhenyu Zhang, Zhifang Liu, and Xiaopeng Gao. Enhance fault localization using a 3d surface representation. In *2010 Second International Conference on Computer Research and Development*, pages 720–724, 2010.

- [130] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. Mfl: Method-level fault localization with causal inference. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 124–133, 2013.
- [131] Spectrum-based fault localization (sfl) simulator. <https://github.com/SERG-Delft/sfl-simulator/>. Accessed: 01-10-2021.
- [132] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.
- [133] Xiaobing Sun, Bixin Li, and Wanzhi Wen. Clps-mfl: Using concept lattice of program spectrum for effective multi-fault localization. In *2013 13th International Conference on Quality Software*, pages 204–207, 2013.
- [134] Attila Szatmári, Qusay Idrees Sarhan, and Árpád Beszédes. Interactive fault localization for python with charmfl. In *the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST)*, pages 33–36, 2022.
- [135] Thorvald Julius Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *København: I kommission hos E. Munksgaard.*, pages 1–34, 1948.
- [136] Shailesh Tiwari, K.K. Mishra, Anoj Kumar, and A.K. Misra. Spectrum-based fault localization in regression testing. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 191–195, 2011.
- [137] Jingxuan Tu, Lin Chen, Yuming Zhou, Jianjun Zhao, and Baowen Xu. Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs. In *2012 12th International Conference on Quality Software*, pages 1–8, 2012.
- [138] Jingxuan Tu, Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. On the analysis of spectrum based fault localization using hitting sets. *Journal of Systems and Software*, 147:106–123, 2019.
- [139] Bela Vancsics, Tamas Gergely, and Arpad Beszedes. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness. In *IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 28–35. IEEE, 2020.
- [140] Bela Vancsics, Ferenc Horvath, Attila Szatmari, and Arpad Beszedes. Call frequency-based fault localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 365–376, 2021.

- [141] Dániel Vince, Renáta Hodován, and Ákos Kiss. Reduction-assisted fault localization: Don't throw away the by-products! In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT)*, pages 196–206, 2021.
- [142] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.
- [143] Nan Wang, Zheng Zheng, Zhenyu Zhang, and Cheng Chen. FLAVS: A fault localization add-in for visual studio. In *Proceedings - 1st International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*, pages 1–6, 2015.
- [144] Yong Wang, Zhiqiu Huang, Bingwu Fang, and Yong Li. Spectrum-Based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking. *IEEE Access*, 6:8925–8933, 2018.
- [145] Yong WANG, Zhiqiu HUANG, Yong LI, RongCun WANG, and Qiao YU. Spectrum-based fault localization framework to support fault understanding. *IEICE Transactions on Information and Systems*, E102–D(4):863–866, 2019.
- [146] Yong Wang, Zhiqiu HUANG, Rongcun WANG, Qiao Yu, and Qiao Yu. Spectrum-based fault localization using fault triggering model to refine fault ranking list. *IEICE Transactions on Information and Systems*, E101–D(10):2436–2446, 2018.
- [147] Martin Weiglhofer, Gordon Fraser, and Franz Wotawa. Using spectrum-based fault localization for test case grouping. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 630–634, 2009.
- [148] Mark Weiser. *Program slicing: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, USA, 1979.
- [149] Wanzhi Wen. Software fault localization based on program slicing spectrum. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1511–1514, 2012.
- [150] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1556–1560. ACM, 2020.

- [151] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, pages 1–10. ACM Press, 2014.
- [152] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 42–51, 2008.
- [153] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.
- [154] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, 2012.
- [155] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.
- [156] W. Eric Wong, Yu Qi, Lei Zhao, and Kai Yuan Cai. Effective fault localization using code coverage. In *Proceedings - International Computer Software and Applications Conference*, pages 449–456, 2007.
- [157] Yong-Hao Wu, Zheng Li, Yong Liu, and Xiang Chen. Fatoc: Bug isolation based multi-fault localization by using optics clustering. *Journal of Computer Science and Technology*, 35:979–998, 2020.
- [158] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. Automated debugging considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 267–278, 2016.
- [159] Yan Xiaobo, Liu Bin, and Wang Shihai. How negative effects a multiple-fault program can do to spectrum-based fault localization. In *2019 Prognostics and System Health Management Conference (PHM-Qingdao)*, pages 1–6, 2019.
- [160] Yan Xiaobo, Bin Liu, and Wang Shihai. An analysis on the negative effect of multiple-faults for spectrum-based fault localization. *IEEE Access*, 7:2327–2347, 2018.
- [161] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. A universal data augmentation approach for fault localization. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 48–60. ACM, 2022.

