Enhancing Spectrum-Based Fault Localization Using Contextual Knowledge

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

Attila Szatmári
Supervisor: Dr. Árpád Beszédes

Doctoral School of Computer Science
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged
2025

1 Introduction

Software testing is a crucial part of any software project. Mature software programs have hundreds of test cases and high code coverage. Higher coverage indicates better quality, meaning that coverage is an indicator of how well the code is tested. Despite high coverage and a large number of tests, no program is free of bugs. This is not surprising since humans inevitably make mistakes when coding.

When code is not working properly, developers need to debug it, which means finding the exact location of the bug and fixing it. The former is often the most difficult part of the process, depending on the maturity and complexity of the code Fault localization is a technique that automates the location part of debugging, making the developer's job easier. One of these techniques is Spectrum-Based Fault Localization (SBFL). SBFL builds on coverage information and test results, combining them to provide suspiciousness scores for each program element. A suspiciousness score indicates how likely an element is to contain the fault. SBFL uses coverage data to compute suspiciousness scores for program. To assist developers, elements are ranked by their scores, helping in debugging. Consider the program components $E = \{e_1, e_2, \ldots, e_m\}$ and the set of test cases $T = \{t_1, t_2, \ldots, t_n\}$. The program spectrum is represented by a binary matrix $S = (s_{ij}) \in \{0, 1\}^{m \times n}$. Here, each element s_{ij} in the matrix reflects whether the program element e_j is executed when running the test case t_i, indicating its coverage. Specifically, $s_{i,j}$ equals 1 if the element is executed; otherwise, it is 0 when it is not covered. Consequently, considering the outcome of each test case, a binary vector $R = \{r_1, r_2, \ldots, r_n\}$ is developed. The vector holds a 0 if a test case is successful and a 1 if it encounters a failure.

$$S = \begin{array}{c} \\ t_1 \\ t_2 \\ \vdots \\ t_m \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_n \end{array} \\ \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,n} \end{pmatrix} \end{array} \qquad R = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{pmatrix}$$

Using program spectra, for each program element e, the following statistical numbers, called the spectrum metrics, are then computed:

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

With these statistics, we can calculate the suspiciousness scores for each statement. SBFL algorithms generate a prioritized list of potentially problematic code elements to help developers identify the actual bug during the debugging process. Currently, SBFL relies only on program spectra, excluding other data. Contextual information suggests that some code elements, bug traits, and fixes are more prone to problems. Using patterns [14], machine

learning models [15], code structure [12, 29], etc. that reflect context, SBFL performance can be improved.

This Ph.D. thesis presents ways to leverage contextual information to make SBFL algorithms more accurate. While working on this project, I began exploring different aspects of software that could improve the efficiency of SBFL algorithms. The results were validated on various datasets and programming languages.

The dissertation is divided into four main sections. The first thesis point outlines an empirical study on how different types of bug fixes impact SBFL's efficiency in JavaScript. It discusses which statement types SBFL can more easily localize using three distinct formulas and provides a quantitative analysis of these cases. The second thesis point details how to identify the optimal properties and levels of tests for efficient fault localization. This section presents a new approach in which fault localization is based on the structure of tests, allowing for more accurate localization of faulty elements. In addition, it shows the percentage of unit and unit-like tests needed to localize SBFL accurately. The third thesis point examines how SBFL algorithms respond to predicate-related errors and suggests a new method of reorganizing the list according to the importance of the statements. The thesis point also presents a quantitative and qualitative analysis of these cases. To evaluate the effectiveness of this approach in helping developers identify faults, we conducted a user study. Finally, the last thesis point examines the essential characteristics of SBFL tools that are necessary for them to be widely usable and user-friendly. The thesis point also examines how SBFL implementations address the zero-division issue and provides a quantitative analysis to determine the optimal solution.

1.1 Challenges

This thesis addresses the challenges of integrating contextual information and implementing spectrum-based fault localization algorithms, offering solutions to these issues. The thesis addresses the following challenges:

- **Challenge 1: Effects of statement types on SBFL.** Although various methods aim to assist developers by automating debugging processes, state-of-the-art techniques often fall short. This is because they do not consider that some statement types are easier to localize than others.

- **Challenge 2: Finding suitable tests for efficient SBFL.** Efficient fault localization in SBFL depends heavily on the quality and composition of the test suite. The challenge lies in identifying which test properties and levels contribute most to SBFL's efficiency and balancing the composition of the test suite to optimize fault localization results.

- **Challenge 3: User-centric SBFL to help fault comprehension.** A key challenge to the widespread acceptance of SBFL techniques is their sole focus on determining the location of the fault. While identifying a fault's location is important, effective debugging also requires developers to understand its nature and context. Bridging the gap between fault localization and comprehension is essential to improving the debugging experience and making SBFL more practical.

- **Challenge 4: Make SBFL user-friendly and easy to implement.** For SBFL to be widely adopted, it must be user-friendly, easy to implement and integrate, and effi-

cient in addressing the needs of developers. One key challenge is addressing technical issues, such as the division-by-zero problem that can arise in certain SBFL formulas. Robust handling of these issues is necessary to ensure accurate results. In addition, it is crucial to identify which features and metrics are the most useful to developers. Finally, integrating user knowledge through interactive SBFL remains a challenge.

2 Bug-Fix Analysis in Spectrum-based Fault Localization

This thesis point examines how different statement types impact accurate fault localization. By analyzing the bug fixes and suspiciousness rankings of faulty statements, we can measure the correlation between the SBFL algorithm's accuracy and the types of statements. We evaluated the performance of the SBFL algorithm using the JavaScript bug benchmark, BugsJS. Our findings reveal notable differences in SBFL algorithm efficiency for bugs involving if-related and sequence-related changes compared to other bug fixes. Additionally, there is significant variance in accurate fault localization among if-related error subtypes. Some subtypes are easier for SBFL algorithms to identify, and certain groups are more efficiently detected by specific algorithms.

