Effective Spectrum Based Fault Localization Using Contextual Based Importance Weight*

Qusay Idrees Sarhan[1,2][0000−0001−8708−0063] and Árpád Beszédes[1][0000−0002−5421−9302]

[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
[2] Department of Computer Science, University of Duhok, Duhok, Iraq
{sarhan, beszedes}@inf.u-szeged.hu

Abstract. In Spectrum-Based Fault Localization (SBFL), a suspicion score for each program element (e.g., statement, method, or class) is calculated by using a risk evaluation formula based on tests coverage and their results. The elements are then ranked from most suspicious to least suspicious based on their scores. The elements with the highest scores are thought to be the most faulty. The final ranking list of program elements helps testers during the debugging process when seeking the source of a fault in the program under test. In this paper, we present an approach that gives more importance to program elements that are executed by more failed test cases and appear in different contexts of method calls (both as callees and as callers) in these tests compared to other elements. In essence, we are emphasizing the failing test cases factor because there are comparably much less failing tests than passing ones. We multiply each element's suspicion score obtained by a SBFL formula by this importance weight, which is the ratio of covering failing tests over all failing tests combined with the so-called method calls frequency. The proposed approach can be applied to SBFL formulas without modifying their structures. The experimental results of our study show that our approach achieved a better performance in terms of average ranking compared to the underlying SBFL formulas and comparable approaches. It also improved the Top-N categories and increased the number of cases in which the faulty method became the top-ranked element.

Keywords: Debugging · fault localization · spectrum-based fault localization · importance weight · method calls.

1 Introduction

Many aspects of our daily lives are automated by software. They are, however, far from being faultless. Software bugs can result in dangerous situations, in-

* The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004) and the project no. TKP2021-NVA-09 which was implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

cluding death. As a result, various software fault localization techniques, such as spectrum-based fault localization (SBFL) [14], have been proposed over the last few decades. SBFL calculates the likelihood of each program element of being faulty based on program spectra collected from executing test cases and their results. SBFL, on the other hand, is not yet widely used in the industry due to a number of challenges and issues [11].

One of such issues is that program elements are ranked from most to least suspicious in order of their suspicion scores. Testers check each element starting at the top of the ranking list to determine whether it is faulty or not. Thus, the faulty element should be placed near the top of the ranking list to aid testers in discovering it early in the evaluation process and with least effort. Many times, SBFL formulas place the faulty elements far from the ranking list top.

In this paper, we are addressing this issue by presenting an approach that gives more importance to program elements that are executed by more failed test cases and appear in different contexts of method calls (both as callees and as callers) in these tests compared to other elements. The intuition is the following. A typical SBFL matrix is unbalanced in the sense that there are much more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly. Also, program elements might behave differently when appearing in different calling contexts. We propose to emphasize the factor of the failing tests in the formulas, which is achieved by introducing a multiplication factor to any SBFL formula. This factor is called *the importance weight*, and is given as the ratio of executed failing tests for a program element with respect to all failing tests combined with the so-called method calls frequency. In other words, a program element will be more suspicious if it is affected by a larger portion of the failing tests and appears in a variety of calling contexts during such test cases. The proposed approach can be applied to any SBFL formula without modifying it.

The experimental results of our study show that our approach achieved a better performance in terms of average ranking and Top-N categories compared to well-known underlying SBFL formulas and Vancsics et al's approach in [12].

The following are the main contributions of the paper:

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

We defined the following Research Questions (RQs) for this paper:

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

The rest of the paper is structured as follows: Section 2 introduces SBFL's work and its key concepts in a nutshell. Section 3 provides a summary of the most

relevant works. Section 4 introduces our approach of enhancing SBFL formulas. Section 5 provides an overview on the used subject programs, data collection, and the evaluation baselines. Section 6 presents the experimental results of this study compared to the existing approaches and provides some analysis about the effectiveness of our proposed approach. Section 7 reports the potential threats to validity. Finally, we present our conclusions and potential future works in Section 8.

2 Background of SBFL

This section explains SBFL and how it can be used to find software faults by ranking program elements according to their likelihood of being faulty.

2.1 SBFL process

Many techniques have been proposed in the literature to automate the process of software fault localization [14]. However, SBFL is the most dominant because of its straightforward but potent nature, i.e. it only uses test coverage and results to calculate the suspiciousness of each program element of being faulty.

