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
Canada
yi_li@ntu.edu.sg

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

Abstract*—Numerous program slicing approaches aim at helping 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 these techniques are key considerations toward their applicability in practice: inspecting only the statements identified in the slice should be faster and more efficient than inspecting the code in full. This paper reports on our experiment applying one of the most recent and accurate slicing approaches, dual slicing, to the task of troubleshooting regression failures in eight large, open-source software projects. The results of our experiments show that slices produced in this setup are still very large to be comfortably managed. Moreover, we observe that most statements in the slice deal with 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 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 77% shorter than the original ones for our case-study projects (350 vs. 2,617 code-level statements, on average), thus, reducing the amount of information developers need to inspect without losing the necessary contextual information. We believe our study and the proposed approach will help promote the efficient integration of slicing-based techniques in debugging activities and will inspire further research in this area.

***Index Terms*—Program slicing, regression failure, minimization, 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.

A large fraction of these approaches is based on Program slicing [1], [2] – a technique for computing a subset of program statements that affect a particular program variable or statement of interest via control and data dependencies. The classic slicing idea is further extended by numerous slicing variants, such as dicing [3], chopping [4], slicing with barriers [5], and thin slicing [6]. 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 [7], [8], [9] 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, (a) removing from the classic slice code statements that do not lead to diverging behavior between the versions, while (b) keeping code that is *not* executed in the regression version, if this code was executed in the baseline version and skipping it could lead to the failure.

Slicing approaches are typically evaluated either on relatively small code samples [4], [6], [10] or curated benchmarks of real faults [8], [9], such as, Defects4J [11] for Java programs. These studies demonstrate the effectiveness of the evaluated techniques, i.e., that they include in the slice changes responsible for the failure while keeping the size of the slice reasonably small. The latter is an important criterion as large slices are unlikely to be more helpful than the program itself. Yet, despite the promising evaluation results, slicing-based approaches did not make it into mainstream development and debugging tools yet.

Case Study. In an attempt to reveal and mitigate the reasons for this lack of adoption, in our work, we first conduct a study to investigate the applicability of slicing-based approaches for identifying regression failures in large software systems. In particular, we focus on the task of troubleshooting failing open-source software library upgrades. We picked this task for two reasons: first, our goal was to evaluate the approaches in realistic debugging scenarios, where multiple changes distributed in the program can simultaneously affect the test results. In contrast, Defects4J is tailored to study individual faults in isolation and the changes between program versions in this dataset are small [12], [13], [14].

The second reason for selecting our case study is its practical relevance and importance: while open-source libraries are commonly used in software development, a large fraction of library client projects do not use the latest versions of the libraries, even though more advanced versions contain fixes for critical bugs and security-related vulnerabilities [15], [16], [17]. For example, Wang et al. [18] showed that 53.4% of the clients they studied upgraded their dependencies within more than four months and 60.3% of the clients use libraries that are more than 20 versions away from the latest available version.

Developers delay library upgrades because these upgrades are often time-consuming and error prone [19], [20], [21]. Moreover, some library changes can be defective, causing developers to revert to the original library version [18].

Both client and library developers spend days, if not months, to identify the reasons for the failures and how/where to fix these failures. For example, the `org.jdbi` client developer spent more than 10 days figuring out why their code failed after 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 [22].» Another client and library developers debugged a failure for more than 20 days: «At this point I don't have time to start digging through bigger tests, commenting out things and so on – but if someone could help here trim tests down into core failure(s), more minimal/isolated case, I'd happily merge a PR [23].»

Motivated to investigate how slicing-based approaches can help developers troubleshoot such failures, we selected for our case study eight large and popular open-source library-client pairs, which include projects such as `jackson` and `antlr4`. We identified a library upgrade failure case for each client and applied dual slicing for producing slices relevant for the failure.

Our experiment showed that although slicing helped substantially reduce the scope of code that developers need to inspect in order to troubleshoot a failure, in the majority of cases, slices can still contain thousands of source-level code statements: 2,617 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 inspected. As such, techniques to further reduce the amount of information presented to developers are needed.

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

Slice Minimization Through Summarization. Inspired by the observations 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 while upgrading a popular database connection library, alibaba-druid, 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 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).

