CoReX: Context-Aware Refinement-Based Slicing for Debugging Regression Failures

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

Julia Rubin
mjulia@ece.ubc.ca
The University of British Columbia
Vancouver, Canada

Abstract

Troubleshooting regression failures is one of the most frequent yet time-consuming development tasks. To help with this task, numerous approaches aim to narrow down developers' attention to the subset of code statements relevant to the failure. The most prominent of these approaches are based on program slicing, as slicing not only identifies a subset of relevant statements but also maintains the flow of information between these statements.

By surveying more than 50 practitioners from eight different countries, we observe that existing dual-version slicing-based approaches have two main limitations: (a) to minimize the number of statements presented to the developer, they omit contextual information required to truly understand the failure and (b) to keep information propagation between the statements in the slice intact, they include lengthy computations that are not necessary to understand the failure. We use these observations to define a new dual-version slicing approach for regression scenarios, called CoReX. We evaluate CoReX on a large number of subjects, comparing it with other existing dual-version slicing approaches. The results of our evaluation show that CoReX outperforms these approaches both in the quantitative metrics used in prior work and in alignment with developers' expectations. We believe our work provides grounds for efficient integration of slicing-based approaches into development workflows.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Program debugging, regression failures, program slicing, user study

ACM Reference Format:
Sahar Badihi and Julia Rubin. 2026. CoReX: Context-Aware Refinement-Based Slicing for Debugging Regression Failures. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764577

1 Introduction

Regression failures occur when changes to software, e.g., to add new features or fix security vulnerabilities, unintentionally break existing functionality [3] (think, "it worked yesterday but does not work today"). Debugging regression failures often demands significant time and effort, especially in large-scale software systems: developers need to go over hundreds, if not thousands, lines of code to understand which of the changes caused the failure and why.

Numerous fault localization approaches are designed to minimize the quantity of code that needs to be inspected when debugging such a failure. They can largely be divided into spectrum-based [1, 25], delta-debugging-based [69, 70], program-slicing-based [2, 32, 48, 50, 52, 56], and combinations of the above [10, 16, 36, 45, 48].

Spectrum- and delta-debugging-based approaches aim to pinpoint statements responsible for the failure. However, prior research shows that focusing only on a set of responsible statements in isolation, without capturing the connection between these statements and their relationship to the failure itself, does not improve the developers' ability to understand and repair faults [23, 31, 40, 44, 64, 68]. In fact, Ko et al. [30] observed that, while debugging, developers not only rely on code search and relevant information gathering but also perform dependency analysis.

Slicing-based approaches aim to address this need by capturing the dependencies and flow of information between the produced subset of statements, supporting the mental-model developers use when debugging failures [5, 7, 13, 34, 45, 56, 64]. A variety of single-program slicing approaches, such as dicing [56], chopping [16], slicing with barriers [33], and thin slicing [46], aim at minimizing the size of the slice developers need to inspect while ensuring it contains the relevant information needed to analyze the failure.

DualSlice [48, 50, 52] and InPreSS [6] are *dual-version* slicing techniques that target the space of regression failures. They analyze the base and regression versions of a program simultaneously, retaining only the execution statements that lead to divergent execution behavior between the versions.

While both dual-version slicing techniques focus on shortening the slice by excluding information identical in both program versions, we conjecture that not all such information is irrelevant and a more nuanced distinction is necessary. Specifically, both techniques omit *contextual* information needed to understand the divergent behaviors between two program versions. Omitting such information, in fact, eliminates one of the main benefits of slicing-based techniques: focusing the developers' attention on the *sequence* of relevant program statements. Moreover, while InPreSS summarizes the propagation of information in code that does not change between two versions of the program, the practical usefulness of such summarization has not been established.

To gain further insights into what information developers deem important when debugging regression failures, we started by conducting a comprehensive study with 55 experienced software developers from eight different countries. The study gathered information about developer needs and evaluated the appropriateness of design decisions made by existing dual-version slicing techniques. To the best of our knowledge, this is the first such study conducted for dual-version slicing. Moreover, among over 200 papers on slicing-based debugging approaches, only very few and relatively older papers [8, 12, 13, 29, 34, 54, 56] focused on its practical applicability.

The results of our study show, among other findings, that participants indeed deemed contextual information omitted by existing dual-version slicing approaches highly important for understanding regression failures. They also found information summarized by InPreSS relevant but with a lower level of importance.

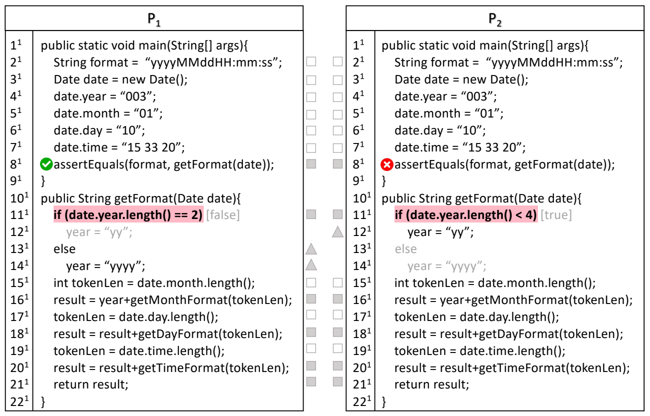
Based on our observations, as well as earlier work showing that developers tend to debug and analyze code iteratively, in multiple phases [30, 34, 37, 40, 64], we designed a novel dual-version slicing approach named CoReX. CoReX builds upon and extends existing dual-version slicing techniques, making a more nuanced distinction between *primary* statements essential for understanding the failure and *secondary* statements of potentially lower relevance. Effectively, the sets of primary and secondary statements produced by CoReX do not overlap with those of DualSlice and InPreSS.

To evaluate CoReX, we compared it with existing dual-version slicing techniques on a set of 286 real-world Java regression failures borrowed from Defects4J [26] and InPreSS [6] repositories. We measured the length of slices each tool produces and the prioritization ratio – the fraction of statements in the slice out of all statements in the trace (often also referred to as "reduction rate"). These metrics are commonly used to evaluate slicing tools as they are indicative of the number of "debugging steps" developers need to take when inspecting a failure.

Our results show that, when considering primary statements only, slices produced by CoReX are around x50 shorter than the entire execution traces (665 vs. 33,884 statements on average) and x3 shorter than those produced by DualSlice (665 vs. 2,007 statements on average). While they contain x3.2 more statements than slices produced by InPreSS (665 vs. 207 statements on average), these additional statements are essential for preserving contextual information, as demonstrated below.

When measuring the execution time of each tool, we observed that the extra analysis performed by CoReX leads to an increased execution time compared to both DualSlice and InPreSS. Specifically, extra processing related to the added contextual information results in a x1.5 and x1.15 overhead compared with InPreSS, on the (smaller) Defects4J and (larger) InPreSS benchmarks, respectively. While we believe this is acceptable given the increased tool accuracy, additional performance improvements can be subject of future work, especially in the underlying slicing components.

Finally, to "close the loop", we assess the ability of the heuristics employed by CoReX to produce slices that would be deemed relevant by the developers of our original study, compared with other dual-slicing tools. To this end, we computed the precision, recall, and F-measure of each tool in identifying statements developers selected as relevant to the failure on the six subjects they analyzed



P1

```
1^1  public static void main(String[] args){
2^1     String format =  "yyyyMMddHH:mm:ss";
3^1     Date date = new Date();
4^1     date.year = "003";
5^1     date.month = "01";
6^1     date.day = "10";
7^1     date.time = "15 33 20";
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() == 2) [false]
12^1       year = "yy";
13^1    else
14^1       year = "yyyy";
15^1    int tokenLen = date.month.length();
16^1    result = year+getMonthFormat(tokenLen);
17^1    tokenLen = date.day.length();
18^1    result = result+getDayFormat(tokenLen);
19^1    tokenLen = date.time.length();
20^1    result = result+getTimeFormat(tokenLen);
21^1    return result;
22^1 }
```

P2

```
1^1  public static void main(String[] args){
2^1     String format =  "yyyyMMddHH:mm:ss";
3^1     Date date = new Date();
4^1     date.year = "003";
5^1     date.month = "01";
6^1     date.day = "10";
7^1     date.time = "15 33 20";
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() < 4) [true]
12^1       year = "yy";
13^1    else
14^1       year = "yyyy";
15^1    int tokenLen = date.month.length();
16^1    result = year+getMonthFormat(tokenLen);
17^1    tokenLen = date.day.length();
18^1    result = result+getDayFormat(tokenLen);
19^1    tokenLen = date.time.length();
20^1    result = result+getTimeFormat(tokenLen);
21^1    return result;
22^1 }
```

(a) $\mathbb{T}_{P_1}$. **(b)** $\mathbb{T}_{P_2}$.

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

during the study. To the best of our knowledge, this is the first evaluation that focuses on the accuracy of the tools, in addition to the prioritization ratio that they achieve. Our results show that CoReX achieves 93% of alignment with developers' opinion, compared with 63% of alignment for DualSlice and 74% – for InPreSS.

Summary of Contributions.

