Responsibility in Context: On Applicability of Slicing in Semantic Regression Analysis

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

Khaled Ahmed
University of British Columbia, Canada
khaledea@ece.ubc.ca

Yi Li
Nanyang Technological University, Singapore
yi_li@ntu.edu.sg

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

***Abstract*—Numerous program slicing approaches aim to help developers troubleshoot regression failures – one of the most time-consuming development tasks. The main idea behind these approaches is to identify a subset of interdependent program statements relevant to the failure, minimizing the amount of code developers need to inspect. Accuracy and reduction rate achieved by slicing are the key considerations toward their applicability in practice: inspecting only the statements in a slice should be faster and more efficient than inspecting the code in full.**

In this paper, we report on our experiment applying one of the most recent and accurate slicing approaches, dual slicing, to the task of troubleshooting regression failures. As subjects, we use projects from the popular Defects4J benchmark and a systematically-collected set of eight large, open-source client-library project pairs with at least one library upgrade failure, which we refer to as LibRench. The results of our experiments show that the produced slices, while effective in reducing the scope of manual inspection, are still very large to be comfortably analyzed by a human. When inspecting these slices, we observe that most statements in a slice deal with the propagation of information between changed code blocks; these statements are essential for obtaining the necessary context for the changes but are not responsible for the failure directly.

Motivated by this insight, we propose a novel approach, implemented in a tool named INPRESS, for further reducing the size of a slice by accurately identifying and summarizing the propagation-related code blocks. Our evaluation of INPRESS shows that it is able to produce slices that are 76% shorter than the original ones (207 vs. 2,007 execution statements, on average), thus, reducing the amount of information developers need to inspect without losing the necessary contextual information.

***Index Terms*—Program slicing, slice minimization, regression failures, case study**

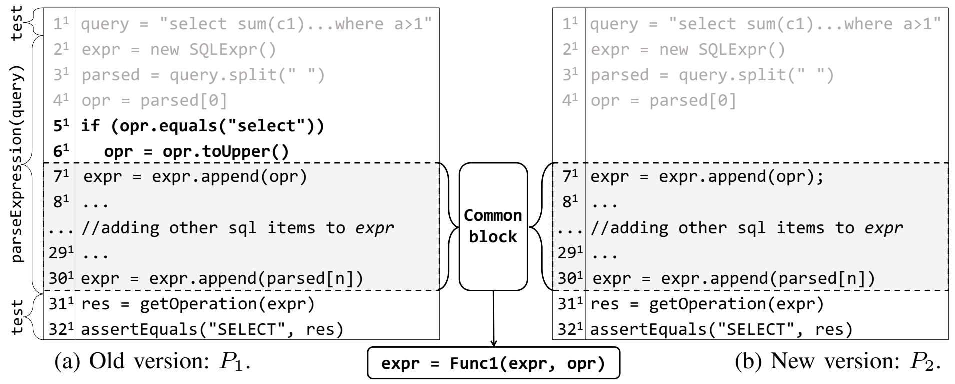
I. INTRODUCTION

Understanding and troubleshooting software regression failures is one of the most time-consuming development tasks, especially in large, production-level software systems. To help developers with this task, numerous change impact analysis and fault localization approaches have been proposed. Their goal is to focus the developers' attention on the minimal subset of program statements *relevant* to the failure. The majority of these approaches can be largely divided into spectrum-based [1], [2] and slicing-based [3], [4]. Unlike spectrum-based approaches that output a ranked list of statements relevant to the failure, slicing captures the dependencies and flow of information between the outputted statements, supporting the mental-model developers build when debugging failures [5], [6], [7]. For this reason, slicing is shown to be useful in locating the source of failures more easily [8], [9], [10].

The classic program slicing idea – to compute a subset of program statements that affect a particular program variable or statement of interest (the failure site, in our case) via control and data dependencies – is further extended by numerous slicing variants, such as dicing [5], chopping [11], slicing with barriers [12], and thin slicing [13]. These variants propose methods and heuristics to minimize the size of the slice developers need to inspect while ensuring it contains the relevant information needed to analyze the failure. Dual slicing [14], [15], [16] is probably one of the most recent and accurate dynamic slicing variants that was shown to be effective for identifying regression failures. It simultaneously analyses the base and regression versions of the program, keeping in the slice only execution statements that lead to diverging behavior between the versions.

The usefulness of the slicing approaches is typically evaluated on relatively small code samples [11], [13], [17]. Work that involves larger programs, typically from curated benchmarks of real faults, such as Defects4J [18], focuses on demonstrating that the produced slices include statements responsible for the failure and are smaller than the original program traces [15], [16]. However, they do not systematically assess whether the size of the produced slices is reasonably small – an important criterion as inspecting large slices is unlikely to be more helpful in practice than debugging the program itself.

Case Study. Motivated to investigate the usefulness of slicing-based approaches and promote their adoption in mainstream software debugging tools, in this work, we start by conducting a study on the properties of slices. In particular, we focus on dual slicing, which is applied to identify regression failures. As our study subjects, we use projects from the Defects4J benchmark. However, projects in this benchmark are explicitly designed to focus on individual faults in isolation, with changes between program versions being small [19], [20], [21]. To experiment



```
1^1  query = "select sum(c1)...where a>1"
2^1  expr = new SQLExpr()
3^1  parsed = query.split(" ")
4^1  opr = parsed[0]
5^1  if (opr.equals("select"))
6^1    opr = opr.toUpper()
7^1  expr = expr.append(opr)
8^1  ...
...  //adding other sql items to expr
29^1 ...
30^1 expr = expr.append(parsed[n])
31^1 res = getOperation(expr)
32^1 assertEquals("SELECT", res)
```

(a) Old version: P_1.

```
1^1  query = "select sum(c1)...where a>1"
2^1  expr = new SQLExpr()
3^1  parsed = query.split(" ")
4^1  opr = parsed[0]

7^1  expr = expr.append(opr);
8^1  ...
...  //adding other sql items to expr
29^1 ...
30^1 expr = expr.append(parsed[n])
31^1 res = getOperation(expr)
32^1 assertEquals("SELECT", res)
```

(b) New version: P_2.

Fig. 1: Dynamic execution traces for P_1 and P_2 program versions.

with slicing in a more realistic setup, we collect an additional set of projects where multiple changes distributed in the program can simultaneously affect the test results. Specifically, we identify eight large open-source library-client project pairs and use them as case studies of troubleshooting library upgrade failures. We refer to this benchmark as LibRench.

We pick the library upgrade task for our experiments because of its relevance and practical importance: currently, a large fraction of open-source library clients do not use the latest versions of the libraries, even though more advanced library versions contain critical bug and security vulnerability fixes [22], [23], [24], [25]. Developers delay library upgrades because upgrades are often time-consuming and error-prone [26], [27], [28]. Moreover, some library changes can be defective, causing developers to revert to the original library version [24].

Both client and library developers spend days, if not months, identifying the reasons and looking for ways to fix upgrade failures. For example, an `org.jdbi` client developer spent more than 10 days troubleshooting the upgrade of the `com.fasterxml.jackson` library: «I fully intend to submit a PR once I narrow down the root cause and come up with an acceptable fix [29].» Another group of developers debugged a failure for more than 20 days: «At this point I don't have time to start digging through bigger tests <...> but if someone could help here trim tests down into core failure(s), more minimal/isolated case, I'd happily merge a PR [30].» Inspecting how slicing can help developers troubleshoot such upgrade failures is thus an ideal task for our study.

We applied dual slicing for projects in Defects4J and LibRench, to produce a slice relevant to each failure. Our analysis of the slices showed that although slicing helped substantially reduce the number of execution-level statements that developers need to inspect in order to understand a failure (i.e., debugging steps they need to make), in the majority of cases, slices are still very large: 2,007 execution-level statements on average, with a maximum of 9,767 statements (see Section III for details). We believe this number is still very large to be comfortably analyzed by a human. As such, *techniques to further reduce the size of slices are needed.*

By inspecting the produced slices, we realized that slices often contain statements that are *responsible* for the failure and statements that *propagate* information between responsible statements, to ensure adequate context and information flow is preserved. This observation inspired the slice minimization technique proposed in this work.

