Poster: Improving Spectrum Based Fault Localization For Python Programs Using Weighted Code Elements

Qusay Idrees Sarhan[1, 2] and Árpád Beszédes[1]
[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 this paper, we present an approach for improving Spectrum-Based Fault Localization (SBFL) by integrating static and dynamic information about code elements. This is achieved by giving more importance to code elements that include mathematical operators compared to other types of elements (e.g., declaration, selection, iteration, or function call) and appear in failed tests. The intuition is that these elements are more likely to have bugs than others. The proposed approach is applicable to any SBFL formula without requiring any modifications to their structures because the weighting is done on the ranking list and not on the formulas. The experimental results of a preliminary study show that our approach achieved a much better performance in terms of average ranking compared to the underlying SBFL formulas. It also improved the Top-N categories; it doubled the number of cases in which the faulty method became the top-ranked element, and in all cases the fault became part of Top-5 of the ranking list.**

***Index Terms*—Debugging, fault localization, spectrum-based fault localization, importance weight, suspiciousness score.**

I. 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, including death. As a result, various software fault localization techniques, such as spectrum-based fault localization (SBFL) [1], 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 [2], [3].

In SBFL, code elements are ranked from most to least suspicious based on their suspicion scores. In the basic approach, only the execution information about each element is used in calculating these scores. As a result, other sorts of information (such as the types of statements, the relationships between statements, how many times a statement is executed, etc.) are disregarded, which lowers SBFL's effectiveness. Therefore, many studies involved other types of information to improve its effectiveness.

In this paper, we improve the effectiveness of SBFL by involving static information about each code element into the SBFL process. We give more importance to code elements that include mathematical operators compared to other types of elements (e.g., declaration, selection, iteration, jump, function return, or function call). The intuition is that these elements are more error prone due to the computations they include which are critical parts of any algorithm. Other kinds of statements can generally be treated as "more simple" which means the probability of a mistake is lower [4]. The proposed approach is applicable to any SBFL formula without requiring any modifications to their structures because the score values are adjusted after the rankings are produced.

We performed a limited experimental study by implementing the method for Python programs, and used a subject system with seeded faults. The results show that our proposed approach achieved a better performance in both the average ranking positions and in Top-N measurements compared to the underlying SBFL formulas. Namely, the average ranks dropped to around 1.5 from 10-20, the number of highest rank positions doubled, while all faults have been ranked in Top-5 in the worst case.

II. Background on SBFL

To extract the spectra for the subject program, the execution of tests on program elements is first recorded as part of the SBFL process. The relationship between the tests and the program elements is represented by a two-dimensional matrix called a "program spectra". Its rows correspond to the program elements, while its columns show the tests.

In the matrix, a cell shows if a relevant test (column) covers a corresponding program element (row). The matrix also stores test results, whether passed or failed (in an extra row).

From the spectra, the following four fundamental statistical counts are made for each program element e: (a) ep: number of passed tests executed e; (b) ef: number of failed tests executed e; (c) np: number of passed tests not executed e; (d) nf: number of failed tests not executed e.

Then, an SBFL formula [5], e.g., Tarantula, Ochiai, or Barinel, can utilize these four fundamental statistics to calculate each element's suspicion score.

As an output, a ranking list based on the scores is created. The element with the highest ranking on the list is the one with the greatest likelihood of having a bug. Thus, SBFL helps developers to identify problematic elements in their programs.

III. Related Works

Based on the hypothesis that some failed tests may reveal more information than other failed tests, the authors of [6] used this information with SBFL formulas. Different weights for failed tests were therefore allocated for each formula in use, and these weights were subsequently employed with multi-coverage spectra.

Another interesting way is to add new information to existing SBFL formulas. The authors in [7] utilized the method calls frequency of the subject programs during the execution of failed tests to add new contextual information to the standard SBFL formulas. Here, the function call frequency was incorporated into the ef value in each formula. The results of their study demonstrated that employing new information from method calls into the underlying formulas can improve SBFL effectiveness.