- [162] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31:1–31:40, 2013.
- [163] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 385–392, 2010.
- [164] Sihan Xu, Jing Xu, Hongji Yang, Jufeng Yang, Chenkai Guo, Liying Yuan, Wenli Song, and Guannan Si. An improvement to fault localization technique based on branch-coverage spectra. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, pages 282–287, 2015.
- [165] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21(6):803–827, 2011.
- [166] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 191–200, 2014.
- [167] Xiaozhen Xue and Akbar Siami Namin. Measuring the odds of statements being faulty. In *International Conference on Reliable Software Technologies*, pages 109–126, 2013.
- [168] Qian Yang, J. Jenny Li, and David M. Weiss. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 52(5):589–597, 2009.
- [169] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. *Lecture Notes in Computer Science*, 7515 LNCS:244–258, 2012.
- [170] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 26(1):1–30, 2017.
- [171] Yi-Sian You, Chin-Yu Huang, Kuan-Li Peng, and Chao-Jung Hsu. Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 180–189, 2013.
- [172] Rongwei Yu, Lei Zhao, Lina Wang, and Xiaodan Yin. Statistical fault localization via semi-dynamic program slicing. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 695–700, 2011.

- [173] Xiaoran Yu, Huanrong Tang, Juan Zou, and Fan Yu. An efficient software faults localization method based on program spectrum. In *2020 IEEE International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*, pages 88–93, 2020.
- [174] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 13th international conference on Software engineering (ICSE)*, pages 201–210. ACM Press, 2008.
- [175] Abubakar Zakari, Shamsu Abdullahi, Nura Modi Shagari, Abubakar Bello Tambawal, Nuruddeen Musa Shanono, Jaafar Zubairu Maitama, Rasheed Abubakar Rasheed, Alhassan Adamu, and Salish Mamman Abdulrahman. Spectrum-based Fault Localization Techniques Application on Multiple-Fault Programs: A Review. *Global Journal of Computer Science and Technology*, 20:41–48, 2020.
- [176] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology*, 124:106312–106332, 2020.
- [177] Abubakar Zakari, Sai Peck Lee, and Chun Yong Chong. Simultaneous localization of software faults based on complex network theory. *IEEE Access*, 6:23990–24002, 2018.
- [178] Long Zhang, Lanfei Yan, Zhenyu Zhang, Jian Zhang, W.K. Chan, and Zheng Zheng. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. *Journal of Systems and Software*, 129:35–57, 2017.
- [179] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering*, 47(6):1089–1113, 2021.
- [180] Xiaohong Zhang, Ziyuan Wang, Weifeng Zhang, Hui Ding, and Lin Chen. Spectrum-based fault localization method with test case reduction. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, pages 548–549, 2015.
- [181] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Junhao Wen. Improving deep-learning-based fault localization with resampling. *J. Softw. Evol. Process.*, 33(3), 2021.
- [182] Guyu Zhao, Hongdou He, and Yifang Huang. Fault centrality: boosting spectrum-based fault localization via local influence calculation. *Applied Intelligence*, 52:1–23, 2021.

- [183] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2021.

Appendices

Appendix A: Summary

Software fault localization is a significant research topic in software engineering. Despite starting in the late 1950s, software fault localization research has gained more attention in the last few decades. This is reflected in the increase in the number of techniques, tools, and publications. The main reason for the increased attention is the dramatic increase in software systems size due to the newly added functionalities and features they provide. This also has led to an increase in the complexity of these systems. As a result, more faults have also been reported. Here, software fault localization is a good approach for reducing the number of faults and ensuring software quality. This PhD thesis aims to improve the effectiveness of Spectrum-based Fault Localization (SBFL), the most common fault localization technique, by addressing some of the most important challenges and issues posed by this technique.