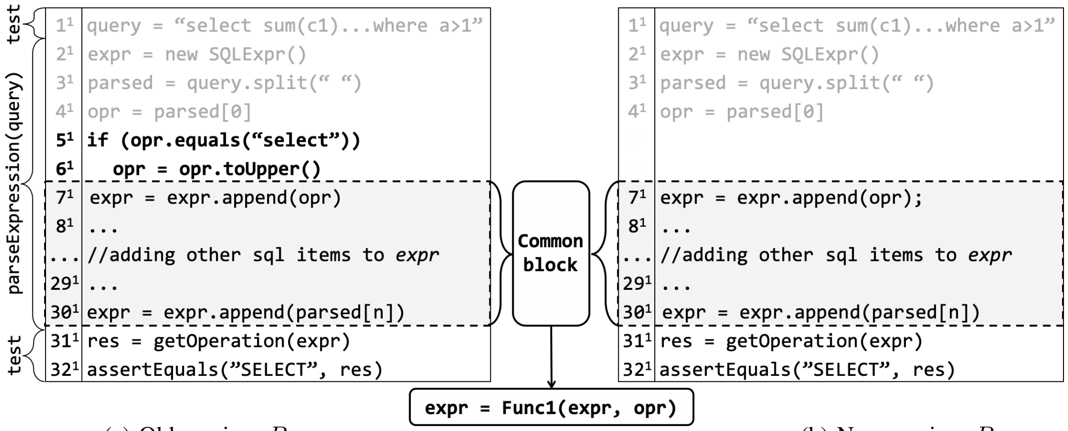
The expected result is successfully obtained in P_1 but the assertion fails in P_2. That is because P_1 ensured to capitalize the name of the operation (lines 5-6) while P_2 omitted these statements. The remainder of the code (lines 7-30) parse additional parts of the input query, building the output 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 this 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 in form of high-level input-output functions, while ensuring that variables needed for later computations are preserved and variables used for "internal" computations are hidden. In this example, we summarize the code in lines 7-30 as a statement `expr = Func1(opr, expr)`, which exposes the relationship between the inputs and the outputs of the summarized blocks without including unnecessary details.

We implemented the proposed approach for identifying propagation blocks and replacing them with their corresponding summaries in a tool named INPRESS (stands for *Context*



```
1¹  query = "select sum(c1)...where a>1"
2¹  expr = new SQLExpr()
3¹  parsed = query.split(" ")
4¹  opr = parsed[0]
5¹  if (opr.equals("select"))
6¹    opr = opr.toUpper()
7¹  expr = expr.append(opr)
8¹  ...
... //adding other sql items to expr
29¹ ...
30¹ expr = expr.append(parsed[n])
31¹ res = getOperation(expr)
32¹ assertEquals("SELECT", res)
```

```
1¹  query = "select sum(c1)...where a>1"
2¹  expr = new SQLExpr()
3¹  parsed = query.split(" ")
4¹  opr = parsed[0]
7¹  expr = expr.append(opr);
8¹  ...
... //adding other sql items to expr
29¹ ...
30¹ expr = expr.append(parsed[n])
31¹ res = getOperation(expr)
32¹ assertEquals("SELECT", res)
```

```
expr = Func1(expr, opr)
```

(a) Old version: P_1. (b) New version: P_2.

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

Summarization in Slicing). Our evaluation of InPreSS shows that it achieves around 77% reduction in the sizes of the slice, on average (max: 97.8%), on our large-scale case studies. As our work help further reduce slice sizes and, thus, the amount of information developers need to inspect when troubleshooting regression failures, we believe it can help promote more efficient integration of slicing-based techniques in debugging activities.

Contributions. This paper makes the following contributions:
1. We conduct a study investigating the applicability of slicing-based techniques for troubleshooting regression failures in eight large open-source library-client project pairs. Our study shows that the produced slices are large – up to 9,766 source code 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, our case studies, and all experimental data available online [24], 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 [25] 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 while slicing is typically done at the Java bytecode level. Thus, we assume that all conditionals other than `if`s, such as `for` and `while` loops, are expressed in terms of `if` and `jump` statements. Also, for simplicity of presentation, 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 the sequence of statement instances executed in a particular program run as 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 [26]. 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_i dominates s_j post-dominates s_i if s_j is always executed following the execution of s_i [27], i.e., either both statements are in the same method and s_j follows s_i in the control-flow graph or they are in different methods and s_i invokes the method whose first statement is s_j. For example, s_3 dominates s_4 in Figure 1. We say that a statement instance s_i^k dominates s_j^m if s_j^m is executed immediately after s_i^k in the same thread and s_i dominates s_j.

