IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 25, 2021, accepted January 9, 2022, date of publication January 18, 2022, date of current version January 28, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3144079

A Survey of Challenges in Spectrum-Based Software Fault Localization

QUSAY IDREES SARHAN[1,2] **AND ÁRPÁD BESZÉDES**[1]
[1]Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary
[2]Department of Computer Science, University of Duhok, Duhok, Kurdistan Region 42001, Iraq

Corresponding author: Qusay Idrees Sarhan (sarhan@inf.u-szeged.hu)

This work was supported in part by the University of Szeged Open Access Fund under Grant 5459; in part by the Ministry of Innovation and Technology, NRDI Office, Hungary within the framework of the Artificial Intelligence National Laboratory Program (MILAB); and in part by the Hungarian National Research, Development and Innovation Office, through the "Security Enhancing Technologies for the IoT," under Grant 2018-1.2.1-NKP-2018-00004. The work of Qusay Idrees Sarhan was supported by the Stipendium Hungaricum Scholarship Program.

ABSTRACT In software debugging, fault localization is the most difficult, expensive, tedious, and time-consuming task, particularly for large-scale software systems. This is due to the fact that it requires significant human participation and it is difficult to automate its sub-tasks. Therefore, there is a high demand for automatic fault localization techniques that can help software engineers effectively find the locations of faults with minimal human intervention. This has led to the proposal of implementing 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. In SBFL, the probability of each program element (e.g., statement, block, or function) being faulty is calculated based on the results of executing test cases and their corresponding code coverage information. However, SBFL techniques are not yet widely adopted in the industry. The rationale behind this is that they pose a number of 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. There are massive studies on SBFL that cover its usage, formulas, performance, etc. So far, no dedicated survey points out comprehensively the issues of SBFL. In this paper, various SBFL challenges and issues have been identified, categorized, and discussed alongside many directions. Also, the paper raises awareness of the works being achieved to address the identified issues and suggests some potential solutions too.

INDEX TERMS Program spectra, spectrum based fault localization, software testing, challenges and issues, survey.

I. INTRODUCTION

Software cover many aspects of our everyday life as they are used in different application domains such as healthcare, military, automobile, and transportation. Thus, our modern life cannot be imagined without software. The extensive use of different software products in our day-to-day activities has led to a significant increase in their size and complexity [1]. As a result, the number and types of software faults have also increased. Software faults not only lead to financial losses; but also loss of lives. 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 [2]. 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 programming time on finding faults alone [3]. Therefore, there is a serious need for developing automatic fault localization techniques that can help developers effectively find the locations of faults in software systems with minimal human intervention. Different types of such techniques have been proposed and implemented by researchers and developers. However, Spectrum Based Fault Localization (SBFL) is considered amongst the most prominent techniques in this respect due to its lightweight, language-agnostic [4], easy to use [5], and relatively low overhead in test execution time [6] characteristics.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

In SBFL, the probability of each program element (e.g., statement, block, or function) being faulty is calculated based on the results of executing test cases and their corresponding code coverage information [7]. Currently, SBFL techniques are not yet widely adopted in the industry as they pose a number of issues and their performance is affected by several influential factors [8], [9]. 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 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 the paper can be summarized as follows:

1) Conducting s systematic literature survey on the challenges of SBFL.
2) Identifying and presenting 18 SBFL challenges and issues.
3) The paper also raises awareness of the works being achieved to address the identified challenges and issues, and suggests some potential solutions in order 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 in order to answer its 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.

The remainder of this paper is organized as follows: Section II briefly introduces the background of SBFL and its main concepts. Section III presents the most relevant works. Section IV describes the research methodology employed to perform the study. Section V presents the study's findings. Section VI outlines the threats to validity and the steps considered to overcome them. Finally, in Section VII, the conclusions of the study are given.

II. BACKGROUND OF SBFL

SBFL is a dynamic program analysis technique that is performed through program execution [10], [11]. The goal of SBFL is to address the problem of finding the root causes of bugs by utilizing information from program elements executed by test cases, in particular the outcomes of tests and their code coverage [12]. Thus, to identify and locate elements more likely to be faulty. In SBFL, code coverage information (also called program spectra), which is obtained from executing a set of test cases with recording their results, is used, by a ranking formula, to calculate the probability of each program element (e.g., statement, block, or function) being faulty [13]. Code coverage provides information on which program element has been executed and which one has not during the execution of each test case, while test results are classified as passed or failed test cases. Passed test cases are executions of a program whose outputs are expected, whereas failed test cases are executions of a program whose outputs are unexpected [14].

The idea of program spectra was mentioned for the first time in 1987 [15]. However, the use of program spectra for fault localization was first proposed in a study on the year 2000 problem (also known as the Y2K problem) aimed at discovering errors in calendar data formatting and storage for dates in and after the year 2000 [16]. It is worth mentioning that Tarantula is one of the first approaches, proposed in 2002 [17], that uses a ranking formula to calculate elements' suspiciousness in SBFL [12]. Afterwards, many other formulas have been introduced and the roots for many of them are from biology such as Ochiai [18] and Binary [19].

To illustrate the work of SBFL, assume that there is a Python *mid*() function that takes three numbers as input and returns the median value. The *mid*() function, which is a well-known code segment used for describing fault localization [20], comprises twelve statements S_i $(1 \leq i \leq 12)$ and six test cases T_j $(1 \leq j \leq 6)$ as shown in Figure 1 that have been executed and the spectra (the execution information of statements in passed and failed test cases) have been recorded as presented in Table 1. There is a fault in statement 7 (the correction is m = x), and only two test cases, T1 and T6, hit that faulty statement. An entry of 1 in the cell corresponding to statement S_i and test case T_j means that the statement S_i has been executed by the test case T_j, and it is 0 otherwise. Also, an entry of 1 in the row labeled R, which represents test results, means that the corresponding test case failed, and it is 0 otherwise. Intuitively, a statement that is executed by more failed test cases is more likely to be considered as a faulty statement.

```
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
        else:
8:          if x>y:
9:              m = y
10:         elif x>z:
11:             m = x
12:     return m
```

```
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
```

FIGURE 1. Running example – code and test cases.

The spectra information is then used by a spectra formula (also called a ranking metric [21], a suspiciousness metric [22], a risk evaluation metric [23], or a fault locator [24])

TABLE 1. Running example – spectra and four counters.

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
R	0	0	0	0	0	1				

to compute how suspicious each element is of being faulty. Table 2 presents a few numbers of spectra formulas proposed in the literature.

TABLE 2. Several spectra formulas with their formulas.

Name	Formula
Tarantula	$\dfrac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$
Ochiai	$\dfrac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$
Overlab	$\dfrac{ef}{min(ef,nf,ep)}$
Wong2	$ef-ep$
Goodman	$\dfrac{2\cdot ef-nf-ep}{2\cdot ef+nf+ep}$

Often, a formula is expressed in terms of four counters [25] that are calculated from the spectra as follows:

- **ef:** the number of times a statement is executed (e) in failed tests.
- **ep:** the number of times a statement is executed (e) in passed tests.
- **nf:** the number of times a statement is not executed (n) in failed tests.
- **np:** the number of times a statement is not executed (n) in passed tests cases.

Finally, the statements are ranked based on their computed suspiciousness scores and then examined by developers in descending order, ranging from the most suspicious to the least suspicious. Statements with the highest scores are considered the most likely to be buggy. Table 3 presents the scores and ranks of applying different spectra formulas on the spectra information of our running example. In the case of the Tarantula formula, for example, statements 6 and 7 are ranked as the most suspicious elements by the formula as they have the highest scores compared to others. The third most suspicious element is 4 and so forth. It is worth mentioning that the statement 11 has not been included in the scoring as it has not been executed by any test case. Figure 2 shows the steps of the SBFL process and how the developer examines the suggested list of suspicious program elements.

In the previous example, we used statements as the basic code elements for fault localization. However, it is important to note that different kinds of granularities are frequently used as well such as functions and code blocks. Technically, the granularity is determined by the granularity of code coverage measurement.

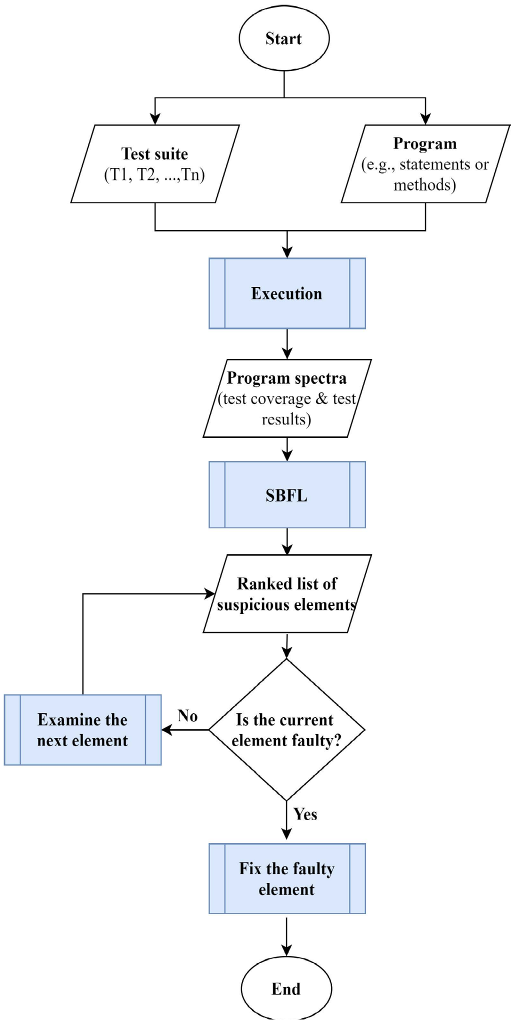

Start
Test suite
(T1, T2, ...,Tn)
Program
(e.g., statements or methods)
Execution
Program spectra
(test coverage & test results)
SBFL
Ranked list of suspicious elements
Examine the next element
No
Is the current element faulty?
Yes
Fix the faulty element
End

FIGURE 2. SBFL process.

