Experimental Evaluation of A New Ranking Formula for Spectrum based Fault Localization

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*—Spectrum-Based Fault Localization (SBFL) uses a mathematical formula to determine a suspicion score for each program element (such as a statement, method, or class) based on fundamental statistics (e.g., how many times each element is executed and not executed in passed and failed tests) taken from test coverage and results. Based on the calculated scores, program elements are then ordered from most suspicious to least suspicious. The elements with the highest scores are thought to be the most prone to error. The final ranking list of program elements aids developers in debugging when looking for the source of a fault in the program under test.**

In this paper, we present a new SBFL ranking formula that enhances a base formula by ranking code elements slightly higher than others that are executed by more failed tests and less passing ones. Its novelty is that it breaks ties between the elements that share the same suspicion score of the base formula. Experiments were conducted on six single-fault programs of the Defects4J dataset to evaluate the effectiveness of the proposed formula. The results show that our new formula when compared to three widely-studied SBFL formulas, achieved a better performance in terms of average ranking. It also achieved positive results in all of the Top-N categories and increased the number of cases where the faulty element became the top-ranked element by 13–23%.

***Index Terms*—Debugging, fault localization, spectrum-based fault localization, formulas, ranking list.**

I. Introduction

Software still has a long way to go before being flawless. Software faults may result in serious undesirable events, including the loss of life. So, during the past few decades, a variety of strategies for locating software faults have been presented, such as Spectrum-based fault localization (SBFL) [1]–[3]. According to SBFL, program spectra produced by running tests are used to determine the likelihood that each program element will be faulty. However, due to the problems it raises, SBFL is not yet utilized substantially [4]. One of such issues is that in SBFL program elements are ranked in order of their suspicion scores from the most suspicious to the least, and it is not guaranteed that the faulty element is easy to find in this list. Programmers evaluate each program element starting at the top of the ranking list to see whether it is faulty or not. The faulty element should be placed close to the top of the ranking with no shared suspicion scores with other elements, so that developers may easily and quickly identify it early on in the examination process.

SBFL algorithms are based on suspiciousness formulas that statistically compare the number of failing and passing test cases that cover a particular code element, and the number of failing and passing test cases not covering the element. There have been numerous formulas proposed in the literature, but the issue of ties remains with all of them. This is the situation where two elements are equally suspicious.

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

Experimental results of our study show that our proposed formula achieved a better performance in terms of average ranking compared to three widely-studied SBFL formulas. Buggy element rankings reduced by an average of ten positions. Also, it achieved positive improvements in the Top-N categories and in particular, increased the number of cases where the buggy method was the highest ranked element in the ranking list by 13–23%.

This paper's main contributions are as follows:

1) A new SBFL formula that improves the performance of SBFL in many cases, which is a good candidate for tie breaking in combination with other formulas as well.
2) The analysis of the impact of the new SBFL formula on the overall SBFL effectiveness is discussed.

While the Research Questions (RQs) are as follows:

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

The remaining sections of the paper are structured as follows. Section II concisely explains the SBFL and its core idea. Section III provides a summary of relevant works. Section IV introduces our novel approach of enhancing SBFL formulas. Section V provides an overview on the used subject programs, data collection, and the evaluation baselines. Section VI presents the experimental results of this study and provides some analysis about the effectiveness of our proposed formula. Finally, we offer our conclusions and potential directions for future research in Section VIII.

II. Background of SBFL

In software debugging, fault localization takes a lot of time. Therefore, automating it is crucial for software developers to easily find the location of a faulty element (e.g., statement, method, or class) in their programs during the debugging process. Several approaches were proposed to perform software fault localization process automatically [2]. Because SBFL is straightforward but effective—it simply relies on tests coverage and their results—we concentrate on it.

To obtain the spectra (i.e., tests coverage and tests results) for the subject program, tests execution on program elements is recorded. Program spectra represents the relationship between tests and program elements. It is represented as a matrix where its rows demonstrate the program elements and its columns represent the tests. An element of the matrix is 1, if it is covered by a test, otherwise it is 0. The matrix stores the test results as well, where 0 indicates a passed test and 1 indicates a failed test. For each program element e in the matrix, the following four statistical values are computed:

- ep: represents the number of passed test cases covering the program element e.
- ef: represents the number of failed test cases covering the program element e.
- np: represents the number of passed test cases not covering the program element e.
- nf: represents the number of failed test cases not covering the program element e.

Then, these four basic statistics can be used by an SBFL formula such as Tarantula [5], [6] in Equation 2 to produce a ranking list of program elements. Whichever element came in first on the list is the one that is most likely to have a fault. Therefore, SBFL can make it simpler for developers to identify the problematic code in the target program.