The execution of test cases on program elements is recorded to extract the spectra (i.e., tests coverage and test results) for the program under test. Program spectra information is a two-dimensional matrix that demonstrate the relationship between test cases and program elements. Its columns depict the test cases, while its rows depict the program elements. If a test case covers an element in the matrix, it is assigned a value of 1; otherwise, it is assigned a value of 0. The test results are also stored in the matrix, where 0 means the test case is passed and 1 when it is failed. For each program element e, the following four basic statistical numbers are frequently calculated from the program spectra: (a) ef: number of failed tests executing e; (b) ep: number of passed tests executing e; (c) nf: number of failed tests not executing e; (d) np: number of passed tests not executing e.

Then, these four basic statistics can be used by a SBFL formula to output a ranked list of program elements. Whichever element is at the top of the list is the most likely to be buggy. As a result, SBFL can assist testers in locating the faulty element in the target program's code.

2.2 Code example

To demonstrate SBFL's work, consider a Java program, adopted from [12], which consists of four main methods (a, b, f, and g), and its four test cases ($t1$, $t2$, $t3$, and $t4$) as shown in Figure 1. It can be noted that there is a fault in method g (the correct statement is _x+=i) and only $t1$ and $t4$ execute that faulty method.

```
public class Example{
  private int _x = 0;
  private int _s = 0;
  public int x() {return _x;}

  public void a(int i){
    _s = 0;
    if (i==0) return;
    if (i<0)
      for (int y=0;y<=4;y++)
        f(i);
    else
      g(i);
  }

  public void b(int i){
    _s = 1;
    if (i==0) return;
    if (i<0)
      a(Math.abs(i));
    else
      for (int y=0;y<=1;y++)
        g(i);
  }

  private void f(int i){
    _x -= i;
  }

  private void g(int i){
   //should be _x += i;
    _x += (i+_s);
  }
}
```

A - Program Code

```
public class ExampleTest {
 @Test public void t1() {
  Example tester = new Example();
  tester.a(-1);
  tester.a(1);
  tester.b(1);
  // failed -> 8
  assertEquals(9, tester.x());
  }

  @Test public void t2() {
  Example tester = new Example();
  tester.a(1);
  tester.b(1);
  // failed -> 3
  assertEquals(4, tester.x());
  }

  @Test public void t3() {
  Example tester = new Example();
  tester.a(1);
  tester.b(0);
  assertEquals(1, tester.x());
  }

  @Test public void t4() {
  Example tester = new Example();
  tester.a(-1);
  tester.a(1);
  tester.b(-1);
  assertEquals(7, tester.x());
  }
}
```

B - Test Cases

Fig. 1. Running example – program code and test cases

2.3 Program spectra and basic statistics

Assume the tests were run on the program and the program spectra (i.e., information on how the four program methods were executed in passed and failed test cases) were captured. This data is presented in Table 1.

Table 1. Program spectra and four basic statistics

	t1	t2	t3	t4	ef	ep	nf	np
a	1	1	1	1	2	2	0	0
b	1	1	1	1	2	2	0	0
f	1	0	0	1	1	1	1	1
g	1	1	1	1	2	2	0	0
Results	1	1	0	0				

A 1 in the cell corresponding to the method a and the test case $t1$ indicates that $t1$ has covered the method a, while a 0 indicates that the method a has not been covered. A 1 in the "Results" row indicates that the relevant test case failed, and a 0 indicates that it passed. The $t2$ test case, for example, calls the methods a, b, and g, but it fails because the output of this calls sequence should be 3, not 4.

Table 1's last four columns represent the four basic statistics (i.e., ef, ep, nf, and np) that are calculated from the program spectra. For example, the value of ef of the method a is 2 because it has been executed by two failed tests $t1$ and $t2$.

2.4 SBFL formulas

A SBFL formula is a mathematical expression that often uses these four basic statistics to compute the suspicion score of each program element of being faulty. We apply various popular formulas [9] for the experimental evaluation in this paper, as shown in Table 2.