TABLE 3. Running example – scores and ranks.

	Tarantula score	Rank	Ochiai score	Rank	Overlab score	Rank	Wong2 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

III. RELATED WORKS

The SBFL has been an important and active research field for decades. However, a survey study on the issues and challenges in this research field is 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 [26] provided an overview of coverage-based testing and compared between 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 [27], 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 own 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 the accuracy, including fault injection, instrumentation, multiple faults, and unrealistic assumptions.

In [28], 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 issues and challenges have been discussed.

In [29], 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, data mining, model-based debugging, and additional techniques. They also listed popular subject programs used to study the effectiveness of different fault localization techniques. Their survey also addresses 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 levels selection.

In [12], the authors conducted a survey on 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 entities, faults introduced by missing code, and coincidental correctness) on SBFL research that have to be tackled to improve the SBFL to be used in real development settings.

In [30], the authors briefly discussed two issues, granularity levels and entities having the same suspiciousness, based on what the authors encountered in their collaboration with the industry. They highlighted that there are many different granularity levels that 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 entities with the same suspiciousness which needs more attention in order to propose new strategies for tie-breaking.

In [31], [32], 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 in the SBFL. Some of them briefly highlighted a very limited number of issues. Also, most of them were not conducted systematically. In contrast, our paper provides a thorough and systematic survey based on a detailed research methodology to examine different issues in the SBFL alongside possible solutions or research gaps for further investigations. As a result, our paper extends the details of the aforementioned studies by identifying, categorizing, and discussing 18 important issues comprehensively.

IV. RESEARCH METHODOLOGY

The systematic process followed in this study is based on the guidelines provided by [33] and [34]. It consists of several stages as presented in the following subsections.

A. IDENTIFICATION OF RESEARCH OBJECTIVE

The objective of this paper is to answer the following research question:

"What are the challenges and issues posed by spectrum-based fault localization (SBFL)?"

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

B. SEARCH STRATEGY

1) LITERATURE SOURCES

Five well-known online literature sources indexing publications of software engineering and computer science were used. Table 4 lists these sources as well as links to their websites.

TABLE 4. 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

2) 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 [35]. Where the "OR" operator was used to link synonyms or related terms and the "AND" operator was used to link the major terms. It is worth mentioning that no publication time span was set during the search.

C. PAPER SELECTION

1) 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 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 challenges.
- Papers published online from 2002-2021.

Exclusion criteria:

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

2) SNOWBALLING

In this paper, the snowballing technique [36] 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 shows the paper selection process and its outcome at each stage. In addition, Table 10 lists all papers (with their references, titles, and publication years) obtained after applying the paper selection process.

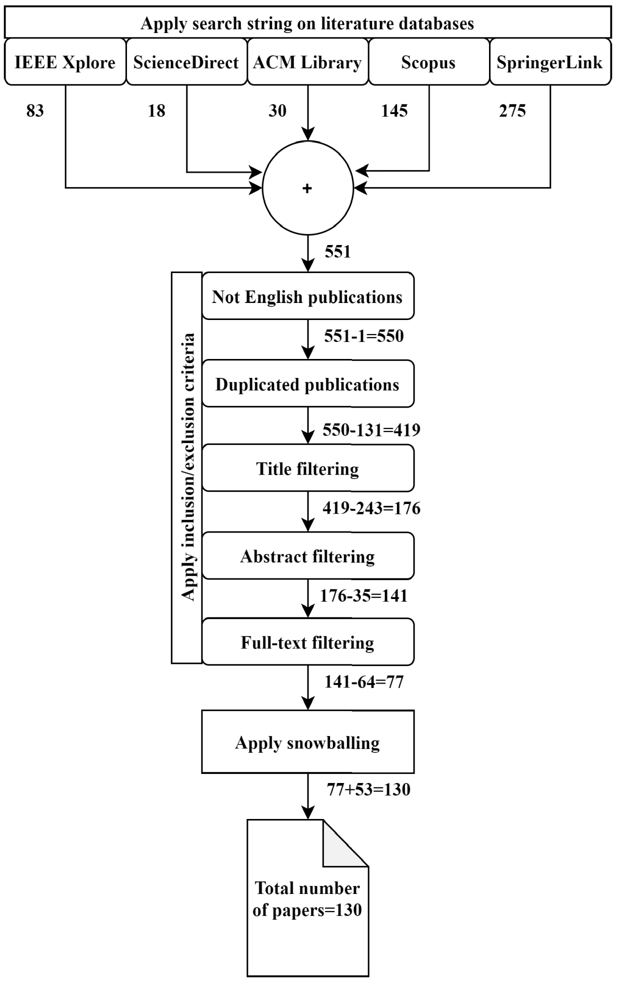


FIGURE 3. The outcome of the paper selection process.

V. RESULTS

To answer the identified research question of this study, all the related publications were extensively read and analyzed. Thus, several challenges and issues posed by the SBFL have been identified alongside many directions, as shown in Figure 4, classified into several categories, and then discussed as follows.

A. STATISTICAL ANALYSIS

In SBFL, statistical analysis is used to correlate program elements with failures [37], 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 technique is significantly better than the state of the art by applying their technique and the state of the art technique on one or more faulty programs. Then, 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 [38].

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 [38], [39] presented the first such tools 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 in order 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

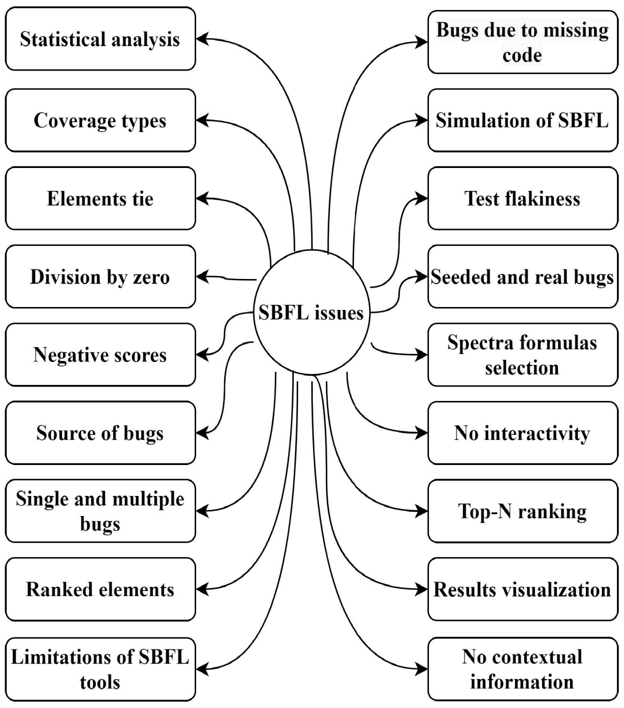


FIGURE 4. Challenges and issues of SBFL.

were discovered to have potentially misstated the significance of their findings because of the selection of inappropriate statistical techniques.

B. COVERAGE TYPES

Since the granularity of fault localization is determined by the granularity of code coverage, the selection of which coverage type to use in the SBFL is crucial as each coverage type influences the performance of SBFL techniques in one way or another [29]. Program coverage elements can be divided into several common types as follows:

- Statement coverage: There are different lines of code that can be considered for statement coverage. Thus, the issue is which line of code can be considered as the most suitable choice. In [40] for example, all the lines of code in the target program are considered for statement coverage. While in [41], 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 of any spectra formula based on the location of the buggy line. To illustrate this, let us consider that we have two versions of the same target program: (a) version A with 1000 lines of code which include all different types of lines of code. (b) version B with 100 lines of code which does not include variable declarations, function declarations, etc. Then, we suppose that the buggy line of code was located at the 500th position in the ranking list of version A. The same buggy line of code was at the 4th position in the ranking list of version B. Using the *Exam* measure in Equation 1 [42], which measures the percentage of statements that the programmer needs to examine before the actual bug is found, program B gives 4% as compared to program A, which gives 5%. This indicates that the fewer ranked statements a program has, the fewer statements the programmer has to examine to find the buggy statement.

$$\text{Exam} = \left(\frac{\text{E}}{\text{N}}\right) \cdot 100\% \qquad (1)$$

where E is the position of the faulty statement in the ranking list and N is the total number of statements in the ranking list.
Therefore, comprehensive experimental studies have to be conducted to distinguish between different types of lines of code and 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 a target program. However, statement coverage is one of the most used coverage types as it often provides the exact locations of faults [43]
- 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 [27], [44], [45].
- Block coverage: Here, a number of program statements are considered for block coverage [46]. 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 [1]. 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.
- 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 [47], [48]: (a) it provides more global contextual information about the investigated program entity, (b) it is scalable to large programs and executions, (c) some studies report that it is a better granularity for the users [49], (d) it reduces the number of program tied elements too, (e) it is also one of the most commonly adopted approaches as the basic program elements [43]. 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.
- 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 case execution [50].