In the work [8], the presented approach gives more importance to program elements that are executed by more failed test cases 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 an SBFL formula by this importance weight, which is the ratio of covering failing tests over all failing tests. The proposed approach can be applied to SBFL formulas without modifying their structures.

The approach proposed in [9] is similar to ours as it also employed statements types in the SBFL process. In the following, we summarize the main differences. (1) It is for C programs, while our approach targets programs written in Python; which is considered the most popular programming language nowadays [10], [11]. (2) Statements are classified into four groups (i.e., conditional, assignment, return, and other), while we are seeking for mathematical operations only. (3) The types of statements are extracted using a Python script, written by the authors, that searches for the language keywords (e.g., if, for, while, return, etc.), while we are using the functions provided by the official AST module. (4) The statements x=5 and x=x+y are considered as the same type of statements, i.e. assignment, while in our case, x=x+y is considered a statement that has a mathematical operator and x=5 is an initialization statement. (5) Weights are learnt using optimization methods such as simulated annealing, grid search, or brute force search based on training data, which we do not require. (6) There is no relationship between the test statistics (i.e., ef) extracted from the dynamic information and the elements types extracted from the static information, while we consider the element type when the element appears in failed test only.

IV. The Proposed Approach

Our proposed approach is depicted in Figure 1. Using the selected SBFL formulas on the program spectra, we calculate the suspicion scores of code elements (i.e., statements in our study). The output is the initial suspicion scores of statements. Then, we extract the Abstract Syntax Trees (AST) of each

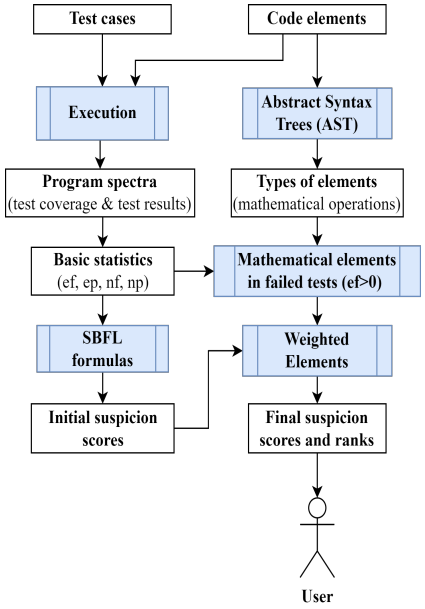

Fig. 1. The proposed approach

Python subject program using the ast[1] module. After that we walk through the generated AST and collect only the mathematical statements (i.e., statements that perform mathematical operations) and ignore other types of statements (e.g., statements that are used for declaration, selection, iteration, or function call). We then filter the obtained mathematical statements by taking only the mathematical statements that appeared in failed test cases (i.e., when $ef > 0$). Then, we give them more importance (weight) compared to others by using Equation 1, and finally we rank all the code elements. The $Max\_Initial\_Score$ is the maximum score in the initial ranking list. It is used to ensure that the mathematical statements will always be examined before the elements that have the highest scores in the initial list of suspicion scores that has been generated before applying our approach.

$$\text{Final\_Score} = Initial\_Score + Max\_Initial\_Score \quad (1)$$

This will improve the initial ranking list by giving more importance to the mathematical statements that are executed by failing tests and lowering the rank of other types of statements. A final, better ranking list is then generated for the user.

To show how our proposed approach works and how it achieves improvements, we will illustrate it with the basic statistics extracted from the spectra of the code example shown in Figure 2, which has the faulty statement S5 (it should be $avg = s/n$), as presented in Table I.