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 [28]. 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 [29], [2] 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 [30]. 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 statements 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 [30]. If V is omitted, it is assumed to include all variables used by c. A *backward dynamic slice* [30] 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.

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 [31]. Several trace alignment techniques have been proposed, based on string matching [32], memory indexing [33], and structural indexing [34], [9]. 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 [7] and later used for regression failures [8], [9]. 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. An example of such alignment is given in Section IV.

III. Case Study

We now describe the study we conducted to evaluate dual slicing in practice. We start by describing our selection of subjects and evaluation metrics; we then discuss the results.

A. Subjects

To identify our subjects, we started by collecting the five most-used libraries from each of the 30 different categories in Maven [35] (150 libraries in total). We further selected 128 libraries whose source code is available in GitHub. The size of these libraries, in terms of lines of code (LoC) as calculated by the JaCoCo tool [36], 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 pom.xml files of the remaining 427 projects, to get 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 each library available on Maven and filtered out client-library pairs with build errors (150 pairs), test errors (167 pairs), and all passing tests (2694 pairs). At the end of this process, we obtained a set of 168 client-library pairs, for which at least one client test fails after the upgrade. This set of pairs contains 84 unique libraries.

Building, running, and testing real open-source systems, especially those that use older library versions, is a labor-intensive task. We thus had to further restrict our choice of subjects and applied the following strategy: we ordered all 84 unique libraries used in our dataset by size and grouped them into 8 bins (four bins of 11 and four bins of 10 libraries). We then randomly selected a library from each bin and a client-library pair for further inspection. This process made sure that we have represented libraries of different sizes in our dataset.

Table I describes the selected subject client-library pairs. The first column of the table shows the subject id. The next six columns show the name of the library, its size, the number of

TABLE I: Subject Programs.

Subj.	Library	LoC	#Clients	P_1	P_2	#Ver. in Between	Client	LoC	Client Ver.
1	jettison	9,281	8,678	1.2	1.4	9	xstream	53,334	1.4.11
2	square-dagger	15,188	424	0.9.1	1.2.5	6	modelmapper	39,863	2.3.4
3	moshi	27,974	599	1.8.0	1.9.2	2	retrofit	26,252	2.6.0
4	jackson-core	48,868	192,377	2.9.9	2.10.1	5	mockserver	48,012	5.6.1
5	antlr4-runtime	56,918	4,113	4.5.3	4.7.2	3	fizzed/rocker	30,667	1.2.1
6	httpcomponents-client	72,440	438	4.5.1	4.5.10	9	wasabi	111,923	1.0.2019
7	jackson-databind	131,477	404,220	2.9.10	2.10.1	11	openAPI-generator	255,981	4.2.2
8	alibaba-druid	373,028	188,020	1.1.14	1.1.21	6	dble	166,021	2.19.07
Avg.	-	91,897	99,859	-	-	6	-	91,507	-

clients it has according to GitHub, and the version numbers for both the old and the new versions of the library that we used in our experiments. We also show the intermediate number of versions between the old and the new versions of each library. The table shows that our library selection ranges from relatively small to large libraries and includes one of the largest libraries in our dataset – `Druid`. Our full data set is available online [24].

B. Methodology

We implemented a version of the dual slicing algorithm based on recent work by Wang et al. [9]. 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 another recently developed tool, Slicer4J [37]. As Slicer4J relies on Soot [38] to instrument the bytecode of its input programs, it produces traces of Jimple statements [39]. We further map these statements to their corresponding source code statement instances using Soot.

For each of the study subjects, we ran the tool on both the base and the regression versions of the program, P_1 and P_2, respectively, using the failing regression test as an input. We recorded sizes of the execution trace (#T) and the produced dual slice (#DSlice) in terms of the number of statements they contain, and calculated the reduction rate achieved by dual slicing as $\frac{\#T - \#DSlice}{\#T}$ (%Reduct.). We also calculated the number of consecutively changed statement instances that a slice contains (#Chg). E.g., lines 5-6 in the example in Figure 1 are counted as one consecutive change. We report these metrics for each library-client pair collected in our work.

TABLE II: Applying DSlice on the Subject Programs.