The main contributions of this thesis point are as follows.

1. We refined our earlier categorization of bug fixes using a more detailed level of bugs produced in BugsJS.
2. We investigated how low-level bug-fix types relate to fault localization effectiveness in terms of the SBFL ranking.
3. We further analyzed the if-related and sequence-related bug-fix types.

2.1 Findings

We demonstrated that IF (predicate-related) bug fix types are significantly more effective than other high-level types [30]. We were interested in whether any of the IF subcategories (change of condition, added new branch, etc.) affect the efficiency of fault localization algorithms more than others. We used Fisher's exact test to determine if any IF categories significantly outperform or underperform the others. Let H_0 be the case that the fault localization algorithms perform similarly to any labels in the IF category. In addition to this, let H_1 be that there is a significant difference. The significance level was chosen to be $\alpha = 0.05$ and if the p-value given by Fisher's exact test is less than or equal to α then we reject the null hypothesis (H_0).

We counted the bugs where the rank fell into a non-overlapping interval of [1], (1, 3], (3, 5], (5, 10] or (10, . . .]. Looking at Figure 1, we can predict that Tarantula performs worse on IF-RBR (branch removed in fix) and IF-ABR (additional branch in fix) in the (1, 3] interval and on IF-RBR in the (3, 5] interval. Table 1 shows that IF-ABR in the top 3 and IF-RBR in the top 5 are significantly different; therefore, in these two cases, we reject the null hypothesis. Therefore, IF-RBR and IF-ABR perform worse in terms of the SBFL algorithm's efficiency. Furthermore, bug fixes labeled IF-RMV (removing the predicate in

the fix) are more likely to be in the top three categories, i.e., have a rank of one, two, or three. Also, bugs whose fixes are labeled IF-CC (predicate condition change in the fix) are easier for Tarantula to find and are more likely to be put in the top 5 or top 10.

Table 1: *Significance In Top-n Within The If Category Based On Fisher Exact Test*

Name	top-1	top-3	top-5	top-10	other
IF-ABR	0.7583	**0.0182**	0.7083	0.6033	0.6033
IF-APC	0.8366	1.0000	0.3099	0.6984	0.6984
IF-APCJ	0.4501	0.2716	1.0000	0.1445	0.1445
IF-APTC	1.0000	0.1327	0.3299	1.0000	1.0000
IF-CC	0.6542	0.6666	0.5909	0.4156	0.4156
IF-RBR	1.0000	0.1399	**0.0194**	0.2427	0.2427
IF-RMV	0.4629	0.2369	0.1342	1.0000	1.0000

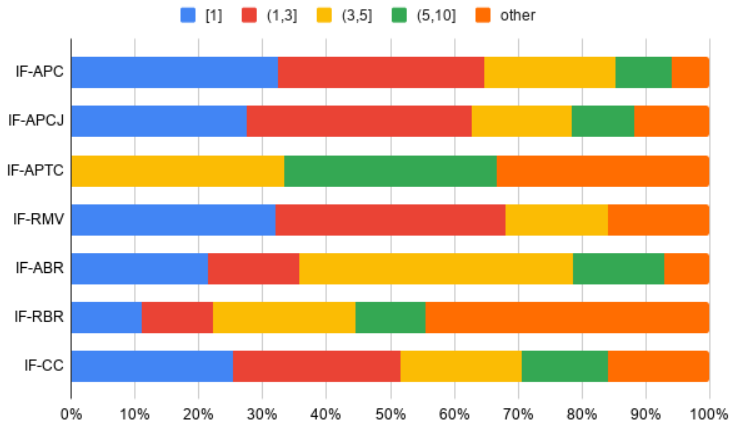

Figure 1: *Ochiai Ranks in IF labels*

Similarly, we will present an additional analysis of labels with SQ (changes in a sequence of operations) prefixes. Table 2 shows the p values from running Fisher's exact test. As we can see, none of them are below the significance level ($\alpha = 0.005$). However, Figure 2 shows that only SQ-AMO (Addition of operations to the sequence of method calls to an object) and SQ-AFO (Addition of operations in an operation sequence of field settings) were found in the top 1, meaning they are easier for Ochiai to find.

Table 2: *Significance In Top-n Within The SQ Category Based On Fisher Exact Test*

Name	top-1	top-3	top-5	top-10	other
SQ-AFO	0.1250	0.7241	0.4713	0.3774	0.3774
SQ-AMO	0.2311	1.0000	1.0000	1.0000	1.0000
SQ-AROB	0.2713	0.4670	1.0000	1.0000	1.0000
SQ-RFO	1.0000	1.0000	0.6431	1.0000	1.0000
SQ-RMO	1.0000	0.4882	0.5417	1.0000	1.0000

The results show that Tarantula identified an equal number of bugs labeled with AS-CE (Change of assignment expression) and AS. Tarantula identified the IF-APCJ (Addition of