C. 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 spectra formulas causes an issue called elements tie [51].

Elements tie means having a similar suspiciousness score for more than one program element in the target program [30]. Tied elements are usually ranked based on three approaches [52] as follows:

- MIN measure (also known as the worst case): it refers to the bottom-most position of the elements that share the same suspiciousness score.
- MAX measure (also known as the best case): it refers to the topmost position of the elements that share the same suspiciousness score.
- MID measure (also known as the average case): it refers to the average of the position of the elements that share the same suspiciousness score and it is calculated using Equation 2.

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

where S is the tie starting position and E is the tie size. It is worth noting that the MID measure considers the average of all possible positions. Therefore, it is more widely used than the other two approaches. Also, the MID measure must be applied on ascending sorted suspiciousness scores.

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 [53], [54]. 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.

Ties among program elements can be divided into two types 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 [53]. In this type, the internal order of examination affects the SBFL 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.

Many approaches can be used to handle the ties problem in the ranking list of program elements. In [55], the authors proposed a tie-breaking strategy that firstly sorts program statements according to their suspiciousness scores and then breaks ties by sorting them according to applying a confidence formula. Such formula is designed to measure the degree of confidence in a given suspiciousness score. When two or more statements have the same suspiciousness score, the score assigned to the statements with higher confidence is more reliable, and thus the statements are more likely to be faulty.

In [51], the authors proposed a grouping-based strategy that employs another influential factor alongside statements' suspiciousness scores. 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 scores to generate the final ranking list. Thus, the statements are examined firstly based on their group order and secondly based on their suspiciousness scores. Their results show that ranking based on several factors can improve the SBFL effectiveness. Thus, the grouping-based strategy could be effective in tie-breaking as well.

In [53], the authors proposed many tie-breaking approaches and also suggested using dynamic program slicing as a promising solution to break ties. To illustrate this, consider the *test*() function in Figure 5. The function has nine statements (1-9) and a fault in statement 3 (it should be b = y). Table 5 presents the function spectra after executing four test cases (T1-T4) and the suspiciousness scores for all the statements after applying Tarantula. It can be noted that eight statements are critically tied (i.e., having the same suspiciousness score of 0.5). Here, dynamic slicing can be used for tie-breaking based on the following steps:

- Constructing the dynamic slice of each failed test case.
- Taking the intersection set of statements from the constructed slices. From Table 5, this set will be the statements 3, 6, and 9.

```
1:  def test(x, y, z):
2:      x = x + y
3:      b = z
4:      if x > 1:
5:          b = x - 4
6:      if b > 0:
7:          o = y + z
        else:
8           o = y + z + 1
9       return o
```

FIGURE 5. A test() function with a faulty statement.

TABLE 5. A test() function – spectra, four counters, and scores.

Statement	T1	T2	T3	T4	ef	ep	nf	np	Scores
1	1	1	1	1	2	2	0	0	0.5
2	1	1	1	1	2	2	0	0	0.5
3	1	1	1	1	2	2	0	0	0.5
4	1	1	1	1	2	2	0	0	0.5
5	0	1	1	0	0	2	2	0	0
6	1	1	1	1	2	2	0	0	0.5
7	1	0	1	0	1	1	1	1	0.5
8	0	1	0	1	1	1	1	1	0.5
9	1	0	0	1	2	2	0	0	0.5
R	1	0	0	1					

- Giving higher priority to the statements in the obtained intersection set.
- Examining the statements with higher priority (i.e., 3, 6, and 9) before the other statements (i.e., 1, 2, 4, 7, and 8). Since the set with higher priority does include the faulty statement, it can be found that the dynamic slicing has reduced the size of the critical tie from eight to three statements.

D. DIVISION BY ZERO

There is always a possibility of the denominators of some spectra formulas having zero. As a result, error messages are produced. For example, when the formula "Overlab" is applied to the information presented in Table 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 3. 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 entity in which its denominator is zero [41], [56], [57].
- Adding a small fixed constant such as 10^{-6} to the denominator [7], [58].
- Adding a larger value such as the number of tests plus 1 to the denominator. Such value is larger than any value which can be returned with a non-zero denominator [7], [57].

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 totally 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.

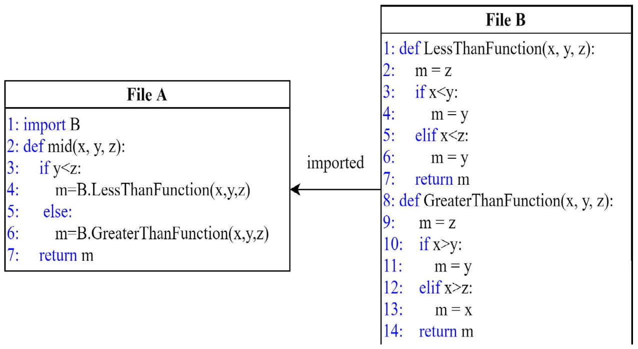
E. NEGATIVE SUSPICIOUSNESS SCORES

In SBFL, most of the formulas used to compute suspiciousness scores of program elements produce positive scores. However, few formulas (e.g., Wong2 and Goodman) produce both positive and negative scores. This may cause a critical 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 in 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 Wong2 formula in Table 3. It can be noted the statements 4, 5, and 10 are assigned with the same score (i.e., -1) and the same rank order (i.e., 3); but we would like to consider the 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 weighting method that multiplies the score of 4 by the weighting value 0.9 (more suspicious) and the scores of 5 and 10 by the weighting value 0.1 (less suspicious). The results of applying our weighting method will decrease the score of 4 and thus put it in the worst position in the ranking list (i.e., 5 rank instead of 3); while it does the opposite with both 5 and 10. A possible solution to this issue is to apply the weighting method to each score generated by such formulas; the absolute of each score has to be taken before ranking the scores.

F. 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 6 for example, File B includes two functions, *LessThanFunction() and GreaterThanFunction()*, with a bug in the statement 6, it should be m = x instead of m = y, of the first function. These two functions



File A
```
1: import B
2: def mid(x, y, z):
3:     if y<z:
4:         m=B.LessThanFunction(x,y,z)
5:     else:
6:         m=B.GreaterThanFunction(x,y,z)
7:     return m
```

File B
```
1: def LessThanFunction(x, y, z):
2:     m = z
3:     if x<y:
4:         m = y
5:     elif x<z:
6:         m = y
7:     return m
8: def GreaterThanFunction(x, y, z):
9:     m = z
10:    if x>y:
11:        m = y
12:    elif x>z:
13:        m = x
14:    return m
```

FIGURE 6. **Fault propagation.**

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 s/he will not be able to know which statement in the called function caused the bug in order to be fixed.

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

G. SINGLE AND MULTIPLE BUGS

A bug is a program element that shows unexpected behavior when executed by a test case. In general, program failures are caused either by a single bug or multiple bugs [31], [32]. 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 statement is mostly not to be ranked high; making it difficult to be identified [44].

To address this issue, further studies are required to know whether a program really 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 it has been used at. It is worth mentioning that many SBFL techniques are designed for programs with 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 [59], where an empirical investigation on multiple-fault versions from different open-source programs has been conducted in order 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 [60], [61]. Other ways to address the issue of multiple bugs in a program is to design novel suspiciousness formulas as in [62], 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 [63]–[65].

H. RANKED ELEMENTS

1) THE RANKED LIST OF ELEMENTS IS HUGE

Mostly, a large number of program elements are included in the ranking list generated by SBFL techniques [44], [66]. 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 the speciousness score of 0 due to the issue of division by zero.
- A huge number of elements that are unrelated as 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 as mentioned in a previous section 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 the statement 9 and an optimized version of the code with the same bug in the statement 8, as shown in Figures 7 and 8. 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.

Table 6 presents the spectra and test case information of all the statements alongside their speciousness scores before optimizing the code, and Table 7 presents the same information but after applying code optimization. It can be noted that code optimization reduces the number of ranked statements. Besides, we can see that it eliminates some ties completely and reduces some others. And, no statement has been scored with the value 0.