Subj.	P_1				P_2			
	#$\mathbb{T}$	#DSlice	**%Reduct.**	#Chg	#$\mathbb{T}$	#DSlice	**%Reduct.**	#Chg
1	20,667	856	95.85	34	22,374	2,829	87.35	169
2	5,891	474	91.95	36	1,275	309	75.76	27
3	1,112	194	82.55	8	1,150	201	82.52	14
4	9,035	1,146	87.32	6	2,213	1,141	48.44	6
5	128,965	8,872	93.12	58	128,468	9,767	92.39	133
6	127	63	50.39	3	307	30	90.22	2
7	30,813	5,964	80.64	56	47,747	8,162	82.9	78
8	52,957	923	98.25	10	54,781	949	98.26	14
Avg.	31,196	2,311	85.01	26	32,289	2,923	82.23	55

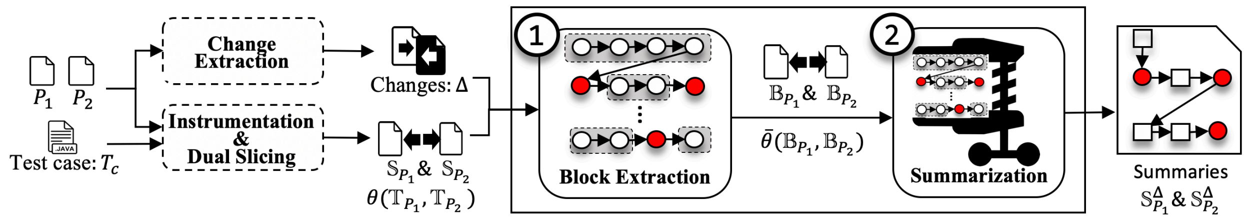
C. Results

Table II shows the results of our analysis. Each line of the table is divided into two main sections presenting the results for the old and the new version of the program - P_1 and P_2. The table shows that dual slicing is indeed very effective in reducing the size of the original execution trace: around 83.6% reduction rate on average (when considering P_1 and P_2 executions together). The main reason for this reduction is that traces are relatively large: more than 30,000 code-level statements, on average. Yet, they contain only a small number of changes: 24 consecutive changes, on average.

Moreover, we observe that, in several cases, e.g., for subject #8 where dual slicing achieves the highest reduction, changes are located at the end of the trace, i.e., close to the failure. for example, the first changed statement for the P_1 execution of subject #8 appears at position 50,693 of the trace. This gives dual slicing an opportunity to remove numerous statements preceding the change: more than 50,000 in this case.

However, despite the great reduction rate, the slice size for the majority of the subjects (including #8) is still very big: 2,617 source-level code statements on average, with a maximum of 9,767 for P_2 of subject #5. 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, in fact, inspired by the real case we observed. The common code block actually includes 223 statements in this case, which propagate manipulations related to managing expressions and do not directly contribute to the failure.

In another example, for subject #1 – the `jettison` library that converts between XML and JSON formats, the change resulted in a different structure of the JSON object, which further went through a long chain of serialization and deserialization operations (152 statements). These operations are, again, of low relevance to the failure. In yet another case, for subject #7 – the `jackson-databind` library that is used for data processing tasks, the change relates to removing an `if` condition validating certain properties of the input. This change further propagated to more than 2,000 statements that dual



P_1 P_2
Test case: T_c
Change Extraction
Instrumentation & Dual Slicing
Changes: Δ
$\mathbb{S}_{P_1}$ & $\mathbb{S}_{P_2}$
$\theta(\mathbb{T}_{P_1}, \mathbb{T}_{P_2})$
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. 2: INPRESS Overview.

slicing keeps due to a transitive control dependency on the removed `if`. *While in all these examples, common code blocks have little relevance to the failure, understanding the inputs and outputs of these blocks is necessary to put the relevant information about the failure in context, keeping the flow of information in the program.*

Our findings correlate with the observation that, in evolving software, common code blocks appear more frequently than changed code. In fact, the average size of common code blocks in the dual slice of the studied projects is 96 statement instances (max: 1032) and a slice contains 32 of such common blocks, on average. For comparison, a slice contains 24 changed blocks, on average, while the average size of the changed block is only 2.6 statement instances.

This presents an opportunity for slice minimization, e.g., by identifying and summarizing these large chunks of common code blocks, which mostly *propagate* information related to the change. Such minimization can further reduce the size of the slices, improving the effectiveness of slicing for further regression analysis. We discuss our approach, INPRESS, for performing such summarization next.

IV. SLICE MINIMIZATION APPROACH

Figure 2 shows the overview of INPRESS. It receives as input two slices, $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$, 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. It 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 the detailed propagation of information inside blocks of common code is not directly responsible for the test failure; only its effect on the remaining program statements needs to be preserved. The two main challenges of INPRESS are thus to (1) accurately identify those blocks of code that can be summarized and (2) concisely summarize each block while hiding internal data propagation and exposing those variables that are used in the following blocks.