III. Related Works

There are many approaches proposed in the literature to enhance the performance of SBFL. One enhancing approach is to improve SBFL formulas to more accurately guide and pinpoint faults in the fault localization process. This is achieved by introducing new SBFL formulas, modifying currently used SBFL formulas, or combining exiting ones. The most important efforts that aim to improve SBFL by modifying its formulas are briefly presented in this section. Thus, we classify these previous approaches into several main categories as follows.

A. Modifying existing SBFL formulas

The authors in [5] improved the performance of the Tarantula formula by modifying some part of it to amplify its scores. However, the improved Tarantula does not make any improvements in the ranking. Also, the authors did not evaluate the improved Tarantula using well-known evaluation metrics. Based on the hypothesis that some failed test cases may yield more testing information than other failed test cases, the authors in [7] updated three well-known SBFL formulas. Different weights for failed test cases were therefore allocated and then used for each of the three formulas, thus; improving the performance.

B. Combining existing SBFL formulas

The authors in [8] proposed a method for generating a new SBFL formula tailored to a certain program by combining 40 different formulas. The proposed method extracts information from the program using mutation testing and then combines multiple formulas based on the gathered information using different voting systems to generate a new formula. Their findings demonstrate that the formula they produced is superior to a number of existing ones. The fact that different SBFL formulas can be combined into a single new formula is important to note. A hybrid formula, which combines the benefits of other existing formulas that have been employed in the combination, is the end product. A hybrid formula should therefore perform better than the others [9]

C. Adding new information to existing SBFL formulas

The authors in [10] utilized the method calls frequency of the subject programs, in call stack instances, during the execution of failed test cases to add new contextual information to the standard SBFL formulas. As a result, the frequency ef was substituted for the ef in each formula. Their test results demonstrated that the efficacy of SBFL might be increased by incorporating this new knowledge into the formulas already in use. This method can only be used with formulas that have the ef numerator, though. Additionally, it is regarded as heavy due to the requirement to trace each method call in failed test cases, whether the caller or callee.

Our proposed approach improves the SBFL performance by introducing a new SBFL formula. The main advantage of our proposed formula over others is that it ranks program elements that are executed in more failed test cases and less passed test cases higher that other elements, and at the same time it effectively breaks ties in many cases.

IV. The proposed SBFL formula

In this section, we present a new SBFL formula to enhance the effectiveness of SBFL and we show advantage over other SBFL formulas. Then, we present its effectiveness when applied on a motivation example.

A. The proposed SBFL formula

Our proposed formula is a sum of two parts: a base component and a tie-breaking enhancement part. At present, we use the simplest possible SBFL formula ef for the base part, but this can be replaced in theory by any other existing formula. The second component serves the purpose of modifying the base part by a slight amount, thus breaking ties with higher probability, and giving higher scores to elements with more failing tests and/or less passing tests:

$$\text{New Formula} = ef + \left(\frac{ef - nf}{ef + nf + ep}\right) \tag{1}$$

TABLE I
MOTIVATION EXAMPLE'S BASIC STATISTICS

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

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

B. An illustrative example from Defects4J

To show how our proposed approach works and how it achieves improvements, several bugs from the used Defects4J dataset were carefully examined. Bug 6 from the "Chart" project was one of the more interesting cases we looked into[1]. Thus, we will illustrate on the basic statistics extracted from the spectra of 26 methods (M1-M26), including the faulty method M21, as presented in Table I.

Tarantula formula was applied on the extracted execution information to compute the suspicion score of each method as presented in Table II. It can be seen that Tarantula formula cannot put the faulty method M21 near the top of the ranking list suggested by the formula (it is ranked 13 based on Equation 5). The reason is that Tarantula assigned higher scores to other 11 methods (i.e., M6, M7, M15-M17, M19, and M22-M26) that have been executed by less number of failed test cases (i.e, one failed test). As a result, these methods got higher ranks in the ranking list and will be examined before the actual faulty method M21.

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

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

TABLE II
MOTIVATION EXAMPLE – SCORES AND RANKS

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

V. EVALUATION

A. Subject programs

Here, we used the single faulty programs (i.e., 302 faults) of the dataset Defects4J v1.5.0 [11]. However, 5 faults were excluded due to instrumentation issues. Thus, the final dataset used contained a total of 297 faults. Table III presents the subject programs.