This PhD thesis consists of three parts. The first part (Chapters 1-2) is the introductory part of our work, which defines the aim of this PhD thesis and the basic definitions that are needed to understand the thesis points presented in the subsequent chapters. The second part (Chapters 3-7) is the thesis points part, which presents our contributions to enhance the effectiveness of SBFL by addressing some of its main challenges and issues. The third part (Chapter 8) is the conclusions and future work part, which concludes our contributions and proposes different paths for future exploration. The scientific results I achieved and report in this thesis are grouped into several thesis points, as presented below. The relationship between the thesis points, the supporting publications, and the chapters is presented in Table A.1.

Thesis Point I: Systematic Survey of SBFL Challenges

In **Chapter 3**, we started our thesis with an important systematic survey study on the topic. As a result, several important issues and challenges of SBFL have been identified and categorized in this survey study. In each category, the most important issues have been briefly presented with possible solutions. The experimental contributions discussed in the following chapters were built upon the findings of this comprehensive survey study.

In this chapter, the following points summarize my main contributions to the topic of thesis point I. The results of this chapter were published in [122].

- Providing a theoretical background on the topic of SBFL and its main concepts.

- Conducting a systematic survey study that discussed the papers related to SBFL and the challenges and issues preventing it from being widely used. The results of the systematic survey showed that still, SBFL poses many problems that have not been addressed yet despite their importance to the effectiveness of SBFL.
- Categorizing the identified challenges and issues into 18 categories. Practically, addressing SBFL challenges can enhance the performance of SBFL in many directions, as will be seen in the subsequent chapters.

Thesis Point II: Tie-Breaking Method for SBFL

In **Chapter 4**, we proposed a method to break the ties between program elements when they are ranked by an SBFL formula. Rank ties in SBFL are very common regardless of the formula employed, and by breaking these ties, improvements to localization effectiveness can be expected. We propose the use of method call contexts for breaking critical ties in SBFL. We rely on instances of call stack traces, which are useful software artifacts during runtime and can often help developers in debugging. The frequency of the occurrence of methods in different call stack instances determines the position of the code elements within the set of other methods tied together by the same suspiciousness score.

In this chapter, the following points summarize my main contributions to the topic of thesis point II. The results of this chapter were published in [60].

- Providing the idea of using tie-breaking to improve the effectiveness of SBFL.
- Providing a thorough background on the problem of ties in SBFL.
- Gathering and discussing the related papers.
- Developing a tie-breaking method based on method call frequency to enhance the performance of SBFL.
- Evaluating the experimental results of the proposed tie-breaking method.

Thesis Point III: Emphasizing SBFL Formulas with Importance Weights

In **Chapter 5**, we enhanced SBFL by proposing the use of emphasis on the failing tests that execute the program element under consideration in SBFL. We rely on the intuition that if a code element gets executed in more failed test cases compared to the other elements, it will be more suspicious and be given a higher ranking. This is achieved by multiplying the initial suspicion score, computed by underlying SBFL formulas, of each program method by an importance weight that represents the rate of executing a method in failed test cases.

In this chapter, the following points summarize my main contributions to the topic of thesis point III. The results of this chapter were published in [119] and [121].

- Providing the idea of using importance weights to improve the effectiveness of SBFL.
- Gathering and discussing the related papers.
- Developing two methods based on importance weight. The first method improves SBFL without using any contextual information and the second method improves SBFL by using contextual information.
- Evaluating the experimental results of the proposed methods of importance weights.

Thesis Point IV: New Formulas for SBFL

In **Chapter 6**, we proposed a new SBFL ranking formula to automatically lead developers to the locations of faults in programs. It is based on the intuition that ties often happen because of shared ef and nf values, and in this case, more failing tests (larger ef) and/or fewer passing ones (smaller ep) will determine the outcome. Via an evaluation across 297 different single-fault programs of Defects4J, the proposed formula is shown to be more effective than all the selected SBFL formulas in this study. It approves the average rank and the Top-N categories as well.