However, it would be interesting to study code optimization and its impact on performance alongside many

```
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 7. 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 8. Running example – optimized code.

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 be reported with evidence. A possible solution to the issue of the suspicious elements is not related logically as code 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 the SBFL performance. Here, we list out some factors that we believe will have a positive impact on the ranking effectiveness as follows:

TABLE 6. 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 7. 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.

2) 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

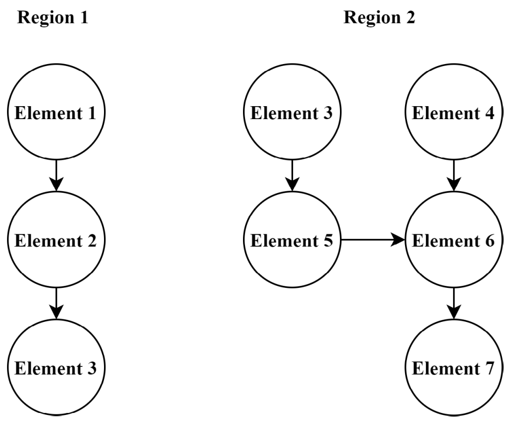
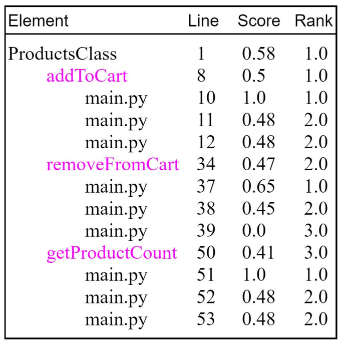


FIGURE 9. Suspicious program regions.

another one from function b(), and so forth. As a result, the ranking list suggested by SBFL is not followed linearly by developers [66] because they have trouble understanding the context of the bug, since they are only given each bug location in isolation [67]. 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.

In [67], the authors proposed a technique that reports the most suspicious program regions instead of a single program element which 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 9 shows what the hierarchical ranked list of program elements looks like.

Recently, the authors in [68] proposed a hierarchical ranked list of elements to solve this issue as well. 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 at line 37 will not be examined before the statement at line 11 because the latter belongs to a function of higher rank in the ranking list. This hierarchical grouping of program elements gives additional useful information about the suspiciousness scores on all layers to the user. They can exclude whole methods or even classes from the list. Figure 10 shows how the Hierarchical ranked list of program elements looks like.



Element	Line	Score	Rank
ProductsClass	1	0.58	1.0
addToCart	8	0.5	1.0
main.py	10	1.0	1.0
main.py	11	0.48	2.0
main.py	12	0.48	2.0
removeFromCart	34	0.47	2.0
main.py	37	0.65	1.0
main.py	38	0.45	2.0
main.py	39	0.0	3.0
getProductCount	50	0.41	3.0
main.py	51	1.0	1.0
main.py	52	0.48	2.0
main.py	53	0.48	2.0

FIGURE 10. Hierarchical ranked list of elements.

I. LIMITATIONS OF SBFL TOOLS

SBFL techniques require suitable tools to automatically collect spectra data and testing information from the target programs [29]. However, the currently available tools [17], [69]–[76] suffer from some limitations as follows [77]:

- Mostly, they only collect abstract and trivial testing information, such as whether a program element is executed by a specific test case or not.
- 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 cannot be used 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++ [26]. 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 [78].
- Proposing and developing tools or plug-ins for specific IDEs is considered as a practical limitation of usage as not all the developers use the same 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 IDEs that are calling this generic tool.

In order to make SBFL tools more useful and practical, they should be developed with some important features as follows [26]:

- A user-friendly graphical interface is a crucial feature for the users nowadays as such interfaces act as the gates into using software systems interactively and efficiently [79]. Thus, a proposed fault localization tool should also be run in a 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 into various file formats according to the user's needs (e.g., XML, 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.

J. BUGS DUE TO MISSING CODE

Generally, software bugs appear due to wrong 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) [12]. In some open source projects, it has been found that missing code faults form the majority of the total faults in these projects [80].

Locating a bug that is introduced by a missing code is a challenging task in SBFL. This is due to the fact that 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 [81]. 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 [52], [82]. However, more work is needed to propose techniques to address the issue of bugs caused by missing code.

K. 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 tests 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 [83]. They should be able to simulate various program structures and their behaviours, 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.

L. 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 [84]. This issue negatively impacts the effectiveness of SBFL techniques as it provides misleading signals during the fault localization process [85]. It has been found that the flakiness of individual test cases influences fault localization scores and ranks, and that some SBFL spectra 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 [86]. 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 [87], each test case has been re-run 10 times. In [88], 30 times. In [84], 100 times. In [89], 4000 times. In [90], 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 [91] 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) it may also inject a delay between each re-run to allow the cause of failure (e.g., a network outage) to occur. (d) 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 [91].

M. SEEDED AND REAL BUGS

Artificial faults (also called seeded faults) are made when a researcher intentionally seeds 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 can not 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 [92] for Java programs, BugsJS [93] for JavaScript programs, and BugsInPy [94] for Python programs, where the passed test cases are much more common than the failed test cases. It is worth mentioning that they are widely used in multiple fault localization studies with about 70.91% of the selected studies utilizing them. 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 [31].

N. 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 [95], 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 [96], several formulas generated by genetic algorithms have been evaluated, and it has been found that the GP13 formula is one of the best performing formulas of its kind. In [84], it has been found that some SBFL spectra 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. It is worth mentioning that multiple spectra 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 [20].

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 further exploration.

O. 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 testing is not utilized to improve the fault localization performance [97]. This issue can be addressed by involving user interactivity. Involving the user and considering his/her feedback on the suspicious elements and their ranks can help to re-rank them, thus improving the fault localization process.

Figure 11, which is adopted from [98], shows the difference between the static SBFL (i.e., without user interactivity) and the interactive SBFL (i.e., with user interactivity).

In [99], 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 his/her 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 really 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 [71], [98], the authors also proposed an interactive fault localization approach that leverages simple user feedback.

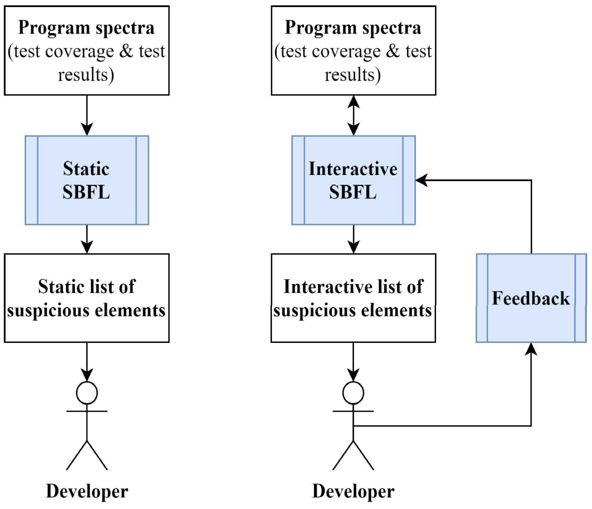

Program spectra
(test coverage & test results)
Static SBFL
Static list of suspicious elements
Developer
Program spectra
(test coverage & test results)
Interactive SBFL
Interactive list of suspicious elements
Feedback
Developer

FIGURE 11. Static vs. interactive SBFL.

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 [100], 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 [101], the authors proposed an interactive approach to use the 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 also investigate cases when developers or users make the wrong estimation and give incorrect feedback due to mistakes or them 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 the scenarios where multiple bugs exist is also recommended.

P. TOP-N RANKING

Several studies including [102], [103] report that developers think that inspecting the first 5 program elements in the ranking list produced by an SBFL technique is acceptable, and that the first 10 elements are the upper bound for inspection before ignoring the ranking list. Therefore, the performance of SBFL can also be evaluated by focusing on these rank positions, collectively called Top-N, as follows:

- Top-1: When the rank of a faulty element is the first in the ranking list.
- Top-3: When the rank of a faulty element is less or equal to three in the ranking list.
- Top-5: When the rank of a faulty element is less or equal to five in the ranking list.
- Top-10: When the rank of a faulty element is less or equal to ten in the ranking list.
- Other: When the rank of a faulty element is more than ten in the ranking list.

Also, there is a special non-accumulating variant of Top-N categories, in which the cases where the bug fell into non-overlapping intervals of $[1]$, $(1, 3]$, $(3, 5]$, $(5, 10]$ or $(10, ...]$ are counted. These categories show in how many cases an SBFL approach moves a bug into a better (for example, from $(5, 10]$ to $(1, 3]$) or a worse (for example, from $[1]$ to $(1, 3]$) category. In other words, in how many cases do the bugs get into a higher-rank category (this kind of improvement is also known as *enabling improvement* [43]) and in how many cases do they downgrade the category. Thus, an SBFL approach that achieves improvements in all categories by moving many bugs to higher ranked categories has better performance.

However, due to the nature of SBFL, the faulty element cannot always be ranked at higher-ranked Top-N categories. This issue is the biggest obstacle to the usefulness of SBFL in practice [22]. 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 8 with a brief description of each proposed solution.

Q. 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 [17]:

- 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 8. 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.	[22], [104]
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 to assign the risks of unsuspicious statements as the lowest value.	[105]
Using program slicing	Deleting program elements that have no dependence on faulty elements to improve the precision of locating faults.	[106]–[112]
Introducing new ranking formulas	Proposing new risk evaluation formulas that outperform the existing ones.	[23], [113]–[123]
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.	[20], [124], [125]
Optimizing test cases	Optimization methods can maximize SBFL performance using a minimum (e.g., by removing redundant test cases) or balanced number of test cases used by SBFL formulas.	[126]–[134]
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).	[135]–[139]
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.	[140]
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.	[141]–[144]

TABLE 9. 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

table of many fields as in Table 9. 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.

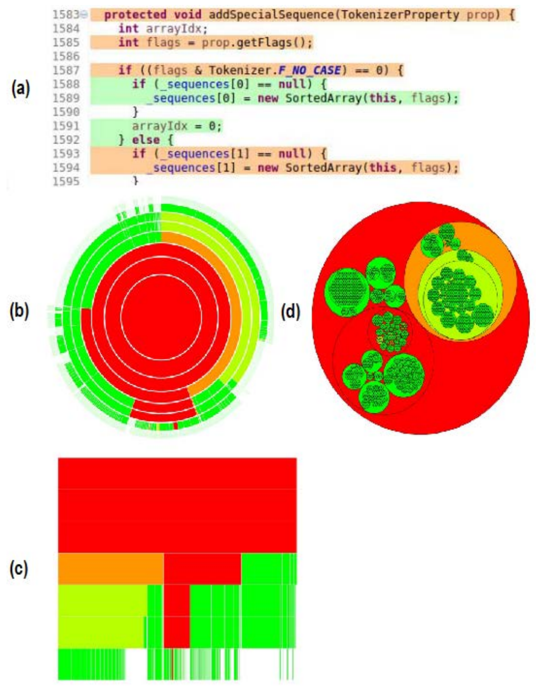
However, there are two main issues with this approach of displaying the results of SBFL, as follows:

- 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 color, and the developer is not provided with helpful hints about the location of faults. The red, green, and yellow colors were selected because they are the most natural and the best for viewing [17].
- 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 in 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 more green (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 not safety) 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 12: (a) coloring program elements in the source code itself [17], [72], [75], [76]. (b) visualizing the results as a Sunburst [73], [145], [146]. (c) visualizing the results as a Treemap [73], [145]. (d) visualizing the results as a Bubble Hierarchy [145].

However, more studies are required to propose new approaches or to improve the usability and effectiveness of the existing approaches alongside 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 the users with interactive visualization filtering options is an interesting area to be investigated.