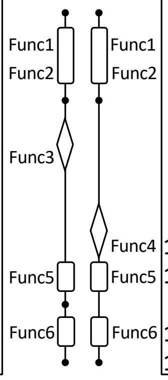
We now describe these two components using an example in Figure 3, which we created for illustrating different features of INPRESS. Figures 3a and 3b show the successful and failing runs of program versions P_1 and P_2, respectively. Like in Figure 1, we mark statements changed between the versions in bold and gray out statements removed by the dual slicing algorithm. This figure also exemplifies an interesting feature of dual slicing that was discussed in Section II – the ability to incorporate information *missing* from the run: as s^1_3 is in the backward slice of the P_1 trace in Figure 3a (as it is a data-flow dependency of the variable `h` in s^1_{13} through execution of the statements in lines 6,7,11-13), s^1_3 is also added to the slice for P_2 in Figure 3b, even though it does not impact the execution of any statement in P_2.

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. These mostly correspond to the changed statements (bolded in the figure). We refer to the set of preserved statement instances

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. 3: 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. 4: Summarized slices for the example in Figure 3.

as the *retained sets* and denote it by ρ_{P_1} and ρ_{P_2}, for $\mathbb{S}_{P_1}$ and $\mathbb{S}_{P_2}$, respectively. More specifically, we define the retain set ρ_{P_1} (ρ_{P_2}) to contain all statement instances corresponding to the statements added and modified in Δ_1 (Δ_2) and also add to each retained set the statement instance corresponding to the passing (failing) assertion. For the example in Figure 3a, ρ_{P_1} contains s^1_2, s^1_5, s^1_{12}, and s^1_{14}. We represent the retained set elements by dots in the schematic representation at the center of the figure.

Next, INPRESS identifies the set of *blocks* within each slice. Intuitively, a block is the longest sequences 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 instance 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 3a, $\mathbb{T}_{P_1}$, 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 3b, $\mathbb{T}_{P_2}$, by definition, has the exact same matched blocks and also an unmatched block: statement instances in lines 9-10. The retained set elements divide the blocks and are shown as dots in this schematic representation. 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. [40], 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 second unmached block in Figure 3a 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 are 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 4a (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.

Algorithm 1: Slice summarization algorithm.

```
Input: $\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\bar{\bar{\mathbb{B}}}$
Output: $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$
begin
    $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$ $\leftarrow \emptyset$
    $W_{P_1}$, $W_{P_2}$ $\leftarrow \emptyset$                          ▷ definitions to look for
    while $\mathbb{B}_{P_1} \neq \emptyset \vee \mathbb{B}_{P_2} \neq \emptyset$ do
        ProcessUnmatched ($\mathbb{B}_{P_1}$, $\mathbb{S}^{\Delta}_{P_1}$, $W_{P_1}$)
        ProcessUnmatched ($\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_2}$)
        ProcessMatched ($\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_1}$ $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1}$, $W_{P_2}$)
    return $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$
Procedure ProcessUnmatched($\mathbb{B}$, $\mathbb{S}^{\Delta}_{L}$, $W$)
    begin
        while $\mathbb{B} \neq \emptyset$ do
            $e \leftarrow get\_last(\mathbb{B})$                    ▷ get the last element
            if $e \in \bar{\bar{\mathbb{B}}}$ then
                return                      ▷ will be processed with its matched block
            else if $e \in \rho$ then
                $\mathbb{S}^{\Delta}_{L} \leftarrow e + \mathbb{S}^{\Delta}_{L}$        ▷ prepend the retained statement instance
                $W \leftarrow W \cup \{use(e)\}$            ▷ look for the variables it uses
                if $e$ is assignment then
                    $W \leftarrow W \setminus \{def(e)\}$   ▷ no need to track redefined variables
                      further
            else
                SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{L}$, $W$)  ▷ unmatched block
            $\mathbb{B} \leftarrow \mathbb{B} \setminus \{e\}$              ▷ remove the processed element
Procedure ProcessMatched($\mathbb{B}_{P_1}$, $\mathbb{B}_{P_2}$, $\mathbb{S}^{\Delta}_{P_1}$, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1}$, $W_{P_2}$)
    begin
        $e \leftarrow get\_last(\mathbb{B}_{P_1})$                   ▷ get the matched block
        SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{P_1}$, $W_{P_1} \cup W_{P_2}$)
        SummarizeDefinitions (e, $\mathbb{S}^{\Delta}_{P_2}$, $W_{P_1} \cup W_{P_2}$)
        $\mathbb{B}_{P_1} \leftarrow \mathbb{B}_{P_1} \setminus \{e\}$;   $\mathbb{B}_{P_2} \leftarrow \mathbb{B}_{P_2} \setminus \{e\}$    ▷ remove the processed
          elements
Procedure SummarizeDefinitions(block, $\mathbb{S}^{\Delta}_{L}$, $W$)
    begin
        $A \leftarrow summarize(block)$      ▷ represent the block as a set of assignments
        foreach $a \in A$ do
            if $def(a) \in W$ then
                ▷ a summary assignment defines a variable of interest
                $\mathbb{S}^{\Delta}_{L} \leftarrow a + \mathbb{S}^{\Delta}_{L}$          ▷ prepend the summary statement
                $W \leftarrow W \setminus \{def(a)\}$      ▷ not tracking redefined variables further
                $W \leftarrow W \cup \{use(a)\}$            ▷ look for the variables it uses
```