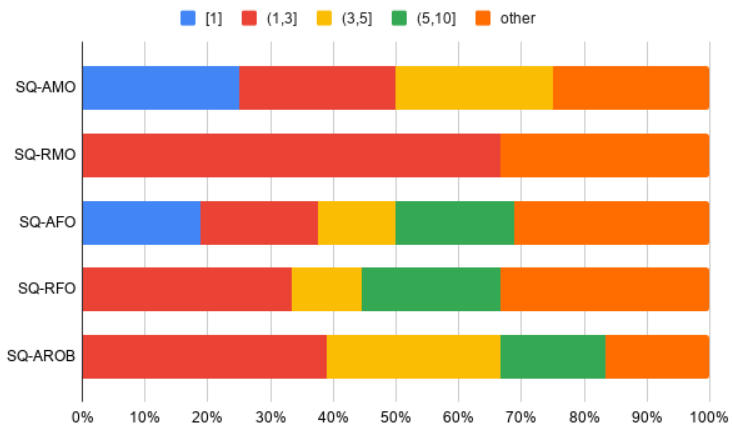

Figure 2: *Ochiai Ranks in SQ labels*

precondition check with jump) label 32 times in the ESLint project, while DStar and Ochiai identified it 31 times. Tarantula discovered bugs with MD-ADD (Method addition) bug fixes four times, compared to three times by Ochiai and DStar. Tarantula detected fewer bugs (88) with the IF-CC label than the other SBFL algorithms (89). SBFL algorithms rank three bug-fix types significantly better and four significantly worse. SQ-related changes are harder to detect with SBFL. IF-related changes differ: faults requiring (else) branch modifications are difficult to localize, while IF-CC and IF-RMV faults rank higher in the top five and top 10.

2.2 Conclusion

We analyzed the relationship between the three most popular SBFL algorithms (Tarantula, Ochiai and DStar) and the bug fix types. This chapter addresses Challenge 1 from Section 1.1, which focuses on identifying specific statements that can enhance the accuracy of fault localization algorithms. Our findings indicate that bugs in predicate-related statements are more easily localized by SBFL. Additionally, buggy branches within conditional statements appeared in the top-5 rankings more frequently than other statement types. This insight can serve as valuable contextual information for improving fault localization. This chapter's findings were published in [27] and [30].

3 Leveraging Test Levels and Properties to Enhance Spectrum-Based Fault Localization

This thesis point discusses identifying the most appropriate test level for efficient SBFL. The goal is to determine which types of tests and what ratio are needed for efficient fault localization. Sometimes, fault localization at the method level may not be the best technique for detecting faults. One such scenario occurs when a software program's test suite contains only unit tests, which simplifies fault detection. In this case, the faulty method would be identified by a single failed test that calls it directly. However, the cost of creating and maintaining a unit-only test suite could easily outweigh the benefits. This would

require extensive object mocking, since few methods operate in complete isolation.

Developers often do not strictly adhere to the ISTQB rules when writing unit tests, leaving the test suite with only a small fraction of tests that meet the strict definition of unit tests. This creates the challenge of recognizing tests that meet the unit test criteria and distinguishing them from similar tests that do not. Many of these ”unit-like” tests are helpful in fault localization because, without fully adhering to the formal definitions, they provide the benefits of unit testing, such as isolated coverage of specific functionality. To address this challenge, we introduce Pure Unit Tests (PUTs), a precise criterion for identifying unit tests in the context of fault localization. In addition, we propose three other heuristics: The Single Method Chain Test (SMC), the Limited Method Chain Test (LMC), and the Short Multichain Test (SMT) to evaluate how much of the test suite can be considered Unit-Like Tests (ULTs).

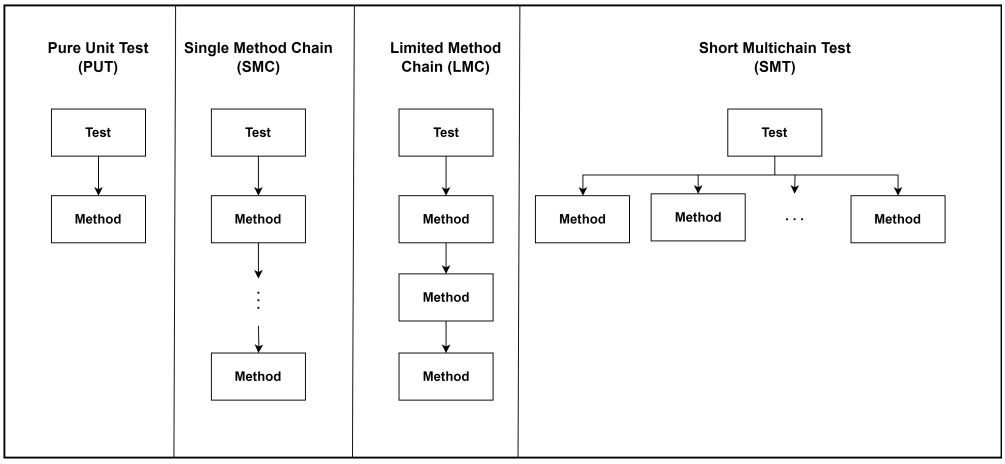

Figure 3: *Pure Unit Test (PUT) and Unit-Like Test (ULT) heuristics*

The main contributions to this thesis point are the following.

1. We developed heuristics to detect pure and unit-like tests.
2. We determined the ideal ratio of these tests for effective SBFL performance.
3. We evaluated the most effective weights of these tests to optimize the Call Frequency-based FL algorithm.
4. Lastly, we designed a new algorithm from these insights and evaluated it on Defects4J.

3.1 Optimal Proportion of Pure Unit Tests for Fault Localization Effectiveness

We analyzed the PUT percentage distributions in disjoint top N rank categories to see how much of the relevant test cases need to be pure unit tests so that SBFL provides accurate localization. To consistently place buggy methods in the top 3, at least 20% of the relevant tests must be PUTs. For example, if a method has five relevant test cases, at least one of them should be a PUT to maximize the likelihood that SBFL will place it in the top three.

If this threshold is not met, developers may need to write additional PUTs to improve fault localization accuracy.