Introducing new SBFL formulas is an interesting line of research. Sometimes we can get good results from not-so-obvious formulas or a simple combination of ef , ep , nf , and np . Therefore, we performed a more systematic approach to finding new formulas.

We proposed a systematic approach to search for new SBFL formulas using only the four basic statistical numbers from the spectra. For this purpose, formula templates are determined and the possible formulas are generated automatically. As a demonstration, we used a formula template to systematically generate all formulas for that template, then these were analyzed and their effectiveness was evaluated on the Defects4J dataset. Interestingly, the analysis has shown that in theory several formulas generated from the same template are equivalent to or should similarly rank elements to each other, while the handling of special cases (like division-by-zero) can significantly influence the practical performance of the formulas and thus the relations among them. In the aforementioned preliminary study, we found formulas that outperformed some existing ones but failed to achieve significant improvement over the most successful existing techniques. However, since the template we used was very simple, this is not surprising.

Thus, we extended the effort to systematically search for SBFL formulas in [123]. We defined new formula templates, which are more elaborate and can cover more existing formulas. The results of our extended formula templates show that the proposed approach led to new formulas (i.e., SGF-1 and SGF-2) that were not reported

in the literature and that outperformed many well-known existing ones. In particular, formula SGF-2 performed very well in all measurements, and being surprisingly simple, we think that it is very competitive to many previously advised and widely used manually crafted formulas.

This proves that the concept is valid and research on systematic SBFL formula generation is a promising direction. Compared to the GP-generated approaches or ML, our approach generates readable and explainable formulas.

In this chapter, the following points summarize my main contributions to the topic of thesis point IV. The results of this chapter were published in [120], [123], and [124].

- Providing the idea of introducing new formulas to improve the effectiveness of SBFL.
- Gathering and discussing the related papers.
- Developing several new formulas and comparing their effectiveness to the existing formulas.
- Evaluating the experimental results of the proposed new SBFL formulas.

Thesis Point V: Supporting Tools for SBFL

In **Chapter 7**, we present “CharmFL”, an open-source fault localization tool for Python programs. The tool is developed with many interesting features that can help developers debug their programs by providing a hierarchical list of ranked program elements based on their suspiciousness scores. Also, we present “SFLaaS”, a fault localization tool for Python programs, which is provided in the form of software as a service. It is implemented with many helpful and practical characteristics to aid developers in debugging their programs. The applicability of both tools has been evaluated via different use cases. The tools have been found to be useful for locating faults in different types of programs and they are easy to use.

In this chapter, the following points summarize my main contributions to the topic of thesis point V. The results of this chapter were published in [127], [134], [125], and [126].

- Regarding the “SFLaaS” tool, I did the following:
 - Developed the fault localization tool as a service to support SBFL for Python developers.
 - Performed the literature review of the currently available tools.
 - Prepared the use cases of the tool.
- Regarding the “CharmFL” tool, I participated in the following:

- Developed the fault localization tool to support SBFL for Python developers.
- Performed the literature review of the currently available tools.
- Prepared the use cases of the tool.

Table A.1: *Mapping of PhD thesis points, chapters, and publications*

No.	PhD Thesis Points	Chapters	Publications
I.	Systematic Survey of SBFL Challenges	Chapter 3	[122]
II.	Tie-Breaking Method for SBFL	Chapter 4	[60]
III.	Emphasizing SBFL Formulas with Importance Weights	Chapter 5	[119], [121]
IV.	New Formulas for SBFL	Chapter 6	[120], [123], [124]
V.	Supporting Tools for SBFL	Chapter 7	[127], [134], [126], [125]