Table 2. SBFL formulas used in the study

Name	Formula
Jaccard (J)	$\frac{ef}{ef+nf+ep}$
Barinel (B)	$\frac{ef}{ef+ep}$
SorensenDice (S)	$\frac{2*ef}{2*ef+nf+ep}$
DStar (DS)	$\frac{ef*ef}{ep+nf}$
Dice (D)	$\frac{2*ef}{ef+nf+ep}$
Interest (I)	$\frac{ef}{(ef+nf)*(ef+ep)}$
Kulczynski1 (K)	$\frac{ef}{nf+ep}$
Cohen (C)	$\frac{2*(ef*np)-2*(nf*ep)}{(ef+ep)*(ep+np)+(nf+np)*(ef+nf)}$

2.5 Suspiciousness scores

We can get the suspiciousness score for each method in Table 3 by applying some formulas to the spectra of our Java program example in Table 1. It is worth noticing that for several methods in this example, each SBFL formula returns the same suspiciousness score. To put it another way, SBFL formulas in this circumstance are unable to distinguish the techniques just on the basis of their pure scores. Thus, the buggy method g is hardly distinguishable from the other methods. As a result, in this scenario, the SBFL effectiveness is reduced by the tie problem among program methods [11].

Table 3. Program example scores and average ranks

Method	J	Rank	B	Rank	S	Rank
a	0.5	2	0.5	2.5	0.67	2
b	0.5	2	0.5	2.5	0.67	2
f	0.33	4	0.5	2.5	0.5	4
g	0.5	2	0.5	2.5	0.67	2

2.6 Suspiciousness ranking

We use the average rank approach in Equation 1, where S denotes the tie's starting position and E denotes the tie's size, to analyze SBFL efficiency in general. Here, the program elements with the same suspicion score are ranked using the average rank, such elements are called *tied elements*, by taking the average of their positions after they get sorted, in descending order, based on their scores.

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

Table 3 presents the average ranks of the sample program using the SBFL formulas that were chosen. Ranks that are part of a tie are highlighted in gray. It can be noted that based on the ranks, Barinel (B) is unable to distinguish the methods from each other, while the other formulas result in a tie-group of three methods.

As a result, such methods are grouped together in the ranking and cannot be distinguished from one another in terms of which one should be investigated first. Therefore, additional information besides the basic hit-spectra are required to break these ties. For example, with a satiable additional information, the buggy method can be moved to a higher place in the ranking list.

3 Related Works

This section summarizes the most important efforts to improve SBFL by focusing on its formulas.

One strategy to improve SBFL is to create new SBFL formulas that outperform the current ones. The authors of [13] presented a new SBFL formula named "DStar", for example. The proposed formula was compared to a number of commonly used formulas, and it outperformed them all. Using Genetic Programming (GP), SBFL formulas can also be created automatically. The authors in [1] employed GP to create SBFL formulas automatically based on program spectra. The authors were able to come up with a total of 30 formulas. According to their findings, the GP is a good strategy for producing effective SBFL formulas.

Improvements can also be achieved by modifying existing SBFL formulas. The authors in [16] also tweaked three well-known SBFL formulas to account for the possibility that some failed tests yield more information than others. As a result, different weights for improving SBFL performance for failed tests were allocated to the three formulas and then used using multi-coverage spectra.

Combining existing SBFL formulas with one another is a different technique. The authors in [3] developed a method for mixing 40 distinct SBFL formulas to create a new SBFL formula suitable to a certain program. The suggested method pulls information from the program via mutation testing, and then uses different voting systems to merge numerous formulae depending on the acquired information to build a new formula. Experiments reveal that the formula created by their method is superior to a number of current formulas. It is worth noting that researchers attempted to combine multiple formulas in order to build new ones. The new formula is regarded as a hybrid formula since it combines the benefits of multiple previous formulations. As stated in [7, 10], the performance of a hybrid formula should be superior to that of existing formulas.

Another way is to supplement existing SBFL formulas with new data. The authors in [12] added new contextual information to the underlying SBFL formulas by using the method calls frequency of the subject programs during the execution of failed tests. In each formula, the frequency ef was substituted for the ef. Their findings showed that incorporating additional data from method calls into the underlying formulas can boost SBFL effectiveness. In addition, the authors in [17] proposed a method for improving SBFL by applying the PageRank algorithm to differentiate tests. Their method takes the original program spectrum information and recomputes it using PageRank, taking into account the contributions of various test cases. The standard SBFL formulas on the recomputed spectrum information can be used to improve fault localization.