3.2 Assessing the Impact of Relaxed Unit Tests on Fault Localization Efficiency

Although a high proportion of the PUT in the test suite indicates accurate SBFL, it is very strict and tests are rarely written to follow it exactly, especially when unit testing methods require extensive object mocking. ULTs play a crucial role in enhancing the efficiency of SBFL, particularly when the percentage of ULTs in relevant cases is sufficiently high ($\geq 50\%$ for SMC and LMC, and $\geq 60\%$ for SMT). In such cases, SBFL will always rank buggy methods in the top 3.

3.3 Optimal Weights for Tests in Call Frequency-Based Fault Localization

In our previous papers [28, 29], we introduced the Call Frequency-based Fault Localization method. However, this method ignores the scope of the test while increasing the efficacy of SBFL. To improve the fault localization ability of FL techniques, we propose identifying unit- and unit-like tests and evaluating the method's frequency with test-type weights. Initially, we use the following weights, which we then gradually increase. 1. Failing PUT = 100, 2. Failing ULT = 10, 3. Passing PUT = 10, 4. Passing ULT = 1. The results show that assigning weights to method call frequencies improves the algorithm's effectiveness.

While assigning weights can improve the Call Frequency-based FL algorithm, it is unclear whether adjusting the weights further can yield better results. To investigate this, we gradually increased the weights of failing PUTs (F-PUTs), ULTs (F-ULTs), and passing PUTs (P-PUTs). The weights we used in this study for {F-PUT, F-ULT, P-PUT, P-ULT} are the following. 1. {1000, 100, 100, 1} 2. {10000, 1000, 1000, 1} 3. {100000, 10000, 10000, 1}. Increasing weights with relatively low values produces better results in the call frequency-based FL algorithm, however, the differences between the weights are not notable.

3.4 Using Test Call Structures to Enhance The Accuracy of Spectrum-Based Fault Localization

Lastly, we created a new SBFL algorithm that uses the size of the call chains for more accurate fault localization by combining insights from previous results. Overall, the new Test Call-Structure-based fault localization algorithm produced better ranks for more than 130 bugs than the hit-based counterpart. Furthermore, it produced lower average ranks, that is, better places in the suspiciousness list, for 75% of the projects.

3.5 Conclusion

We investigated how different heuristics for unit tests correlate with the accuracy of SBFL. Therefore, we proposed heuristics to approximate unit tests based on their method call

chains. These are the Pure Unit Test (PUT), the Single Method Chain (SMC), the Limited Method Chain (LMC), and the Short Multichain Test (SMT). To examine the proportion of such tests in the relevant tests of defects (tests covering the same methods as the analyzed tests), we conducted a study on Defects4J. The results showed that including more than 20% PUTs in the test cases significantly improves the accuracy of the SBFL, placing the faulty methods in the top 3 of the suspicion rankings. However, if one of the failed tests is a PUT, then identifying the faulty method becomes easy even without using SBFL. This chapter addresses Challenge 2 from Section 1.1, which focuses on identifying the suitable tests for Fault Localization. Our findings indicate that having a substantial number of unit tests will help SBFL place buggy methods in the top-3 rankings. Our findings suggest that tests that have the smallest and fewest call chains contribute to more accurate Fault Localization. These insights can serve as valuable contextual information for improving fault localization. This chapter's findings were published in [24], [25] and [26].

4 A New Method for Better Localization of Predicate-Related Faults Using Spectrum-Based Fault Localization

This thesis point discusses how focusing on specific statement types, particularly predicate statements, can improve the accuracy of SBFL algorithms. We apply the *Predicate Promotion* technique, whereby we assign ranks to a ranked list of suspicious elements by promoting predicate statements selectively when their dependent code blocks appear at higher ranks than the predicate itself. The ranking of other statements remains intact.

The intuition behind the approach is as follows: When programmers investigate code to find the exact locations of errors, they need to understand the code as a whole, rather than as individual statements. The smallest unit of context for a statement is its syntactic block. If that block is embedded in a predicate, such as a selection or a loop, then the predicate is essential for understanding the program logic. We argue that comprehending program logic and thus localizing the fault is usually more efficient when the programmer is first offered the predicate and then its dependent block in the SBFL's rank list.

The main contributions of this thesis point are as follows.

1. We examined the frequency of predicate statements in two datasets (Java and Python).
2. We evaluated how the *Predicate Promotion* algorithm impacts the ranking of SBFL.
3. We carried out a user study to determine if our method helps in fault understanding.

4.1 Prevalence of Predicate Statements in Bug Fixes

Tables 3 and 4 show the distribution of the statement types and the distribution of the type of changed statements (i.e., faulty). We distinguish between the following categories: 'Predicate' refers to *if, else if, for, while* statements, 'Body' refers to statements that are in a dependent block of some Predicate, while 'Other' are all the remaining statements, e.g. import, expression, assignment statements, etc. The difference in the ratios of predicate statements in the two measurements is clearly visible: predicates and predicate bodies make up more than half of the modified code, as opposed to only 2% (Table 3) in the

code base in general. Interestingly, there are more predicate statements in the Defects4J projects (19%), but a high portion of the changeset is still predicate-related (Table 4).