1. This paper presents the results of a comprehensive developer study that investigates the relevance of different types of program statements in understanding regression failures. The results obtained through a number of complementary experiments show that 84% of developers find contextual information, omitted by existing techniques, highly relevant. They found propagation information preserved by DualSlice and summarized by InPreSS of secondary importance (Section 4).
2. It implements and empirically evaluates a novel dynamic dual-version slicing approach called CoReX, which incorporates the results of the developer study. Our evaluation of CoReX on 286 real-world Java regressions shows that it achieves a prioritization ratio comparable with existing techniques while focusing on statements deemed more relevant by the developers (Sections 5 and 6).
3. It makes our implementation, study, and evaluation results publicly available [49] to encourage further work in this area.

2 Running Example

Figure 1 shows the old (left) and the new (right) versions of the *Lang-18* program, taken from the popular Defects4J regression dataset [11] and simplified for presentation purposes. The goal of this code is to calculate the format of a given date. The assertion in line 8 checks if the method `getFormat` outputs the same value as in the variable `format` defined in line 2.

The expected result is successfully obtained in the old version, but the assertion fails in the new version. That is because 3-digit years were previously deemed to have the `"yyyy"` format: the `if` statement in line 11 determines that $3 \neq 2$ and the value of `year` is then assigned in line 14 to be `"yyyy"`. Due to the change in line 11, in the new version, the `if` statement now evaluates to *true* ($3 < 4$), which leads to the value of `year` being assigned in line 12 to be `"yy"`. The remainder of the code (lines 15-21) further parses the input `date`, extracting and returning its format. However, this additional code has no effect on the failure.

3 Background

We now provide the relevant background information about slicing and introduce the two state-of-the-art dual-version slicing techniques: DualSlice [52] and InPreSS [6].

3.1 Slicing Fundamentals

Code Representation. We assume a common Java-bytecode-style programming language representation [76], with method calls and return statements, assignments to local and global variables, conditionals, i.e., `if`s, and `jump`s. For the simplicity of the presentation, we discuss our examples on the source-code level. However, slicing is performed on Java bytecode; thus, all conditionals other than `if`s, such as `for` and `while` loops, are expressed in terms of `if` and `jump` statements. We refer to a statement in line i of a program as s_i, e.g., the `if` statement in line 11 of Figure 1a is denoted by s_{11}. Moreover, in dynamic analysis, each statement of a program can be triggered multiple times during the program execution, e.g., in multiple iterations of a loop or in different calls to the same method. We refer to each individual execution of a statement as a *statement instance* and denote the k^{th} execution of a statement s_i as s_i^k. We refer to each statement instance as an *execution-level statement* or, simply *execution statement*. A sequence of execution statements (i.e., statement instances executed in a particular program run) constitutes an *execution trace*. In fact, Figure 1 shows the execution traces of the old and the new versions of the program.

We annotate `if` statements (e.g., s_{11}) with a label showing whether the `if` condition is evaluated to *true* or *false* in the dynamic execution. Gray lines indicate statements that are not executed because they are encapsulated by an `if` statement that evaluates to *false* (e.g., s_{12} in Figure 1a).

Classical Single-Version Program Slicing. Classical slicing [53, 55] computes the set of statements that affect a particular variable of interest, often referred to as a *slicing criterion*. Slicing can be performed statically or dynamically [32]. While static slicing considers all possible program paths leading to the slicing criterion, dynamic slicing focuses on one concrete execution. The main idea behind dynamic slicing is to first collect an execution trace of a program, and then inspect the control and data dependencies of the trace statements, identifying statement instances that affect the slicing criterion and omitting the rest. A slicing criterion for an execution trace is a tuple (c, V), where c is a statement instance and V is a set of all variables of interest used in this statement instance [32]. If V is omitted, it is assumed to include all variables used by c.

A *backward dynamic slice* [32] uses the notion of control- and data-flow dependencies. A control dependency means that the execution of the later statement (or statement instance) depends on the outcome of the former, e.g., statements within the `if` block are control-dependent on the `if` statement itself. There is a data-flow dependency between two statements (or statement instances) if the former defines a variable used in the latter. A slice is then defined as the set of statement instances whose execution affects the slicing criterion, i.e., the set of instances on which the slicing criterion is control- or data-flow-dependent, either directly or transitively. We denote by $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$ the two slices of the corresponding traces $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$, which were obtained by running the test T_c. Intuitively, a dynamic slice corresponds to the sequence of steps developers need to analyze when troubleshooting a failure.

3.2 Dual-Version Slicing

For regression failures, it is valuable to consider both the old and the new versions of the code simultaneously, as some faults could be caused by *omitting* certain code in the new version. Working on both versions simultaneously is the focus of dual-version slicing techniques. In the rest of the paper, we refer to the old and the new versions of the code as P_1 and P_2, respectively; we denote a test case that passes on P_1 and fails on P_2 by T_c. We assume that the execution of T_c is deterministic. We denote by $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$ the two traces that correspond to the execution of T_c on versions P_1 and P_2, respectively, e.g., the traces shown in Figure 1.

The set of changes between the versions is denoted by Δ. Δ includes statements that have to be *added (A)*, *removed (R)*, and *modified (M)* to transform one version of the program to another. In the example in Figure 1, Δ={M(s_{11})}. We boldface changed statements in our examples and highlight them in red (line 11).

Trace Alignment. Trace alignment obtains as input two traces, $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$, and identifies pairs of the corresponding trace statement instances (s, s'), where s comes from $\mathbb{T}_{P_1}$ and s' comes from $\mathbb{T}_{P_2}$ [75]. When a statement instance s or s' cannot be *matched* to any instance in the other trace, it is considered *unmatched*. There are several trace alignment techniques, which are based on string matching [43], memory indexing [47], and structural indexing [50, 65].

For example, s_{14}^1 in Figure 1a is unmatched as its corresponding statement in P_2 is not executed (and thus grayed out in Figure 1b). In contrast, s_2^1 is a matched statement instance. In our examples, we annotate unmatched and matched statement instances with triangles and squares, respectively (see Figure 1).

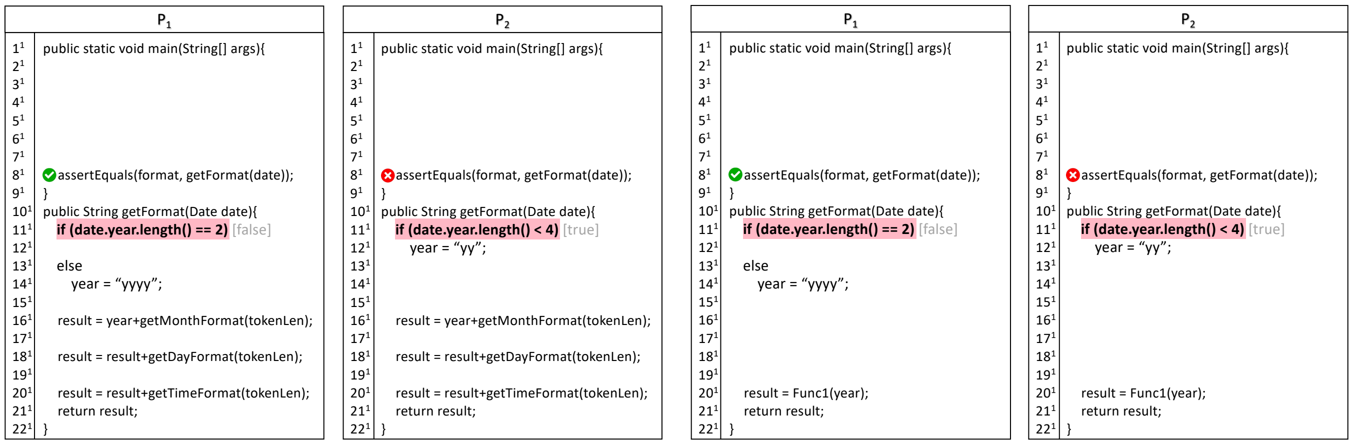
DualSlice is a symmetric slicing technique that works on two traces simultaneously. It was first introduced for debugging concurrency bugs [52] and later used for regression failures [48, 50]. Its goal is to produce a minimum sequence of statement instances that are causally connected, leading from the root cause to the failure.

Given two execution traces, DualSlice first aligns the traces and then separates matched statement instances into those that produce the same vs. different data values. We annotate statement instances that produce different values by filled squares, e.g., s_{16}^1 in Figure 1a, where the variable `result` has a different value due to the change in the value of `year`. Statements that produce the same values are annotated by empty squares. The main idea of the DualSlice technique is to focus only on differences between executions, i.e., unmatched statements and matched statements that produce different data values (triangles and filled squares).

Figure 2a shows the slices produced by DualSlice for the example in Figure 1. As there is no divergence in the execution of statement instances s_2^1-s_7^1, s_{15}^1, s_{17}^1, and s_{19}^1, DualSlice concludes that they are not part of these slices (and thus denoted by empty lines in the figure) and are irrelevant for the failure.

Another key idea behind DualSlice is to compute the transitive closure of dependencies across both traces: once a statement is added to a slice in one of the traces, its corresponding aligned statement is also added to the slice of the other trace. This is done to incorporate information *missing* from the run, as that could explain the reason for a failure.