```
protected void addSpecialSequence(TokenizerProperty prop) {
  int arrayIdx;
  int flags = prop.getFlags();

  if ((flags & Tokenizer.F_NO_CASE) == 0) {
    if (_sequences[0] == null) {
      _sequences[0] = new SortedArray(this, flags);
    }
    arrayIdx = 0;
  } else {
    if (_sequences[1] == null) {
      _sequences[1] = new SortedArray(this, flags);
    }
```

FIGURE 12. **Different visualization schemes. (a) Code coloring (b) Sunburst (c) Treemap (d) Bubble Hierarchy.**

R. NO CONTEXTUAL INFORMATION

In SBFL, the ranking is performed only based on the suspiciousness score of each 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 [147]. 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 [44].

Recently, the authors in [148] 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 [24], [149], 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, propagating 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 to the caller for boosting SBFL.

A method call sequence mining with a slide-window method has been used in [150] 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 [151]–[154], 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. However, many such studies and other contextual information can be considered to improve the effectiveness of SBFL.

VI. THREATS TO VALIDITY

There are different threats that might affect the validity of each survey study. For this study, different internal and external threats were avoided by considering the following actions:

- Internal validity
  - Finding related papers: There is no guarantee that all the papers related to the topic of this study have been found. Therefore, a search string containing different term synonyms was applied to various literature sources to obtain the related publications. Despite that, there may be some important relevant papers left. To address this threat, the snowballing search technique was used in order to lower the possibility of missing them.
  - Paper inclusion/exclusion criteria: Applying paper selection criteria can pose a threat of personal bias. Thus, only after the authors reached an agreement were the papers included or excluded in/from this study.
- External validity
  - Study reproducibility: Another threat to consider is whether or not other researchers will be able to replicate this study and obtain similar results. This issue can be addressed by providing the details

TABLE 10. List of all papers included in this study.

No.	Ref.	Paper Title	Year
1	[26]	Survey of Coverage-Based Testing Tools	2009
2	[27]	Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators	2013
3	[28]	Fault-localization techniques for software systems: A Literature Review	2014
4	[29]	A Survey on Software Fault Localization	2016
5	[12]	Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenge	2016
6	[30]	Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective	2017
7	[31]	Multiple fault localization of software programs: A systematic literature review	2020
8	[32]	Spectrum-based Fault Localization Techniques Application on Multiple-Fault Programs: A Review	2020
9	[37]	A practical evaluation of spectrum-based fault localization	2009
10	[38]	Validation of software testing experiments: A meta-analysis of icst 2013	2014
11	[39]	Traceability Challenge 2013: Statistical analysis for traceability experiments	2013
12	[40]	An evaluation of similarity coefficients for software fault localization	2006
13	[41]	Empirical evaluation of the tarantula automatic fault-localization technique	2005
14	[42]	Enhance fault localization using a 3d surface representation	2010
15	[43]	Leveraging Contextual Information from Function Call Chains to Improve Fault Localization	2020
16	[44]	Refining spectrum-based fault localization rankings	2009
17	[46]	On the accuracy of spectrum-based fault localization	2007
18	[47]	An empirical study of fault localization families and their combinations	2021
19	[48]	A learning-to-rank based fault localization approach using likely invariants	2016
20	[51]	Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques	2010
21	[52]	Effective fault localization using code coverage	2007
22	[53]	Ties within Fault Localization rankings: Exposing and Addressing the Problem	2011
23	[55]	An empirical study of the effects of test-suite reduction on fault localization	2008
24	[56]	Scalable statistical bug isolation	2005
25	[57]	Reduction-assisted fault localization:Don't throw away the by-products!	2021
26	[58]	Duals in spectral fault localization	2013
27	[7]	Model for spectra-based software diagnosis	2011
28	[66]	Are automated debugging techniques actually helping programmers?	2011
29	[68]	Charmfl: A fault localization tool for python	2021
30	[17]	Visualization of test information to assist fault localization	2002
31	[69]	Crisp - A fault localization tool for Java programs, 2007	
32	[70]	Debugging reinvented: Asking and answering why and why not questions about program behavior	2008
33	[71]	Interactive Fault Localization Using Test Information	2009
34	[72]	Zoltar: A spectrum-based fault localization tool	2009
35	[73]	Gzoltar: An eclipse plug-in for testing and debugging	2012
36	[74]	FLAVS: A fault localization add-in for visual studio	2015
37	[75]	UnitFL: A fault localization tool integrated with unit test	2016
38	[76]	Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software	2018
39	[77]	Exploring machine learning techniques for fault localization	2009
40	[78]	Code coverage differences of Java bytecode and source code instrumentation tools	2019
41	[80]	Emulation of software faults: A field data study and a practical approach	2006
42	[84]	Simulating the Effect of Test Flakiness on Fault Localization Effectiveness	2020
43	[85]	A study on the lifecycle of flaky tests	2020
44	[86]	Modeling and ranking flaky tests at apple	2020
45	[87]	Does refactoring of test smells induce fixing flaky tests?	2017
46	[88]	Wait, wait.no, tell me. analyzing selenium configuration effects on test flakiness	2019
47	[89]	Understanding reproducibility and characteristics of flaky tests through test reruns in java projects	2020
48	[90]	Flakeflagger: Predicting flakiness without rerunning tests	2021
49	[91]	DeFlaker: Automatically Detecting Flaky Tests	2018
50	[92]	Defects4J: a database of existing faults to enable controlled testing studies for Java programs	2014
51	[93]	BugsJS: a benchmark of javascript bugs	2019
52	[94]	BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies	2020
53	[95]	Empirical evaluation of existing algorithms of spectrum based fault localization	2014
54	[96]	Evolving human competitive spectra-based fault localisation techniques	2012
55	[20]	A new hybrid algorithm for software fault localization	2015
56	[97]	Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE	2019
57	[98]	Interactive fault localization leveraging simple user feedback	2012
58	[99]	A New Interactive Fault Localization Method with Context Aware User Feedback	2019
59	[100]	Enlightened debugging	2018
60	[101]	Tester Feedback Driven Fault Localization	2012
61	[102]	Practitioners' expectations on automated fault localization	2016
62	[103]	Automated debugging considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems	2016
63	[22]	Spectrum-Based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking	2018
64	[106]	Software fault localization based on program slicing spectrum	2012
65	[23]	Evolving suspiciousness metrics from hybrid data set for boosting a spectrum based fault localization	2020
66	[145]	Using HTML5 visualizations in software fault localization	2017
67	[146]	Pangolin: An SFL-Based Toolset for Feature Localization	2019
68	[148]	Call frequency-based fault localization	2021
69	[113]	A Crosstab-based Statistical Method for Effective Fault Localization	2008
70	[126]	A New Spectrum-based Fault Localization With the Technique of Test Case Optimization	2016
71	[127]	A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches	2017
72	[141]	A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization	2017
73	[124]	An Approach to Generate Effective Fault Localization Methods for Programs	2019
74	[128]	An Effective Strategy to Build Up a Balanced Test Suite for Spectrum-Based Fault Localization	2016
75	[114]	An Efficient Software Faults Localization Method based on Program Spectrum	2020
76	[115]	Towards Better Fault Localization: A Crosstab-Based Statistical Approach	2012

TABLE 10. ***(Continued.)*** **List of all papers included in this study.**

77	[142]	The Impact of Rare Failures on Statistical Fault Localization: The Case of the Defects4J Suite	2019
78	[107]	Statistical Fault Localization via Semi-dynamic Program Slicing	2011
79	[116]	Statistical fault localization in decision support system based on probability distribution criterion	2013
80	[117]	Statistical Fault Localization Based on Importance Sampling	2015
81	[129]	Spectrum-Based Fault Localization Method with Test Case Reduction	2015
82	[108]	Demystifying the Combination of Dynamic Slicing and Spectrum-based Fault Localization	2019
83	[130]	Using Spectrum-Based Fault Localization for Test Case Grouping	2009
84	[104]	Spectrum-Based Fault Localization Using Fault Triggering Model to Refine Fault Ranking List	2018
85	[131]	Spectrum-based fault localization tool with test case preprocessor	2013
86	[109]	Slice-based statistical fault localization	2014
87	[118]	Research on improved the Tarantula spectrum fault localization algorithm	2014
88	[132]	Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization	2018
89	[110]	On the analysis of spectrum based fault localization using hitting sets	2019
90	[140]	Mitigating the Effect of Coincidental Correctness in Spectrum Based Fault Localization	2012
91	[119]	Measuring the Odds of Statements Being Faulty	2013
92	[105]	Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques	2010
93	[143]	Incremental spectrum cloning algorithm for optimization of spectrum-based fault localization	2014
94	[133]	Improving the Accuracy of Spectrum-Based Fault Localization for Automated Program Repair	2020