Table 3: *Distribution of statement categories in BugsInPy*

	Number of statements	Percentage	Number of changed statements	Percentage
Predicate	42,944	1%	653	27%
Body	41,284	1%	636	26%
Other	4,223,986	98%	1125	47%

Table 4: *Distribution of statement categories in Defects4J*

	Number of statements	Percentage	Number of changed statements	Percentage
Predicate	2,931,820	19%	614	27%
Body	4,604,227	29%	800	35%
Other	8,152,450	52%	899	39%

4.2 Quantitative and Qualitative Evaluations

Our *Predicate Promotion* algorithm enhances SBFL by prioritizing predicate statements in ranked fault lists. In BugsInPy, the number of predicate statements increased notably in the top-1 (from 2 to 15), top-3 (from 11 to 45), and top-5 (from 14 to 52) ranks. Defects4J also showed improvements, particularly at the top-1 rank (increasing from 1 to 43). On average, the rank increased by 29.36 positions in BugsInPy and by 70.24 positions in Defects4J, demonstrating the robustness and effectiveness of the method in improving fault localization precision.

To understand how our technique could benefit or hinder fault localization success, we conducted a qualitative analysis of a sample of bugs in the benchmark. We manually examined 31 BugsInPy bugs in detail. These were cases in which predicate promotion placed or moved elements outside the top 10 (24 and 7, respectively). Similarly, we examined 42 Defects4J bugs: 29 moved into the top-10 section, and 13 moved out of it. After reviewing every case, we identified clear patterns explaining when the algorithm succeeds by moving faulty statements to better ranks and when it fails by pushing them lower.

4.3 User Study for Fault Comprehension

The goal of this study was to determine whether participants could identify bugs more quickly and accurately while gaining a better understanding of the problem. We evaluated the time each participant took to complete their tasks and then calculated the average times for both the predicate-promoting algorithm and the original algorithm. We also analyzed statements that participants chose that might be faulty and used the changeset to verify the correctness of their responses. Lastly, we examined their explanations and bug resolutions to determine whether either algorithm facilitated deeper insight into the problem. Our findings suggest that the modified algorithm provides developers with a clearer perspective on the problem. Participants using the predicate-promoting algorithm

demonstrated a better understanding of the fault and provided more detailed explanations. However, the differences in the time taken to locate the bug and the number of successfully identified faults were not substantial.

4.4 Conclusion

Our approach, Predicate Promotion, is to rearrange the original ranking list produced by an SBFL formula by assigning predicate statements higher ranks than any of their dependent code blocks. Our analysis showed that the *Predicate Promoting* algorithm increases the number of faulty statements in the top 10 compared to the original in both Java and Python. We carried out a qualitative assessment of the bugs whose statements were shifted into the top 10 and those that were shifted out of the top 10. A user study involving 15 participants was conducted to evaluate the benefits of the predicate-promoting algorithm in debugging. This chapter addresses Challenges 1 and 3 of Section 1.1, which focus on identifying specific statements that can enhance the accuracy of fault localization algorithms and creating a user-centric SBFL to help fault comprehension. Our findings indicate that bugs in predicate-related statements are more easily localized by SBFL. Additionally, buggy branches within conditional statements appeared in the top-5 rankings more frequently than other statement types. This insight can serve as valuable contextual information to improve fault localization. Our method aids fault comprehension by prioritizing the predicate statement ahead of its dependent block, encouraging developers to examine the logic and trace the fault to the most suspicious element. The results of this chapter have been submitted to the IEEE Access Journal, however the paper is still under review [20].

5 Aiding the Development and Integration of Spectrum-Based Fault Localization Techniques

This thesis point discusses the challenges of implementing SBFL algorithms. While this topic is more technical than others, we believe identifying and providing solutions to these challenges will help future researchers and developers create more reliable, user-friendly, and accurate fault localization tools.

Despite the immense literature on SBFL, it is still not widely used in the industry due to several challenges that need to be addressed [11, 18]. There is a lack of SBFL tools to help developers debug programs written in popular languages like Python. Another major problem is that SBFL approaches typically calculate suspiciousness scores of program elements without consulting the user. This is considered one of the main issues that reduces their applicability. Meeting technical requirements can sometimes be difficult, and SBFL's statistical nature can occasionally mislead developers during the debugging process, leading to the technique's rejection by practitioners [16, 17]. Another problem SBFL tool developers face is how they handle the division by zero problem when they are implementing SBFL formulas.

The main contributions to this thesis point are the following.

1. We developed an SBFL tool integrated in PyCharm, a leading Python IDE, to address the challenge of the lack of Python-specific SBFL tools.

2. We implemented interactiveness in this tool to make SBFL more user-friendly.

3. We analyzed the most popular SBFL tools and why they are not well integrated and gave recommendations on how to create good and user-friendly SBFL tools that can be integrated into IDEs.

4. Finally, we analyzed the zero-division problem in SBFL formulas and how tools implement them, and made recommendations about what SBFL tool developers need to practice.

5.1 Bridging Usability and SBFL with The Interactive Design of CharmFL

CharmFL is an integrated SBFL tool in the PyCharm IDE for Python programs. To create the program's spectra, our tool measures code coverage. While this tool offers a user-friendly integrated plugin in the PyCharm IDE, it lacks certain features that would enhance its ease of use. Our tool provides the possibility to generate a static call graph for the investigated Python program. Whenever the user selects a program element, its callers and callees will be highlighted. This is useful for developers since they can visualize the context, thus helping them efficiently find the buggy element. When developers are debugging, they investigate the ”close” and ”far” contexts as well. One context might seem buggier than the other, i.e., we know the bug is not in the method, but the callee seems faulty, or the other way around.

5.2 Recommendations for Developing SBFL Tools

We gained experience developing an SBFL tool for the PyCharm IDE and learned what developers need to do for seamless integration. We also collected the most useful features to make SBFL tools widely usable. This part of the thesis is a step towards practically usable SBFL in IDEs. Out of the total of 29 expectations, only 18 were implemented (62%) in the most used SBFL tools. We compiled a list of concrete recommendations for future tool builders based on existing tools, user opinions, and our own experiences. It is without doubt that the underlying technology should also be further researched to reach the efficiency and precision required in a practical, everyday context, not only in a lab setting. However, our concerns regarding the implementation into IDEs should be equally considered. Hopefully, parallel development of the two can give birth to usable SBFL plug-ins in some of the popular IDEs in the future.