InPreSS builds on top of DualSlice. The main idea behind InPreSS is to look at code that is common vs. different between the



P1
```
1^1  public static void main(String[] args){
2^1
3^1
4^1
5^1
6^1
7^1
8^1  ✓assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() == 2) [false]
12^1
13^1    else
14^1       year = "yyyy";
15^1
16^1    result = year+getMonthFormat(tokenLen);
17^1
18^1    result = result+getDayFormat(tokenLen);
19^1
20^1    result = result+getTimeFormat(tokenLen);
21^1    return result;
22^1 }
```

P2
```
1^1  public static void main(String[] args){
2^1
3^1
4^1
5^1
6^1
7^1
8^1  ✗assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() < 4) [true]
12^1       year = "yy";
13^1
14^1
15^1
16^1    result = year+getMonthFormat(tokenLen);
17^1
18^1    result = result+getDayFormat(tokenLen);
19^1
20^1    result = result+getTimeFormat(tokenLen);
21^1    return result;
22^1 }
```

(a) **DualSlice for P_1 and P_2.**

P1
```
1^1  public static void main(String[] args){
2^1
3^1
4^1
5^1
6^1
7^1
8^1  ✓assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() == 2) [false]
12^1
13^1    else
14^1       year = "yyyy";
15^1
16^1
17^1
18^1
19^1
20^1    result = Func1(year);
21^1    return result;
22^1 }
```

P2
```
1^1  public static void main(String[] args){
2^1
3^1
4^1
5^1
6^1
7^1
8^1  ✗assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1    if (date.year.length() < 4) [true]
12^1       year = "yy";
13^1
14^1
15^1
16^1
17^1
18^1
19^1
20^1    result = Func1(year);
21^1    return result;
22^1 }
```

(b) **InPreSS for P_1 and P_2.**

Figure 2: DualSlice and InPreSS Results for the Example in Figure 1.

versions. It postulates that matched common code, albeit producing different data values, is not directly responsible for the test failure. For example, s^1_{16}, s^1_{18}, and s^1_{20} are affected in the exact same manner by the value of `year` (set in s^1_{14} in P_1 and s^1_{12} in P_2) and thus are of less relevance when reasoning about the failure.

However, instead of removing such common code, InPreSS identifies and summarizes blocks of common code in the form of high-level input-output functions, where outputs are variables needed for later computations and inputs are their dependencies on variables used earlier in the slice. This is done to keep the flow of information within the slice, which developers might need when troubleshooting failures. Figure 2b shows the slices produced by InPreSS for the same example, where a statement `result = Func1(year)` in line 20 captures the dependency between the year and the output result. Such summaries eliminate (potentially very long) blocks of internal computations, better focusing the developers' attention on the part directly responsible for the failure.

4 Study: Developers' Needs for Debugging Regression Failures

Despite the valuable insights and the effectiveness of both dual-version slicing techniques in prioritizing the number of statements developers need to examine, we observe two main limitations. First, while InPreSS correctly points out that preserving the flow of information in a slice is important for the developers' ability to build a correct mental model of the code [9, 40, 63], the slices produced by both this approach and DualSlice miss contextual information. For example, the variable `format` used in line 8 in Figure 2b is never defined and it is unclear what this target format is expected to be. Even more challenging, the value of `date` is also omitted, which further complicates debugging (especially when dealing with larger and more complex software [27]).

Second, the importance of neither such contextual information nor statements hidden behind InPreSS summaries was ever evaluated with real developers. That is, the validity of the assumptions on which statements are relevant for the developer when debugging regression failures was not yet evaluated with real users. Instead, techniques are mostly evaluated on the prioritization ratio that they achieve.

To address these issues, we conducted a large-scale randomized study with 55 developers from eight countries, gathering their opinions on the design decisions made by existing techniques and, more generally, the information they need to efficiently debug regression failures. The study was conducted through an interactive online questionnaire, to reach a wide audience and explore a non-trivial set of case study code snippets. Full details about the study, including a copy of the questionnaire, are available online [49]. The study was approved by the ethics board in our institution.

We discuss our study methodology and results next.

4.1 Methodology

Program Subjects. We started from the Defects4J [26] benchmark (v1.5) – a large and popular collection of reproducible real-world failures in Java programs. Like prior work that conducted user studies using this benchmark [21, 63, 72] (for program repairs, single-version analysis, etc.), we excluded two projects that rely on domain-specific knowledge. We picked one failure at random from the remaining four projects (*chart*, *lang*, *math*, and *time*). Additionally, we selected two failures from the LibRench dataset, contributed by InPreSS [6] – a set of real-world client-library project pairs with at least one library upgrade failure. Table 1 presents our six selected subjects, together with the failure id for the Defects4J projects, a short description of each failure, and the number of study participants who were assigned this subject. To prioritize depth over breadth, each participant was assigned one subject only but studied it from multiple dimensions, as described below.

As the size of the execution trace for these subjects ranges between 263 and 54,781 statements (20,125 on average), we shortened and simplified code snippets, to ensure participants could fully comprehend the code and answer numerous questions within a reasonable time frame of about 30 minutes to one hour. The goal of this simplification was to preserve the essence of the changes and the failure (similar to the example in Figure 1) while removing unnecessary implementation details. The simplification was performed by the first author of the paper and reviewed by another

Table 1: User Study Subjects

Sub. ID	Project (Failure ID)	Short Description	# Part.
#S1	JFree**Chart** (8)	Modifying "timeZone" variable reference	9
#S2	Commons-**Lang** (18)	Modifying "year" format	9
#S3	Commons-**Math** (37)	Deleting conditional calculations	9
#S4	Joda-**Time** (8)	Modifying arithmetic expression	9
#S5	**Jackson**DataBind/OpenAPI	Removing reference shortcut	10
#S6	Alibaba-**Druid**/Dble	Modifying SQL Parsing	9

author, to ensure consistency with the original implementation. The size of the resulting snippets ranged from 15 to 26 statements, comparable to those used in similar debugging studies [44].

Study Questionnaire. After collecting optional background and demographic information, Each study participant was assigned one subject (out of 6), for which they answered 12 questions across three main study parts. Also, to minimize bias, we randomized the order of study parts presented to each participant and the order of code snippets presented in each part, as described below.

Part 1: Information participants deem relevant when debugging regression failures. To investigate this direction, we showed study participants two program traces, similar to the example in Figure 1, and asked them to first identify and describe the failure. Then, we asked them to select statements that they deemed relevant for identifying and explaining the failure, and provide a rationale for their selection.

Part 2: Design decisions made by existing techniques. To complement the previous questions looking at the problem from a different angle, we showed the participants three code pair views, similar to the DualSlice and InPreSS views in Figure 2. Each view had a different combination of statements describing the failure, i.e., statements selected by DualSlice, statements selected by InPreSS, and one of the above views augmented with contextual information. We asked participants to rank the views based on the following two criteria: (1) *Completeness*: whether the view includes all essential information needed to explain and debug the failure and (2) *Conciseness*: whether the view excludes only the necessary information needed to explain and debug the failure. We also asked participants to provide a written rationale for their ranking, detailing the advantages and disadvantages of each view.

Part 3: Value of textual explanations. To further investigate ways to assist developers in identifying and troubleshooting failures, we studied how helpful textual summaries can be for the task. To initiate such discussion, we provided line-by-line textual explanations that correspond to the code and showed them to the participants in an experiment similar to Part 2. We further asked the participants to provide insights into whether a code or text version was more helpful and why.

To generate textual explanations, we started from using GenAI techniques and further refined the generated explanations manually, as the initial GenAI-produced results were not fully satisfactory. Specifically, the initial explanations were revised by the first author of this paper and then peer-reviewed by the researchers in our group and the second author, to produce complete and consistent explanations across all case studies.

Finalizing. At the end of the questionnaire, respondents could optionally submit free-text comments and suggestions on how to improve fault localization and the survey itself.

A copy of our questionnaire for each subject is available online [49]. To minimize biases participants might have from viewing full code before or after analyzing the views or from analyzing the views in a particular order, we created multiple versions of the questionnaire for each of the six subjects: "trace first" and "views first" and, within each, different order of views. Each participant was automatically assigned one questionnaire version, selected at random from the version with the lowest number of responses.

Study Participants. As common with this kind of study [28], we started with a pilot. It included six participants from our home university. The goal of the pilot study was to test our survey design and infrastructure, which allowed us to identify and address obvious issues at no additional cost. The pilot study was also used to estimate the study duration and the necessary compensation. Our pilot participants found the survey generally easy to understand and provided feedback for further improvements, leading to minor modifications. Responses from the pilot study were discarded. Based on participant response times, we estimated the study duration to be between 20 and 45 minutes.

To recruit a more diverse group of participants, we invited Computer Science and Engineering *graduate* students at our university, who have at least one year of prior hands-on full-time development experience and who are familiar with code debugging tools. We further reached out to our network of collaborators and colleagues who, in turn, distributed a call for participation in their organizational and personal networks. All participants filled out a pre-study questionnaire and signed a consent form. The study lasted about two months, in late 2024. We randomly selected ten participants who completed the study to receive a $30 Amazon gift card.

4.2 Results

We received 103 responses, of which 48 (47%) were incomplete and we excluded them from our analysis. For the 55 complete responses, nine responses each were for subjects #S1-#S4 and #S6, and ten responses were for subject #S5. The responses came from eight countries across four continents: Canada, the United States, Germany, Denmark, France, the Netherlands, Iran, and Brazil, with Canada and Germany being the most represented. Among the participants, 16 were under 25, 33 were aged 25-34, and 3 were aged 35-44. Most of them were students (26 PhD, 13 Master's, and 11 Bachelor's), with some having concurrent appointments in industry. Additional five were professional software and testing engineers. Around 40% of the participants had over three years of professional industry experience, while most others have completed one or more industrial internships (1-3 years experience: 18%, 3-5 years: 13%, 5-10 years: 53%, >10 years: 16%). More than 80% self-assessed their programming skills as "Intermediate" and "Advanced". We thus believe our sample is diverse and representative.