The Tarantula formula was applied to the extracted execution information to compute the suspicion score of each statement as presented in Table II. It can be seen that Tarantula cannot put the faulty statement S5 very near the top of the

[1]https://docs.python.org/3/library/ast.html

```
def do_avg(nums):
    S1: n=len(nums)
    S2: s = 0
    S3: for i in nums:
    S4:     s = s + i
    S5: avg = s / 2
    S6: return round(avg, 1)
```

```
import avg
def test_T1():
    nums = [0, 0, 0]
    assert avg.do_avg(nums) == 0.0
def test_T2():
    nums = [2, 2, 2]
    assert avg.do_avg(nums) == 2.0
def test_T3():
    nums = [2, 2, 3]
    assert avg.do_avg(nums) == 2.3
def test_T4():
    nums = [4, -2, -2]
    assert avg.do_avg(nums) == 0.0
```

Fig. 2. Running example – code and test cases

TABLE I
RUNNING EXAMPLE – SPECTRA AND BASIC STATISTICS

	T1	T2	T3	T4	ef	ep	nf	np
S1	1	1	1	1	2	2	0	0
S2	1	1	1	1	2	2	0	0
S3	1	1	1	1	2	2	0	0
S4	1	1	1	1	2	2	0	0
S5	1	1	1	1	2	2	0	0
S6	1	1	1	1	2	2	0	0
Results	0	1	1	0				

TABLE II
RUNNING EXAMPLE – SCORES AND RANKS

	Tarantula score	Rank	Tarantula *	Rank *
S1	0.5	3.5	0.5	4.5
S2	0.5	3.5	0.5	4.5
S3	0.5	3.5	0.5	4.5
S4	0.5	3.5	1.0	1.5
S5	0.5	3.5	1.0	1.5
S6	0.5	3.5	0.5	4.5

ranking list suggested by the formula (it is ranked 3.5 based on Equation 5). The reason, in this case, is that Tarantula assigned the same score to all the statements in our code example. However, after applying our proposed approach denoted with *, the faulty statement got a higher rank than before. Thus, it will be examined before most of the other statements.

V. Evaluation and Discussion

The Research Questions (RQs) we address in this study are the following:

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

Our proposed approach has been evaluated on 17 seeded single-fault versions of the "bottle.py"[2] micro-framework for Python web-based applications. Each seeded version has a bug in a statement that has a mathematical operator. As this concept paper focuses on this types of bugs, all the 17 bugs are mathematical. For future work, we plan to add other types of statements and use existing datasets instead of seeded programs to perform a more comprehensive and realistic evaluation.

"bottle.py" is a 168 KB project that has about 4.4 KLOC and about 350 tests. It has been selected in this study because it is a well-known Python framework that has 100 contributors and has been forked 1.4k times. The open-source tool "CharmFL" [12] was used to extract the statement-level granularity as a coverage type. Also, we compared our proposed approach to the well-known formulas presented in Equations (2-4) respectively; to measure its effectiveness.

$$\text{Tarantula} = \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf} + \frac{ep}{ep+np}} \tag{2}$$

$$\text{Ochiai} = \frac{ef}{\sqrt{(ef+nf)*(ef+ep)}} \tag{3}$$

$$\text{Barinel} = \frac{ef}{ef+ep} \tag{4}$$

The overall effect of our proposed strategy on SBFL efficacy is presented and discussed in this section. For this, we employ evaluation metrics that have previously been employed by other researchers in the literature [13], [14].

A. Achieved improvements in average ranks (RQ1)

The program elements with the same suspicion score are ranked using the average rank, such elements are called *tied elements* [15], by averaging their positions after they get sorted in descending order according to their scores. We use the average rank approach in Equation 5, where S denotes the tie's starting position and E denotes the tie's size.

$$\text{MID} = S + \left(\frac{\text{E - 1}}{2}\right) \tag{5}$$

Table III presents the average ranks before and after (columns 2 & 3) applying our proposed approach and the difference between their average ranks (column 4). If the difference is negative, it indicates that our proposed approach improved the baseline.

TABLE III
COMPARISON OF AVERAGE RANKS

	Before	After	Diff.
Tarantula	24.8	1.5	-23.3
Ochiai	10.3	1.3	-9.0
Barinel	24.8	1.5	-23.3

With all of the selected SBFL formulas, we can observe that our proposed approach improved the average rank; reduced by about 19 positions in overall. Considering the formulas that have lower average ranks after applying our proposed approach, Ochiai is the best one. This indicates that using an importance weight could have a positive impact and enhances the SBFL results.