5.3 Mitigating the Division by Zero Problem in SBFL

Even the best fault localization techniques encounter difficulties when division by zero occurs. In this thesis point, we use mathematical analysis to learn more about the SBFL techniques published in the last three to five decades and how division by zero affects them. Based on our analysis, we propose categorizing the collected formulas into classes and providing reasonable, useful, context-dependent solutions to avoid division by zero for each problematic formula class. Additionally, we conducted an empirical study on the effects of division by zero on score calculation in SBFL formulas, using the Defects4J and

JerryScript datasets. Thus, we ensure that existing formulas function properly, even in corner cases such as division by zero.

From our analysis, three distinct categories emerge from the formulas: no division by zero occurs (25 formulas), limits exist (21), and limits are indeterminate (29). Also, 25 formulas flagged as safe during analysis showed no division by zero during the experiments. For 39 out of the remaining 50 formulas, division by zero was an issue in benchmark programs, both faulty and non-faulty elements, proving that underscoring our study has relevance. Finally, excluding those program elements from scoring ($ef + ep = 0$) decreased division by zero instances (average reduction: 59.84%, best case: 99.97%), though issues persisted. Adopting existing division modes over simply setting a score of zero (denoted as *naive*) proved advantageous, despite varying effectiveness across formulas. Tailoring the division mode to each formula resulted in 27.21% boost in top-5 ranks on average.

5.4 Conclusion

We created a tool called CharmFL used for SBFL-related experiments. We implemented features that enhance user interactivity which allows feedback on suspicious elements to improve fault localization rankings. Although SBFL holds potential for automated debugging, our assessment of current tools and user feedback reveals challenges that must be addressed for industry acceptance. Of 29 identified expectations for SBFL tool implementation in IDEs, only 62% were met, revealing a gap between research and industry needs. In addition, we investigated the problem of division by zero in SBFL formulas, analyzing 75 formulas and multiple division handling strategies. Our results suggest that eliminating zero division occurrences through execution-based exclusions reduces their frequency by 59.84%, while empirically selecting the best-performing division mode improves ranking effectiveness by a 27%. Based on our results, we propose guidelines to mitigate zero division problems and improve SBFL reliability. This chapter addresses Challenge 4 from Section 1.1, which focuses on making SBFL user-friendly and easy to integrate. We suggested essential features for SBFL tools to enhance user-friendliness and ease of integration for users, developers, and researchers. Additionally, we addressed the division by zero challenge in SBFL, aiming for more reliable algorithms and enhanced user experience. The results of this chapter have been published in [19, 22, 23, 31].

6 Contributions of The Thesis

In the **first thesis group**, the contributions are related to bug fix analysis in Spectrum-Based Fault Localization.

I/1. I manually labeled and categorized bug fixes in BugsJS.

I/2. I performed a correlation analysis of low-level bug fix types and precise bug localization in JavaScript programs.

In the **second thesis group**, the contributions are related to using test levels and properties for more accurate Spectrum-Based Fault Localization.

II/1. I provided the heuristics for identifying Pure Unit and Unit-like Tests

II/2. I analyzed the Defects4J benchmark and found the optimal amount of PUT and ULT tests for which SBFL can easily find errors.

II/3. I provided a new test-call-structure-based fault localization method.

II/4. I evaluated the weighted call-frequency and test call-structure based SBFL methods.

In the **third thesis group**, the contributions are related to the enhancing Spectrum-based Fault Localization's efficiency by rearranging the suspiciousness list based on statement types such as predicates.

III/1. I provided the idea of promoting predicates based on their dependent blocks.

III/2. I evaluated quantitative and qualitative experiments on Python and Java programs.

III/3. I participated in conducting a user study to assess how the new method aids in fault comprehension. However, evaluating the results of the user study was my contribution.

In the **fourth thesis group**, the contributions are related to the challenges in implementing Spectrum-Based Fault Localization algorithms and integrating them into IDEs.

IV/1. I developed the bug localization tool and its interactivity feature to support SBFL for Python programs.

IV/2. I performed a literature review of SBFL tools, their features, and how they are integrated into IDEs.

IV/3. I made recommendations based on developers' and my own experience on how to create reliable and user-friendly SBFL tools.

IV/4. I collected and implemented 75 SBFL formulas used in the literature, and made an empirical evaluation of how different partitioning modes and proposed ideas change the accuracy of SBFL. I participated in the mathematical classification of these formulas.

The author states that while the thesis results are primarily his own work, the pronoun ***we*** is used instead of ***I*** to recognize the input of co-authors in the papers forming the basis of this thesis.

Table 5 summarizes the relationship between the thesis points and the corresponding publications.

Table 5: *Correspondence between the thesis points and my publications.*

Publication	Thesis point												
	I/1	I/2	II/1	II/2	II/3	II/4	III/1	III/2	III/3	IV/1	IV/2	IV/3	IV/4
[30]	•												
[27]		•											
[25]			•	•									
[24]					•								
[26]						•							
[20]							•	•	•				
[19]										•			
[22]										•			
[23]											•	•	
[31]													•

The author's publications on the subjects of the thesis

Journal publications

[20] **Szatmári, Attila**, Balogh, Gergő, and Beszédes, Árpád. Towards Better Localization of Predicate-Related Faults in Spectrum-Based Fault Localization. *Under review*, 2025

[29] Vancsics, Béla and Horváth, Ferenc and **Szatmári, Attila** and Beszédes, Árpád. Fault Localization Using Function Call Frequencies. *The Journal of Systems and Software*, VOL(193), 111429, 2022.

