Towards Precise Fault Localization: Expanding Spectrum Analysis with Variables and Branches

Hyeongjun Jeon°, Gyeongju Lee, Aditi and Sang-Ki Ko
Department of Artificial Intelligence, University of Seoul
{113bommy, lkj011218, aditimzu16, sangkiko}@uos.ac.kr

Abstract

Automated Program Repair (APR) and Fault Localization (FL) are vital for efficient software debugging, yet localizing logical errors remains challenging. Standard Spectrum-Based Fault Localization (SBFL) often lacks the precision to effectively guide Large Language Models (LLMs) for repair tasks. This paper introduces an "Expanded SBFL" methodology designed to improve the localization of logical errors and enhance LLM-based APR. On the other hand, this paper focuses on a subset of error cases structurally well-suited to SBFL to ensure that our enhanced methodology can be applied more reliably and yield meaningful improvements in APR. We evaluate our method using the DeepSeek-Coder-V2-Lite-Instruct LLM on a curated dataset of Python logical errors derived from the Google DeepMind CodeContest dataset. The experimental results demonstrate that our Expanded SBFL methodology improves APR performance for specific errors.

1. Introduction

Automated Program Repair (APR) and Fault Localization (FL) are crucial for reducing manual debugging effort in software engineering. The software defects can be broadly divided into syntactic and logical errors. This work focuses on repairing logical errors in Python programs by integrating FL and APR. We utilize Spectrum-Based Fault Localization (SBFL), a widely studied FL technique that utilizes code coverage information from test case executions to identify potentially faulty code locations statistically. SBFL methods correlate program element execution with test outcomes (pass/-fail) to rank code lines by suspiciousness. Despite its advantages, such as being lightweight and language-agnostic, standard SBFL often suffers from limited precision. The top-ranked lines may not always identify the actual fault, leading to inefficiencies when inaccurate information guides subsequent repair attempts.

We propose an enhanced SBFL methodology to improve the localization of logical errors in Python programs. Recognizing that standard SBFL formulas can be unreliable for certain code structures, we filter out cases where spectrum-based analysis is ineffective, focusing on issues better suited for SBFL localization. Additionally, we define SBFL expansion strategies for these areas, offering more relevant information for the APR task to Large Language Models (LLMs). By refining the FL output and providing richer context, we significantly enhance the effectiveness of LLM-based APR for logical errors in Python.

2. Related Work

Deep Learning-based APR APR has progressed from earlier deep learning models trained on bug-fix datasets, which often learned limited repair patterns, to modern LLMs. The advent of powerful LLMs such as GPT-4, Codex, and CodeT5, pre-trained on extensive code and text corpora, represents a significant leap forward. These models excel in code generation and modification, driven by context or prompts, and often outperform traditional APR techniques due to their deeper understanding of code structures and repair strategies [1]. Our research utilizes an open-source LLM (DeepSeek-Coder-V2-Lite-Instruct [4]) for APR, guided by fault locations identified through our enhanced FL technique.

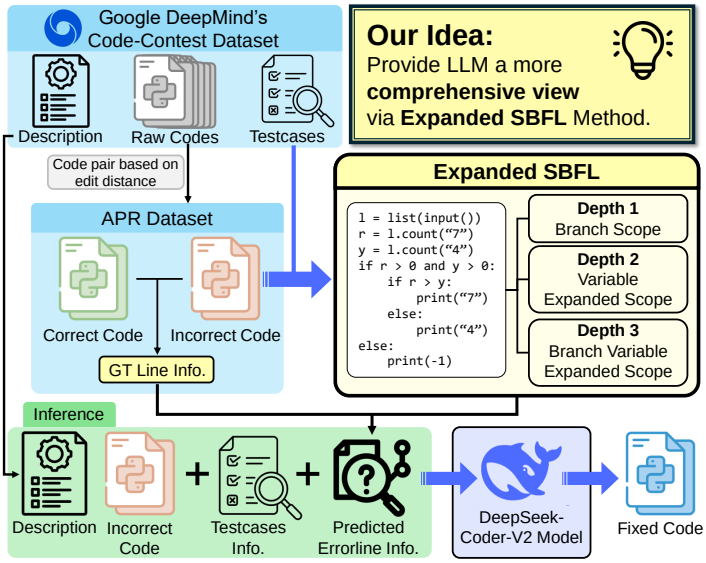

Figure 1: Overview of the proposed APR framework

Spectrum-Based Fault Localization (SBFL) SBFL is a fault localization technique that analyzes code coverage data from test executions, correlating it with pass/fail outcomes [3]. The core principle is that code lines executed more frequently by failing tests and less by passing tests are considered suspicious. SBFL employs statistical formulas (e.g., Tarantula, Ochiai) to calculate suspiciousness scores and rank lines accordingly. Despite its utility, SBFL has inherent limitations in addressing certain fault types. This paper identifies the faults suitable for SBFL and proposes an extended methodology to enhance its localization capability, with the goal of improving downstream APR performance.