One important aspect of INPRESS is the ability to synchronize summaries of matched blocks. E.g., in Figure 3b, 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 4a. 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 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, $\mathbb{S}^{\Delta}_{P_1}$ and $\mathbb{S}^{\Delta}_{P_2}$, 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 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 3) or unmatched blocks (represented with diamonds in Figure 3). 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 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 3b, 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. We implemented the block extraction and summarization algorithms in Java. To identify common vs. changed code statements, we used GumTree [41], a state-of-the-art code differencing tool. GumTree identifies inserted, deleted, changed, and moved code statements, using an Abstract Syntax Tree (AST) structure [42] rather than a textual structure. We then mapped changed source code lines to Jimple statement instances.

For block summarization, we adapted the implementation of Slicer4J [37] 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 $\text{Func}_v(X)$ 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.

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?

RQ3 (Generalisability). How generalized is InPreSS when applied to general type of regression failures?

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 report both

TABLE III: Slice reduction rate achieved by INPRESS.

Subj.	P_1			P_2			Time (Min.)	
	#DSlice	#INPRESS	**%Reduct.**	#DSlice	#INPRESS	**%Reduct.**	DSlice	INPRESS
1	856	160	81.3	2,829	728	74.26	107.3	9.0
2	474	224	52.74	309	187	39.48	21.8	0.4
3	194	31	84.02	201	50	75.12	12.1	0.07
4	1,146	292	74.52	1,141	25	97.80	409.1	1.1
5	8,872	903	89.82	9,767	1,322	86.46	778.8	19.8
6	63	24	61.9	30	15	50	0.7	0.1
7	5,964	672	88.73	8,162	844	89.65	704.7	0.5
8	923	54	94.14	949	81	91.46	168.8	0.2
Avg.	2,311	295	78.4	2,923	406	75.53	274.8	3.6

the dual slicing and the slice summarization times, for all cases, in minutes. We performed all our experiments on an Ubuntu 18.04.4 Virtual Machine with 4 cores and 16 GB of RAM running on an in-house Ubuntu 16.04 server with 64 cores and 512 GB of memory.

Table III shows the results of our analysis, which we further separate into results for the old and new slices. The table shows that our slice summarization approach achieves a high reduction rate for both versions of each program: around 77%, on average, in both versions. Interestingly, like in the case of dual slicing, INPRESS achieves the maximal reduction rate of 92.8% on average for both versions of subject #8. This is because the original slice is relatively large, 923 statements in P_1, but the changes it contains are few and sparse: only 10 changed blocks, with 2.1 statement instances each, on average. At the same time, there are 14 blocks of common code, with 62 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.

The smallest reduction rate of 39.48% is for the P_2 version of subject #2, 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 traces of 309 statement instances in total. This provides only a limited opportunity for minimization: 45% of the original slice cannot be removed in any case. 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 almost 40% by summarizing common blocks of 4.8 statements, on average, into blocks of 1.1 statements 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 #5, 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 subject #6, with the smallest input trace and slice sizes.

Answer to RQ1: INPRESS is able to produce slice summaries that are around 77% shorter than the original slice, on average, in a matter of a few minutes. 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 on the overall effectiveness of INPRESS. To this end, we calculated the number of matched and unmatched blocks in each subject and the reduction rate achieved by summarizing only blocks of that type.