SBFL can also be improved by breaking ties. Ties in SBFL are dominant; thus it is unlikely that any of the known SBFL formulas will generate distinct scores for all program elements. The authors in [5] proposed an approach, also based on method calls frequency, to break tied program elements. Their experimental results showed that employing information from method calls frequency in failed tests cases for tie breaking can improve the effectiveness of SBFL.

SBFL's performance was improved in several ways as a result of the aforementioned studies. Our proposed approach improves the SBFL performance by giving more importance to program elements that are executed by more failed test cases and appear in different contexts of method calls (both as callees and as callers) in these tests. The advantages of our proposed approach over others are: (a) It does not modify the existing SBFL formulas. Thus, it can be applied to any SBFL formula to enhance its effectiveness. This is very important as it makes the proposed approach more applicable than other approaches. (b) It solves the issue of unbalanced SBFL matrix in the sense that there are much more passing tests than failing ones, and many SBFL formulas treat passing and failing tests similarly. (c) Finally, it also involves information outside the regular SBFL matrix, namely the calling context information.

4 The proposed SBFL enhancing approach

In this section, we present the concept of our proposed approach to enhance the effectiveness of the underlying SBFL formulas and how it works. Then, we present its effectiveness when applied on our motivational example.

4.1 The frequency-based ef (ϕ)

To obtain the frequency-based ef (ϕ), we first create the *frequency-based* SBFL matrix, which replaces the traditional hit-based one. As a result, instead of $\{0, 1\}$, each element will receive an integer reflecting the number of occurrences of the given element in the unique call stacks while running in various calling contexts. In other words, unique call stacks are data structures that store call stack state information during test case execution and count the number of method occurrences within these structures [12].

Table 4 presents the frequency-based matrix for our Java example. The unique call stacks of $t1$, for example, are (a, f), (a, g), and (b, g), hence the frequency of g for test $t1$ will be 2.

Table 4. Frequency-based matrix

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

ϕ is determined by adding the frequency-based values for the failing test cases in the matrix. The greater the value of ϕ for a method, the more suspicious is. For instance, adding the frequency-based values of the faulty method g (i.e., 2 and 2) in the matrix for the failing test cases (i.e., $t1$ and $t2$) will yield 4 as the value of ϕ for the method g, which is the biggest ϕ value compared to others.

4.2 The proposed approach

Using the selected SBFL formulas on the program spectra, we calculate the suspicion scores of program methods. The output are the initial suspicion scores of methods. Then, we multiply each initial score of each method by its importance weight which is computed via Equation 2.

$$\text{Importance Weight} = \left(\frac{\text{ef} * \phi}{\text{ef} + \text{nf}} \right) \tag{2}$$

The order of methods in the initial ranking list will be rearranged based on the value of each method's importance weight, resulting in a final improved ranking list. From Table 5, it can be seen that the faulty method g will get the rank 1 after applying our proposed approach instead of 2 (in case of J and S) or 2.5 (in case of B) as its weight is greater than others. The rationale behind using the ϕ is that if a method appears in a lot of calls during a failed test, it will be considered more suspicious and will be given a higher rank than other methods. We combine the ϕ with $ef/(ef + nf)$ because the later emphasizes the failing test cases factor because there are comparably much less failing tests than passing ones.

Table 5. Program example scores and average ranks after applying our approach

Method	J**	Rank	B**	Rank	S**	Rank
a	1.5	2	1.5	2	2.0	2
b	1.0	3	1.0	3	1.33	3
f	0.17	4	0.25	4	0.25	4
g	2.0	1	2.0	1	2.67	1

5 Evaluation

5.1 Subject programs

In this study, we used the faulty programs of version v1.5.0 of Defects4J [6]; where 6 open-source Java programs had 438 actual faults found in their repositories[3]. However, due to instrumentation issues or incorrect test results, 27 defects were eliminated from this analysis. As a result, the final dataset used contained a total of 411 faults. Each program's primary characteristics are presented in Table 6.

Table 6. Subject programs

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

[3] `https://github.com/rjust/defects4j/tree/v1.5.0`

5.2 Granularity of data collection

Method-level granularity was used as a program spectra/coverage type in this work. It provides users with a more understandable level of abstraction [2, 18]. However, in terms of the proposed approach, there is no theoretical barrier to investigate other granularity levels as well.