†) This work was supported by the Ministry of Education of the Republic of Korea and the National Research Foundation of Korea (NRF-RS-2023-00208094).

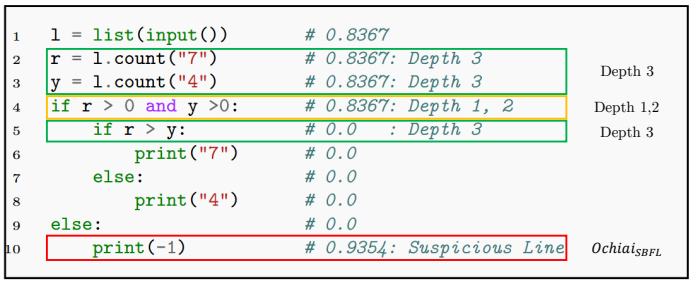

Figure 2: Running example of context expansion

3. Main Contribution

We created a dataset using the Google DeepMind CodeContest Python dataset [2] to evaluate our approach to detecting logical errors. We identified pairs of submissions, one correct (passing all test cases) and one incorrect (failing at least one), and filtered them based on edit distance (≤ 9) to focus on small logical errors.

SBFL prediction reliability depends on line type and context. Loop coverage can be mixed, and standard SBFL methods focus only on line execution, limiting control flow accuracy. To improve localization precision, we applied a heuristic filter, considering predictions 'confident' if the top SBFL-ranked line was a conditional statement (if/elif) outside a loop (for/while). Only programs meeting this criterion were included in our final dataset.

Following standard SBFL procedures, we executed test suites and used coverage.py to collect line-level coverage data. However, SBFL often produces noisy results, with the highest-ranked line not necessarily being the fault location. To address this, we used an 'Expanded SBFL' method, incorporating heuristics to provide the LLM with a more accurate view of the faulty code region, reducing misleading noise in the APR process.

- **Depth 1 (Branch Scope):** Include the entire control flow branch containing the initially estimated faulty line.
- **Depth 2 (Variable Expanded Scope):** Identify all variables used in the initially estimated faulty line. Then, include all lines that use these identified variables.
- **Depth 3 (Branch Variable Expanded Scope):** Identify all variables used in the initially estimated faulty line and those used within the Depth 1 branch. Then, include all lines that use these identified variables.

Figure 2 illustrates a Python code example, with each line annotated by its corresponding Ochiai score. The red box highlights the lines identified as suspicious by SBFL, while the yellow and green boxes represent additional lines included through expansion. Yellow denotes Depth 1 and Depth 2 expansions, and green indicates lines added at Depth 3.

We capped SBFL expansion at 50% of the total code lines to prevent dilution of fault information. Depth 3 expansion was attempted first, with a fallback to Depth 2 and then Depth 1 if the expansion limit was exceeded. Our experiments showed that this length-limited, prioritized expansion strategy improved localization precision, significantly boosting the subsequent APR success of the DeepSeek-Coder-V2-Lite-Instruct over baseline SBFL. The table below reports the average proportion of lines flagged as suspicious, as well as the precision and recall of including the actual faulty line. N represents the total number of evaluated samples, and $|\cdot|$ denotes the length of each element.

Table 1: Key statistics of different fault localization methods

Expansion	Localization Rate	Recall	Precision
Ochiai$_{\text{SBFL}}$	6.62%	49.56%	**49.56%**
+ Depth 1	15.78%	76.11%	31.95%
+ Depth 2	19.09%	78.00%	27.06%
+ Depth 3	**22.55%**	**80.78%**	23.73%

Table 2: Overall experimental results

Combination	Extra Info	Confidence Filtered Dataset		
		Pass@1	Pass@5	Pass@10
Code + Description	-	29.44%	62.33%	72.00%
	+ Depth 1	**31.67%**	62.89%	73.22%
	+ Depth 2	31.11%	62.33%	73.33%
	+ Depth 3	30.33%	**62.89%**	**73.89%**
	+ GT Line	33.78%	64.44%	75.56%
Code + Description + Test cases	-	34.44%	65.33%	74.89%
	+ Depth 1	36.67%	**69.78%**	**77.44%**
	+ Depth 2	**37.89%**	67.00%	77.22%
	+ Depth 3	**37.89%**	67.11%	76.33%
	+ GT Line	38.67%	67.78%	77.78%

$$\text{Localization Rate} = \frac{\sum_{i=1}^{N} |\text{Suspicious Lines}_i|}{\sum_{i=1}^{N} |\text{Total Lines}_i|}$$

$$\text{Recall} = \frac{\sum_{i=1}^{N} \mathbf{1}\left[\text{Fault}_i \cap \text{Suspicious Lines}_i \neq \emptyset\right]}{\sum_{i=1}^{N} |\text{Fault}_i|}$$

$$\text{Precision} = \frac{\sum_{i=1}^{N} \mathbf{1}\left[\text{Fault}_i \cap \text{Suspicious Lines}_i \neq \emptyset\right]}{\sum_{i=1}^{N} |\text{Suspicious Lines}_i|}$$

4. Experiments

To evaluate the effectiveness of our Expanded SBFL approach in LLM-based APR, we designed a comprehensive experiment comparing various levels of fault localization information provided to the repair model in two different contexts.