Table IV shows the results of this analysis. Similar to Table III, each row of the table corresponds to a subject program in Table I and further separates results for the old and new slices. The table shows that summarizing only unmatched blocks achieves a higher reduction rate for most of the cases: 69% and 66% on average for P_1 and P_2, respectively while the reduction rate for matched blocks is only 8% and 11%, on average. That is because the number of unmatched blocks is generally larger than the number of matched blocks (30 vs. 17 for P_1, on average) and their size is also larger (67.8 vs. 26.3, on average), giving more opportunities for reduction.

Our inspection of the case studies shows that there are three factors contributing to this behavior: first, the most prevalent change type in our subjects, appearing in 38% of the changes, is of conditional blocks, e.g., *if* and *loops*. An example of such cases is in the subject #7, discussed in Section III. The next common change type is the method call replacements (28%), which occurs in the subjects #1 and #4. As changes to the conditional blocks and method calls only cause control execution divergence, larger blocks of unmatched common code remain.

Second, several issues related to trace alignment (a generally hard problem) prevent some expected matches. For example, in subject #3, a refactoring early in the trace (method rename) led to a chain of eight unmatched blocks with more than 100 statements.

Finally, our decision to synchronize the size and type of matched blocks between versions resulted in several chunks of common code being split, producing smaller blocks, with fewer opportunities for eliminating internal calculations and variables. For example, in #7, 22 of 76 (28.9%) and 24 of 76 (31.5%) common blocks in P_1 and P_2, respectively, are generated because of the block synchronization.

Interestingly, for subject #6 in P_2, there is only one unmatched block and its size is one statement. Upon inspection, there are only two consecutive changes in the assignment statements in P_2, which leads to most of the executed code being matched.

TABLE IV: Applying INPRESS$_{\bar{\bar{\mathbb{B}}}}$ and INPRESS$_{\bar{\mathbb{B}}}$ on the Subject Programs.

Subj.	P_1						P_2					
	$\bar{\bar{\mathbb{B}}}$			$\bar{\mathbb{B}}$			$\bar{\bar{\mathbb{B}}}$			$\bar{\mathbb{B}}$		
	#	Avg. (Max)	**%Reduct.**	**#**	Avg. (Max)	**%Reduct.**	**#**	Avg. (Max)	**%Reduct.**	**#**	Avg. (Max)	**%Reduct.**
1	24	5.9 (37)	8.99	26	25.1 (247)	68.22	24	5.9 (37)	2.82	160	14.2 (475)	70.34
2	9	5.3 (13)	9.49	39	6.8 (20)	43.24	9	5.3 (13)	14.56	26	4.2 (18)	24.91
3	4	2.7 (6)	3.09	9	19 (91)	80.92	4	2.7 (6)	2.98	14	12.1 (91)	72.13
4	0	0 (0)	0.0	5	227 (1090)	74.52	0	0 (0)	0.0	5	224.8 (1095)	97.80
5	9	188.7 (1208)	11.1	54	130.5 (3569)	74.52	9	188.7 (1208)	10.77	131	59.1 (2035)	86.46
6	3	7.6 (13)	23.8	2	13 (25)	38.09	3	7.6 (13)	50	1	1 (1)	0
7	76	2.4 (34)	1.57	90	62.9 (1269)	87.15	76	2.4 (34)	1.12	102	76.5 (1360)	89.7
8	16	2.6 (13)	2.81	14	61.7 (292)	91.3	16	2.6 (13)	2.73	17	51.1 (290)	88.72
Avg.	17	26.3 (165)	7.59	30	67.8 (825)	69.68	17	26.3 (165)	10.62	57	55.1 (670)	66.1

Answer to RQ2: Factors affecting reduction rate include types of code statements that are changed (conditional v.s. assignment), types of changes made in the code (additions, removals, renames), and their relative position in the input traces (which influences the splitting algorithm), and the quality of the alignment algorithm. The combination of approaches employed by INPRESS increases its effectiveness for the majority of cases w.r.t. reduction rate.

C. *RQ3 (Applicability).*

To answer RQ3 and and validate the generalizability of our approach, we further applied INPRESS to Defects4J benchmark [11], which is typically used for large-scale evaluation of fault localization, software testing, and program repairs. This benchmark is composed of of 395 regression failures from six distinct Java programs. For each bug in the repository, Defects4J provides a faulty version and a fixed version, a minimized change set between two versions that represent the failure fix, and a test case triggering the failure. In our experiment, we regard the fixed version as the original, correct version (P_1) and faulty version as the regression version (P_2).