Slice Minimization Through Summarization. Building up on the insights from the study, we propose an approach for further minimizing program slices while preserving the code necessary for understanding the regression failure. The main idea behind our approach is to summarize the (generally large) propagation blocks of code *while keeping their effect on the responsible statements.*

Consider, for example, two versions of a program, P_1 and P_2, in Figure 1 – a simplified version of a regression failure which occurred when upgrading a popular database connection library, `alibaba-druid`, from version 1.1.14 to 1.1.21. The figure shows dynamic execution traces of the old (passing) and new (failing) executions, where lines 1, 31, and 32 correspond to the test and the remaining lines correspond to the changed method `parseExpression` in the library code. We mark code changed between the two versions in bold (lines 5-6).

The (unchanged) test calls the library `parseExpression` method, with a String representing an SQL query as its input (line 1 in both figures). The goal of the method is to parse the input expression and return it in an SQL format. The test then verifies that parsing was successful and that the operation of the returned query is indeed `SELECT` (lines 31-32).

The expected result is successfully obtained in P_1 but the assertion fails in P_2. That is because P_1 ensured to capitalize the name of the operation (lines 5-6) while P_2 omitted these statements. The remainder of the code (lines 7-30) parses additional parts of the input query, building the output SQL query returned by the `parseExpression` method. Yet, these additional parts are not checked by the test and thus have no effect on the failure.

The dual slicing approach applied to this example will return all but the four grayed-out statements in lines 1-4. That is because it correctly identifies that the execution of these grayed-out statements is identical between the versions and thus cannot be the reason for the failure. It will also correctly keep the changed code in lines 5 and 6, as this code is responsible for the failure. However, it will also keep the code in lines 7-30 as part of the slice, as the program output is data-dependent on this code (see Section II for details).

Understanding how exactly the change in the value of `opr` in line 6 affects the value of `expr` in line 31 through the calculations in lines 7-30 does not help in understanding the source of the failure: the value of `expr` is computed in the exact same manner in both versions P_1 and P_2, with only the initial value of `opr` being different. Simply removing statements in lines 7-30, which did not change between the versions, e.g., like *git diff* does, is not useful either: that would hide the important fact that the value of `expr` in line 31 depends on the value of `opr` line 6 of P_1 and line 4 of P_2. Such *context* is necessary to understand the reasons for the failure.

Thus, instead of removing such blocks of code, we propose to *summarize* their effect on the rest of the program. The summaries consist 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. Such summarization eliminates unimportant internal computations while preserving the flow of information in the slice. In our example, the code in lines 7-30 is summarized as a statement `expr = Func1(opr, expr)`, which captures the relationship between the inputs and the outputs of the summarized block without including unnecessary details.

We implemented the proposed approach for identifying and replacing propagation blocks with their corresponding summaries in a tool named INPRESS (stands for *Information-Preserving Slice Summarization*). Our evaluation of INPRESS shows that it produces summarized slices that are around 76% shorter than the original ones, on average (max: 97.89%), helping to further reduce the amount of information developers need to follow when troubleshooting regression failures.

Contributions. This paper makes the following contributions:
1. We conduct a study investigating the applicability of slicing-based techniques for troubleshooting regression failures. Our study shows that the produced slices are large – up to 9,767 execution-level statements (Section III).
2. We define the notion of *responsible* and *propagation* statements of the slice: the former are necessary to understand the failure and the latter are used to propagate contextual information between the responsible statements (Section IV).
3. We propose an approach for reducing the size of slices by abstracting contextual information while preserving its effects on the failure through propagation (Section IV).
4. We implement the proposed approach in a tool, called INPRESS, and evaluate its effectiveness (Section V).
5. We make our implementation of INPRESS, Defects4J and LibRench case studies that we used, and all experimental data available online [31], to encourage future work in this area.

II. BACKGROUND

In this section, we provide the necessary background and definitions used in the rest of the paper.

Code Representation. We assume a common Java-bytecode-style programming language representation [32], with method calls and return statements, assignments to local and global variables, conditionals, i.e., `if`s, and `jump`s. For simplicity, 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 our examples as s_i, e.g., the `if` statement in line 5 of Figure 1a is denoted by s_5.

A *library* is a standalone distribution of a program which exposes a number of APIs (i.e., method calls) to its *clients*. A client program can use one or more libraries. We refer to the two versions of a program (a client-library pair) as P_1 and P_2 and the set of changes between the versions as Δ. These changes include statements that need to be *added (A)*, *removed (R)*, and *modified (M)* to transform one version of the program to another. More specifically, we define two symmetric operators, Δ_2 and Δ_1. Δ_2 captures additions, modifications, and removals in P_2 compared with P_1. Applying Δ_2 on the P_1 version will produce P_2, i.e., Δ_2 $(P_1) \rightarrow P_2$. For the example in Figure 1, Δ_2 =$\{$R(s_5-s_6)$\}$. Similarly, Δ_1 captures additions, modifications, and removals in P_1 compared with P_2, i.e., Δ_1 $(P_2) \rightarrow P_1$. In our example, Δ_1 =$\{$A(s_5-s_6)$\}$.

Tests and Execution Traces. 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 Figure 1a, the execution trace consists of s_1^1, s_2^1, s_3^1, etc. For regression failures, we denote a test case that passes with P_1 and fails with 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 with P_1 and P_2, respectively.

Control, Control-flow, and Data-flow Dependencies. In static analysis, for two statements s_i and s_j, we say that s_j is *control-dependent* on s_i if, during execution, s_i can directly affect whether s_j is executed [33]. An `if` statement is a common example of a statement that affects the execution of its subsequent statements. For the example in Figure 1a, s_6 is control-dependent on s_5. We say that statement instance s_j^m is control-dependent on s_i^k if s_j is control-dependent on s_i. That is, s_6^1 is control-dependent on s_5^1.

We say that statement s_j is control-flow-reachable from s_i if s_j is always executed following the execution of s_i, i.e., either both statements are in the same method and s_j appears after s_i in the control-flow graph or they are in different methods and s_i (transitively) invokes the method whose statement is s_j. For example, s_4 is control-flow-reachable from s_3 in Figure 1. We say that a statement instance s_j^m is control-flow-reachable from s_i^k if s_j is control-flow-reachable from s_i and s_j^m is, in fact, executed after s_i^k in the same thread.

We say that a statement instance s_i^k is a *dynamic reaching definition* of a variable v in s_j^m if and only if (a) s_j^m is control-flow-reachable from s_i^k, (b) there exists a variable v s.t. v is used in s_j and defined in s_i, and (c) there is no redefinition of v along the control-flow edges between s_i^k and s_j^m. In that case, we also say that the statement instance s_j^m is *data-flow-dependent* on s_i^k w.r.t. the variable v [34]. For example, the dynamic reaching definition of the variable `parsed` used in s_4^1 is s_3^1 and, thus, s_4^1 is data-flow-dependent on s_3^1 w.r.t. `parsed`.

Program Slicing. Program slicing [35], [4] 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 [36]. While static slicing considers all possible program paths leading to the slicing criterion, dynamic slicing focuses on one concrete execution. The main idea behind a 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.

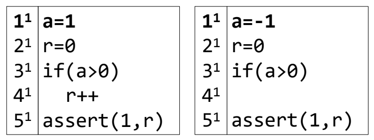
A slicing criterion for an execution trace is a tuple (c, V), where c is a statement instance and V is a set of all variables of interest used in this statement instance [36]. If V is omitted, it is assumed to include all variables used by c. A *backward dynamic slice* [36] is the set of statement instances 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.

Trace Alignment. Trace alignment gets as input two traces, $\mathbb{T}_{P_1}$ and $\mathbb{T}_{P_2}$, and establishes correspondence of execution points across the traces [37]. Several trace alignment techniques have been proposed, based on string matching [38], memory indexing [39], and structural indexing [40], [16]. They produce a set of pairs of aligned trace statement instances, which we denote by $\theta(\mathbb{T}_{P_1}, \mathbb{T}_{P_2})$.

Dual Slicing. Dual slicing is a symmetric slicing technique that works on two traces simultaneously; it was first introduced for debugging concurrency bugs [14] and later used for regression failures [15], [16]. 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 – one from the passing and the other from the failing execution, the main idea behind dual slicing is to first align the traces and then focus only on their differences. The approach defines two types of *differences*: statement instances that are not aligned across executions (e.g., s_5^1-s_6^1 in Figure 1a, as these are only executed in one of the traces) and statement instances that are aligned but produce different data values (e.g., s_7^1 in Figures 1a and 1b, as the execution of this statement instance results in different values of `expr`).