Appendix B: Összegzés

A szoftverhibák lokalizálása jelentős kutatási téma a szoftverfejlesztésben. Annak ellenére, hogy az 1950-es évek végén kezdődött, a (szoftver)hibalokalizációs kutatások az utóbbi évtizedekben egyre nagyobb figyelmet kaptak. Ezt tükrözi a technikák, eszközök és publikációk számának növekedése. A fokozott figyelem fő oka a szoftverrendszerek méretének drámai növekedése az általuk biztosított újonnan hozzáadott funkciók miatt. Ez egyben a rendszerek komplexitásának növekedéséhez is vezetett. Ennek eredményeképpen több hibát is jelentettek. Itt a szoftverhibák lokalizálása jó megközelítés a hibák számának csökkentésére és a szoftver minőségének biztosítására. A jelen doktori értekezés célja a spektrumalapú hibalokalizáció (SBFL), a legelterjedtebb hibalokalizációs technika hatékonyságának javítása azáltal, hogy foglalkozik a technika által felvetett legfontosabb kihívásokkal és problémákkal.

Ez a doktori értekezés három részből áll. Az első rész (1-2. fejezetek) munkánk bevezető része, amely meghatározza a doktori értekezés célját és azokat az alapvető definíciókat, amelyek szükségesek a későbbi fejezetekben bemutatott tézispontok megértéséhez. A második rész (3-7. fejezetek) a tézispontokat tartalmazza, amely bemutatja az SBFL hatékonyságának növeléséhez való hozzájárulásomat azáltal, hogy néhány fő kihívással és problémával foglalkozik. A harmadik rész (8. fejezet) a következtetések és a jövőbeli munka része, amely lezárja a dolgozatot, és különböző utakat javasol a jövőbeli kutatáshoz. Az általam elért és ebben a szakdolgozatban közölt tudományos eredményeket több tézispontba csoportosítottam, az alábbiakban bemutatottak szerint. A tézispontok, az azokat alátámasztó publikációk és a fejezetek közötti kapcsolatot az A.1. táblázat mutatja be.

Tézis I pont: Az SBFL kihívásainak szisztematikus áttekintése

A dolgozat 3. fejezetében bemutatjuk a terület szisztematikus irodalmi áttekintését. Ennek eredményeképpen az SBFL számos fontos kérdését és kihívását azonosítottuk és kategorizáltuk ebben a felmérő tanulmányban. Minden kategóriában röviden bemutatottuk a legfontosabb problémákat a lehetséges megoldásokkal együtt. A következő fejezetekben tárgyalt kísérleti eredmények ezen tanulmánynak a megállapításaira épültek.

Ebben a fejezetben a következő pontok foglalják össze az I. tézispont témájával kapcsolatos főbb eredményeimet [122].

- Elméleti háttér nyújtása az SBFL témájához és főbb fogalmihoz.
- Szisztematikus irodalmi áttekintés készítése, amely megvitatta az SBFL-hez kapcsolódó cikkeket, valamint a széles körű alkalmazását akadályozó kihívásokat és problémákat. A szisztematikus felmérés eredményei azt mutatták, hogy az SBFL még mindig számos problémát vet fel, amelyekkel még nem foglalkoztak.
- Az azonosított kihívások és problémák 18 kategóriába sorolása. A gyakorlatban az SBFL kihívásainak kezelése számos irányban javíthatja az SBFL teljesítményét, amint azt a következő fejezetekben látni fogjuk.

Tézis II pont: Holtverseny-feloldás módszere az SBFL esetében

A 4. fejezetben javasoltunk egy megoldást a programelemek közötti holtverseny megszüntetésére, amikor azokat egy SBFL-képlet alapján rangsorolják. Az SBFL-ben az azonos gyanúsági értékek az alkalmazott formulától függetlenül nagyon gyakoriak, és ezen holtversenyek megszüntetése által a lokalizációs hatékonyság javulása várható. Javasoljuk a eljárás-hívási kontextusok használatát az SBFL-ben a holtverseny feloldására. A hívási veremek tartalmára támaszkodunk, amelyből hasznos információk nyerhetőek ki a program futására vonatkozóan, és amelyek gyakran segíthetik a fejlesztőket a hibakeresésben. A különböző hívási veremekben előforduló metódusok gyakorisága határozza meg a kódelemek egymáshoz viszonyított pozícióját az azonos gyanúsági érték esetén.