5.3 Evaluation baselines

Several well-studied SBFL formulas were utilized as baselines in this paper, as presented in Table 2, to evaluate and compare our proposed approach to. It is worth mentioning that Vancsics et al's approach proposed in [12] is comparable to ours; thus, we will compare our results to it too.

6 Experimental Results and Discussion

6.1 Achieved improvements in the average ranks

Table 7 presents the average ranks before (column 2) and after (column 3) using our proposed approach (denoted with **) and Vancsics et al's approach in [12] (denoted with *), as well as the difference between them (column 4). If the difference is negative, it indicates that the used approach has the potential to improve.

Table 7. Average ranks comparison

			Diff.	Diff.
J = 38.51	J* = 23.58	J** = 21.83	J-J* = -14.93	J-J** = -16.68
B = 38.5	B* = 23.66	B** = 21.7	B-B* = -14.84	B-B** = -16.8
S = 38.51	S* = 23.77	S** = 21.96	S-S* = -14.74	S-S** = -16.55
DS = 149.03	DS* = 150.59	DS** = 136.67	DS-DS* = 1.56	DS-DS* = -12.36
D = 38.51	D* = 23.58	D** = 21.83	D-D* = -14.93	D-D** = -16.68
I = 38.5	I* = 23.66	I** = 21.7	I-I* = -14.84	I-I** = -16.8
K = 153.34	K* = 138.26	K** = 136.66	K-K* = -15.08	K-K** = -16.68
C = 38.54	C* = 20.76	C** = 17.87	C-C* = -17.78	C-C** = -20.67

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

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

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

It is worth mentioning that only using average ranks as an evaluation metric for SBFL effectiveness has its own set of drawbacks: (a) outlier average ranks could distort the overall information on the performance of any proposed approach. (b) it tells nothing about the distribution of the rank values and their changes before and after applying a proposed approach. Therefore, there is a more important category of evaluation than average ranks: improvements in the *Top-N ranks*, where the advantages are more obvious, as presented below.

6.2 Achieved improvements in the Top-N categories

According to [8] and [15], testers believe that examining the first five program elements in an SBFL ranking list is acceptable, with the first ten elements being the highest limit for inspection before the list is dismissed. Thus, the success of SBFL can also be measured by concentrating on these rank positions, which are collectively known as Top-N, as follows: (a) Top-N: When the rank of a faulty program element is less or equal to N. (b) Other: When the rank of a faulty program element is more than the highest N value used in the categorizations (it is 10 in our experiments).

Figure 2 shows the number of bugs in the Top-N categories for each approach. Here, improvement is defined as a decrease in the number of cases in the "Other" category and an increase in any of the Top-N categories.

It is evident that by relocating many bugs to higher-ranked categories, our proposed approach and Vancsics et al's approach improved all Top-N categories. However, our approach placed more bugs (i.e., **19**–**25** bugs) into one of the Top-N categories from the "Other" category (with rank > 10) compared to Vancsics et al's approach (i.e., **16**–**21** bugs). This is significant since it raises the possibility of finding a bug with our approach while it was not very probable without it. This kind of interesting improvements is also known as *enabling improvements* [4]. Table 8 presents the enabling improvements achieved by each approach.

It can be noted that each new formula achieves enabling improvements, the average enabling improvements was about **5%** of the total number of faults in the used dataset by our approach. In these cases the basic SBFL formulas ranked the faulty method in the other category, but our proposed approach managed to bring it forward into the Top-10 (or better) categories. Note that, the formulas B**, I**, and C** are the best in this aspect. Overall, each formula based on our proposed approach was able to achieve enabling improvements in the possible cases. It can be noted that Vancsics et al's approach improvements was about