Part 1: Statement Relevance. The majority of the participants (44 out of 55, 80%) deemed at least one of the context statements (empty squares) essential for understanding the failure. In their free-form answers, participants mentioned that «beside changes I want to see what concrete values cause the failure in this test» (P10) and «illustrating which variables are involved in the process or are passed as input can streamline things» (P9). On average, participants selected 56% of the context statements, with the highest selection at 80%

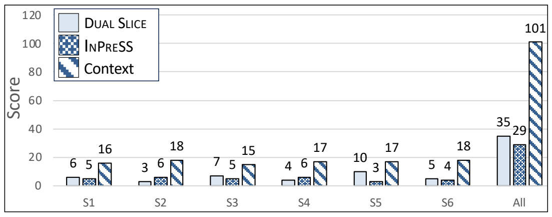


Figure 3: View Scores.

of all such statements. We further discuss the different types of context statements in Section 5, when we describe our proposed slicing approach.

Of the remaining 11 participants who did not select any context statements, nine had completed the *trace first* versions of the questionnaire. We conjecture that looking at complete traces first might have caused them to overlook the importance of these statements. As for the remaining two participants who completed the *view first* version of the questionnaire, their replies differ from those of the majority of participants who completed this version: 22 out of 24 *trace first* version participants (92%) selected at least one contextual statement, leading us to the conclusion that contextual information is important for participants who do not have access to full traces.

Focusing on propagation statements kept by DualSlice (full squares), the majority of the participants (43 out of 55, 78%) did not pick any of these statements as relevant. Nine (16%) selected a subset of these statements; three participants (6%) selected all these statements (as well as all other statements in the trace), saying that «I need to track where the changed value is used and why these places lead to the failure» (P38) and «I checked the content to see how the variables were transformed» (P32). All such cases were for subjects #S3 and #S5, where the change led to the execution of code that was not executed in the old version of the program.

Part 2: Comparing Code Views. Consistently with Part 1 findings, when comparing code views, 84% of participants (46 of 55) ranked the view that contains contextual information as their top choice. The remaining nine participants (16%) ranked it as their second choice, with none ranking it third. In their justifications, many noted that lack of context «requires some time to guess what the failing input is» (P25), which is «not that suitable for a reviewer who did not write the code» (P47).

Four of the participants (7%) selected summarization made by InPreSS as their first choice and 21 (28%) as their second. They noted that «collapsing the logic into a single function call makes it easier to understand» (P16) and «[dual slices] might include extraneous information that is not immediately relevant to resolving the failure» (P1). Yet, five participants (9%) preferred slices produced by DualSlice as their first choice and 25 (46%) as a second as they «want to choose myself what I want to see. So an optional collapse would be more helpful for me» (P34).

To quantitatively compare three views, we used a simple scoring mechanism: two points for the first-place ranking, one for second place, and zero for last place. Figure 3 shows the scores for three views for each of the six subjects and for all subjects combined (labeled as "All"). The overall scores are 101 for the slices with context, compared with 35 and 29 for the slices produced by DualSlice and InPreSS, respectively. The results indicate that the vast majority of participants found contextual information important, while there is no clear consensus on whether and when to summarize the propagation of information, as done by InPreSS. The preference for a view that contains contextual information was statistically significant; however, we found no statistically significant differences between DualSlice and InPreSS rankings. More details about statistical tests can be found online [49].

Part 3: Textual Explanations. When comparing textual explanations (which either include or exclude descriptions of contextual and propagation statements), participants again preferred explanations that contain context, albeit to a smaller extent: 36 of the participants (65%) ranked the view with context as their first and 16 (29%) – as their second choice. The scores of explanations corresponding to the slices produced by InPreSS were significantly higher (Wilcoxon test, $p < 0.05$) than those for DualSlice because participants favored shorter textual explanations that abstract away unnecessary details.

While none of the participants preferred textual explanations of regression over code, more than half would like to see both code and textual explanations combined: «if I debug code that I'm unfamiliar with, I like as much explanation as possible, so both would be nice» (P34) and «ideally, I would like to view the explanation first, then look at the code for additional context» (P46).

Additional Feedback. In the open-ended questions, the most frequently discussed issue (23 participants, 42%) was the integration of the provided information into an IDE: «If it were integrated into the IDE, I would likely use it» (P5). This aligns with earlier feedback about recommendation tools [38], where participants emphasized that to be useful, such tools must be embedded into their natural workflows.

Overall, developers appreciated support directing their attention to parts of the code that are more likely to be relevant to the failure but emphasized the need to have access to the entire code on demand. Some study participants indicated that simultaneous code and textual explanations would be effective. Others pointed to a preference for on-demand explanations that do not clutter the code view but are available, when needed. Interestingly, participants also suggested to use textual explanations for summarized blocks. This feedback and suggestions call for follow-up usability studies on the desired integration into IDEs, e.g., how to visualize different types of statements (via colors, highlighting, "collapsing" code parts, etc.) and what format of the additional visual and contextual cues is needed, which extends beyond the scope of this paper.

Summary: In a variety of experiments, we observed that the vast majority of participants found some contextual information critically important for debugging regression failures. While slice summarization performed by InPreSS is beneficial for some cases, several participants favored access to the entire code. Finally, participants favored textual explanations that contain contextual information and preferred integrating textual and code views.

5 CoReX: Context-Aware Refinement-Based Slicing

Our study showed that contextual information is essential for understanding and debugging regression failures. Yet, it showed that not all contextual information (e.g., not all empty squares in Figure 1) is equivalently important. We also observed that developers seek flexibility to decide whether and when to view certain

```
Inputs: $P_1, P_2, T_c$
1: $\mathbb{T}_{P_1}, \mathbb{T}_{P_2} \leftarrow$ runTest($P_1, P_2, T_c$)
2: $M_{P_1P_2}, UM_{P_1}, UM_{P_2} \leftarrow$ alignTraces($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}$)
3: $M^{=}_{P_1P_2}, M^{\neq}_{P_1P_2} \leftarrow$ analyzeMatched($M_{P_1P_2}$)
4: $\mathbb{S}_{P_1}, \mathbb{S}_{P_2} \leftarrow$ getSlice($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}, UM_{P_1}, UM_{P_2}, M^{\neq}_{P_1P_2}$)
Outputs: $\mathbb{S}_{P_1}, \mathbb{S}_{P_2}$
```

(a) DualSlice.

```
Inputs: $P_1, P_2, T_c$
1: $\mathbb{T}_{P_1}, \mathbb{T}_{P_2} \leftarrow$ runTest($P_1, P_2, T_c$)
2: $M_{P_1P_2}, UM_{P_1}, UM_{P_2} \leftarrow$ alignTraces($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}$)
3: $M^{=}_{P_1P_2}, M^{\neq}_{P_1P_2} \leftarrow$ analyzeMatched($M_{P_1P_2}$)
4: $\mathbb{S}_{P_1}, \mathbb{S}_{P_2} \leftarrow$ getSlice($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}, UM_{P_1}, UM_{P_2}, M^{\neq}_{P_1P_2}$)
5: $\Delta \leftarrow$ extractChanges($P_1, P_2$)
6: $\mathbb{S}^{sum}_{P_1}, \mathbb{S}^{sum}_{P_2} \leftarrow$ summarizeSlices($\mathbb{S}_{P_1}, \mathbb{S}_{P_2}$)
Outputs: $\mathbb{S}^{sum}_{P_1}, \mathbb{S}^{sum}_{P_2}$
```

(b) InPreSS.

```
Inputs: $P_1, P_2, T_c$
1: $\mathbb{T}_{P_1}, \mathbb{T}_{P_2} \leftarrow$ runTest($P_1, P_2, T_c$)
2: $M_{P_1P_2}, UM_{P_1}, UM_{P_2} \leftarrow$ alignTraces($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}$)
3: $M^{=}_{P_1P_2}, M^{\neq}_{P_1P_2} \leftarrow$ analyzeMatched($M_{P_1P_2}$)
4: $\mathbb{S}_{P_1}, \mathbb{S}_{P_2} \leftarrow$ getSlice($\mathbb{T}_{P_1}, \mathbb{T}_{P_2}, UM_{P_1}, UM_{P_2}, M^{\neq}_{P_1P_2}, M^{=}_{P_1P_2}$)
5: $\Delta \leftarrow$ extractChanges($P_1, P_2$)
6: $\mathbb{S}^{sum'}_{P_1}, \mathbb{S}^{sum'}_{P_2} \leftarrow$ summarizeSlices++($\mathbb{S}_{P_1}, \mathbb{S}_{P_2}$)
Outputs: $\mathbb{S}^{sum'}_{P_1}, \mathbb{S}^{sum'}_{P_2}$
```

(c) CoReX.

Figure 4: Overview of Dual-version Slicing Techniques.

pieces of code. The last observation is also consistent with prior studies, showing that developers often analyze and debug failures iteratively [30, 34, 37], first quickly skimming through code and information flows to determine which parts are relevant to the failure and then following with a more in-depth investigation.

Based on these observations, we design a novel dual-version slicing approach named CoReX, which refines and extends design decisions made by prior work, to better align with developers' needs. Figure 4 shows the high-level workflow of DualSlice, InPreSS, and CoReX and emphasizes the main differences between these approaches and the novel ideas behind CoReX, which we discuss next.