Unlike classical slices, dual slices only contain statement instances that differ between traces. In Figure 1, s_1^1, s_2^1, s_3^1, and s_4^1 are not part of the slice (and thus grayed out in the figure): these instances are aligned across executions and define the exact same data values. As there is no divergence in the execution of these instances, dual slicing concludes that they are not the reason for the failure.

Another key difference between dual and classical slicing is that dual slicing computes the transitive closure of dependencies across both traces. This means that 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. For the example in Figure 2, the statements in lines 1 and 3 of P_2 are added to the dual slice even though they have no effect on the failing assertion statement in line 5 of P_2 and, thus, are not part of the classical backward slice of P_2.



```
1¹ a=1
2¹ r=0
3¹ if(a>0)
4¹   r++
5¹ assert(1,r)
```

(a) Trace of P_1.

```
1¹ a=-1
2¹ r=0
3¹ if(a>0)
4¹
5¹ assert(1,r)
```

(b) Trace of P_2.

Fig. 2: Classical vs. dual slice.

III. Case Study

We now describe the methodology, selection of subjects, and results of our study for evaluating the properties of dual slices.

A. Methodology

We implemented a version of the dual slicing algorithm based on recent work by Wang et al. [16]. This work focused on improving trace alignment, which is the main building block for dual slicing. The authors released their implementation of the dual slicing algorithm in a tool named ERASE. We borrowed the trace alignment algorithm from ERASE. We opted not to use the slicing algorithm implemented in the tool as it does not support implicit control dependencies caused by exceptional flows (i.e., runtime exceptions which, in fact, were the main reasons for failures in our case studies), multi-threading, and data flows through Java framework calls, which were necessary for our study. Instead, we implemented the slicing part on top of Slicer4J [41]. Slicer4J relies on Soot [42] to instrument the bytecode of its input programs and produces traces of Jimple statements [43]. We further map these statements to their corresponding source code statement instances using Soot. We performed all our experiments 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.

B. Subjects

As our study subjects, we use two sets of programs described below and summarized in Table I.

Defects4J. We started with the popular Defects4J benchmark [18], which consists of 395 regression failures from six Java projects. For each regression failure, Defects4J provides a faulty and a correct version of the program, a minimal set of changes required to fix the fault, and a test that triggers the fault. We had to exclude 41 program pairs whose traces cannot be processed by the ERASE trace alignment algorithms, as was also done by the authors of that tool [16] and 4 pairs that could not run under Java 8 runtime used by Slicer4J [41]. Like Lin et al. [44], we also excluded 16 pairs with nondeterministic

TABLE I: Subjects.

ID	Project	#Failures	LoC	
			P_1	P_2
Defects4J				
D1	JFreeChart	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				
L1	jettison / xstream	1	62,615	63,631
L2	square-dagger / modelmapper	1	55,051	56,995
L3	moshi / retrofit	1	54,229	54,987
L4	jackson-core / mockserver	1	96,880	100,567
L5	antlr4-runtime / fizzed-rocker	1	87,585	84,232
L6	httpcomponents-client / wasabi	1	184,363	186,329
L7	jackson-databind / openAPI-generator	1	391,254	387,458
L8	alibaba-druid / dble	1	440,011	445,216
Avg.	-	-	171,498	172,426
All				
Avg.	-	-	115,848	116,314

test results. Finally, we excluded 56 "trivial" failures, which occur at the last statement of the trace, as a dual slice for these failures contains one statement only. At the end, our evaluation included 278 Defects4J faulty and correct program pairs. Due to space limitations, in Table I, we group the pairs by their corresponding project and report the number of correct and faulty program pairs in each project (#Failures). The full list of subjects that we used is available online [31].

As the table shows, the size of the correct (P_1) and faulty (P_2) versions of the programs in the Defects4J benchmarks (subject ids D1-D6), is almost identical, in terms of lines of code (LoC) calculated by the JaCoCo tool [45]. This is because the Defects4J benchmark is tailored to study individual faults in isolation, with each correct and faulty program pair containing only one fault and only the changes needed to fix the fault. That is, Defects4J might not represent realistic debugging scenarios, where numerous types of changes, made for different purposes, co-exist in the program [46].

LibRench. To evaluate the applicability of slicing for identifying regression failures in more realistic debugging scenarios, we collected an additional set of programs, which we refer to as LibRench. To this end, we focused on the task of troubleshooting failing open-source software library upgrades, collecting the five most-used libraries from each of the 30 different categories in Maven [47] (150 libraries in total). We further selected 128 libraries whose source code is available in GitHub. The size of these libraries ranges between 2,135 and 973,435 LoC (average: 123,484, median: 56,366).

To collect clients, we started from the 1,000 top-starred Java projects in GitHub that use the Maven dependency manager. We filtered out 573 projects that could not pass the build and test phases successfully under JDK version 8. We parsed the `pom.xml` files of the remaining projects to obtain the list of Maven libraries they use and selected projects with at least one library in our list of top libraries. We then attempted to upgrade the client to the latest version of the library available on Maven, identifying 168 client-library pairs for which at least one client test fails after the upgrade. This set of pairs contained 84 unique libraries, which we further ordered by size and grouped into 8 bins We then randomly selected a library from each bin and a client-library pair for further inspection. This process ensured that we have represented libraries of different sizes in our dataset.

Subjects L1-L8 in Table I correspond to the selected client-library pairs, where we list the name of the library and the client for each pair. Detailed information about the selected projects, such as their versions and more, is available online [31].

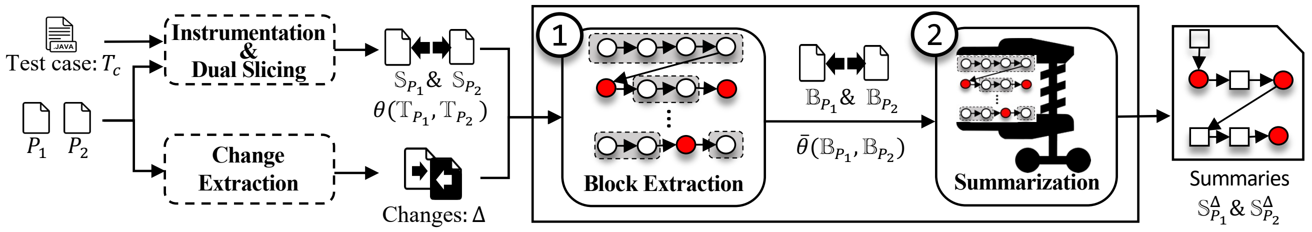
C. Results

We applied dual slicing on each base and regression subject program pair, (P_1, P_2), using the failing regression test as an input. Table II shows the size of the execution trace (#T) and the produced dual slice (#DSlice), in terms of the number of execution statement instances they contain. It also shows the reduction rate achieved by dual slicing, calculated as $\frac{\#T-\#DSlice}{\#T}$ (%Reduct.). We report the metrics for each library-client pair individually (subjects #L1-#L8) and aggregate the metrics to report average numbers for all Defects4J cases in the same project (subjects #D1-#D6). Detailed results for each individual Defects4J case are available online [31]. The last column of the table captures the number of changes, i.e., consecutively changed statement instances, that a trace contains (#Chg). E.g., lines 5-6 in the example in Figure 1 are counted as one consecutive change.

The table shows that dual slicing is indeed very effective in reducing the size of the original execution trace: around 88.28% reduction rate on average (row "P_1 +P_2 Avg.", which considers P_1 and P_2 executions together, for both Defects4J and LibRench). The main reason for this reduction is that traces are relatively large: more than 30,000 statement instances, on average. Yet, they contain only a small number of changes: 23 consecutive changes, on average. A small number of changes, especially when located towards the end of the trace, i.e., close to the failure, gives dual slicing an opportunity to remove numerous statements preceding the change.

TABLE II: Reduction Rate Achieved by DSlice.