95	[144]	Improving spectrum-based fault-localization through spectra cloning for fail test cases beyond balanced test suite	2014
96	[134]	Improving Spectrum-Based Fault Localization using quality assessment and optimization of a test suite	2020
97	[120]	Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis	2017
98	[111]	HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices	2014
99	[121]	Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme	2015
100	[122]	Evaluation and Analysis of Spectrum-Based Fault Localization with Modified Similarity Coefficients for Software Debugging	2013
101	[112]	Effective Statistical Fault Localization Using Program Slices	2012
102	[54]	A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System	2017
103	[81]	A Revisit of a Theoretical Analysis on Spectrum-Based Fault Localization	2015
104	[82]	A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization	2013
105	[59]	An Analysis on the Negative Effect of Multiple-Faults for Spectrum-Based Fault Localization	2018
106	[147]	An evaluation of pure spectrum-based fault localization techniques for large-scale software systems	2019
107	[151]	Contextualizing Spectrum-Based Fault Localization	2018
108	[150]	On the Use of Sequence Mining within Spectrum Based Fault Localisation	2018
109	[152]	Lightweight Defect Localization for Java	2005
110	[153]	Leveraging Method Call Anomalies to Improve the Effectiveness of Spectrum-Based Fault Localization Techniques for Object-Oriented Programs	2012
111	[154]	Fine-tuning spectrum based fault localisation with frequent method item sets	2016
112	[24]	Enhancing Spectrum-Based Fault Localization Using Fault Influence Propagation	2020
113	[44]	An Improvement to Fault Localization Technique Based on Branch-Coverage Spectra	2015
114	[62]	CLPS-MFL: Using Concept Lattice of Program Spectrum for Effective Multi-fault Localization	2013
115	[60]	Spectrum-based multi-fault localization using Chaotic Genetic Algorithm	2021
116	[49]	Spectrum-Based Fault Localization Framework to Support Fault Understanding	2019
117	[67]	Software Fault Localization via Mining Execution Graphs	2011
118	[135]	Proximity based weighting of test cases to improve spectrum based fault localization	2011
119	[136]	On the Integration of Test Adequacy, Test Case Prioritization, and Statistical Fault Localization	2010
120	[137]	Exploring the Triggering Modes of Spectrum-Based Fault Localization: An Industrial Case	2021
121	[138]	Effects of Class Imbalance in Test Suites: An Empirical Study of Spectrum-Based Fault Localization	2012
122	[63]	FTFL: A Fisher's test-based approach for fault localization	2021
123	[64]	Debugging in Parallel	2007
124	[61]	Fault density, fault types, and spectra-based fault localization	2015
125	[65]	FATOC: Bug Isolation Based Multi-Fault Localization by Using OPTICS Clustering	2020
126	[50]	Evaluating data-flow coverage in spectrum-based fault localization	2019
127	[149]	Fault centrality: boosting spectrum-based fault localization via local influence calculation	2021
128	[139]	An empirical study of boosting spectrum-based fault localization via pagerank	2019
129	[123]	Simultaneous Localization of Software Faults Based on Complex Network Theory	2018
130	[125]	A Framework for Improving Fault Localization Effectiveness Based on Fuzzy Expert System	2021

of how this study has been conducted. Therefore, Section IV thoroughly describes each step of the research methodology that was used in this study.

VII. CONCLUSION

Software cover many aspects of our day-to-day life and our world cannot 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, the SBFL is considered the most prominent technique. It computes the suspiciousness of each program entity of being faulty based on information gathered from test cases, their results, and their corresponding code coverage. Several important issues and challenges have been identified and categorized in this study. In each category, the most important issues have been briefly presented with some possible ideas to address them.

In conclusion, this survey aims to provide a clearer understanding of the most important challenges and issues in spectrum based fault localization, such that additional studies can be carried out to overcome these issues or possible avenues can be suggested for further exploration. Also, the results of this paper may be of great interest to novice testers and researchers who would like to provide contributions to this interesting topic. We hope that this paper will be regarded as a primary source of useful and relevant information on the issues and challenges in SBFL.

REFERENCES

[1] A. Maru, A. Dutta, K. V. Kumar, and D. P. Mohapatra, "Software fault localization using BP neural network based on function and branch coverage," *Evol. Intell.*, vol. 14, no. 1, pp. 87–104, Mar. 2021.

[2] M. Golagha, A. Pretschner, and L. C. Briand, "Can we predict the quality of spectrum-based fault localization?" in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 4–15.

[3] Y. Sasaki, Y. Higo, S. Matsumoto, and S. Kusumoto, "SBFL-suitability: A software characteristic for fault localization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 702–706.

[4] A. Perez and R. Abreu, "A qualitative reasoning approach to spectrum-based fault localization," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 372–373.

[5] S. Tiwari, K. K. Mishra, A. Kumar, and A. K. Misra, "Spectrum-based fault localization in regression testing," in *Proc. 8th Int. Conf. Inf. Technol.*, Apr. 2011, pp. 191–195.

[6] H. L. Ribeiro, P. A. R. de Araujo, M. L. Chaim, H. A. D. Souza, and F. Kon, "Evaluating data-flow coverage in spectrum-based fault localization," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2019, pp. 1–11.

[7] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–32, Aug. 2011.

[8] Y. Xiaobo, L. Bin, and W. Shihai, "How negative effects a multiple-fault program can do to spectrum-based fault localization," in *Proc. Syst. Health Manage. Conf. (PHM-Qingdao)*, Oct. 2019, pp. 1–6.

[9] M. Jia, Z. Cui, Y. Wu, R. Xie, and X. Liu, "SMFL integrating spectrum and mutation for fault localization," in *Proc. 6th Int. Conf. Dependable Syst. Appl. (DSA)*, Jan. 2020, pp. 511–512.

[10] C. Ma, T. Tan, Y. Chen, and Y. Dong, "An if-while-if model-based performance evaluation of ranking metrics for spectra-based fault localization," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf.*, Jul. 2013, pp. 609–618.

[11] P. Li, M. Jiang, and Z. Ding, "Fault localization with weighted test model in model transformations," *IEEE Access*, vol. 8, pp. 14054–14064, 2020.

[12] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," 2016, *arXiv:1607.04347*.

[13] Z. Cui, M. Jia, X. Chen, L. Zheng, and X. Liu, "Improving software fault localization by combining spectrum and mutation," *IEEE Access*, vol. 8, pp. 172296–172307, 2020.

[14] Y. Lei, C. Wang, X. Mao, and Q. Wu, "Enhancing contexts for automated debugging techniques," in *Proc. 7th Int. Conf. Softw. Eng. Adv. Enhancing (ICSEA)*, 2012, pp. 1–7.

[15] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *Proc. Int. Workshop Manag. Requirements Knowl.*, Los Alamitos, CA, USA, 1987, pp. 539–550.

[16] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proc. 6th Eur. Softw. Eng. Conf. Held Jointly*. Berlin, Germany: Springer-Verlag, 1997, pp. 432–449.

[17] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. 24th Int. Conf. Softw. Eng. (ICSE)*, 2002, pp. 467–477.

[18] A. Ochiai, "Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions-II," *Bull. Jpn. Soc. Sci. Fisheries*, vol. 22, no. 9, pp. 526–530, 1957.

[19] A. H. Cheetham and J. E. Hazel, "Binary (presence-absence) similarity," *J. Paleontol.*, vol. 43, no. 5, pp. 1130–1136, 1969.

[20] J. Kim, J. Park, and E. Lee, "A new hybrid algorithm for software fault localization," in *Proc. 9th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2015, pp. 1–8.

[21] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proc. 30th Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2014, pp. 191–200.

[22] Y. Wang, Z. Huang, B. Fang, and Y. Li, "Spectrum-based fault localization via enlarging non-fault region to improve fault absolute ranking," *IEEE Access*, vol. 6, pp. 8925–8933, 2018.

[23] A. A. Ajibode, T. Shu, and Z. Ding, "Evolving suspiciousness metrics from hybrid data set for boosting a spectrum based fault localization," *IEEE Access*, vol. 8, pp. 198451–198467, 2020.

[24] H. He, J. Ren, G. Zhao, and H. He, "Enhancing spectrum-based fault localization using fault influence propagation," *IEEE Access*, vol. 8, pp. 18497–18513, 2020.

[25] G. Laghari, K. Dahri, and S. Demeyer, "Comparing spectrum based fault localisation against test-to-code traceability links," in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, Dec. 2018, pp. 152–157.

[26] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *Comput. J.*, vol. 52, no. 5, pp. 589–597, 2007.

[27] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2013, pp. 314–324.

[28] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A literature review," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–8, Sep. 2014.

[29] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[30] M. Golagha and A. Pretschner, "Challenges of operationalizing spectrum-based fault localization from a data-centric perspective," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Mar. 2017, pp. 379–381.

[31] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Inf. Softw. Technol.*, vol. 124, pp. 106312–106332, Jan. 2020.

[32] A. Zakari, S. Abdullahi, N. M. Shagari, A. B. Tambawal, N. M. Shanono, J. Z. Maitama, R. A. Rasheed, A. Adamu, and S. M. Abdulrahman, "Spectrum-based fault localization techniques application on multiple-fault programs: A review," *Global J. Comput. Sci. Technol.*, vol. 4, pp. 41–48, Mar. 2020.

[33] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Inf. Softw. Technol.*, vol. 64, pp. 1–18, Aug. 2015.