[2]http://bottlepy.org/

B. Achieved improvements in the Top-N categories (RQ2)

According to [16] and [17], developers believe that examining the first five elements in the 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 a buggy program element's rank is N or less. (b) Other: If a buggy program element's rank is higher than the highest N value used for categorizations (it is 10 in our study).

Table IV presents the number of bugs in each of the Top-N categories for all the buggy versions of our subject program, before and after applying our proposed approach, as well as the differences between them. Here, improvement is defined as a decrease in the number of cases in the "Other" category and an increase in any of the Top-N categories.

TABLE IV
TOP-N CATEGORIES

	Top-1	Top-3	Top-5	Top-10	Other
Tarantula	4	7	9	11	6
Tarantula*	9	16	17	17	0
Diff.	5	9	8	6	-6
Ochiai	5	9	10	15	2
Ochiai*	12	17	17	17	0
Diff.	7	8	7	2	-2
Barinel	4	7	9	11	6
Barinel*	9	16	17	17	0
Diff.	5	9	8	6	-6

It can be noted that there were noticeable improvements in terms of the Top-N categories with positive results in all the categories. Also, we were successful in increasing the number of cases in which the faulty method was ranked first: 5–7 bugs moved to Top-1 category. Another interesting finding is that 2–6 bugs were moved from the "Other" category into one of the higher-ranked categories. In particular, no elements left beyond Top-5 in our case, and looking at Top-1, the number of identified elements were at least double than for the baseline.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented an approach to improve the effectiveness of SBFL by involving static information (i.e., types of code elements) into the SBFL process for Python programs. This way, we combine static and dynamic information about code elements. Presently, we considered one type of statements: mathematical statements, such statements were given more weight/importance compared to other types of statements. Based on the positive results for this approach, we believe that it could be explored more in future such as by experimenting with other statement types and other ways for the weighting.

In particular, we want to carry out the following studies:

- Extending the statements types to more fine-grained types and considering other features that could be extracted from the source code. Finding out the most frequent faulty elements [4] can be a good starting point to define more elaborate weighting strategies.
- Comparing the effectiveness of each statement type on SBFL and then combining the best statements types.
- Including other SBFL formulas and involving other benchmark datasets in the evaluation.

Our measurement data and the seeded program files are available at: https://bit.ly/41QerL2

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, aug 2016.

[2] Q. I. Sarhan and A. Beszedes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022.

[3] R. Abreu, "The bumpy road of taking automated debugging to industry," *CoRR*, vol. abs/2212.01237, 2022.

[4] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark and taxonomy of javascript bugs," *Software Testing, Verification and Reliability*, vol. 31, no. 4, p. e1751, 2021, e1751 stvr.1751.

[5] Neelofar, "Spectrum-based Fault Localization Using Machine Learning," 2017. [Online]. Available: https://findanexpert.unimelb.edu.au/scholarlywork/1475533-spectrum-based-fault-localization-using-machine-learning

[6] 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 *2013 IEEE 37th Annual Computer Software and Applications Conference*, 2013, pp. 180–189.

[7] B. Vancsics, F. Horvath, A. Szatmari, and A. Beszedes, "Call frequency-based fault localization," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 365–376.

[8] Q. I. Sarhan, "Enhancing spectrum based fault localization via emphasizing its formulas with importance weight," in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022, pp. 53–60.

[9] N. Neelofar, L. Naish, J. Lee, and K. Ramamohanarao, "Improving spectral-based fault localization using static analysis," *Software: Practice and Experience*, vol. 47, no. 11, pp. 1633–1655, 2017.

[10] "Tiobe index," https://www.tiobe.com/tiobe-index/, accessed: 06-03-2023.

[11] "Pypl index," https://pypl.github.io/PYPL.html, accessed: 06-03-2023.

[12] Q. I. Sarhan, A. Szatmari, R. Toth, and A. Beszedes, "Charmfl: A fault localization tool for python," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 114–119.

[13] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 502–514.

[14] A. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 468–479.

[15] Q. I. Sarhan, B. Vancsics, and A. Beszedes, "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)*, 2021, pp. 103–113.

[16] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176.

[17] 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 *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 267–278.