ID	#$\mathbb{T}$		#DSlice		**%Reduct.**		#Chg	
	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
Defects4J								
D1	6,080	5,745	69	53	89.95	89.87	3	1
D2	100,517	84,809	3,193	3,489	97.09	96.08	6	5
D3	3,612	2,570	60	26	93.42	92.21	3	1
D4	11,212	6,390	444	1,033	92.15	85.3	6	19
D5	3,628	2,600	542	213	90.47	91.93	7	4
D6	14,176	11,899	407	630	95.21	94.61	3	2
Avg.	39,491	32,543	1,291	1,505	93.97	91.94	5	7
LibRench								
L1	20,667	22,374	856	2,829	95.85	87.35	34	169
L2	5,891	1,275	474	309	91.95	75.76	36	27
L3	1,112	1,150	194	201	82.55	82.52	8	14
L4	9,035	2,213	1,146	1,141	87.31	48.44	6	6
L5	128,965	128,468	8,872	9,767	93.12	92.39	58	133
L6	127	307	63	30	50.39	90.22	3	2
L7	30,813	47,747	5,964	8,162	80.64	82.9	56	78
L8	52,957	54,781	923	949	98.25	98.26	10	14
Avg.	31,196	32,289	2,311	2,923	85.01	82.23	26	55
All								
Avg.	35,343	32,416	1,801	2,214	89.49	87.08	15	31
P_1 +P_2 Avg.	33,879		2,007		88.28		23	



Test case: T_c
Instrumentation & Dual Slicing
P_1 P_2
Change Extraction
$\mathbb{S}_{P_1}$ & $\mathbb{S}_{P_2}$
$\theta(\mathbb{T}_{P_1}, \mathbb{T}_{P_2})$
Changes: Δ
1
Block Extraction
$\mathbb{B}_{P_1}$ & $\mathbb{B}_{P_2}$
$\bar{\theta}(\mathbb{B}_{P_1}, \mathbb{B}_{P_2})$
2
Summarization
Summaries
$\mathbb{S}^{\Delta}_{P_1}$ & $\mathbb{S}^{\Delta}_{P_2}$

Fig. 3: InPreSS overview.

One of the main reasons the reduction rate for Defects4J is higher than that for LibRench is that, by design, Defects4J benchmarks include only a minimal set of changes, which are mostly located close to the failure. In fact, in more than 90% of the Defects4J subjects, the first changed statement instance appears in the last third of the trace, which gives dual slicing an opportunity to remove numerous statements preceding the change. This is not the case for the majority of LibRench subjects. Interestingly, only the largest LibRench subject, #L8, exhibits the same property: the first change in the P_1 execution of 52,957 statement instances appears at position 50,693 of the trace. Thus, dual slicing removes more than 50,000 statement instances preceding the change, achieving a very high reduction rate of 98.25% in this case.

However, despite the high reduction rate, the slice size for the majority of the subjects (including #L8) is still very large: 2,007 statement instances, on average, with a maximum of 9,767 instances for P_2 of subject #L5. This means that, even after slicing, developers still need to follow a large number of execution steps when debugging failures.

By manually inspecting the generated slices, we observe that slices contain large blocks of statement instances *common* between different program versions, similar to statements in Figure 1, which was inspired by a real case we observed. In this case, the common code block, in fact, included 223 statement instances that propagate manipulations related to managing expressions and do not directly contribute to the failure. In another example, the `jettison` library for converting between XML and JSON formats (subject #L1), the change resulted in a different structure of the JSON object that further went through a chain of serialization and deserialization operations (152 statement instances), which are, again, of low relevance to the failure. In yet another case, the `jackson-databind` library used for data processing tasks (subject #L7), the change related to removing an `if` condition validating certain properties of the input. This change further propagated to more than 2,000 statement instances that dual slicing keeps due to a transitive control dependency on the removed `if`.

This presents an opportunity for slice minimization through summarizing these large blocks of common code, which have little relevance to the failure and mostly *propagate* information related to the change. Such minimization can further reduce the size of the slices while keeping the flow of information in the program, improving the effectiveness of slicing for regression analysis. We discuss our approach, InPreSS, for performing such summarization next.

IV. Slice Minimization Approach

Figure 3 shows the overview of InPreSS. It receives as input two slices, $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$, which are produced by a dual slicing technique for the execution of the test T_c on P_1 and P_2, respectively. It also receives the trace alignment, $\theta(\mathbb{T}_{P_1},\mathbb{T}_{P_2})$, and the set of changes between program versions, Δ. InPreSS produces as output summarized slices, $\mathbb{S}^{\Delta}_{P_1}$ and $\mathbb{S}^{\Delta}_{P_2}$, which capture the effects of changes in P_2 on the failure of T_c.

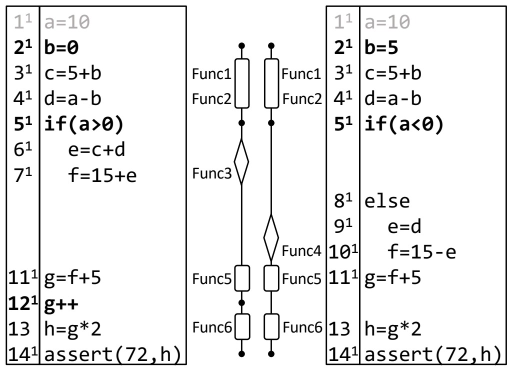
In the current version of InPreSS, input slices and their alignment are produced by our instantiation of the dual slicing algorithm. However, any pair of aligned execution sequences, including classical backward slices and the complete execution trace itself, can be used instead.

InPreSS leverages the insight that computations performed by code that is common between two versions of the program are not directly responsible for the test failure. Yet, simply removing common code is not desirable as developers rely on the context and flow of information in a program when troubleshooting failures [7], [9]. Thus, instead of removing common code, InPreSS ① accurately identifies blocks of common code that can be individually summarized in terms of high-level input-output functions and ② concisely summarizes the identified blocks while keeping the flow of information in the program intact. As such, InPreSS minimizes the slice while preserving the dependencies between all its statements.

We now describe these two parts of our approach (boxes ① and ② in Figure 3) using Figure 4 as an example that we created to illustrate different features of InPreSS. Figures 4a and 4b show the successful and failing runs of program versions P_1 and P_2, respectively. Like in Figure 1, we mark statements changed between the versions in bold and gray out statement instances removed by the dual slicing algorithm.

A. Block Extraction

Given the input slices ($\mathbb{S}_{P_1}$,$\mathbb{S}_{P_2}$) and changes Δ=(Δ_1,Δ_2), InPreSS first marks statement instances within each slice that have to be preserved in the summary in their original form. We refer to the set of preserved statement instances as the *retained set* and denote it by ρ_{P_1} and ρ_{P_2}, for $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$, respectively. We define the retained set ρ_{P_1} (ρ_{P_2}) to contain all statement instances corresponding to the statements added and modified in Δ_1 (Δ_2). These statement instances are bolded in the figure. We also add to retained sets the statement instances corresponding to the test assertion. For the example in Figure 4a, ρ_{P_1} contains statement instances in lines 2, 5, 12, and 14. We represent the retained set elements by dots in the schematic representation at the center of the figure.



```
1^1  a=10
2^1  b=0
3^1  c=5+b
4^1  d=a-b
5^1  if(a>0)
6^1    e=c+d
7^1    f=15+e



11^1 g=f+5
12^1 g++
13   h=g*2
14^1 assert(72,h)
```

(a) Dual slice: P_1.

```
1^1  a=10
2^1  b=5
3^1  c=5+b
4^1  d=a-b
5^1  if(a<0)


8^1  else
9^1    e=d
10^1   f=15-e
11^1 g=f+5

13   h=g*2
14^1 assert(72,h)
```

(b) Dual slice: P_2.

Fig. 4: Example regression failure.

```
2^1  b=0
3^1  c=Func1(b)
4^1  d=Func2(a,b)
5^1  if(a>0)
7^1   f=Func3(c,d)


11^1 g=Func5(f)
12^1 g++
13   h=Func6(g)
14^1 assert(72,h)
```

(a) Slice summary: P_1.

```
2^1  b=5
3^1  c=Func1(b)
4^1  d=Func2(a,b)
5^1  if(a<0)

8^1  else
10^1  f=Func4(d)
11^1 g=Func5(f)

13   h=Func6(g)
14^1 assert(72,h)
```

(b) Slice summary: P_2.