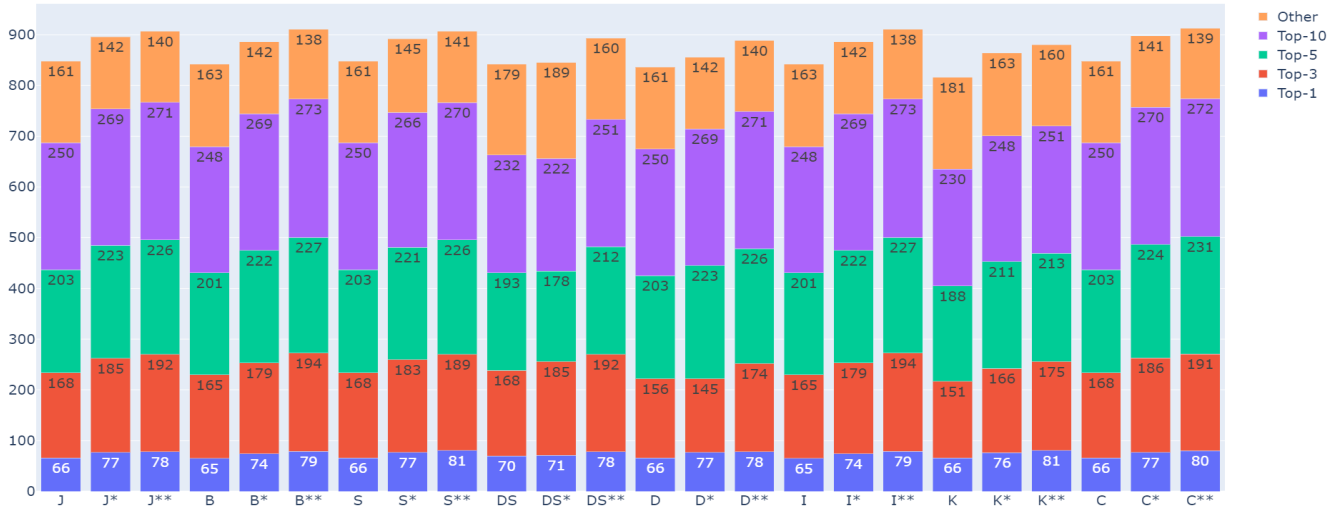

Fig. 2. Top-N categories

Table 8. Enabling improvements

	Rank > 10 (%)	Enab. impr. (%)	Enab. impr. (%)
J vs. J* vs. J**	161 (39.2%)	19 (4.6%)	21 (5.1%)
B vs. B* vs. B**	163 (39.7%)	21 (5.1%)	25 (6.0%)
S vs. S* vs. S**	161 (39.2%)	16 (3.9%)	20 (4.9%)
DS vs. DS* vs. DS**	179 (43.6%)	10 (2.4%)	19 (4.6%)
D vs. D* vs. D**	161 (39.2%)	19 (4.6%)	21 (5.1%)
I vs. I* vs. I**	163 (39.7%)	21 (5.1%)	25 (6.0%)
K vs. K* vs. K**	181 (44.0%)	18 (4.4%)	21 (5.1%)
C vs. C* vs. C**	161 (39.2%)	20 (4.9%)	22 (5.4%)

4% of the total number of faults in the used dataset with the formulas B*, I*, and C* as the best ones. Here, this improvement seems modest considering the fact that only an importance weight was used. Other, more complex weights may yield much more improvement which will be investigated in the future. Higher categories have significant improvements as well, with roughly **8–15** bugs moving to Top-1 by our approach compared to Vancsics et al's approach with **1–11** bugs, for example. Here also, our approach outperformed Vancsics et al's approach by moving more bugs to the Top-1 category.

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

7 Threats to validity

In software engineering, each experimental study has some threats to its validity. In this work, the following actions were considered to avoid or mitigate the threats of validity:

- Selection of evaluation metrics: to be certain that our findings and conclusions are correct, we selected well-known evaluation metrics (i.e., average ranks and Top-N categories) that have been utilized in prior studies too.
- Correctness of implementation: a code review was performed numerous times to guarantee that our experiment implementation was correct. Furthermore, we have executed our proposed strategy multiple times to ensure that it is properly implemented.
- Selection of subject programs: we used Defects4J as a benchmark dataset in our study. Therefore, our findings cannot be generalized to other Java programs. However, we believe that the programs of Defects4J are representative and contain real faults of varied types and complexity. Defects4J is also extensively utilized in other software fault localization research.
- Exclusion of faults: due to technical limits, we had to eliminate 27 faults from the Defects4J dataset (about 6% of the total number of faults). The question is whether or not other researchers working with the same dataset will be able to reproduce our results. Our findings were not influenced in any way by this exclusion and the excluded faults were scattered almost uniformly throughout the dataset, thus we believe that this threat is very low.
- Selection of SBFL formulas: we used a collection of well-known SBFL formulas in our experiment to evaluate the effectiveness of our proposed approach, which represents only a small percentage of the reported formulas in the literature. The results demonstrate that all of them have improved. However, we cannot guarantee that using other different formulas would yield

the same results. We used the formulas which are extensively used in other software fault localization research to limit the effect of this issue.