All three approaches obtain the same input: two versions of a program, P_1 and P_2, and a test case T_c, which passes in P_1 and fails in P_2. They then run the test on each version of the program, to produce execution traces $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$ (step 1), perform trace alignment to identify matched and unmatched statements (step 2), and further divide matched statements into those with the same and those with different data values (step 3), as discussed in Section 3.2.

We denote by $M_{P_1P_2}$ the set of matched trace statements, which are further divided into $M^{=}_{P_1P_2}$ and $M^{\neq}_{P_1P_2}$: statements with the same and different data values. In fact, $M^{=}_{P_1P_2}$ are statements that we informally refer to as context earlier in the paper. We denote by UM_{P_1} and UM_{P_2} the unmatched statements in $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$, respectively. In Figure 1, $M^{=}_{P_1P_2}$ and $M^{\neq}_{P_1P_2}$ statements are represented by the empty and filled squares. The unmatched statements are represented by triangles. The grayed-out lines, e.g., s^1_{12} in Figure 1b, represent source code statements executed in one but not the other program version and thus are not part of that version's trace.

Slicing. The first major difference between CoReX and both DualSlice and InPreSS lies in how they compute a synchronized slice over the traces (step 4): while both DualSlice and InPreSS do not include $M^{=}_{P_1P_2}$ statements in their produced slices, CoReX does, as shown in line 4 of Figure 4c. For our running example, the slice produced by CoReX in this step will, in fact, include the entire trace in Figure 1, while slices produced by the two other tools will exclude statements s^1_2, s^1_3, s^1_4, s^1_5, s^1_6, s^1_7, s^1_{15}, s^1_{17}, and s^1_{19}.

At this stage, DualSlice stops and outputs the produced synchronized slice, while both InPreSS and CoReX proceed to their corresponding slice summarization steps.

Slice Summarization. As discussed in Section 3.2, the key idea behind InPreSS is to summarize certain statements of the slice in the form of high-level input-output functions, focusing specifically on abstracting blocks of code that are unchanged between the two versions of the program (steps 5 and 6 in Figure 4b). The rationale behind such a summarization is to prioritize statements that developers inspect first, assuming that common code propagates the effects of changes but is not directly responsible for the failure.

A straightforward idea at this stage would be to apply the original InPreSS summarization algorithm on the slice augmented with $M^{=}_{P_1P_2}$ statements. We refer to such a solution as CoReX^I – an initial version of CoReX – and show its output in Figure 5a. In this example, statement instances s^1_{15}-s^1_{20}, which propagate the changed value of the variable `year` through numerous calculations identical between the versions, are summarized (as they are less important to understand, at least for the "first debugging pass"). Statement instances s^1_2-s^1_7 are kept because each defines a variable used later in the code (i.e., they are the context statements).

However, according to the study in Section 4, not all contextual statements are equivalently important to developers. In fact, the majority of our study participants have not selected statements like s^1_5-s^1_7 as important. More specifically, out of nine developers who received this example in their questionnaire, five picked all three statements s^1_2-s^1_4 as relevant and three picked statements s^1_2 and s^1_4; none of the nine developers picked any of the statements s^1_5-s^1_7.

Analyzing the intuition behind this selection, we define the summarization approach used by CoReX (step 6 in Figure 4c) to better distinguish between *primary* statements, which are essential for understanding the failure during the "first scan" and should be kept in the slice as-is and *secondary* statements, which are potentially relevant but not immediately crucial and thus should be summarized. Unlike InPreSS or CoReX^I, when calculating input-output summarization functions, CoReX checks whether an input variable v used in the summarized block is defined in a statement s from the $M^{=}_{P_1P_2}$ set. If so, it effectively "moves" s inside the block, treating it as a secondary statement; consequently, it also does not include v in the signature of the summary function. Effectively, it treats as primary the transitive closure of $M^{=}_{P_1P_2}$ statements that flow into the *changed* statements and the assertion itself (these primary statements are s^1_2-s^1_4 in our example); it treats as secondary the transitive closure of $M^{=}_{P_1P_2}$ statements flowing into summarized blocks (statements s^1_5-s^1_7). The output (primary statements) produced by CoReX on our running example is shown in Figure 5b. Unlike DualSlice and InPreSS, which do not include statements s^1_2-s^1_4, CoReX includes these statements in its output. Unlike CoReX^I, which keeps statements s_5^1-s_7^1 as primary, CoReX deems them as secondary, making it more aligned with the developers' intention.

Implementation Details. CoReX is implemented for Java version 8. It utilizes the InPreSS implementation, which is based on the trace alignment algorithm from ERASE [50] and Slicer4J Java slicer [4].

6 Evaluation

We quantitatively evaluate CoReX by comparing it with other dual-version slicing approaches, DualSlice and InPreSS; we also conduct an internal evaluation by comparing it with CoReX^I. Our evaluation is driven by the following research questions:

RQ1 (Prioritization Ratio): What is the size of the slice and the prioritization ratio achieved by each tool?

RQ2 (Execution Time): What is the execution time of each tool?

RQ3 (Alignment): How well do the produced slices match the developers' preferences?

The goal of the first question is to investigate, on a large number of real-world regressions, how effective the filtering implemented by slicing is, compared with the size of the original execution trace. The goal of the second question is to assess the tools' performance. Finally, in our third question, we aim to assess the ability of the tools to produce not only concise but also relevant slices, w.r.t. the preferences of the 55 developers who participated in the study described in Section 4. To the best of our knowledge, this is the first evaluation that focuses on the accuracy of the tools, in addition to the prioritization ratio that they achieve.

6.1 Methods and Metrics

To answer **RQ1** and **RQ2**, we borrow the InPreSS evaluation dataset, which includes 278 failures from six projects of the popular Defects4J benchmark [26] and 8 large client-library project pairs with upgrade failures from LibRench. Table 2 shows the list of projects and the average lines of code (LoC) for each individual subject within a project: both the passing and failing versions of the code for a subject. The full list of subjects is available online [49].

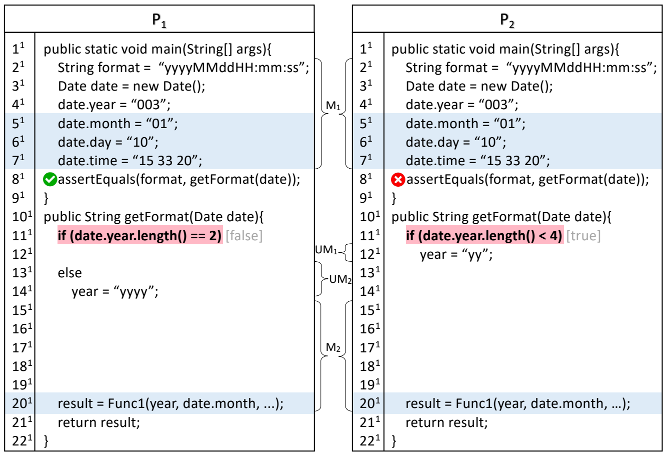
Table 2: RQ1 and RQ2 Evaluation Subjects

ID	Project	#Failures	Avg. LoC	
			P_1	P_2
Defects4J				
D1	JFree**Chart**	23	96,522	96,517
D2	**Closure** Compiler	95	90,604	90,601
D3	Commons **Lang**	49	22,756	22,750
D4	Commons **Math**	63	85,623	85,617
D5	**Mockito**	26	37,281	37,277
D6	Joda-**Time**	22	28,428	28,422
Avg.	-	-	60,202	60,197
LibRench	Client-Libraries	8	171,498	172,426
All Avg.	-	-	115,848	116,314

For **RQ1**, for each subject and tool, we calculate the prioritization ratio (Rat.) – a metric commonly used by existing techniques as: $\frac{\#\mathbb{T}-\#Tool}{\#\mathbb{T}}$, where $\#\mathbb{T}$ is the size of the trace and #Tool is the size of the slice produced by each tool.

For **RQ2**, we report the average execution time for each tool, where each case is executed five times. Our experiments are conducted on an Ubuntu 18.04.4 Virtual Machine with 4 cores and 32 GB of RAM, running on an in-house Ubuntu server with 64 cores and 512 GB of memory.