Fig. 5: Summarized slices for the example in Figure 4.

Next, INPRESS identifies the set of *blocks* within each slice. Intuitively, a block is the longest sequence of statement instances that correspond to a chunk of common code, with the retained statements used as "dividers". We further distinguish between two types of blocks: *matched blocks* ($\bar{\bar{\mathbb{B}}}$), which contain statement instances aligned between slices, i.e., in $\theta(\mathbb{T}_{P_1},\mathbb{T}_{P_2})$, and *unmatched blocks* ($\bar{\mathbb{B}}$), which contain unaligned statement instances. In the example in Figure 4a, $\mathbb{T}_{P_1}$, has three matched blocks, represented by rectangles in the figure: statement instances in lines 3-4, 11, and 13. It also has an unmatched block represented by a diamond: (unaligned) statement instances in lines 6-7. The slice in Figure 4b, $\mathbb{T}_{P_2}$, by definition, has the exact same matched blocks and also an unmatched block: in lines 9-10. Furthermore, even though there are no retained set elements between statement instances in lines 11 and 13 of $\mathbb{T}_{P_2}$, these instances are split into two blocks, to match the blocks in $\mathbb{T}_{P_1}$.

The block extraction step outputs "chunked" slices, $\mathbb{B}_{P_1}$ and $\mathbb{B}_{P_2}$, as well as the alignment between their matched blocks, $\bar{\theta}$. Such an abstraction of blocks allows us to identify regions of code between the retained statements that can be summarized without losing critical information related to the statements of interest to the client developers, as discussed next.

B. Slice Summarization

The goal of this step is to accurately summarize the slice, expressing the mapping between inputs and outputs for each block, as well as dependencies between the block summaries and the retained statements. More specifically, inspired by Person et al. [48], we summarize each block as a set of functions representing the mapping between the block's input variables (i.e., variables used in the block) and its output variables (i.e., variables defined in the block).

For example, the first unmatched block in Figure 4a consists of two statement instances in lines 6 and 7. It uses two variables: `c` and `d` in line 6 and sets two variables: `e` and `f`. However, only one of these variables is used in the remainder of the code – f (in line 11). The variable `e` is used for internal calculations only and thus is not part of the external block definitions. Thus, the summary of this block is represented by a function Func3(`c,d`), as shown in Figure 5a (line 7). That is, the summary of the block abstracts the internal computations, only exposing input-output mappings of variables defined in the block and used in the rest of the code.

One important aspect of INPRESS is the ability to synchronize summaries of matched blocks. For example, in Figure 4b, only the variable `d` is used in the rest of $\mathbb{S}_{P_2}$ (line 9), while in $\mathbb{S}_{P_1}$ both variables, `c` and `d`, are used in line 6 and, thus, are represented as Func1(`b`) and Func2(`a,b`) in lines 3 and 4 in Figure 5a. However, to ensure that matched blocks (of aligned statement instances) are summarized in an identical way, INPRESS also generates the summary for the variable `c` in $\mathbb{S}_{P_2}$. Besides ensuring that matched code is summarized with the same functions, this also demonstrates that, unlike in $\mathbb{S}_{P_1}$, `c` was not used in $\mathbb{S}_{P_2}$ for calculating the value of the variable `f` (because of the change in the `if` condition in line 5).

Each abstraction function contains all statement instances used for computing the variable, which are all statement instances that are in the backward slice performed on the common block code, with the defined variable being the slicing criterion. For example, the body of Func1(`b`) consists of one statement instance in line 3, while the body of Func3(`c,d`) consists of two instances in lines 6 and 7.

To produce the summaries, INPRESS processes both execution slices backwards, starting from the failing assertion statement. It transitively collects definitions of variables used in the retained statement instances while summarizing each block as a set of assignments capturing the mapping of the block input to output variables. While doing that, it also synchronizes the summaries of the matched blocks. In a sense, the tool performs a backward dynamic slicing over the set of retained statement instances and blocks, producing a set of statement instances required to retain the flow of information (1) from the input variables to the changed statement instances, (2) between the changed statement instances, and (3) from the changed instances to the failed assertion.

This process is shown in Algorithm 1, which accepts $\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, and the set of matched blocks $\bar{\bar{\mathbb{B}}}$ as input and produces the minimized versions of both slices, $\mathbb{S}^{\triangle}_{P_1}$ and $\mathbb{S}^{\triangle}_{P_2}$, as output. It works simultaneously on both slices and, similar to dynamic backward slicing techniques, maintains working sets W_{P_1} and W_{P_2} (line 4) to track the ids of the variables that have to be defined in each of the summaries. It leverages the fact that input slices are aligned around common blocks and thus summarizes the unmatched parts of each slice first, starting from the failing

Algorithm 1: Slice summarization algorithm.

```
1  Input: $\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\bar{\bar{\mathbb{B}}}$
   Output: $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$
2  begin
3      $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$ $\leftarrow \emptyset$
4      $W_{P_1}$, $W_{P_2}$ $\leftarrow \emptyset$                         ▷ definitions to look for
5      while $\mathbb{B}_{P_1} \neq \emptyset \vee \mathbb{B}_{P_2} \neq \emptyset$ do
6          ProcessUnmatched ($\mathbb{B}_{P_1}$, $\mathbb{S}^{\Delta}_{P_1}$, $W_{P_1}$)
7          ProcessUnmatched ($\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_2}$)
8          ProcessMatched ($\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1}$, $W_{P_2}$)
9      return $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$
10 Procedure ProcessUnmatched($\mathbb{B}$, $\mathbb{S}^{\Delta}_{L}$, $W$)
11     begin
12         while $\mathbb{B} \neq \emptyset$ do
13             $e \leftarrow get\_last(\mathbb{B})$                    ▷ get the last element
14             if $e \in \bar{\bar{\mathbb{B}}}$ then
15                 return                      ▷ will be processed with its matched block
16             else if $e \in \rho$ then
17                 $\mathbb{S}^{\Delta}_{L} \leftarrow e + \mathbb{S}^{\Delta}_{L}$         ▷ prepend the retained statement instance
18                 $W \leftarrow W \cup \{use(e)\}$              ▷ look for the variables it uses
19                 if $e$ is assignment then
20                     $W \leftarrow W \setminus \{def(e)\}$     ▷ no need to track redefined variables
                           further
21             else
22                 SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{L}$, $W$)   ▷ unmatched block
23             $\mathbb{B} \leftarrow \mathbb{B} \setminus \{e\}$                ▷ remove the processed element
24 Procedure ProcessMatched($\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1}$, $W_{P_2}$)
25     begin
26         $e \leftarrow get\_last(\mathbb{B}_{P_1})$                    ▷ get the matched block
27         SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{P_1}$, $W_{P_1} \cup W_{P_2}$)
28         SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1} \cup W_{P_2}$)
29         $\mathbb{B}_{P_1} \leftarrow \mathbb{B}_{P_1} \setminus \{e\}$;   $\mathbb{B}_{P_2} \leftarrow \mathbb{B}_{P_2} \setminus \{e\}$      ▷ remove the processed
           elements
30 Procedure SummarizeDefinitions(block, $\mathbb{S}^{\Delta}_{L}$, $W$)
31     begin
32         $A \leftarrow summarize(block)$       ▷ represent the block as a set of assignments
33         foreach $a \in A$ do
34             if $def(a) \in W$ then
                   ▷ a summary assignment defines a variable of interest
35                 $\mathbb{S}^{\Delta}_{L} \leftarrow a + \mathbb{S}^{\Delta}_{L}$               ▷ prepend the summary statement
36                 $W \leftarrow W \setminus \{def(a)\}$        ▷ not tracking redefined variables further
37                 $W \leftarrow W \cup \{use(a)\}$              ▷ look for the variables it uses
```

assertion statement (lines 6-7), then synchronizes on both slices to summarize matched blocks (line 8), and repeats until both slices are fully processed (lines 5-9).

Unmatched parts of the slice are either individual statement instances from the retained set (represented with dots in Figure 4) or unmatched blocks (represented with diamonds in Figure 4). They are handled by the `ProcessUnmatched` procedure of the algorithm (lines 10-23). Specifically, this procedure traverses each individual slice, represented by $\mathbb{B}$, backwards until it reaches a matched block (lines 14-15). When a matched block is reached, the procedure terminates, to make sure matching blocks are handled together.

