Effective Minimization of Failure-Inducing Tests Using Convention-Aware Slicing

Sara Baradaran and Mukund Raghothaman
Thomas Lord Department of Computer Science, University of Southern California, Los Angeles, CA, USA
{sbaradar, raghotha}@usc.edu

***Abstract*—The minimization of failing tests is among the first steps in the debugging process. In this paper, we propose a slicing-based approach to this problem. The technique utilizes a new data flow analysis which leverages coding conventions to efficiently approximate the behavior of callee procedures without being aware of their code. This allows us to quickly derive smaller test versions which trigger the same failure as the original tests. We also incorporate static and dynamic analysis to preserve necessary control dependencies, which further guarantees the well-formedness and executability of the output tests.**

We have implemented our approach in a tool named FITSLICER, which performs test case minimization for Java programs. When applied to real-world failing tests from the Defects4J dataset, our evaluation shows that FITSLICER reduces the size of these tests on average by 40%. Moreover, when the minimized tests are executed, in 53% of the cases, they create failing traces which are on average 24% shorter than the original traces. These simplified test versions also allow DStar, the most effective spectrum-based fault localization formula, to identify 11% and 7% more true faults within its top-1 and top-5 predictions, respectively.

***Index Terms*—Test case minimization, program slicing, test simplification, test failure debugging**

I. INTRODUCTION

Alongside failure-triggering statements, failing tests typically execute portions of code that are unrelated to the actual defect, introducing noise and misleading information that hinders effective fault diagnosis. Therefore, test case minimization is arguably the first step in the debugging process. Besides its value to software developers, test minimization also promises to enhance the effectiveness of downstream software engineering tools for fault localization [1]–[5], vulnerability detection [6], and program repair [7]. Beyond improving the accuracy of these tools, the minimized tests are also faster to compile and execute. This helps accelerate the debugging process, particularly when repeated executions of failing tests are necessary, such as with mutation-based fault localization (MBFL) techniques [8], [9].

There are three basic requirements of any test case minimization algorithm: First, the minimized test must be *executable*, so as to allow developers to single-step through the program using a debugger, and for automated tools to collect coverage profiles. Furthermore, the minimized test must be *correct*, i.e., it should trigger the same failure as the original test. Finally, it must achieve substantial test size reduction, and it will be further helpful if the minimization algorithm itself runs quickly.

All three requirements are non-trivial: For example, in the case of Java programs, minimized tests are subject to type safety and control flow integrity checks by the Java compiler or the JVM bytecode verifier. As we will see, the minimized tests produced by traditional slicing algorithms would fail these checks. In addition, sound and precise data and control flow analysis of object-oriented programs is tricky, resulting in minimized tests that are either insufficiently small or exhibit different behaviors. This becomes even more intricate when test cases invoke API functions with black-box code.

In this paper, we introduce FITSLICER as a slicing approach for test minimization in Java software. It uses a new strategy for the data flow analysis of object-oriented programs with black-box code components. This allows the system to efficiently approximate the behavior of callee procedures at call sites, without being aware of their source code. These approximations are arranged in two levels, with different degrees of conservatism regarding possible data dependencies through black-box code. FITSLICER also uses a novel hybrid approach to calculate control dependencies by combining static and dynamic information about the test. In this way, it is able to produce minimized test versions that can be executed by the JVM and exhibit the same behaviors as the original unminimized tests.

We have implemented FITSLICER and evaluated its applicability on real-world failing tests from the Defects4J benchmark suite [10]. The results show that FITSLICER can generate test versions which are on average 40% smaller in size than the original tests. Also, in 53% of the cases, these minimized versions execute on average 24% fewer instructions to trigger the same failures as the original test cases.

To highlight the effectiveness of such test minimization in the debugging process, we compare the performance of well-known spectrum-based fault localization (SBFL) approaches in two distinct scenarios where failing coverage information is obtained from: (*a*) the original failing tests and (*b*) the minimized versions generated by FITSLICER. Our evaluation shows that in the second scenario, SBFL formulas can find on average 9.25% more true faults in their top-1 predictions.