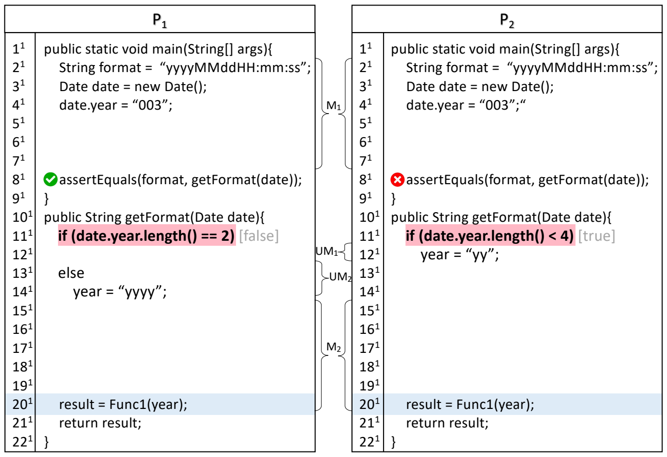
To answer **RQ3**, we re-consider code samples used in the study described in Section 4 (see Table 1) and check how well the automated heuristics applied by CoReX fit the developers' expectations, compared with those of other dual-version slicing tools. To this end, for each sample, we deem as *Relevant* statements selected by at least 50% of the participants who evaluated this subject (i.e., at least five participants). This is done to ensure a reasonable consensus between developers and exclude outliers. We then calculate the *Precision* (P), *Recall* (R), and *F-Measure* (F) for each tool, w.r.t. their ability to include in the slice statements deemed relevant by the developers. Specifically, Precision measures the proportion of relevant statements included in the slice out of all slice statements: $P = \frac{\#(Relevant \cap Tool)}{\#Tool} \times 100$. Recall measures the proportion of relevant statements included in the slice out of all relevant statements: $R = \frac{\#(Relevant \cap Tool)}{\#Relevant} \times 100$. F-measure is a harmonic mean aiming



```
P1
1^1  public static void main(String[] args){
2^1    String format = "yyyyMMddHH:mm:ss";
3^1    Date date = new Date();
4^1    date.year = "003";
5^1    date.month = "01";
6^1    date.day = "10";
7^1    date.time = "15 33 20";
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1   if (date.year.length() == 2) [false]
12^1
13^1   else
14^1     year = "yyyy";
15^1
16^1
17^1
18^1
19^1
20^1   result = Func1(year, date.month, ...);
21^1   return result;
22^1 }

P2
1^1  public static void main(String[] args){
2^1    String format = "yyyyMMddHH:mm:ss";
3^1    Date date = new Date();
4^1    date.year = "003";
5^1    date.month = "01";
6^1    date.day = "10";
7^1    date.time = "15 33 20";
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1   if (date.year.length() < 4) [true]
12^1     year = "yy";
13^1
14^1
15^1
16^1
17^1
18^1
19^1
20^1   result = Func1(year, date.month, ...);
21^1   return result;
22^1 }
```

(a) CoReX^I for P_1 and P_2.



```
P1
1^1  public static void main(String[] args){
2^1    String format = "yyyyMMddHH:mm:ss";
3^1    Date date = new Date();
4^1    date.year = "003";
5^1
6^1
7^1
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1   if (date.year.length() == 2) [false]
12^1
13^1   else
14^1     year = "yyyy";
15^1
16^1
17^1
18^1
19^1
20^1   result = Func1(year);
21^1   return result;
22^1 }

P2
1^1  public static void main(String[] args){
2^1    String format = "yyyyMMddHH:mm:ss";
3^1    Date date = new Date();
4^1    date.year = "003";"
5^1
6^1
7^1
8^1  assertEquals(format, getFormat(date));
9^1  }
10^1 public String getFormat(Date date){
11^1   if (date.year.length() < 4) [true]
12^1     year = "yy";
13^1
14^1
15^1
16^1
17^1
18^1
19^1
20^1   result = Func1(year);
21^1   return result;
22^1 }
```

(b) CoReX for P_1 and P_2.

Figure 5: Comparing Slices Produced by CoReX^I and CoReX for the Example in Figure 1.

Table 3: Prioritization Ratio

ID	$\mathbb{T}$	Sync.Slice		DualSlice		InPreSS		CoReX^I		CoReX	
		Size	Rat.	Size	Rat.	Size	Rat.	Size	Rat.	Size	Rat.
Defects4J											
D1	5,913	2,506	57.62	61	98.97	7	99.88	32	99.46	19	99.69
D2	92,663	45,796	50.58	3,341	96.39	164	99.82	779	99.16	395	99.57
D3	3,072	691	77.49	43	98.60	8	99.74	117	96.20	75	97.57
D4	8,801	2,812	68.04	738	91.61	39	99.55	165	98.13	121	98.63
D5	3,213	1,278	60.22	377	88.25	26	99.18	85	97.36	48	98.51
D6	13,064	5,312	59.34	519	96.03	31	99.77	84	99.36	50	99.62
Avg.	36,025	17,156	52.38	1,398	96.12	72	99.80	341	99.05	185	99.49
LibRench											
Avg.	31,743	13,782	56.58	2,618	91.75	342	98.92	1,593	94.98	1,145	96.39
All											
Avg.	33,884	15,469	54.35	2,007	94.07	207	99.39	967	97.15	665	98.04

to balance between precision and recall, calculated as F = $2 \times \frac{P \times R}{P+R}$. Precision, Recall, and F-Measure calculated for each developer's selection individually are in our online appendix [49].

6.2 Results

RQ1: Prioritization Ratio. Table 3 shows the sizes of the trace, the synchronized slice Sync.Slice, and the slice produced by each of the compared techniques. Here, we focus only on primary statements produced by InPreSS, CoReX^I, and CoReX, to estimate the "first pass" effort. By design, the combination of primary and secondary statements for both CoReX^I and CoReX includes the entire slice, as one of our main observations is that developers do not wish any of the statements to be completely removed.

The table also shows the prioritization ratio for each technique when compared with the size of the trace. The metrics in the table are aggregated for all Defects4J faults in the same project and across the eight projects in the LibRench benchmark, for both the old and the new version of the program, combined. The last row of the table shows the aggregated results for all projects combined. All raw results from our evaluation are available online [49].

CoReX achieves a relatively high prioritization ratio for all subject programs: 98.04% on average. This means that, for an average project, a developer would only need to inspect around 665 execution steps during a debugging session instead of 33,884.

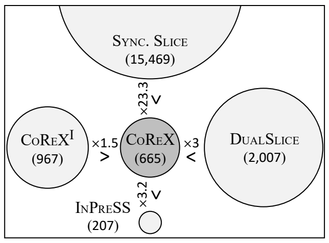


Figure 6: Slice Sizes.

CoReX produces slices that are x23.3 shorter than the full synchronized slice (665 vs. 15,469 statements, on average), attesting to its ability to reduce debugging overhead, i.e., when slicing is integrated into the debugging workflows of IDEs and slices are manually inspected by the developers during an interactive debug session. Moreover, despite including contextual statements, it produces slices that are x3 shorter than those of DualSlice (665 vs. 2,007 statements, on average) and only x3.2 longer than those of InPreSS (665 vs. 207 statements, on average), placing it "in the middle area" between these two tools. Finally, CoReX's slices are x1.5 shorter than those of CoReX^I (665 vs. 967 statements, on average), demonstrating the presence of a large number of secondary contextual statements in real-world projects (between 1 and 2,677 statements).

Table 4: Execution Time (Minutes)

Benchmark	DualSlice		InPreSS		CoReX^I		CoReX	
	Defects4J	LibRench	Defects4J	LibRench	Defects4J	LibRench	Defects4J	LibRench
Slice Sync.	24.91	364.94	24.91	364.94	39.25	398.06	39.25	398.06
Slice Sum.	-		10.99	59.15	19.84	97.36	15.69	88.16
Total	24.91	364.94	35.9	424.09	59.09	495.42	54.94	486.22

RQ2: Execution Time. Table 4 shows the runtime measurements for the four tools. We separate the tools' runtime into that for the synchronized slicing (steps 1-4 in Figure 4) and slice summarization (steps 5-6). We report the results separately for Defects4J and LibRench, as projects within these benchmarks are of comparable size and complexity.

CoReX's overall runtime is x2.21 longer than that of DualSlice and x1.53 longer than that of InPreSS for the Defects4J projects. For larger LibRench projects, the difference is less pronounced: x1.33 and x1.15 for DualSlice and InPreSS, respectively. The longer execution time is because CoReX has to process longer slices, to include statements from the $M^{=}_{P_1P_2}$ set.

Looking into details, we observe that the majority of execution time is spent in the synchronized slicing stage, especially for larger LibRench projects (our approach inherits performance bounds of Slicer4J, which we use as the underlying slicing implementation). The overhead added by the summarization step on top of the slicing time is relatively minor and is comparable in CoReX and InPreSS (14% for InPreSS vs. 18% for CoReX on LibRench projects, as CoReX is processing longer slices). Moreover, while the synchronized slice is, by definition, identical for CoReX and CoReX^I, CoReX summarization time is shorter as it excludes a number of $M^{=}_{P_1P_2}$ statements from the block inputs, eliminating the need to build additional blocks containing definitions of variables in these statements.

In practice, the CoReX execution time can be accommodated by integrating the most time-consuming parts of the tool, i.e., trace alignment and slicing, into the testing workflow. Specifically, these steps could be performed offline, as soon as a test fails. The resulting slices could then be reused at any time in the follow-up analysis, as necessary.

RQ3: Alignment. Table 5 shows the alignment values for each tool w.r.t. the developer's preferences for the six subjects from the preliminary study. Our goal with this analysis is to "close the loop" and confirm that the algorithm embedded in CoReX can indeed successfully identify the contextual statements that developers deemed important.

Overall, CoReX achieved the highest F-measure (93%) across all subjects, with high precision (90%) and recall (98%), attesting to its ability to identify the majority of statements deemed relevant by the developers (R) while returning few irrelevant statements (P). In fact, CoReX achieved a perfect recall of 100% in all but one subject, #S5, where it deems as secondary a divergent execution statement, which, albeit being unchanged between the program versions, is executed in one version but not in the other. Future work should look at whether and how to deal with such divergent execution statements, given that not all of them are marked as relevant by developers.