Ebben a fejezetben a következő pontok foglalják össze a II. tézispont témájával kapcsolatos főbb eredményeimet [60].

- Az SBFL hatékonyságának javítása érdekében a holtverseny feloldásának ötlete.
- Háttér nyújtása az SBFL-ben a holtversenyek problémájáról.
- A kapcsolódó cikkek összegyűjtése és feldolgozása.
- Az eljárás hívási gyakoriságán alapuló holtverseny döntési módszer kidolgozása az SBFL teljesítményének növelése érdekében.
- A javasolt holtverseny feloldási módszer kísérleti eredményeinek értékelése.

Tézis III pont: SBFL-képletek súlyozása fontossági súlyokkal

Az 5. fejezetben továbbfejlesztettük az SBFL-t azzal, hogy az SBFL-ben a vizsgált programelemet végrehajtó hibás tesztek súlyozását javasoltuk. Arra az intuícióna

támaszkodunk, hogy ha egy kódelemet a többi elemhez képest több sikertelen tesztelésben hajtanak végre, akkor az gyanúsabb lesz, és magasabb rangsorolást kap. Ezt úgy érjük el, hogy az egyes program-eljárások SBFL-formulák alapján kiszámított kezdeti gyanúsági pontszámát megszorozzuk egy fontossági súllyal, amely a módszer sikertelen tesztelésekben történő végrehajtásának arányát jelzi.

Ebben a fejezetben a következő pontok foglalják össze a III. tézispont témájával kapcsolatos főbb eredményeimet. E fejezet eredményeit az [119] és [121] cikkek tartalmazzák.

- Az SBFL hatékonyságának javítása érdekében a fontossági súllyal való súlyozás ötlete.
- A kapcsolódó cikkek összegyűjtése és feldolgozása.
- Két módszer kifejlesztése a fontossági súly alapján. Az első módszer a kontextus információk felhasználása nélkül javítja az SBFL-t, a második módszer pedig ezek felhasználásával javítja az SBFL-t.
- A javasolt fontossági súlyon alapuló módszerek kísérleti eredményeinek kiértékelése.

Tézis IV pont: Új képletek az SBFL számára

A 6. fejezetben egy új SBFL rangsorolási formulát javasoltunk. Ez azon az intuíción alapul, hogy a kötések gyakran a közös *ef* és *nf* értékek miatt következnek be, és ebben az esetben a több sikertelen teszt (nagyobb *ef*) és/vagy a kevesebb átmenő teszt (kisebb *ep*) határozza meg a végeredményt. A Defects4J 297 különböző egyetlen hibát tartalmazó programjának kiértékelésén keresztül a javasolt képlet hatékonyabbnak bizonyul, mint a tanulmányban kiválasztott összes SBFL-képlet, úgy az átlagos rangsor mint a Top-N kategóriák tekintetében.

Az új SBFL-formulák bevezetése érdekes kutatási irányvonal. Néha jó eredményeket kaphatunk nem is olyan nyilvánvaló formulákból vagy a *ef*, *ep*, *nf* és *np* egyszerű kombinációjából. Ezért szisztematikusabb megközelítést végeztünk az új formulák konstruálására.

Javasoltunk egy szisztematikus megközelítést új SBFL-képletek generálására, amely csak a spektrumokból származó négy alapvető statisztikai számot használja. Ehhez képletsablonokat határozzunk meg, és a lehetséges képleteket automatikusan generáljuk. Demonstrációként egy képletsablon segítségével szisztematikusán generáltuk az összes képletet az adott sablonhoz, majd ezeket elemeztük és hatékonyságukat a Defects4J adathalmazon értékeltük. Érdekes módon az elemzés azt mutatta, hogy elméletileg több, ugyanabból a sablonból generált formula egyenértékű, vagy hasonlóan kell rangsorolni az elemeket egymáshoz, míg a speciális esetek (például a nullával való osztás) kezelése jelentősen befolyásolhatja a formulák gyakorlati teljesítményét és