Papers in conference proceedings

[19] Sarhan, Qusay Idrees, **Szatmári, Attila**, Tóth, Rajmond, Beszédes, Árpád. CharmFL: A Fault Localization Tool for Python. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*, 114-119, 2021.

[22] **Szatmári, Attila**, Sarhan, Qusay Idrees and Beszédes, Árpád. Interactive Fault Localization for Python with CharmFL. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*,33-36, 2022.

[23] **Szatmári, Attila**, Sarhan, Qusay Idrees, Soha, Péter Attila, Balogh, Gergő and Beszédes, Árpád. On the Integration of Spectrum-Based Fault Localization Tools into IDEs. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE'24)*, 24-29, 2024.

[24] **Szatmári, Attila**. Harnessing Test Call Structures for Improved Fault Localization Effectiveness. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 623-628, 2025.

[25] **Szatmári, Attila**, Gergely, Tamás and Beszédes, Árpád. Influence of Pure and Unit-Like Tests on SBFL Effectiveness: An Empirical Study. In *Proceedings of the 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 392-399, 2025.

[26] **Szatmári, Attila**, Orban, Aondowase James and Gergely, Tamás. Weighted Call Frequency-based Fault Localization. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 582-586, 2025.

[27] **Szatmári, Attila**, Vancsics, Béla and Beszédes, Árpád. Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency?. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, 16-23, 2020.

[30] Vancsics, Béla, **Szatmári, Attila** and Beszédes, Árpád. Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'20)*, 308-319, 2020.

[28] Vancsics, Béla, Horváth, Ferenc, **Szatmári, Attila** and Beszédes, Árpád. Call Frequency-Based Fault Localization. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)*, 365-376, 2021.

[31] Vince, Dániel, **Szatmári, Attila**, Kiss, Ákos and Beszédes, Árpád. Division by Zero: Threats and Effects in Spectrum-Based Fault Localization Formulas. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS'22)*, 221-230, 2022.

Other not related publications of the author

[13] Horváth, Ferenc, Balogh, Gergő, **Szatmári, Attila**, Sarhan, Qusay Idrees, Vancsics, Béla and Beszédes, Árpád. Hands-On: Interacting with Interactive Fault Localization Tools. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*,61-63, 2022.

[21] **Szatmári, Attila**, Gergely, Tamás and Beszédes, Árpád. ISTQB-based Software Testing Education: Advantages and Challenges. In *Proceedings of the 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*,389-396, 2023.

Other References

[11] Higor A. de Souza, Marcos L. Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges, 2017.

[12] Higor A. de Souza, Danilo Mutti, Marcos L. Chaim, and Fabio Kon. Contextualizing spectrum-based fault localization. *Information and Software Technology*, 94:245 – 261, 2018.

[13] Ferenc Horváth, Gergő Balogh, Attila Szatmári, Qusay Idrees Sarhan, Béla Vancsics, and Árpád Beszédes. Hands-on: Interacting with interactive fault localization tools. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 61–63, November 2022.

[14] Sangwon Hyun, Jiyoung Song, Eunkyoung Jee, and Doo-Hwan Bae. Timed pattern-based analysis of collaboration failures in system-of-systems. *Journal of Systems and Software*, 198:111613, 2023.

[15] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 184–193, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.

[17] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 165–176, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum based software fault localization. *IEEE Access*, 10:10618–10639, 2022.

[19] Qusay Idrees Sarhan, Attila Szatmári, Rajmond Tóth, and Árpád Beszédes. CharmFL: A fault localization tool for Python. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*, pages 114–119, September 2021.

[20] Attila Szatmári, Gergő Balogh, and Árpád Beszédes. Towards better localization of predicate-related faults in spectrum-based fault localization. *UNDER REVIEW*, 1:20, 2025.

[21] Attila Szatmári, Tamás Gergely, and Árpád Beszédes. ISTQB-based software testing education: Advantages and challenges. In *Proceedings of the 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 389–396, April 2023.

[22] Attila Szatmári, Qusay Idrees Sarhan, and Árpád Beszédes. Interactive fault localization for Python with CharmFL. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 33–36, November 2022.

[23] Attila Szatmári, Qusay Idrees Sarhan, Péter Attila Soha, Gergő Balogh, and Árpád Beszédes. On the integration of spectrum-based fault localization tools into IDEs. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments (IDE'24)*, pages 24–29, April 2024.

[24] Attila Szatmári. Harnessing test call structures for improved fault localization effectiveness. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 623–628, 2025.

[25] Attila Szatmári, Tamás Gergely, and Árpád Beszédes. Influence of pure and unit-like tests on sbfl effectiveness: An empirical study. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 392–399, 2025.

[26] Attila Szatmári, Aondowase James Orban, and Tamás Gergely. Weighted call frequency-based fault localization. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 582–586, 2025.

[27] Attila Szatmári, Béla Vancsics, and Árpád Beszédes. Do bug-fix types affect spectrum-based fault localization algorithms' efficiency? In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 16–23, 2020.

[28] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)*, pages 365–376, March 2021.

[29] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Fault localization using function call frequencies. *The Journal of Systems and Software*, 193:111429, 2022.

[30] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in JavaScript programs. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'20)*, pages 308–319, February 2020.

[31] Dániel Vince, Attila Szatmári, Ákos Kiss, and Árpád Beszédes. Division by zero: Threats and effects in spectrum-based fault localization formulas. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS'22)*, pages 221–230, December 2022.