Table 5: Alignment with Developers' Selections

Sub. ID	Sync.Slice			DualSlice			InPreSS			CoReX^I			CoReX		
	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)
#S1	43	100	60	67	67	67	100	67	80	75	100	86	100	100	100
#S2	39	100	56	71	71	71	100	71	83	78	100	88	100	100	100
#S3	37	100	54	50	86	63	67	57	62	64	100	78	70	100	82
#S4	42	100	59	50	60	55	80	80	80	71	100	83	83	100	91
#S5	64	100	78	56	71	63	80	57	67	75	86	80	86	86	86
#S6	39	100	56	57	57	57	100	57	73	78	100	88	100	100	100
Avg.	44	100	60	59	69	63	88	65	74	73	98	84	90	98	93

CoReX's recall matches that of CoReX^I, as both tools retain contextual statements deemed relevant by the developers and also summarize secondary statements. However, CoReX^I includes additional irrelevant contextual statements flowing into secondary statements, which reduces its precision to 73%. As InPreSS retains fewer statements, it achieves a relatively high precision (88%), but it sacrifices recall (65%) by missing some relevant context. DualSlice achieves the lowest Precision and Recall of all tools in our study, followed only by the full synchronized slice itself, which retains all statements related to the failure, thus having perfect recall, but very low precision.

Notably, more than 50% of the context is considered important for effective debugging (Section 4.2) and CoReX can correctly identify the relevant 50%.

Summary: Our results indicate that CoReX achieves a high prioritization ratio, comparable with that of DualSlice and InPreSS. Evaluating the heuristics applied by CoReX on the expected results produced by developers during our study shows that CoReX can align with developers' expectations better than its competitors. Balancing the inclusion of relevant statements while minimizing the inclusion of irrelevant ones comes at the expense of an increase in the execution time. We believe such an increase is acceptable, considering the benefits of the tool.

7 Limitations and Threats to Validity

Limitations. The main limitation of our approach is its command-line nature, which makes the produced traces difficult to inspect. While this limitation is shared with other slicing techniques, our study includes preliminary discussions on how such tools could be integrated into IDEs to improve usability. A large user study evaluating such integration would be useful. However, performing such a study in a meaningful way is a non-trivial effort: one first needs to engage a number of developers who are familiar with big projects to-be-used as our case studies, making sure the developers for each project have comparable skills (for the study and control groups). Then, integrating CoReX with the IDE debugger that the developers are familiar with is necessary. We thus leave conducting such a study, on projects of considerable size and complexity, for separate future work.

Additional limitations of CoReX are inherited from the limitations of the underlying infrastructure: Soot and ERASE cannot support Java versions beyond 9 and 8, respectively. Moreover, while CoReX is not inherently restricted from analyzing executions of concurrent programs, it relies on ERASE for trace recording and alignment, which does not support concurrency.

External validity. As in many user studies in software engineering, our research is inductive in nature and thus might not generalize beyond our study context. Yet, the study includes a large and diverse set of participants coming from varied geographical, educational, and professional backgrounds. This gives us confidence that our study represents central and significant views.

Another threat to validity concerns the representativeness of the program subjects. While we had to minimize their size to allow the study participants to spend substantial time analyzing different code versions in detail, the subjects were heavily inspired by real faults from both the popular Defects4J dataset and the LibRench dataset of large real software systems. Understanding these programs did not require domain-specific knowledge or advanced programming skills, making them suitable for evaluating the impact of debugging tools and code comprehension. We thus believe our subject selection is representative and suitable for this study.

Similarly, the results of our tool evaluation (Section 6) may be influenced by the subject selection and may not generalize to other subjects not included in our study. Again, to mitigate this threat, we relied on a popular externally created dataset from prior work: Defects4J and LibRench. We believe that considering 286 case studies from 14 projects of significant size and complexity makes our findings reliable.

Internal validity. To ensure the validity and effectiveness of our user study, we conducted a pilot study and included several validation questions. These steps were taken to identify potential issues in the study design, such as question ambiguity, and to refine the questionnaire. As a related threat, we might have misinterpreted participants' answers or ideas they expressed. To mitigate this threat, we made sure two authors of the paper performed data analysis independently.

Finally, our evaluation of the tools' accuracy and alignment with developers' expectations relies on the data collected in the study. A follow-up external evaluation of our results on a set of different subjects might be beneficial. To encourage such validation and replication of our results, we have made all experimental data and the tool implementation publicly available [49].

8 Related Work

The most related to ours are other slicing-based fault localization techniques and user studies evaluating these techniques. We also discuss additional fault localization approaches and their user studies. Finally, we discuss studies on developers' debugging practices.

Slicing-based Fault Localization. Following Korel and Laski's introduction of dynamic slicing [32], numerous slicing-based fault localization techniques have been proposed [73, 74]. Most focus on single-version programs and propose slicing variants, such as chopping [16], dicing [56], relevant slicing [17], amorphous slicing [18, 19], and memory-address dependence slicing [35], aiming to minimize the number of statements developers need to inspect. Dual-version slicing techniques, namely DualSlice [48, 50, 52] and InPreSS [6] are already extensively discussed in this paper, as well as how our approach differs.

User Studies Investigating Slicing-based Fault Localization. A number of earlier studies were conducted, albeit with small programs and a small number of participants [13, 30, 34, 56]. Specifically, Weiser and Lyle [56] made the first comparison of debugging with and without slicing, finding no significant improvement. In early 2001, Francel and Spencer [13] reported improved performance with slicing during debugging. Ko and Myers [29] found that the slicing-based Whyline tool they developed helped developers locate bugs faster. These studies focused on a single program version; there remains a lack of research on the usefulness of slicing in regression scenarios, which our paper addresses.

Non-slicing-based Fault Localization Approaches. Spectrum-based fault localization (e.g., Tarantula [25] and other related approaches [1, 39, 61, 62, 67]) uses probabilistic models to rank program statements based on test results, identifying suspicious components involved in failures. Improvements to the spectrum-based fault localization methods integrate program analysis techniques, including program slicing [36, 45, 59], to refine statement rankings. Yet, these tools still focus on providing a ranked list of program statements, without identifying the dependencies between them. Other approaches rely on additional inputs, such as bug reports [58] or version histories [57], framing fault localization as an information retrieval task. Techniques based on delta debugging [69, 71] isolate failure-inducing changes (faults) by systematically reverting subsets of changes between the correct version and the faulty version of the program to identify the smallest subset of responsible changes (faults). While all these approaches primarily target localization by identifying suspicious statements, our goal is to build a comprehensive slice guiding the developer through a step-by-step inspection of the failure.

Another line of work focuses on explaining the differences between the two executions, i.e., via trace comparison [20, 22, 24] and symbolic analysis [41, 42, 66]. As this work does not focus specifically on the failure-related analysis, our approach is both orthogonal and complementary: we aim not only to find differences between executions but also to identify the parts specifically relevant to the failure.

More recently, several LLM-based fault localization techniques have been proposed. In a nutshell, they collect relevant information from the generally-large code base (e.g., Widyasari et al. [60] rely on spectrum-based fault localization for this purpose) and then provide this information as input to an LLM, together with an appropriately crafted prompt, to generate fault explanations. These approaches are mostly orthogonal and complementary to ours. In fact, prior studies show that developers rely on semantic dependencies between statements when reasoning about code [40]. Instead of relying on LLM to "figure out" such semantic dependencies, future work could explore using CoReX output as an alternative input to an LLM in this setting.

User Studies Investigating Non-slicing-based Fault Localization Approaches. Parnin and Orso [40], and subsequent studies [15, 63, 64], found that simply presenting suspicious statements without context or dependencies is not enough to aid developers as they need explanations for the ranking. Similar conclusions were drawn about the effectiveness of delta debugging [23, 68]. Wang et al. [51] explored the effectiveness of information retrieval-based techniques, finding that while more detailed bug reports helped developers identify faulty files faster, they did not improve the overall fault localization process.

User Studies Investigating Developers' Debugging Practices and Expectations. Gilmore [14] argued that comprehension and debugging are interconnected and developers engage in simultaneous code search, dependency analysis, and information gathering during debugging [30]. In fact, developers perform systematic analyses to track variable values to explain crashes [9], reflecting slicing principles. Kochhar et al.[31] surveyed 386 practitioners and found that practitioners expect a fault localization technique to satisfy some criteria in terms of debugging data availability, granularity level, trustworthiness (reliability), scalability, efficiency, ability to provide rationale, and IDE integration. Recently, Soremekun et al. [44] found that many developers view lab-based fault localization assumptions as unsuitable for real-world debugging. We relied on these findings and proposed a solution that aligns with the developers' mental model and provides the information needed to understand the code.

9 Conclusion

In this paper, we investigated the usefulness of existing dual-version slicing-based techniques, namely DualSlice and InPreSS, in focusing developers' attention on a subset of program statements relevant to troubleshooting a regression failure. By conducting a large-scale questionnaire study with 55 participants from eight different countries, we identified the main limitations of the existing techniques and proposed a new slicing approach, called CoReX, which addresses these limitations. Our quantitative evaluation of CoReX on more than 280 programs from the Defects4J and LibRench benchmarks shows that its effectiveness in terms of prioritization ratio is comparable to those of DualSlice and InPreSS. Yet, it achieves a better alignment with developers' expectations by selecting statements deemed relevant by developers. We believe our work will inspire further research in this area, including future developer studies, and will help promote the efficient integration of slicing-based techniques into debugging workflows.

10 Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

11 Data Availability

To support further work in this area, detailed information about our case studies and their analysis, as well as our implementation of CoReX, are available online [49].

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proc. of the International Symposium on Dependable Computing (PRDC)*. 39–46.

