CASS: Context-Aware Slice Summarization for Debugging Regression Failures

Sahar Badihi
University of British Columbia, Canada
shrbadihi@ece.ubc.ca

***Abstract*—Troubleshooting regression failures is a common yet time-consuming task for developers. Program slicing techniques aim to streamline this process by identifying relevant program statements, reducing the code developers need to inspect. By surveying over 50 practitioners from eight countries, we identified two main limitations in existing slicing-based approaches for debugging regressions. First, to minimize the number of statements presented to developers, these approaches often exclude contextual information essential for fully understanding the failure. Second, to maintain information propagation within the slice, they include lengthy computations that are unnecessary for understanding the failure. We further use these observations to propose a new slicing approach for debugging regression failures, named CASS. Our evaluation shows that CASS better aligns with developers' needs while producing more concise slices than existing techniques.**

I. INTRODUCTION

Regression failures occur when software changes unintentionally break existing functionality [1], requiring significant debugging effort. Developers often need to review large amounts of code to identify the root cause of failures. Fault localization approaches assist developers by identifying failure-causing statements. Spectrum-based [2], [3] and delta-debugging-based [4], [5] approaches focus on automatically pinpointing such statements. However, isolating these statements without understanding their relationship to the failure, does not effectively aid debugging [6], [7], [8]. Slicing-based approaches address this limitation by capturing the dependencies and information flow between the produced subset of statements, aligning with developers' mental models during debugging [9], [10], [11], [12]. Single-program slicing approaches, such as dicing [9], chopping [13], and thin slicing [14], aim to minimize the slice size while retaining relevant failure analysis information.

Dual-version slicing techniques, such as DUALSLICE [15] and INPRESS [16], focus on *regression* failures by analyzing both the base and regression versions of a program. While effective in shortening slices by removing identical information from both versions, they risk omitting critical *contextual* information needed to build an accurate mental model of dependencies between slice statements, eliminating one of the main benefits of slicing-based techniques.

As part of this work, we conduct a comprehensive study with 55 experienced software developers to better understand what information is important when debugging regression failures. The results show that participants indeed highly value the contextual information omitted by dual-version slicing techniques for understanding regression failures. They also found that most statements preserved in dual slices provide context for changes rather than directly causing failures.

Based on these observations, we further propose a novel slicing approach named CASS (Context-aware Slice Summarization). Our evaluation shows that CASS concisely retains the contextual statements necessary for understanding the failure while accurately summarizing propagation-related code blocks that are less important from the developers' perspective.

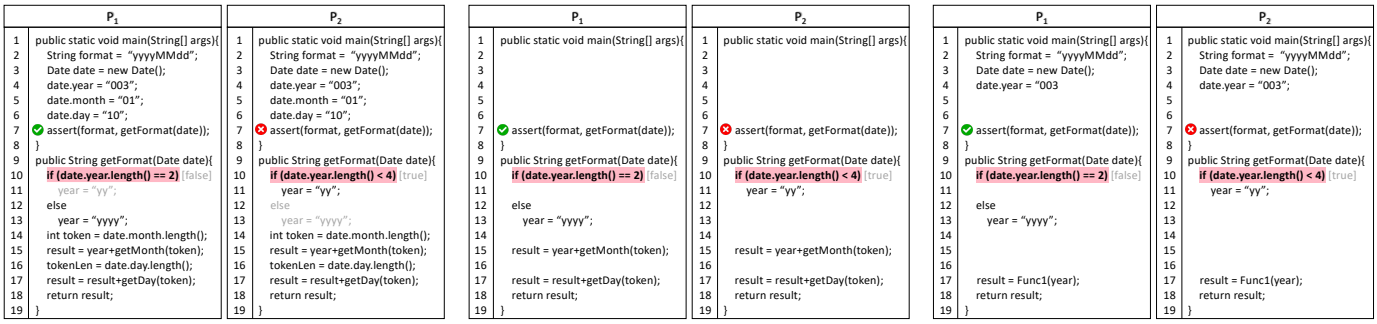
II. DUALSLICE AND ITS LIMITATIONS

Consider, for example, two versions of a program, P_1 and P_2, in Figure 1a – a simplified version of the *Lang-18* failure from the Defects4J dataset [17]. The figure shows dynamic execution traces of passing (P_1) and failing (P_2) executions, with the code changed between the two versions highlighted in red (line 10). The assertion in line 7 checks if `getFormat` matches `format`. In P_1, 3-digit years are set as `"yyyy"`, passing the assertion. In P_2, an updated condition causes `year` to be assigned `"yy"`, altering the value of `"result"` and leading to the failure.