7 Összefoglalás

Szoftverprogramok meghibásodása során a fejlesztő felelőssége, hogy megtalálja a probléma forrását és kijavítsa azt. A spektrumalapú hibalokalizáció (SBFL) egy olyan módszer, amely segít a fejlesztőknek a hibák helyének meghatározásában a lefedettségi információk és a teszteredmények alapján. Együttesen ezek a program spektrumát alkotják. Annak ellenére, hogy az SBFL-t régóta kutatják, az iparban még mindig nem elterjedt technológia. Ennek az egyik oka, hogy az SBFL csak a program spektrumát veszi figyelembe, de nem használja fel a fejlesztés során keletkező információkat. Ezeket a kiegészítő információkat kontextus információnak nevezzük. A disszertáció célja, hogy azonosítsa ezeket a kontextus információkat, és azok használhatóságát az SBFL-ben.

Ebben a disszertációban különböző kontextus információkat használtunk fel az SBFL hatékonyságának és használhatóságának javítására. A disszertáció két részből áll. Az első rész a munka bevezető része, amelyben a doktori értekezés célja kerül bemutatásra, majd bemutatunk néhány kihívást az SBFL algoritmusokkal kapcsolatban. Ezután bevezetjük az alapvető definíciókat, amelyekre az olvasónak szüksége van a tézis pontok megértéséhez (1-2 fejezetek). A második rész (3-6 fejezetek) a tézis pontok bemutatása és értékelése, amely bemutatja a a *kód, a végrehajtás, a tesztelés és a felhasználó* szintű kontextus információk felhasználási módjait és ezáltal az SBFL hatékonyságának javítását.

A 3. fejezetben a hibajavítás típusai és az SBFL hatékonysága közötti kapcsolatot vizsgáltuk. Az elemzéshez a három legnépszerűbb SBFL-formulát (Tarantula, Ochiai, DStar) használtuk. Az eredmények azt mutatják, hogy a predikátummal kapcsolatos kódelemek (**IF kategória**) módosításai pontosabban lokalizálhatók az SBFL segítségével, mint más kategóriák. Megvizsgáltuk alacsonyabb szinten lévő hibajavítási kategóriák és az SBFL pontossága közötti kapcsolatot is. Az eredmények azt mutatják, hogy a **IF** kategórián belül két alkategóriát nehezebb lokalizálni mint a többit. Az összes eset közül azonban bizonyos hibajavítási típusok (IF-ABR, IF-RBR, SQ-AFO, SQ-AROB) nehezebben, míg más típusok (IF-CC, IF-RMV és MC-DAP) könnyebben lokalizálhatók SBFL segítségével.

A 4. fejezetben a hívási láncok használhatóságát elemeztük az SBFL-algoritmusok javításához. A hívási láncok alapján heurisztikákat hoztunk létre az egységtesztek és az egységszerű tesztek azonosítására. Az eredmények azt mutatják, hogy ha a releváns tesztesetek több mint 20%-a tiszta egységteszt (PUT) akkor az SBFL pontossága jelentősen javul. Továbbá egy a heurisztikákon alapuló súlyozási stratégiát alkalmaztunk, hogy javítsunk a Call-Frequency alapú SBFL algoritmuson. Ezen tapasztalatok alapján kijelenthető, hogy minél kisebb területet kell a fejlesztőnek átvizsgálnia annál könnyebben megtalálja a hibát. Ezt az információt alkalmaztuk egy új SBFL algoritmus készítéséhez. Az eredmények azt mutatják, hogy ez az új megközelítés a Defects4j 2.0-ból származó projektek 75%-a esetében jobb lokalizációt eredményezett.

Az 5. fejezetben a predikátumokat és ahhoz kapcsolódó hibákat elemeztük a BugsInPy és Defects4J benchmarkokban. A predikátum utasítások az összes utasításhoz képest kis mennyiségben fordulnak elő a programokban, azonban a hibás utasítások több mint fele predikátummal kapcsolatos. A módszerünk, a *Predicate Promotion* a predikátumokat az SBFL által elkészített gyanússági listában a függő blokkjaik fölé emeli. Az eredmények azt mutatják, hogy a *Predicate Promotion* algoritmus az eredetihez képest több hibás utasítást sorol a top 10 leggyanúsabb elem közé mind Java, mind Python nyelven mint az eredeti algoritmus. Azoknak a hibáknak a kvalitatív elemzését is megnéztük, amelyek az algorit-

mus miatt bekerültek a top 10-be vagy kikerültek onnan. Az algoritmus akkor működik a legjobban, ha a predikátumban szemantikai probléma van. 15 résztvevővel felhasználói tanulmányt végeztünk a *Predicate Promotion* algoritmus előnyeinek értékelésére hibakeresés során. Annak ellenére, hogy néhányan nem azonosították helyesen a hibákat, a résztvevők alaposabb magyarázatot adtak a *Predicate Promotion* algoritmus használatával.

A 6. fejezetben azt vizsgáltuk meg, hogy milyen kihívások merülnek fel az SBFL algoritmusok megvalósítása során, majd megoldási javaslatokat adtunk ezekre a kihívásokra. Létrehoztuk a CharmFL-t, egy IDE-be integrált hibalokalizációs eszközt. Olyan funkciókat implementáltunk, amelyek fokozzák a felhasználói interaktivitást és növeli az eszköz felhasználóbarátságát. Emellett megvizsgáltuk, hogy a nullával való osztás problémája hogyan befolyásolhatja az SBFL-képleteket, és milyen megoldásokat találhatunk ennek elkerülésére. Különböző irányelveket javasoltunk a nullával való osztás problémájának kiküszöbölésére.