When processing elements of an unmatched block, it makes sure to add all elements of the retained set to $\mathbb{S}^{\Delta}_{L}$ (line 17), prepending them to order the summary in chronological rather than reverse order. It then identifies all variables *used* by the retained statement instance and adds them to the working set W, to make sure their definitions are included in the summary (line 18). Moreover, if the retained statement is an assignment (rather than an `if` statement or a method call), it identifies the variable *defined* in the assignment and removes it from the working set, since this variable is being redefined in the current statement (line 20).

Unmatched blocks are treated similarly, with the goal to find and keep definitions of variables in W. Specifically, the algorithm calls the `SummarizeDefinitions` procedure (lines 21-22), to summarize the effect of each block as an unordered set of assignments capturing the data flows from the block input to output variables, as discussed above (line 32). Then, *definitions* of variables in W are added to the summary $\mathbb{S}^{\Delta}_{L}$ (line 35) and removed from W because their definitions are found already (line 36). Instead, variables *used* by the newly added statements are added to W for further tracking (line 37).

Finally, after the unmatched blocks in both $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$ are processed, the algorithm processes matched blocks (line 8, 24-28). Matched blocks are summarized similarly to unmatched ones, except that both working sets W_{P_1} and W_{P_2} are used to specify the variables of interest. This is done to ensure that matched blocks are summarized with identical functions, as in the example of `c` in Figure 4b, discussed above.

The algorithm terminates when it finishes processing both slices (line 12) and returns the produced summaries.

C. Implementation

The implementation of InPreSS works for Java (version 8) and consists of three parts: dual slicing, block identification, and trace summarization. Our implementation of the dual slicing algorithm was discussed in Section III. For block identification, we borrowed the trace alignment algorithm from ERASE and further used GumTree [49] – a state-of-the-art code differencing tool, which identifies inserted, deleted, and modified code statement – to identify common vs. changed code.

For block summarization, we adapted the implementation of Slicer4J [41] to compute control and data-flow dependencies for each variable v defined in the block. It outputs a slice containing statement instances that may affect v. The variables used in the slice, X, (rather than internally defined in the block) are essentially inputs to the block, which may affect the defined variable v. Our implementation then produces the assignment statement mapping v to the variables affecting it, via a function in the form of $v := Func_v(X)$. We further populated the produced function Func$_v(X)$ with all block statement instances from the slice created for v. Finally, we used the same slicing algorithm to compute data-flow dependencies over the trace augmented with the assignments summarizing the effect of each common block. Both block identification and summarization algorithms are implemented in Java.

V. Evaluation

We evaluate our approach on subject programs described in Section III. Our evaluation aims at answering the following research questions:

RQ1 (Effectiveness). How effective is the trace size reduction achieved by InPreSS?

RQ2 (Code Properties). Which code properties affect the size of the produced summaries?

A. *RQ1 (Effectiveness)*

To answer RQ1, we compared the sizes of the dual slice (#DSlice) and the minimized slice produced by InPreSS (#InPreSS). As in Section III, we calculated the reduction rate of InPreSS over the dual slice (%Reduct.): $\frac{\#DSlice - \#\text{InPreSS}}{\#DSlice}$. To further inspect the overhead of InPreSS, we measured both the dual slicing and the slice summarization times for all cases and report the averaged results, in minutes, from five consecutive runs. We, again, performed all our experiments 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.

Table III shows the results of our analysis for each subject program, which we further separate into results for the old and new slices. Like in Section III, due to space limitations, we had to aggregate the metrics and report the averages for all Defects4J faults in the same project. Complete results are available online [31].

The table shows that InPreSS achieves a high reduction rate for all subject programs: 76.07% on average, in both versions (row "All Avg.", which considers P_1 and P_2 executions together, for both Defects4J and LibRench). This means that, instead of inspecting 2,007 execution steps during a debugging session, a developer now needs to inspect only 207, for an average project. Focusing on the LibRench benchmark for more detailed analysis, the maximal reduction rate of 97.89%, for the P_2 version of subject #L4, occurs because the original slice is relatively large, 1,141 statement instances, but the changes it contains are few and sparse: only 6 changed blocks, with 2.5 statement instances each, on average. At the same time, there are 14 blocks of common code, with 80 statement instances each, on average, which can be efficiently summarized by InPreSS, with 2 statement instances in the summary, on average. We observe a similar trend in most of the other cases.

TABLE III: Slice Reduction Rate Achieved by InPreSS.

ID	#DSlice		#InPreSS		**%Reduct.**		Time (Min.)	
	P_1	P_2	P_1	P_2	P_1	P_2	DSlice	InPreSS
Defects4J								
D1	69	53	9	4	69.67	72.12	0.01	0
D2	3,193	3,489	177	151	82.64	88.16	64.92	25.8
D3	60	26	12	5	50.13	57.39	0.03	0.016
D4	444	1,033	35	44	60.57	78.39	1.52	0.01
D5	542	213	30	22	80.59	87.93	0.38	0.03
D6	407	630	31	30	75.18	86.31	0.18	0.05
Avg.	1,291	1,505	77	70	70.05	79.03	22.58	8.86
LibRench								
L1	856	2,829	114	683	86.68	75.85	2.98	0.79
L2	474	309	218	181	54	41.42	3.85	0.1
L3	194	201	31	50	84.02	75.12	0.98	0.02
L4	1,146	1,141	291	24	74.60	97.89	939.88	0.91
L5	8,872	9,767	899	1,318	89.86	86.5	2506.4	461.59
L6	63	30	24	15	61.9	50	0.08	0.01
L7	5,964	8,162	669	841	88.78	89.69	1069.1	0.68
L8	923	949	47	74	94.9	92.2	198.23	0.30
Avg.	2,311	2,923	286	398	79.34	76.08	590.18	0.9
All								
Avg.	1,801	2,214	181	234	74.6	77.55	306.38	4.88
All Avg.	2,007		207		76.07			

The smallest reduction rate of 41.42% is for the P_2 version of subject #L2, where the number of changed statement instances is relatively large compared with the number of statements in the slice: there are 140 changed statement instances in the slice of size 309 in total. As such, the upper bound for a possible reduction rate is 55% only. Furthermore, changes split the slice into smaller blocks of common code, which further reduce the opportunity for minimization. Yet, InPreSS achieved a reduction rate of 41.42% by summarizing common blocks of size 4.8, on average, into blocks of size 1.1, on average.

The runtime measurements show that InPreSS can process even the longest slices in a matter of a few minutes, which is low compared to the runtime of the dual slicing algorithm. The time of dual slicing is generally proportional to the size of both input traces and its high runtime performance is mainly due to slicing the trace and performing the trace matching algorithm. Similarly, the runtime of InPreSS is proportional to the size of both input slices, as it also works on both slices in one pass. Specifically, subject #L5, with the largest input trace and slice sizes, has the highest processing time for both dual slicing and InPreSS; the processing time is the lowest for subjects #D1 and #L6, with the smallest input trace and slice sizes.

Answer to RQ1: InPreSS is able to produce slice summaries that are around 76% shorter than the original slice, on average. The reduction rate is higher for subjects with larger input slices and a smaller number of changes as it increases the chance of building larger blocks of common code and consequently summarizing their internal computations.

B. *RQ2 (Code Properties)*

To answer RQ2, we investigated the relative contribution of matched and unmatched code blocks, denoted by $\bar{\bar{\mathbb{B}}}$ and $\bar{\mathbb{B}}$, respectively, on the overall effectiveness of InPreSS. To this end, we produced two versions of the tool: InPreSS$_{\bar{\bar{\mathbb{B}}}}$, which summarizes matched blocks only, adding unmatched blocks to the retained set and, conversely, InPreSS$_{\bar{\mathbb{B}}}$, which summarizes unmatched blocks only. We then calculated the number and size of blocks of each type and the reduction rate achieved by summarizing only blocks of that type.

Table IV shows the results of this analysis. Each row of the table correspond to a subject program; the columns show the number of blocks of each type, the average and maximal size of the blocks, and the reduction rate achieved by summarizing only the blocks of that type, separately for P_1 and P_2 versions of each subject, and then for all subjects and versions combined. By definition, the number and size of matched blocks are identical in P_1 and P_2.