8 Conclusions

This paper presents the use of importance emphasis on the failing tests that execute the program element under consideration in SBFL. We rely on the intuition that if a code element gets executed in more failed test cases and appear in more calling contexts in such tests compared to other elements, it will be more suspicious and gets a higher rank position. This is achieved by multiplying the initial suspicion score, computed by underlying SBFL formulas, of each program method by an importance weight that represents the rate of executing a method in failed test cases combined with the so-called method calls frequency. The following are the primary characteristics of the proposed approach: (a) it can be used to any SBFL formula without changing the structure or notion of the formula. (b) it overcomes the problem of an unbalanced SBFL matrix since there are far more passing tests than failing tests, and many SBFL formulas treat passing and failing tests in the same way. The findings of this study's experiments reveal that relocating many bugs to the top Top-N rankings improved the average ranks for all formulas studied and surpassed previous approaches.

We would like to evaluate the effectiveness of our approach at different levels of granularity in the future, such as at the statement level. Incorporating other SBFL formulas into the study to determine which formulas produce the greatest results and classifying them into groups would be fascinating to investigate further. We would also like to use other expressions of importance weights and see how they affect SBFL efficacy.

References

1. Ajibode, A.A., Shu, T., Ding, Z.: Evolving suspiciousness metrics from hybrid data set for boosting a spectrum based fault localization. IEEE Access **8**, 198451–198467 (2020)
2. B. Le, T.D., Lo, D., Le Goues, C., Grunske, L.: A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. p. 177–188. ISSTA 2016, Association for Computing Machinery, New York, NY, USA (2016)
3. Bagheri, B., Rezaalipour, M., Vahidi-Asl, M.: An approach to generate effective fault localization methods for programs. In: International Conference on Fundamentals of Software Engineering. pp. 244–259 (2019)
4. Beszédes, A., Horváth, F., Di Penta, M., Gyimóthy, T.: Leveraging contextual information from function call chains to improve fault localization. In: IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 468–479 (2020)
5. Idrees Sarhan, Q., Vancsics, B., Beszedes, A.: Method calls frequency-based tie-breaking strategy for software fault localization. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 103–113 (2021). https://doi.org/10.1109/SCAM52516.2021.00021

6. Just, R., Jalali, D., Ernst, M.D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 437–440. ACM Press (2014)
7. Kim, J., Park, J., Lee, E.: A new hybrid algorithm for software fault localization. In: Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication. pp. 1–8 (2015)
8. Kochhar, P.S., Xia, X., Lo, D., Li, S.: Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. p. 165–176. ISSTA 2016, Association for Computing Machinery, New York, NY, USA (2016)
9. Neelofar: Spectrum-based Fault Localization Using Machine Learning (2017), https://findanexpert.unimelb.edu.au/scholarlywork/1475533-spectrum-based-fault-localization-using-machine-learning
10. Park, J., Kim, J., Lee, E.: Experimental Evaluation of Hybrid Algorithm in Spectrum based Fault Localization. International conference on Software Engineering Research and Practice (SERP) (2014)
11. Sarhan, Q.I., Beszedes, A.: A survey of challenges in spectrum-based software fault localization. IEEE Access **10**, 10618–10639 (2022). https://doi.org/10.1109/ACCESS.2022.3144079
12. Vancsics, B., Horvath, F., Szatmari, A., Beszedes, A.: Call frequency-based fault localization. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 365–376 (2021)
13. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The dstar method for effective software fault localization. IEEE Transactions on Reliability **63**(1), 290–308 (2014)
14. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A Survey on Software Fault Localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (aug 2016)
15. Xia, X., Bao, L., Lo, D., Li, S.: "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: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 267–278 (2016)
16. You, Y.S., Huang, C.Y., Peng, K.L., Hsu, C.J.: Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In: 2013 IEEE 37th Annual Computer Software and Applications Conference. pp. 180–189 (2013)
17. Zhang, M., Li, X., Zhang, L., Khurshid, S.: Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 261–272 (2017)
18. Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L.: An empirical study of fault localization families and their combinations. IEEE Transactions on Software Engineering **47**(2), 332–347 (2021)