[34] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Softw. Eng. Group, School Comput. Sci. Math., Keele Univ., U.K. Dept. Comput. Sci., Univ. Durham EBSE, Mumbai, India, Tech. Rep. EBSE- 2007-01, 2007.

[35] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *J. Syst. Softw.*, vol. 80, no. 4, pp. 571–583, 2007.

[36] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–10.

[37] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[38] M. Hays, J. H. Hayes, and A. C. Bathke, "Validation of software testing experiments: A meta-analysis of icst 2013," in *Proc. 7th Int. Conf. Softw. Test., Verification Validation*, 2014, pp. 333–342.

[39] M. Hays, J. H. Hayes, A. J. Stromberg, and A. C. Bathke, "Traceability Challenge 2013: Statistical analysis for traceability experiments: Software verification and validation research laboratory (SVVRL) of the University of Kentucky," in *Proc. 7th Int. Workshop Traceability Emerg. Forms Softw. Eng. (TEFSE)*, 2013, pp. 90–94.

[40] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput.*, 2006, pp. 39–46.

[41] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2005, pp. 273–282.

[42] Q. Shi, Z. Zhang, Z. Liu, and X. Gao, "Enhance fault localization using a 3D surface representation," in *Proc. 2nd Int. Conf. Comput. Res. Develop.*, 2010, pp. 720–724.

[43] A. Beszedes, F. Horvath, M. Di Penta, and T. Gyimothy, "Leveraging contextual information from function call chains to improve fault localization," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2020, pp. 468–479.

[44] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. Van Gemund, "Refining spectrum-based fault localization rankings," in *Proc. ACM Symp. Appl. Comput.*, 2009, pp. 409–414.

[45] S. Xu, J. Xu, H. Yang, J. Yang, C. Guo, L. Yuan, W. Song, and G. Si, "An improvement to fault localization technique based on branch-coverage spectra," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, pp. 282–287.

[46] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Ind. Conf. Pract. Res. Techn.*, 2007, pp. 89–98.

[47] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 332–347, Feb. 2021.

[48] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 177–188.

[49] Y. WANG, Z. HUANG, Y. LI, R. WANG, and Q. YU, "Spectrum-based fault localization framework to support fault understanding," *IEICE Trans. Inf. Syst.*, vol. E102-D, no. 4, pp. 863–866, 2019.

[50] H. L. Ribeiro, P. A. Roberto de Araujo, M. L. Chaim, H. A. D. Souza, and F. Kon, "Evaluating data-flow coverage in spectrum-based fault localization," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2019, pp. 1–11.

[51] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *Proc. 10th Int. Conf. Qual. Softw.*, Jul. 2010, pp. 13–22.

[52] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, Jul. 2007, pp. 449–456.

[53] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 21, no. 6, pp. 803–827, Sep. 2011.

[54] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. Van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 114–125.

[55] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 201–210.

[56] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[57] D. Vince, R. Hodován, and Á. Kiss, "Reduction-assisted fault localization: Don't throw away the by-products!" in *Proc. 16th Int. Conf. Softw. Technol.*, 2021, pp. 196–206.

[58] L. Naish and H. J. Lee, "Duals in spectral fault localization," in *Proc. 22nd Austral. Softw. Eng. Conf.*, 2013, pp. 51–59.

[59] Y. Xiaobo, B. Liu, and W. Shihai, "An analysis on the negative effect of multiple-faults for spectrum-based fault localization," *IEEE Access*, vol. 7, pp. 2327–2347, 2018.

[60] D. Ghosh and J. Singh, "Spectrum-based multi-fault localization using chaotic genetic algorithm," *Inf. Softw. Technol.*, vol. 133, pp. 1–16, Jan. 2021.

[61] N. DiGiuseppe and J. A. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Softw. Eng.*, vol. 20, no. 4, pp. 928–967, 2015.

[62] X. Sun, B. Li, and W. Wen, "CLPS-MFL: Using concept lattice of program spectrum for effective multi-fault localization," in *Proc. 13th Int. Conf. Qual. Softw.*, Jul. 2013, pp. 204–207.

[63] A. Dutta, K. Kunal, S. S. Srivastava, S. Shankar, and R. Mall, "FTFL: A Fisher's test-based approach for fault localization," *Innov. Syst. Softw. Eng.*, vol. 4, pp. 1–25, Jun. 2021.

[64] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. Int. Symp. Softw. Test. Anal.*, 2007, pp. 16–26.

[65] Y.-H. Wu, Z. Li, Y. Liu, and X. Chen, "FATOC: Bug isolation based multi-fault localization by using OPTICS clustering," *J. Comput. Sci. Technol.*, vol. 35, no. 5, pp. 979–998, Oct. 2020.

[66] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Test. Anal.*, New York, NY, USA, 2011, pp. 199–209.

[67] S. Parsa, S. Arabi, and N. E. Koopaei, "Software fault localization via mining execution graphs," in *Proc. Int. Conf. Comput. Sci. Appl. (ICCSA)*, 2011, pp. 610–623.

[68] Q. Idrees Sarhan, A. Szatmari, R. Toth, and A. Beszedes, "CharmFL: A fault localization tool for Python," in *Proc. IEEE 21st Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2021, pp. 114–119.

[69] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip, "Crisp—A fault localization tool for Java programs," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 775–778.

[70] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 301–310.

[71] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive fault localization using test information," *J. Comput. Sci. Technol.*, vol. 24, no. 5, pp. 962–974, Sep. 2009.

[72] T. Janssen, R. Abreu, and A. J. Van Gemund, "Zoltar: A spectrum-based fault localization tool," in *Proc. ESEC/FSE Workshop Softw. Integr. Evol. Runtime*, 2009, pp. 23–29.

[73] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2012, pp. 378–381.

[74] N. Wang, Z. Zheng, Z. Zhang, and C. Chen, "FLAVS: A fault localization add-in for visual studio," in *Proc. IEEE/ACM 1st Int. Workshop Complex Faults Failures Large Softw. Syst. (COUFLESS)*, May 2015, pp. 1–6.

[75] C. Chen and N. Wang, "UnitFL: A fault localization tool integrated with unit test," in *Proc. 5th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Dec. 2016, pp. 136–142.

[76] H. L. Ribeiro, R. P. A. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon, "Jaguar: A spectrum-based fault localization tool for real-world software," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2018, pp. 404–409.

[77] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring machine learning techniques for fault localization," in *Proc. 10th Latin Amer. Test Workshop*, Mar. 2009, pp. 1–6.

[78] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, "Code coverage differences of Java bytecode and source code instrumentation tools," *Softw. Qual. J.*, vol. 27, no. 1, pp. 79–123, Mar. 2019.

[79] H. B. Hassan and Q. I. Sarhan, "Performance evaluation of graphical user interfaces in Java and C#," in *Proc. Int. Conf. Comput. Sci. Softw. Eng. (CSASE)*, Apr. 2020, pp. 290–295.

[80] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[81] T. Y. Chen, X. Xie, F.-C. Kuo, and B. Xu, "A revisit of a theoretical analysis on spectrum-based fault localization," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 17–22.

[82] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 1–40, 2013.

[83] *Spectrum-Based Fault Localization (SFL) Simulator*. Accessed: Oct. 1, 2021. [Online]. Available: https://github.com/SERG-Delft/sfl-simulator/

[84] B. Vancsics, T. Gergely, and A. Beszedes, "Simulating the effect of test flakiness on fault localization effectiveness," in *Proc. IEEE Workshop Validation, Anal. Evol. Softw. Tests (VST)*, Feb. 2020, pp. 28–35.

[85] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 1471–1482.

[86] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at apple," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Engineering: Softw. Eng. Pract.*, Jun. 2020, pp. 110–119.

[87] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Proc. Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2017, pp. 1–12.

[88] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "Wait, wait. No, tell Me. Analyzing selenium configuration effects on test flakiness," in *Proc. IEEE/ACM 14th Int. Workshop Autom. Softw. Test (AST)*, May 2019, pp. 7–13.

[89] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2020, pp. 403–413.

[90] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *Proc. 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 1572–1584.

[91] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 433–444.

[92] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, New York, NY, USA, 2014, pp. 437–440.

[93] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "BugsJS: A benchmark of Javascript bugs," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 90–101.
[94] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, and J. Phan, "BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1556–1560.
[95] J. Kim and E. Lee, "Empirical evaluation of existing algorithms of spectrum based fault localization," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Feb. 2014, pp. 346–351.
[96] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering* (Lecture Notes in Computer Science), vol. 7515. 2012, pp. 244–258.
[97] G. Balogh, F. Horvath, and A. Beszedes, "Poster: Aiding Java developers with interactive fault localization in eclipse IDE," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 371–374.
[98] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 67–76.
[99] F. Horvath, V. S. Lacerda, A. Beszedes, L. Vidacs, and T. Gyimothy, "A new interactive fault localization method with context aware user feedback," in *Proc. IEEE 1st Int. Workshop Intell. Bug Fixing (IBF)*, Feb. 2019, pp. 23–28.
[100] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 82–92.
[101] A. Bandyopadhyay and S. Ghosh, "Tester feedback driven fault localization," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 41–50.