A *classic backward dynamic slice* [18] inspects control and data dependencies to identify executed statements, directly or transitively, affecting a variable of interest. DUALSLICE [15] is a symmetric slicing technique that begins by aligning the traces and then focuses only on their differences. Specifically, it retains *unmatched* statements (without corresponding instances in the other trace), e.g., the statement in line 13 in P_1, and *matched* statements (producing different values), e.g., statements in lines 15, 17, and 18. It omits matched statements with identical data values, e.g., statements in lines 2-6. Unlike classical slicing, DUALSLICE computes transitive dependencies across both traces, adding aligned statements from the other trace to capture missing information that could explain the failure. Figure 1b illustrates the slices produced by DUALSLICE.

While DUALSLICE reduces the number of statements developers examine, it misses essential context. For example, the variable `format` used in line 7 in Figure 1b is never defined and, even more challenging, the value of `date` is also omitted, further complicating debugging (especially in larger and more complex software). This technique assumes the relevance of statements without validation, relying on reduction rates alone.

INPRESS [16] further minimizes dual slices by summarizing common code blocks. However, similar to DUALSLICE, it omits essential contextual information (e.g., lines 2–6 in Figure 1b) needed to understand the failure. Due to space limitations, this paper focuses on the DUALSLICE approach.



(a) Execution Traces for P_1 and P_2. (b) DUALSLICE for P_1 and P_2. (c) CASS for P_1 and P_2.

Fig. 1: Running Example: a Simplified Lang-18 Failure.

III. STUDY ON DEBUGGING REGRESSION FAILURES

To gather insights into the information developers need to efficiently debug regression failures, we conducted a large-scale study involving 55 experienced software developers from eight countries. The study used an interactive online questionnaire and was approved by the ethics board in our institution.

Program Subjects. We randomly selected 6 failures, four from the widely-used Defects4J [17] benchmark and two from our collected set of real-world client-library upgrade failures. As the execution traces were large (20,125 statements on average), we simplified the code snippets, preserving the changes and failures while removing implementation details.

Study Questionnaire. After collecting background and demographics, each participant was assigned one of the six subjects. We showed participants two program traces, similar to Figure 1a, and asked them first to explain the failure. Next, they selected statements they considered relevant for understanding the failure, followed by providing a rationale for their selection.

Findings. Most study participants (44 out of 55, 80%) deemed at least one of the context statements (e.g., line 2) essential for understanding the failure. Moreover, the majority of the participants (42 out of 55, 78%) did not pick any of propagation statements kept by DUALSLICE (e.g., statements in lines 15 and 17) as relevant.

IV. CONTEXT-AWARE SLICE SUMMARIZATION

Based on the observations from our study, we design CASS, a slicing-based approach that better fits the developers' needs. The output produced by CASS on our running example is shown in Figure 1c. Unlike DUALSLICE, which excludes statements with the same data values from the slice, CASS retains these statements, ensuring the slice remains free of undefined variables (e.g., `format` in line 7). However, analysis of developers' responses reveals that not all contextual statements are equally important. For example, statements in lines 5-6 were not selected by the majority of participants. CASS defines context as the transitive closure of all definitions flowing into the changed statements and the test assertion.

Additionally, CASS abstracts computational code blocks that propagate information between the changed statements, summarizing them with high-level input-output functions [1]. These functions represent outputs as variables needed for subsequent computations and inputs as dependencies from earlier variables in the slice. This summarization removes unimportant internal computations while preserving the critical flow of information. In summary, CASS *preserves essential context while minimizing irrelevant details*, offering developers a more focused view of the failure.