Summarizing only the unmatched blocks leads to a high reduction rate of 58.64% on average, for P_1 and P_2 executions combined; summarizing only the matched blocks leads to a reduction rate of 17.45%, on average. Unmatched blocks thus have a larger contribution to the total reduction rate achieved

TABLE IV: Reduction Rate Achieved by INPRESS$_{\bar{\bar{\mathbb{B}}}}$ and INPRESS$_{\bar{\mathbb{B}}}$.

ID	$\bar{\bar{\mathbb{B}}}$ (Matched Blocks)						$\bar{\mathbb{B}}$ (Unmatched Blocks)					
	# of Blocks		Avg. (Max) Size		%**Reduct.**		# of Blocks		Avg. (Max) Size		%**Reduct.**	
	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
Defects4J												
D1	1	1	1.6 (2)	1.6 (2)	11.3	16.32	3	1	47.7 (52)	27.1 (475)	57.57	56.5
D2	19	19	57.1 (458)	57.1 (458)	31.84	30.9	18	14	120.8 (938)	143.2 (1,651)	50.8	57.26
D3	2	2	6.4 (14)	6.4 (14)	25.03	29.32	3	1	6.7 (11)	5.0 (7)	25.1	28.17
D4	5	5	6.9 (25)	6.9 (25)	17.16	20.94	6	6	160.5 (342)	457.1 (687)	43.39	57.45
D5	4	4	25.5 (56)	25.5 (56)	30.66	37.85	5	4	55.1 (335)	34.5 (63)	49.93	50.08
D6	4	4	25.7 (54)	25.7 (54)	30.46	33.25	4	3	70.9 (175)	102.4 (240)	44.72	53.06
Avg.	8	8	26.1 (168)	26.1 (168)	24.3	25.66	9	8	93.2 (438)	168.2 (727)	46.58	52.14
LibRench												
L1	18	18	8.6 (50)	8.6 (50)	14.71	4.48	26	159	25.1 (247)	14.3 (475)	73.59	71.93
L2	11	11	4.9 (13)	4.9 (13)	10.75	16.5	39	26	6.8 (20)	4.2 (18)	43.24	24.91
L3	4	4	2.7 (6)	2.7 (6)	3.09	2.98	9	14	19 (91)	12.1 (91)	80.92	72.13
L4	1	1	1 (1)	1 (1)	0	0	5	5	227 (1,090)	224.8 (1,095)	74.60	97.89
L5	9	9	188.8 (1,208)	188.8 (1,208)	11.85	10.8	52	131	135.6 (3,569)	59.1 (2,035)	74.52	75.73
L6	3	3	7.6 (13)	7.6 (13)	23.8	50	2	1	13 (25)	1 (1)	38.09	0
L7	77	77	2.4 (34)	2.4 (34)	1.62	1.16	90	102	62.9 (1,269)	76.5 (1,360)	87.15	88.58
L8	13	13	3.5 (15)	3.5 (15)	3.57	3.47	14	17	61.7 (292)	51.1 (290)	91.3	88.72
Avg.	17	17	27.4 (167)	27.4 (167)	8.69	11.18	30	56	68.9 (825)	55.4 (670)	70.86	64.98
All												
Avg.	12	12	26.7 (168)	26.7 (168)	16.49	18.42	19	32	81.05 (631)	111.8 (698)	58.72	58.56
P_1 +P_2 Avg.	12		26.7 (168)		17.45		25		96.42 (664)		58.64	

by INPRESS. This is because the number of unmatched blocks is generally larger than the number of matched blocks (25 vs. 12, on average) and their size is also larger (96.42 vs. 26.7, on average), giving more opportunities for reduction.

Unmatched blocks ($\bar{\mathbb{B}}$). There are two reasons for why unmatched blocks of common code exist. First, code that is common between two versions can be executed in one but not in the other version. This happens because of control divergence, e.g., a change in the value of a variable used in an `if` condition causes the `else` block to be taken instead of the `if` block. Likewise, an existing but previously unused method can be triggered in the new version. In fact, the majority of unmatched blocks in our experiments (89%) are in this category. As one example, in subject #L7 discussed in Section III, a removal of an `if` condition resulted in the execution of 2,000 previously unexecuted statements.

The second reason (11% of unmatched blocks, mostly in LibRench) is related to issues with trace alignment – a generally hard problem which prevents some "expected" matches. The ERASE trace alignment algorithm uses the control path (a sequence of loop conditions and method calls) as ids of statements and then aligns statements based on these ids. As method refactorings lead to a divergence of control paths and, thus, statement ids, the trace alignment algorithm treats statements controlled by this modified code as unmatched. For example, in subjects #L1 and #L4, code was refactored into a new method, which resulted in large portions of unmatched code. In subject #L3, a method rename early in the trace led, again, to a chain of unmatched blocks.

Matched blocks ($\bar{\bar{\mathbb{B}}}$). Our decision to synchronize the size and type of matched blocks between versions resulted in several chunks of common code being split by changed statements executed in one version but not the other. This results in producing smaller matched blocks, with fewer opportunities for eliminating internal calculations and variables. For example, in subject #L7, there are 77 synchronized matched blocks. Without synchronizations, P_1 and P_2 would have 55 and 53 blocks, respectively. In fact, the higher the number of changes between the versions, the more blocks get split, which explains a higher number of matched (and also unmatched) blocks in LibRench compared with Defects4J subjects.

Interestingly, the reduction rate achieved by summarizing matched blocks only is generally higher for Defects4J than LibRench, with the exception of LibRench subject #L6. That is because these subjects contain a relatively small number of changes and it happens more often that changes to assignment lead to data rather than control divergence. As such, the fraction of matched blocks is higher in these subjects, providing more opportunities for reduction.

Answer to RQ2: Factors affecting the reduction rate include the number and types of code statements that are changed and the quality of the alignment algorithm. The combination of approaches employed by INPRESS increases its ability to achieve a high reduction rate for the majority of cases.

VI. LIMITATIONS AND THREATS TO VALIDITY

For **external validity**, our results may be affected by the subject selection and may not necessarily generalize beyond our subjects. We attempted to mitigate this threat by selecting an externally created dataset used in prior work, Defects4J, and further augmenting it with the LibRench dataset of most-popular Java libraries and clients from Maven. As we used different projects of considerable size and complexity, we believe our results are reliable.

For **internal validity**, we controlled for the threat of any implementation defects by having two authors of this paper manually and independently analyze the results and discuss their findings. We make all our experimental data and implementation of the tool publicly available [31] to encourage validation and replication of our results.

The **main limitation of our approach** is its command-line nature, which makes the produced traces difficult to inspect. While we share this limitation with other slicing and trace-based techniques, we intend to explore integration with IDEs as part of future work. Additional limitations are inherited from the limitations of the underlying infrastructure: Soot and ERASE cannot support Java versions beyond 9 and 8, respectively; the trace alignment algorithm might miss some desired alignments, as discussed in Section V; and Slicer4J can consume substantial time, especially when slicing long execution traces.

VII. Discussion and Related Work

We now discuss the related work on fault localization, focusing mainly on slicing-based techniques and techniques for pinpointing failure-inducing changes. We then discuss empirical studies on fault localization approaches developers follow.

Slicing-based fault localization. Program slicing – a technique for reducing the size of the program by identifying only those parts of the code that are relevant w.r.t. a certain slicing criterion – was extensively discussed in this paper. Our work builds up on these approaches; however, to the best of our knowledge, we are the first to propose an approach for summarizing large pieces of code when analyzing regression failures by relying on a common code abstraction.

Perhaps the most related to ours is the slice rewriting approach called amorphous slicing [50], [51]. The main idea behind this approach is to preserve the semantic rather than syntactic behavior of a slice by simplifying the slice statements, with the goal of producing a smaller slice. Integrating such an approach with ours, e.g., by producing simplified slices instead of the summaries produced by InPreSS, could be a valuable direction for possible future work.