FL Inputs We compared three conditions for fault localization (FL) information provided to the LLM. The **Baseline** condition included only buggy code and context, without fault locations. The **Expanded SBFL** condition added localized context from one of three expansion strategies: Depth 1 (Branch), Depth 2 (Variable), or

Depth 3 (Branch & Variable). The **Ground Truth (GT)** condition provided the LLM with the correct fault location for comparison.

Context Variants for LLM Prompting For each FL condition, we tested two contextual variants in the DeepSeek-Coder-V2-Lite-Instruct prompt. The **Program Description** variant included the NL description of the problem, while the **Program Description & Test Case** variant added passing or failing test case examples, detailing inputs, actual outputs, and expected outputs.

Performance Metrics We assess logical error correction by measuring the proportion of modified programs that pass all CodeContest test cases, relative to the total number of samples. A program is considered successfully repaired if it passes all test cases.

Table 3: Error repair performance by error type

Error Type	Description Only			Description + Test Case		
	Base	SBFL+	GT	Base	SBFL+	GT
Output Format Errors	29.5	35.6 (+6.1)	40.2	45.3	47.7 (+2.4)	50.8
Incorrect Conditional Logic	34.9	35.4 (+0.5)	37.1	37.2	39.2 (+2.0)	41.5
Loop Errors	35.5	29.0 (−6.5)	45.2	37.9	34.5 (−3.4)	37.9
Variable Misuse	26.2	28.5 (+2.3)	29.2	30.9	33.1 (+2.2)	33.1
Errors of Omission	34.5	25.0 (−9.5)	39.3	41.0	38.6 (−2.4)	32.5
Flawed Calculations	27.8	30.3 (+2.5)	30.9	30.3	34.4 (+4.1)	33.7
Index Misuse	38.1	31.0 (−7.0)	34.5	34.1	38.8 (+4.7)	43.5

Experimental Results and Analysis Table 2 highlights key insights on APR performance. First, richer context consistently improves repair effectiveness. Adding test cases boosted Pass@k rates across all fault localization (FL) conditions (Baseline, SBFL+, GT), with Pass@1 increasing from 29.44% to 34.44% in the baseline.

Second, our SBFL+ method outperformed the baseline, achieving a Pass@1 of 37.89% with the richest context, closely matching GT's 38.67%. This demonstrates that enhanced localization significantly impacts APR success for SBFL-suited errors.

Comparing expansion depths within SBFL+ showed varying effectiveness. Depths 2 and 3 provided the best Pass@1 (37.89%) with test case context, while Depth 1 was slightly better for Pass@5 and Pass@10. This highlights that the optimal expansion depth may vary depending on the context and metrics, but SBFL+ still outperforms the baseline.

We categorized logical errors into seven logical error types suggested in Table 3. SBFL+ outperformed the baseline for Output Format, Variable Misuse, and Flawed Calculations, often approaching GT performance, but lagged for Loop Errors, Errors of Omission, and Index Misuse. This suggests that SBFL+ struggles with loop-centric faults or missing code, where current expansion cannot fully compensate.

Adding test case information improved localization precision and APR effectiveness for error types such as Output Format and Flawed Calculations but worsened performance for Loop Errors and Errors of Omission. Our manual analysis suggests that the model tends to overfit the failing test case, generating a patch that addresses a couple of failing inputs while ignoring the overall program logic demonstrated by the description. This issue is particularly crucial for Loop Errors and Errors of Omission as these bugs often require broader, structural reasoning rather than a simple, localized fix. In these scenarios, the test case narrowly focuses the LLM's attention, hindering its ability to perform the global code modifications.

5. Conclusions

This paper introduced an enhanced SBFL method to improve LLM-based APR for Python logical errors. By applying a heuristic filter focusing on bugs amenable to SBFL and using context expansion strategies, our "Expanded SBFL" significantly improved APR performance using the DeepSeek-Coder-V2-Lite-Instruct LLM, nearly matching the upper bound set by GT fault localization, particularly with rich contexts like descriptions and test cases.

However, error repair performance analysis by error type revealed limitations. Expanded SBFL struggled with loop errors and errors of omission, where underlying SBFL signals are less reliable compared to baseline approaches without localization assistance. Furthermore, while adding test cases generally improved results, it sometimes constrained LLM for loop-related and omission errors under perfect localization.

Enhancing SBFL with structural heuristics and expanded context effectively supports LLM-driven repair for identifiable logical errors. The key future challenge is to develop more robust localization methods for error types inadequately addressed by standard SBFL.

References

[1] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128, 2023.

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. Science, 378(6624):1092–1097, 2022.

[3] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spectrum-based fault localization: Testing oracles are no longer mandatory. In 2011 11th International Conference on Quality Software, pages 1–10. IEEE, 2011.

[4] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931, 2024.