Evaluation. To demonstrate CASS's effectiveness, we compared it to DUALSLICE. We evaluated *precision* (the proportion of retained statements that are relevant) and *recall* (the proportion of relevant statements that are correctly retained) across the six study subjects. CASS achieved 84% precision and 92% recall, significantly outperforming DUALSLICE (62% precision, 71% recall), as shown in Table I. These results indicate that CASS not only retains relevant statements but also minimizes the inclusion of irrelevant ones.

TABLE I: Comparing Techniques

Subjects	Metrics	Trace	DUALSLICE	CASS
6 Study Subjects	Precision	31%	62%	**84%**
	Recall	100%	71%	**92%**
278 Failures	Size (Reduct.)	36,025 (0%)	1,398 (92%)	**185 (97%)**

In addition to precision and recall, we evaluated the effort required to inspect slices by calculating the reduction rate over 278 failures from Defects4J, presented in the third row of Table I. The reduction rate measures the percentage decrease in slice size (#Slice) compared to the full trace (#$\mathbb{T}$), calculated as: $\frac{\#\mathbb{T}-\#Slice}{\#\mathbb{T}}$. CASS achieved an average reduction rate of 97%, reducing the number of steps for developers to inspect from 36,025 to 185. Interestingly, despite including contextual statements, CASS achieves a slightly higher reduction rate with significantly fewer statements (185 vs. 1,398) in comparison to DUALSLICE. This demonstrates that CASS aligns with developer preferences and offers a manageable set of statements for debugging, making it both efficient and practical.

V. CONCLUSION

This paper investigates the effectiveness of existing slicing techniques, such as DUALSLICE, for debugging regression failures. A study with 55 participants from eight countries revealed key limitations, leading to the development of CASS, a new approach that better aligns with developer preferences while reducing the inspection effort.

[1]The summarization algorithm builds on our previous work, INPRESS [16], with several design and implementation modifications omitted for simplicity.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental Regression Testing," in *Proc. of the Conference on Software Maintenance (ICSM)*, 1993, pp. 348–357.

[2] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for Fault Localization," in *Proc. of the International Conference on Software Engineering Workshop on Software Visualization (ICSE-SV)*, 2001.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proc. of the International Symposium on Dependable Computing (PRDC)*, 2006, pp. 39–46.

[4] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 253–267, 1999.

[5] A. Zeller, "Isolating Cause-effect Chains from Computer Programs," in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2002, pp. 1–10.

[6] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 199–209.

[7] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2016, pp. 808–819.

[8] E. Soremekun, L. Kirschner, M. Böhme, and M. Papadakis, "Evaluating the Impact of Experimental Assumptions in Automated Fault Localization," in *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 159–171.

[9] M. Weiser and J. Lyle, "Experiments on Slicing-based Debugging Aids," in *Workshop on Empirical Studies of Programmers (ESP)*, 1986, pp. 187–197.

[10] M. A. Francel and S. Rugaber, "The Value of Slicing While Debugging," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 151–169, 2001.

[11] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental Evaluation of Program Slicing for Fault Localization," *Empirical Software Engineering (ESE)*, vol. 7, no. 1, pp. 49–76, 2002.

[12] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating Faults with Program Slicing: An Empirical Analysis," *Empirical Software Engineering (ESE)*, vol. 26, no. 3, pp. 1–45, 2021.

[13] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-inducing Chops," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2005, pp. 263–272.

[14] M. Sridharan, S. J. Fink, and R. Bodik, "Thin Slicing," in *Proc. of the International Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 112–122.

[15] H. Wang, Y. Lin, Z. Yang, J. Sun, Y. Liu, J. S. Dong, Q. Zheng, and T. Liu, "Explaining Regressions via Alignment Slicing and Mending," *IEEE Transactions on Software Engineering (TSE)*, 2019.

[16] S. Badihi, K. Ahmed, Y. Li, and J. Rubin, "Responsibility in Context: On Applicability of Slicing in Semantic Regression Analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 563–575.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440.

[18] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.