[102] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2016, pp. 165–176.
[103] X. Xia, L. Bao, D. Lo, and S. Li, "'Automated debugging considered harmful' considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 267–278.
[104] Y. Wang, Z. Huang, R. Wang, Q. Yu, and Q. Yu, "Spectrum-based fault localization using fault triggering model to refine fault ranking list," *IEICE Trans. Inf. Syst.*, vol. E101-D, no. 10, pp. 2436–2446, 2018.
[105] X. Xie, T. Y. Chen, and B. Xu, "Isolating suspiciousness from spectrum-based fault localization techniques," in *Proc. 10th Int. Conf. Qual. Softw.*, Jul. 2010, pp. 385–392.
[106] W. Wen, "Software fault localization based on program slicing spectrum," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 1511–1514.
[107] R. Yu, L. Zhao, L. Wang, and X. Yin, "Statistical fault localization via semi-dynamic program slicing," in *Proc. IEEE 10th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Nov. 2011, pp. 695–700.
[108] S. Reis, R. Abreu, and M. d'Amorim, "Demystifying the combination of dynamic slicing and spectrum-based fault localization," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 4760–4766.
[109] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *J. Syst. Softw.*, vol. 89, pp. 51–62, Mar. 2014.
[110] J. Tu, X. Xie, T. Y. Chen, and B. Xu, "On the analysis of spectrum based fault localization using hitting sets," *J. Syst. Softw.*, vol. 147, pp. 106–123, Jan. 2019.
[111] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices," *J. Syst. Softw.*, vol. 90, pp. 3–17, Apr. 2014.
[112] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf.*, Jul. 2012, pp. 1–10.
[113] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proc. Int. Conf. Softw. Test., Verification, Validation*, Apr. 2008, pp. 42–51.
[114] X. Yu, H. Tang, J. Zou, and F. Yu, "An efficient software faults localization method based on program spectrum," in *Proc. IEEE Int. Conf. Inf. Technol., Big Data Artif. Intell. (ICIBA)*, vol. 1, Oct. 2020, pp. 88–93.
[115] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 3, pp. 378–396, May 2012.
[116] P. Hao, Z. Zheng, Y. Gao, and Z. Zhang, "Statistical fault localization in decision support system based on probability distribution criterion," in *Proc. Joint IFSA World Congr. NAFIPS Annu. Meeting (IFSA/NAFIPS)*, Jun. 2013, pp. 878–883.
[117] A. S. Namin, "Statistical fault localization based on importance sampling," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2015, pp. 58–63.
[118] X. Liang, L. Mao, and M. Huang, "Research on improved the tarantula spectrum fault localization algorithm," in *Proc. 2nd Int. Conf. Inf. Technol. Electron. Commerce*, Dec. 2014, pp. 60–63.
[119] X. Xue and A. S. Namin, "Measuring the odds of statements being faulty," in *Proc. Int. Conf. Reliable Softw. Technol.*, 2013, pp. 109–126.
[120] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 1, pp. 1–30, 2017.
[121] D. Landsberg, H. Chockler, D. Kroening, and M. Lewis, "Evaluation of measures for statistical fault localisation and an optimising scheme," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2015, pp. 115–129.
[122] Y.-S. You, C.-Y. Huang, K.-L. Peng, and C.-J. Hsu, "Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging," in *Proc. 37th Annu. Comput. Softw. Appl. Conf.*, 2013, pp. 180–189.
[123] A. Zakari, S. P. Lee, and C. Y. Chong, "Simultaneous localization of software faults based on complex network theory," *IEEE Access*, vol. 6, pp. 23990–24002, 2018.
[124] B. Bagheri, M. Rezaalipour, and M. Vahidi-Asl, "An approach to generate effective fault localization methods for programs," in *Proc. Int. Conf. Fundamentals Softw. Eng.*, 2019, pp. 244–259.
[125] C.-T. Lin, W.-Y. Chen, and J. Intasara, "A framework for improving fault localization effectiveness based on fuzzy expert system," *IEEE Access*, vol. 9, pp. 82577–82596, 2021.
[126] J. Kim, J. Park, and E. Lee, "A new spectrum-based fault localization with the technique of test case optimization," *J. Inf. Sci. Eng.*, vol. 32, no. 1, pp. 177–196, 2016.
[127] A. Perez, R. Abreu, and A. Van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 654–664.
[128] N. Li, R. Wang, Y. Tian, and W. Zheng, "An effective strategy to build up a balanced test suite for spectrum-based fault localization," *Math. Problems Eng.*, vol. 2016, Apr. 2016, Art. no. 5813490.
[129] X. Zhang, Z. Wang, W. Zhang, H. Ding, and L. Chen, "Spectrum-based fault localization method with test case reduction," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 548–549.
[130] M. Weiglhofer, G. Fraser, and F. Wotawa, "Using spectrum-based fault localization for test case grouping," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 630–634.
[131] P. Daniel and K. Y. Sim, "Spectrum-based fault localization tool with test case preprocessor," in *Proc. IEEE Conf. Open Syst. (ICOS)*, Dec. 2013, pp. 162–167.
[132] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2018, pp. 184–191.
[133] T. Kuma, Y. Higo, S. Matsumoto, and S. Kusumoto, "Improving the accuracy of spectrum-based fault localization for automated program repair," in *Proc. 28th Int. Conf. Program. Comprehension*, Jul. 2020, pp. 376–380.
[134] C. Liu, C. Ma, and T. Zhang, "Improving spectrum-based fault localization using quality assessment and optimization of a test suite," in *Proc. IEEE 20th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Oct. 2020, pp. 72–78.
[135] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 420–423.
[136] B. Jiang and W. K. Chan, "On the integration of test adequacy, test case prioritization, and statistical fault localization," in *Proc. 10th Int. Conf. Qual. Softw.*, Jul. 2010, pp. 377–384.
[137] T. Dao, M. Wang, and N. Meng, "Exploring the triggering modes of spectrum-based fault localization: An industrial case," in *Proc. 14th IEEE Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2021, pp. 406–416.

[138] C. Gong, Z. Zheng, W. Li, and P. Hao, "Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf. Workshops*, Jul. 2012, pp. 470–475.

[139] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via PageRank," *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1089–1113, Jun. 2021.

[140] A. Bandyopadhyay, "Mitigating the effect of coincidental correctness in spectrum based fault localization," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 479–482.

[141] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W. K. Chan, and Z. Zheng, "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization," *J. Syst. Softw.*, vol. 129, pp. 35–57, Jul. 2017.

[142] Y. Kucuk, T. A. D. Henderson, and A. Podgurski, "The impact of rare failures on statistical fault localization: The case of the Defects4J suite," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 24–28.

[143] P. Daniel, K. Y. Sim, and S. Seol, "Incremental spectrum cloning algorithm for optimization of spectrum-based fault localization," *Contemp. Eng. Sci.*, vol. 7, pp. 1649–1655, 2014.

[144] P. Daniel, K. Y. Sim, and S. Seol, "Improving spectrum-based fault-localization through spectra cloning for fail test cases beyond balanced test suite," *Contemp. Eng. Sci.*, vol. 7, pp. 677–682, Jan. 2014.

[145] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," in *Proc. 1st IEEE Work. Conf. Softw. Vis. (VISSOFT)*, 2013, pp. 1–10.

[146] B. Castro, A. Perez, and R. Abreu, "Pangolin: An SFL-based toolset for feature localization," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 2019, pp. 1130–1133.

[147] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Hoorn, A. Filieri, and D. Lo, "An evaluation of pure spectrum-based fault localization techniques for large-scale software systems," *Softw. Pract. Exper.*, vol. 49, no. 8, pp. 1197–1224, Aug. 2019.

[148] B. Vancsics, F. Horvath, A. Szatmari, and A. Beszedes, "Call frequency-based fault localization," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2021, pp. 365–376.

[149] G. Zhao, H. He, and Y. Huang, "Fault centrality: Boosting spectrum-based fault localization via local influence calculation," *Appl. Intell.*, vol. 5, pp. 1–23, Oct. 2021.

[150] G. Laghari and S. Demeyer, "On the use of sequence mining within spectrum based fault localisation," in *Proc. 33rd Annu. ACM Symp. Appl. Comput.*, Apr. 2018, pp. 1916–1924.

[151] H. A. de Souza, D. Mutti, M. L. Chaim, and F. Kon, "Contextualizing spectrum-based fault localization," *Inf. Softw. Technol.*, vol. 94, pp. 245–261, Oct. 2018.

[152] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *Proc. Eur. Conf. Object-Oriented Program.*, 2005, pp. 528–550.

[153] J. Tu, L. Chen, Y. Zhou, J. Zhao, and B. Xu, "Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs," in *Proc. 12th Int. Conf. Qual. Softw.*, 2012, pp. 1–8.

[154] G. Laghari, A. Murgia, and S. Demeyer, "Fine-tuning spectrum based fault localisation with frequent method item sets," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Aug. 2016, pp. 274–285.

QUSAY IDREES SARHAN received the B.Sc. degree in software engineering from the University of Mosul, Iraq, in 2007, and the M.Tech. degree in software engineering from Jawaharlal Nehru Technological University, India, in 2011. Since 2012, he has been a Lecturer at the University of Duhok, Iraq. He is currently working at the Department of Software Engineering, University of Szeged, Hungary. He has a couple of national and international publications. His research interests include software engineering, the Internet of Things, and embedded systems.

ÁRPÁD BESZÉDES received the Ph.D. degree in computer science from the University of Szeged, in 2005. He is currently an Associate Professor at the University of Szeged. He has over 100 publications. His research interests include static and dynamic program analysis with special emphasis on software testing and debugging applications. He is regularly invited to serve in the program committees for various software engineering conferences, and as a Reviewer and an Editor for *Software Engineering* and *Computer Science* journals.