We discarded 128 out of 395 failures due to following reasons: (1) the execution of test case was non-deterministic (e.g. Mockito-20): they should be stable traces to replay the failure; (2) the test cases (e.g. Closure-1) were not compatible with our implementation: the test cases were not reproducible under java 8 or were organized in a non-conventional way, using scripts rather than standard JUnit test cases, preventing us from running with our tool and is left for future work; and (3) the faulty trace or the fixed trace is over-long (i.e., over 1 million steps) (e.g. Math-56): most of the trace alignment algorithms, including ERASE, do not scale for such over-long traces.

Table V shows the results of the evaluation, which we further separate into results for the old and new slices. Similar to our case studies, the table shows that our slice summarization approach achieves a high reduction rate for Defects4J bugs.

Answer to RQ3: INPRESS is not limited to libraries and its effectiveness can be generalized to regression failures generated by other type of applications. INPRESS is able to produce slice summaries that are around 93% shorter than the original slice, on average.

TABLE V: Applying INPRESS to Defects4J benchmarks.

Subj.	#Running/Total Bugs	P_1			P_2		
		#DSlice	#INPRESS	**%Reduct.**	#DSlice	#INPRESS	**%Reduct.**
Chart	25/26	45	8	83.07	47	6	87.69
Closure	78/133	1,999	120	94.01	2,459	106	95.69
Lang	56/65	22	6	73.56	54	13	75.35
Math	63/106	205	49	75.96	932	118	87.35
Mockito	20/38	271	24	91.28	753	20	97.34
Time	25/27	364	560	33	33	90.87	94.11
Avg.	267/395	696	53	92.32	1063	67	93.72

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 selecting top-used projects from the biggest Java library repository, Maven. In addition, to evaluate our approach in the context of prior work, we also applied INPRESS to externally created dataset – i.e., Defects4J – that is used by most of state-of-the-art techniques. 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 analyzing the results and discussing their findings. We make all our experimental data and implementation of the tool publicly available [24] 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 debugging techniques, we intend to explore integration with IDEs as part of future work. Additional limitations are inherited from the limitations of its 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 (several thousands statements).

VII. DISCUSSION AND RELATED WORK

We now discuss the related work, focusing mainly on existing fault localization techniques. We divide these techniques into two main categories: (1) techniques for extracting program slices relevant to the failure, and (2) techniques for pinpointing failure-inducing changes.

Identifying slices relevant to the failure. Program slicing – a technique for reducing the size of the program by identifying only those parts of 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 [43], [44]. The main idea behind this approach is to preserve 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 [45], [46]. These approaches report the smallest subset of changes that can be reverted to recover the original program behavior. Spectrum-based techniques, e.g., Tarantula [47], 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 [48]. 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 [49]. 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 [8].

VIII. CONCLUSION

In this paper, we investigated the applicability of slicing-based program analysis techniques to help narrow down developers' attention to a subset of program statements relevant for troubleshooting a regression failure. We conducted a study applying dual slicing – one of the most advanced and accurate dynamic slicing techniques – to a set of eight large open-source client-library programs with real upgrade failure scenarios. Our study allowed us to make two observations: (a) the slices reported by the technique are still relatively large to be comfortably inspected and (b) there is an opportunity to further reduce the size of the slice by retaining statements *responsible* for the failure and summarizing statements that *propagate* contextual information between the responsible statements. We implemented the proposed approach in a tool named INPRESS and showed its effectiveness in reducing the size of the slices by 77% in our case studies.

IX. 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 [24].

REFERENCES

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] “Jackson-databind (Issue-2789),” https://github.com/FasterXML/jackson-databind/issues/2789.

[23] “Jackson-databind (Issue-2876),” https://github.com/FasterXML/jackson-databind/issues/2876.

[24] “Supplementary Materials. Anonymized for the Review,” https://resess.github.io/artifacts/InPreSS/.

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

[26] 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.

[27] F. E. Allen, “Control Flow Analysis,” *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

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

[35] “Maven Central Repository,” https://maven.apache.org/.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] C. A. Welty, “Augmenting Abstract Syntax Trees for Program Understanding,” in *Proc. of the International Conference Automated Software Engineering (ICSE)*, 1997, pp. 126–133.

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

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.