így a köztük lévő kapcsolatokat. A fent említett előzetes tanulmányban olyan formulákat találtunk, amelyek felülmúltak néhány létező formulát, de nem sikerült jelentős javulást elérni a legsikeresebb létező technikákhoz képest. Ezért kiterjesztettük a módszerünket az SBFL formulák szisztematikus keresésére [123]. Új képletsablonokat definiáltunk, amelyek kidolgozottabbak és több létező képletet képesek lefedni. A kibővített képletsablonjaink eredményei azt mutatják, hogy a javasolt megközelítés olyan új képleteket eredményezett, amelyekről a szakirodalomban nem számoltak be, és számos jól ismert létező képletet felülmúlt. Különösen az SGF-2 formula teljesített nagyon jól minden mérésben, és mivel meglepően egyszerű, úgy gondoljuk, hogy nagyon versenyképes számos korábban javasolt és széles körben használt, kézzel készített formulával szemben.

Ez azt bizonyítja, hogy a koncepció működőképes, és a szisztematikus SBFL-képletgenerálással kapcsolatos kutatás ígéretes irány. A GP-generált megközelítésekhez vagy az ML-hez képest a mi megközelítésünk olvasható és megmagyarázható formulákat generál.

Ebben a fejezetben a következő pontok foglalják össze a IV. tézispont témájával kapcsolatos főbb eredményeimet [120], [123], és [124].

- Az SBFL hatékonyságának javítása érdekében új formulák bevezetésének ötlete.
- A kapcsolódó cikkek összegyűjtése és feldolgozása.
- Több új formula kifejlesztése és hatékonyságuk összehasonlítása a meglévő formulákkal.
- A javasolt új SBFL-képletek kísérleti eredményeinek értékelése.

Tézis V pont: Támogató eszközök az SBFL számára

A 7. fejezetben bemutatjuk a “CharmFL” nyílt forráskódú hibalokalizációs eszközt Python programokhoz. Az eszköz számos hasznos funkcióval rendelkezik, amelyek segíthetik a fejlesztőket programjaik hibakeresésében, mivel a gyanúsági pontszámok alapján rangsorolt programelemek hierarchikus listáját nyújtja. Emellett bemutatjuk az “SFLaaS”-t, egy hibalokalizációs eszközt Python programokhoz, amelyet szoftver mint szolgáltatás formájában nyújtunk. Számos hasznos és praktikus tulajdonsággal van implementálva, hogy segítse a fejlesztőket programjaik hibakeresésében. Mindkét eszköz alkalmazhatóságát különböző felhasználási eseteken keresztül értékeltük. Az eszközök hasznosnak bizonyultak a különböző típusú programok hibáinak lokalizálására, és könnyen használhatóak.

Ebben a fejezetben a következő pontok foglalják össze a V. tézispont témájával kapcsolatos főbb eredményeimet [127], [134], [125], és [126].

- Az “SFLaaS” eszközzel kapcsolatban a következők az eredményeim:
 - A hibalokalizációs eszköz fejlesztése szolgáltatásként az SBFL támogatására Python fejlesztők számára.

- A jelenleg elérhető eszközök szakirodalmi áttekintésének elvégzése.
- Az eszköz használati eseteinek előkészítése.
- A “CharmFL”-lel kapcsolatban a következőkben vettem részt:
 - Hibalokalizációs eszköz fejlesztése az SBFL támogatására Python fejlesztők számára.
 - A jelenleg elérhető eszközök szakirodalmi áttekintésének elvégzése.
 - Az eszköz használati eseteinek előkészítése.

Table A.1: *PhD tézispontok, fejezetek és publikációk kapcsolata*

Szám	PhD tézispontok	Fejezetek	Publikációk
I.	Az SBFL kihívásainak szisztematikus felmérése	Fejezet 3	[122]
II.	Holtpont feloldás módszere SBFL-hez	Fejezet 4	[60]
III.	Az SBFL képletek súlyozása fontossági súlyokkal	Fejezet 5	[119], [121]
IV.	Új képletek az SBFL-hez	Fejezet 6	[120], [123], [124]
V.	Támogató eszközök az SBFL-hez	Fejezet 7	[127], [134], [126], [125]