TABLE III
SUBJECT PROGRAMS

Project	Number of bugs	Size (KLOC)	Number of tests	Number of methods
Chart	16	96	2.2k	5.2k
Closure	113	91	7.9k	8.4k
Lang	47	22	2.3k	2.4k
Math	77	84	4.4k	6.4k
Mockito	25	11	1.3k	1.4k
Time	19	28	4.0k	3.6k
All	297	332	22.1k	27.4k

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

B. Granularity of data collection

The program coverage type used in this study was method-level granularity. It has several advantages [12]: it can handle large-scale programs (i.e., scales well to them), it provides more thorough contextual information about the program element under inquiry, and, in accordance with some research, it also gives users a more intelligible degree of abstraction [13], [14]. However, there is no theoretical barrier preventing the study of lower granularity levels.

C. Evaluation baselines

In this paper, several widely-studied SBFL formulas [6]: Tarantula, Ochiai, and Barinel; which are presented in Equations (2-4) respectively; were used as the baselines of comparison.

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

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

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

VI. Experimental Results and Discussion

This section presents and discusses the overall impact of the proposed formula on SBFL effectiveness. We use evaluation metrics that have been used also by other researchers in the literature for this purpose [15], [16].

A. Achieved improvements in the average ranks

Average rank, calculated using Equation 5, ranks program elements based on their shared (tied) suspicion scores by taking into account the average of their places after being sorted, descendingly.

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

where S denotes the tie's starting position and E denotes its size.

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

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

TABLE IV
Average rank of faulty elements of SBFL formulas compared to our proposed formula

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

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

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, the *Top-N* categories will also be evaluated as presented next.

B. Achieved improvements in the Top-N categories

In this study, we used Top-1, Top-3, Top-5, Top-10, and Other (e.g., rank $>$ 10) as a performance evaluation metric. The well-performing formula should put as much as possible bugs in the higher ranks categories [17]. Table V presents the number of bugs in the Top-N categories (cumulative) as well as their percentages for the entire dataset, of the baseline formulas and our proposed one, as well as the differences between them. There has been improvement if there are fewer bugs in the Other category and more bugs in any Top-N category.

TABLE V
Top-N categories

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

It is clear that by relocating many bugs to higher categories, our new formula improves all Top-N categories. 7–11 bugs were moved from the Other category with rank $>$ 10 into one of the higher Top-N categories. This is important as it gives a "new hope" that a bug will be discovered with our proposed formula while without it, it was not very likely. A significant number of improvements are also visible in higher

categories; for example, about 10 bugs were located in the Top-1 category. Note that the percentages of bugs in each category for each formula were computed based on the number of faults in Defect4J. While the difference percentages were computed based on the number of faults before applying our proposed formula.

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

VII. Implications and Future Plans

We presented the evaluation of a new SBFL formula. According to the findings of our preliminary research, the proposed formula merits further investigation. In addition, we plan to do the following research in the future:

- Extending the benchmark to the new Defects4J programs (i.e., version 2.0), to programs written in other programming languages, etc.
- Involving other existing SBFL formulas in the evaluation.
- Trying the enhancement component of the proposed formula together with other base formulas instead of only using ef.
- Investigating in more detail the effect of our formula, for instance statistics about how many ties are broken, how many times did ep helped, and so on.

VIII. Conclusions

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

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

IX. Acknowledgements

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.

References

[1] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A Literature Review," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, sep 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2659118.2659125

[2] 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. [Online]. Available: http://ieeexplore.ieee.org/document/7390282/

[3] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," pp. 1–46, jul 2016. [Online]. Available: http://arxiv.org/abs/1607.04347

[4] 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.

[5] X. Liang, L. Mao, and M. Huang, "Research on improved the tarantula spectrum fault localization algorithm," in *Proceedings of 2nd International Conference on Information Technology and Electronic Commerce*, 2014, pp. 60–63.

[6] 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

[7] 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.

[8] B. Bagheri, M. Rezaalipour, and M. Vahidi-Asl, "An approach to generate effective fault localization methods for programs," in *International Conference on Fundamentals of Software Engineering*, 2019, pp. 244–259.

[9] J. Kim, J. Park, and E. Lee, "A new hybrid algorithm for software fault localization," in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, 2015, pp. 1–8.

[10] 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.

[11] "A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research," https://github.com/rjust/defects4j/tree/v1.5.0, accessed: 2021-07-01.

[12] G. Shu, B. Sun, A. Podgurski, and F. Cao, "Mfl: Method-level fault localization with causal inference," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 124–133.

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

[14] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," 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. 177–188. [Online]. Available: https://doi.org/10.1145/2931037.2931049

[15] 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.

[16] 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.

[17] 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. [Online]. Available: https://doi.org/10.1145/2931037.2931051