[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic Program Slicing. *ACM SIGPLAN Notices* 25, 6 (1990), 246–256.

[3] Hiralal Agrawal, Joseph Robert Horgan, Edward W Krauser, and Saul A London. 1993. Incremental Regression Testing. In *Proc. of the Conference on Software Maintenance (ICSM)*. 348–357.

[4] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Slicer4J: A Dynamic Slicer for Java. In *Proc. of the International European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1570–1574.

[5] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-localization Using Dynamic Slicing and Change Impact Analysis. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 520–523.

[6] Sahar Badihi, Khaled Ahmed, Yi Li, and Julia Rubin. 2023. Responsibility in Context: On Applicability of Slicing in Semantic Regression Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 563–575.

[7] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *Proc. of the International Conference on Software Engineering (ICSE)*. 572–583.

[8] David Binkley. 2002. An Empirical Study of the Effect of Semantic Differences on Programmer Comprehension. In *Proc. of the International Workshop on Program Comprehension (ICPC)*. 97–106.

[9] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? an Experiment with Practitioners. In *Proc. of the International European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 117–128.

[10] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce Before You Localize: Delta-debugging and Spectrum-based Fault Localization. In *Proc. of the International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 184–191.

[11] Defects4J Benchmark. 2025. https://github.com/rjust/defects4j/tree/v1.5.0.

[12] Margaret Ann Francel and Spencer Rugaber. 1999. The Relationship of Slicing and Debugging to Program Understanding. In *Proc. of the International Workshop on Program Comprehension (ICPC)*. 106–113.

[13] Margaret Ann Francel and Spencer Rugaber. 2001. The Value of Slicing While Debugging. *Science of Computer Programming* 40, 2-3 (2001), 151–169.

[14] David J Gilmore. 1991. Models of Debugging. *Acta psychologica* 78, 1-3 (1991), 151–172.

[15] Carlos Gouveia, José Campos, and Rui Abreu. 2013. Using HTML5 Visualizations in Software Fault Localization. In *Proc. of the Working Conference on Software Visualization (VISSOFT)*. 1–10.

[16] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating Faulty Code Using Failure-inducing Chops. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 263–272.

[17] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. 1999. An Efficient Relevant Slicing Method for Debugging. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*. 303–321.

[18] Mark Harman, David Binkley, and Sebastian Danicic. 2003. Amorphous Program Slicing. *Journal of Systems and Software (JSS)* 68, 1 (2003), 45–64.

[19] Mark Harman and Sebastian Danicic. 1997. Amorphous Program Slicing. In *Proc. of the International Workshop on Program Comprehension (IWPC)*. 70–79.

[20] Kevin J Hoffman, Patrick Eugster, and Suresh Jagannathan. 2009. Semantics-aware Trace Analysis. *ACM Sigplan Notices* 44, 6 (2009), 453–464.

[21] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. 2020. Experiments with Interactive Fault Localization Using Simulated and Real Users. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. 290–300.

[22] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proc. of the International Conference on Software Engineering (ICSE)*. 87–99.

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proc. of the International Conference on Software Engineering (ICSE)*. 672–681.

[24] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *IEEE Symposium on Security and Privacy (S&P)*. 347–362.

[25] James A Jones, Mary Jean Harrold, and John T Stasko. 2001. Visualization for Fault Localization. In *Proc. of the International Conference on Software Engineering Workshop on Software Visualization (ICSE-SV)*.

[26] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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)*. 437–440.

[27] Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. 2020. When Does My Program Do This? Learning Circumstances of Software Behavior. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*. 1228–1239.

[28] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering (ESE)* 20 (2015), 110–141.

[29] Amy J Ko and Brad A Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proc. of the International Conference on Software Engineering (ICSE)*. 301–310.

[30] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering (TSE)* 32, 12 (2006), 971–987.

[31] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 165–176.

[32] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163.

[33] Jens Krinke. 2004. Slicing, Chopping, and Path Conditions with Barriers. *Software Quality Journal* 12, 4 (2004), 339–360.

[34] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. 2002. Experimental Evaluation of Program Slicing for Fault Localization. *Empirical Software Engineering (ESE)* 7, 1 (2002), 49–76.

[35] Xiangyu Li and Alessandro Orso. 2020. More Accurate Dynamic Slicing for Better Supporting Software Debugging. In *Proc. of the International Conference on Software Testing, Validation and Verification (ICST)*. 28–38.

[36] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based Statistical Fault Localization. *Journal of Systems and Software (JSS)* 89 (2014), 51–62.

[37] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic Generation of Natural Language Summaries for Java Classes. In *Proc. of the International Conference on Program Comprehension (ICPC)*. 23–32.

[38] Emerson Murphy-Hill and Gail C Murphy. 2013. Recommendation Delivery: Getting the User Interface Just Right. In *Proc. of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 223–242.

[39] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 3 (2011), 1–32.

[40] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 199–209.

[41] Fabrizio Pastore, Leonardo Mariani, and Alberto Goffi. 2013. RADAR: A Tool for Debugging Regression Problems in C/C++ Software. In *Proc. of International Conference on Software Engineering (ICSE), Tool Demonstration Track*. 1335–1338.

[42] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. Darwin: An Approach to Debugging Evolving Programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 3 (2012), 1–29.

[43] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2006. Sieve: A Tool for Automatically Detecting Variations Across Program Versions. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 241–252.

[44] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Mike Papadakis. 2023. Evaluating the Impact of Experimental Assumptions in Automated Fault Localization. In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 159–171.

[45] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. 2021. Locating Faults with Program Slicing: An Empirical Analysis. *Empirical Software Engineering (ESE)* 26, 3 (2021), 1–45.

[46] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proc. of the International Conference on Programming Language Design and Implementation (PLDI)*. 112–122.

[47] William N Sumner and Xiangyu Zhang. 2010. Memory Indexing: Canonicalizing Addresses Across Executions. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*. 217–226.

[48] William N Sumner and Xiangyu Zhang. 2013. Comparative Causality: Explaining the Differences Between Executions. In *Proc. of the International Conference on Software Engineering (ICSE)*. 272–281.

[49] Supplementary Materials. 2025. https://resess.github.io/artifacts/CoReX.

[50] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining Regressions via Alignment Slicing and Mending. *IEEE Transactions on Software Engineering (TSE)* (2019).

[51] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 1–11.

[52] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing Concurrency Bugs Using Dual Slicing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 253–264.

[53] Mark Weiser. 1979. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. *PhD thesis, University of Michigan* (1979).

[54] Mark Weiser. 1982. Programmers Use Slices when Debugging. *Commun. ACM* 25, 7 (1982), 446–452.

[55] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering (TSE)* 4 (1984), 352–357.

[56] Mark Weiser and Jim Lyle. 1986. Experiments on Slicing-based Debugging Aids. In *Workshop on Empirical Studies of Programmers (ESP)*. 187–197.

[57] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical Spectrum-Based Fault Localization. *IEEE Transactions on Software Engineering (TSE)* 47, 11 (2019), 2348–2368.

[58] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 262–273.

[59] Wanzhi Wen, Bixin Li, Xiaobing Sun, and Jiakai Li. 2011. Program Slicing Spectrum-based Software Fault Localization. In *SEKE*. 213–218.

[60] Ratnadira Widyasari, Jia Wei Ang, Truong Giang Nguyen, Neil Sharma, and David Lo. 2024. Demystifying Faulty Code: Step-by-step Reasoning for Explainable Fault Localization. In *Proc. of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 568–579.

[61] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[62] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering (TSE)* 42, 8 (2016), 707–740.

[63] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 267–278.

[64] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *Proc. of the International Conference on Software Engineering (ICSE)*. 808–819.

[65] Bin Xin, William N Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. *ACM SIGPLAN Notices* (2008), 238–248.

[66] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. 2015. A Synergistic Analysis Method for Explaining Failed Regression Tests. In *Proc. of the International Conference on Software Engineering (ICSE)*, Vol. 1. 257–267.

[67] Shin Yoo. 2012. Evolving Human Competitive Spectra-based Fault Localization Techniques. In *Proc. of the International Symposium on Search Based Software Engineering (SSBSE)*. 244–258.

[68] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Towards Automated Debugging in Software Evolution: Evaluating Delta Debugging on Real Regression Bugs from the Developers' Perspectives. *Journal of Systems and Software (JSS)* 85, 10 (2012), 2305–2317.

[69] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 253–267.

[70] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*. 1–10.

[71] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-inducing Input. *IEEE Transactions on Software Engineering (TSE)* 28, 2 (2002), 183–200.

[72] Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang, Ziyuan Wang, and Lingming Zhang. 2022. Program Repair: Automated vs. Manual. *arXiv preprint arXiv:2203.05166* (2022).

[73] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2007. A Study of Effectiveness of Dynamic Slicing in Locating Real Faults. *Empirical Software Engineering (ESE)* 12 (2007), 143–160.

[74] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental Evaluation of Using Dynamic Slices for Fault Location. In *Proc. of the International Symposium on Automated Analysis-driven Debugging*. 33–42.

[75] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards Locating Execution Omission Errors. In *Proc. of the International Conference on Programming Language Design and Implementation (PLDI)*. 415–424.

[76] Jianjun Zhao. 2000. Dependence Analysis of Java Bytecode. In *Proc. of the International Computer Software and Applications Conference (COMPSAC)*. 486–491.