Pinpointing failure-inducing changes. Techniques based on delta debugging aim to isolate failure-inducing changes by repetitively reverting different subsets of changes between the original, correct version and the current, faulty version of the program [52], [53]. These approaches report the smallest subset of changes that can be reverted to recover the original program behavior. Spectrum-based techniques, e.g., Tarantula [1], are inspired by probabilistic- and statistical-based models. Given a program and a set of failed/passed tests, these approaches use various heuristics to rank the program statements as *suspicious* components which might be involved in a failure, narrowing the search for the faulty component that made the execution fail [54]. Another line of work uses symbolic analysis to automatically find potential root causes of regression failures. Given two versions of a program and an input that fails on the modified version program, such approaches aim to automatically synthesize new inputs that (a) is similar to the failing input and (b) does not fail [55]. By generating additional input cases, these techniques aim to pinpoint reasons for failures, even for hard-to-explain bugs.

Our work is thus orthogonal and complementary to these approaches: we focus on identifying reasons for large slices found in practice and on minimizing slices by summarizing the precise flow of information propagated from these changes to the failure. As any slice-based approach, InPreSS can be augmented by techniques that distinguish between responsible and not responsible changes within the slice, e.g., as in [15].

Developers' approach to fault localization. By investigating how developers debug a never-seen-before program, Weiser [3] found that developers recognize program slices significantly more often than random code fragments presented to them, suggesting that developers tend to follow the flow of execution when investigating a failure. Additional studies found that developers had a better understanding of the program and were more systematic in their inspection of code when using slices [6], [7]. Developers were also shown to perform systematic analysis to track changes in variable values and explain the crash through these changes [56]. Intuitively, this captures the mechanism employed in slicing. A number of authors [9], [57], [58] investigated how developers use automated debugging tools. One of the key findings of these works is that presenting only the potentially relevant statements, as done in spectrum-based fault localization, without context and dependencies between them, is insufficient for developers during debugging sessions. The assumption behind our approach is in full agreement with these studies: we build up on the program slicing mechanisms while ensuring context and flow of information is adequately maintained. However, unlike these works, our main focus is on studying the properties of slicing and using these properties for slice minimization.

VIII. Conclusion

In this paper, we investigated the usefulness of slicing-based techniques in narrowing down developers' attention to a subset of program statements relevant for troubleshooting a regression failure. Specifically, we conducted a study applying dual slicing, one of the most advanced of these techniques, to more than 280 programs from the Defects4J and LibRench benchmarks. Our study showed that: (a) the slices reported by the technique are still relatively large to be comfortably inspected by a human and (b) there is an opportunity to further reduce the size of the slices by retaining statements *responsible* for the failure and summarizing statements that *propagate* contextual information between the responsible statements. Based on these observations, we implemented a slice minimization approach, InPreSS, and showed its effectiveness in reducing the size of the slices by 76%, on average. We believe our work will help promote the efficient integration of slicing-based techniques in debugging and will inspire further research in this area.

Data Availability. To support further work in this area, detailed information about our case studies and their analysis, as well as our implementation of InPreSS, are available online [31].

Acknowledgments. Part of this work was funded by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).

References

[1] 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.

[2] 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.

[3] M. Weiser, "Program Slicing," in *Proc. of the International Conference on Software Engineering (ICSE)*, 1981, pp. 439–449.

[4] ——, "Program Slicing," *IEEE Transactions on Software Engineering (TSE)*, no. 4, pp. 352–357, 1984.

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

[6] 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.

[7] 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.

[8] A. J. Ko and B. A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2008, pp. 301–310.

[9] 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.

[10] 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.

[11] 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.

[12] J. Krinke, "Slicing, Chopping, and Path Conditions with Barriers," *Software Quality Journal*, vol. 12, no. 4, pp. 339–360, 2004.

[13] 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.

[14] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing Concurrency Bugs Using Dual Slicing," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2010, pp. 253–264.

[15] W. N. Sumner and X. Zhang, "Comparative Causality: Explaining the Differences Between Executions," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2013, pp. 272–281.

[16] 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.

[17] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization Using Dynamic Slicing and Change Impact Analysis," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2011, pp. 520–523.

[18] 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.

[19] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4j," in *Proc. of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 130–140.

[20] Y. Küçük, T. A. Henderson, and A. Podgurski, "The Impact of Rare Failures on Statistical Fault Localization: The Case of the Defects4j Suite," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 24–28.

[21] Y. Song, X. Xie, Q. Liu, X. Zhang, and X. Wu, "A Comprehensive Empirical Investigation on Failure Clustering in Parallel Debugging," *Journal of Systems and Software (JSS)*, vol. 193, p. 111452, 2022.

[22] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna, "Spider: Enabling Fast Patch Propagation in Related Software Repositories," in *Proc. of the Symposium on Security and Privacy (SP)*, 2020, pp. 1562–1579.

[23] "Equifax Data Breach," https://en.wikipedia.org/wiki/2017_Equifax_data_breach.

[24] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An Empirical Study of Usages, Updates and Risks of Third-party Libraries in Java Projects," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 35–45.

[25] "Open-Source Library Vulnerabilities," https://www.veracode.com/blog/managing-appsec/six-types-open-source-library-vulnerabilities.

[26] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep Me Updated: An Empirical Study of Third-party Library Updatability on Android," in *Proc. of the Conference on Computer and Communications Security (CCS)*, 2017, pp. 2187–2200.

[27] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do Developers Update their Library Dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[28] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do Developers Update Third-party Libraries in Mobile Apps?" in *Proc. of the International Conference on Program Comprehension (ICPC)*, 2018, pp. 255–265.

[29] "Jackson-databind (Issue-2789)," https://github.com/FasterXML/jackson-databind/issues/2789.

[30] "Jackson-databind (Issue-2876)," https://github.com/FasterXML/jackson-databind/issues/2876.

[31] "Supplementary Materials." https://resess.github.io/artifacts/InPreSS/.

[32] J. Zhao, "Dependence Analysis of Java Bytecode," in *Proc. of the International Computer Software and Applications Conference (COMPSAC)*, 2000, pp. 486–491.

[33] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[34] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic Slicing in the Presence of Unconstrained Pointers," in *Proc. of the Symposium on Testing, Analysis, and Verification (TAV)*, 1991, pp. 60–73.

[35] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," *PhD thesis, University of Michigan*, 1979.

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

[37] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards Locating Execution Omission Errors," in *Proc. of the International Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 415–424.

[38] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A Tool for Automatically Detecting Variations Across Program Versions," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2006, pp. 241–252.

[39] W. N. Sumner and X. Zhang, "Memory Indexing: Canonicalizing Addresses Across Executions," in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 217–226.

[40] B. Xin, W. N. Sumner, and X. Zhang, "Efficient Program Execution Indexing," *ACM SIGPLAN Notices*, pp. 238–248, 2008.

[41] K. Ahmed, M. Lis, and J. Rubin, "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)*, 2021, pp. 1570–1574.

[42] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 1999, pp. 1–11.

[43] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," *Sable Technical Report*, 1998.

[44] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong, "Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 509–519.

[45] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "JaCoCo Java Code Coverage Library," https://www.jacoco.org/jacoco/.

[46] G. An, J. Yoon, and S. Yoo, "Searching for Multi-fault Programs in Defects4j," in *Proc. of the International Symposium on Search Based Software Engineering (ISSBSE)*, 2021, pp. 153–158.

[47] "Maven Central Repository," https://maven.apache.org/.

[48] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential Symbolic Execution," in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[49] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and Accurate Source Code Differencing,” in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[50] M. Harman and S. Danicic, “Amorphous Program Slicing,” in *Proc. of the International Workshop on Program Comprehension (IWPC)*, 1997, pp. 70–79.

[51] M. Harman, D. Binkley, and S. Danicic, “Amorphous Program Slicing,” *Journal of Systems and Software (JSS)*, vol. 68, no. 1, pp. 45–64, 2003.

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

[53] A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-inducing Input,” *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 183–200, 2002.

[54] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A Survey on Software Fault Localization,” *IEEE Transactions on Software Engineering (TSE)*, vol. 42, no. 8, pp. 707–740, 2016.

[55] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, “Darwin: An Approach to Debugging Evolving Programs,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 3, pp. 1–29, 2012.

[56] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, “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)*, 2017, pp. 117–128.

[57] 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.

[58] X. Xia, L. Bao, D. Lo, and S. Li, ““Automated Debugging Considered Harmful” Considered Harmful,” in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 267–278.