II. MOTIVATING EXAMPLE

Consider the program in Figure 1 which is designed to test the calculation of cumulative probabilities. Lines 2–4 start by creating and initializing a standard normal distribution with mean 0 and standard deviation 1, i.e., $\mathcal{N}(0, 1)$. Then, in Lines 5–19, the cumulative probabilities of this distribution up to `i` and `-i` are calculated for increasing values of `i` and checked for appropriateness, i.e., whether $\Pr(x \leq \texttt{-i})$ is small and whether $\Pr(x \leq \texttt{i})$ is large. Unfortunately, this program throws an

```
1  public void testExtremeValues() throws Exception {
2    NormalDistribution distribution = (NormalDistribution) getDistribution();
3    distribution.setMean(0);
4    distribution.setStandardDeviation(1);
5    for (int i = 0; i < 100; i += 5) { // ensure no convergence exception
6      try {
7        double lowerTail = distribution.cumulativeProbability((double)−i);
8        double upperTail = distribution.cumulativeProbability((double) i);
9        if (i < 10) { // make sure not top−coded
10         assertTrue(lowerTail > 0.0d);
11         assertTrue(upperTail < 1.0d);
12       } else { // make sure top coding not reversed
13         assertTrue(lowerTail < 0.00001);
14         assertTrue(upperTail > 0.99999);
15       }
16     } catch (Exception e) {
17       throw new RuntimeException(e.getMessage());
18     }
19   }
20 }
```

Fig. 1: Example test case adapted from the Apache Commons Math library [11]. The test fails on Line 17 in the 9th iteration of the loop because of an unexpected exception which is thrown when calling `distribution.cumulativeProbability(-40)`.

```
1  public void testExtremeValues() throws Exception {
2    NormalDistribution distribution = (NormalDistribution) getDistribution();
3    distribution.setMean(0);
4    distribution.setStandardDeviation(1);
5    for (int i = 0; i < 100; i += 5) { // ensure no convergence exception
6      try {
7        double lowerTail = distribution.cumulativeProbability((double)−i);
8      } catch (Exception e) {
9        throw new RuntimeException(e.getMessage());
10     }
11   }
12 }
```

Fig. 2: A minimized version of the test case in Figure 1.

unexpected exception when calling `cumulativeProbability` method on Line 7, which in turn results in throwing a runtime exception on Line 17 and a test failure.

Debugging this test failure is not easy: If we instrument the Math library using the Soot framework [12] and run the test case in Figure 1, we discover that 241,825 Jimple statements were executed before the crash. Of course, not all portions of the code executed in this process would be relevant to the failure. For example, the upper tail calculation on Line 8 seems unlikely to influence the eventual test outcome. On the other hand, besides the failure-inducing invocation on Line 7, Lines 2–4 are essential for creating and initializing the `distribution` object, and the exception handling block is necessary to propagate this error to the underlying test harness. In addition to these statements, the `for` loop on Line 5 is also necessary to sufficiently increment `i` to the failure-triggering argument `-40`. The central goal of this paper is therefore to automatically minimize the test case so as to aid in debugging.

Of course, the problem of test case minimization is not new, going at least as far back as Zeller's foundational work on delta debugging [13]. Unfortunately, because delta debugging and related techniques need to test a large number of programs,

FitS	S4J
✓	✓
✗	✗
✗	✗
✓	✓
✓	✓
✓	✓
✓	✓
✓	✓
✓	✗
✓	✓

```
1  public void testEqualsHashcode() {
2    boolean equal;
3    String ptrn = "Pattern: {0,testfmt}";
4    ExtendMsgFormat mf1 = null;
5    ExtendMsgFormat mf2 = null;
6    mf1 = new ExtendMsgFormat(ptrn, Locale.US);
7    String ptrn2 = "X" + ptrn; // different pattern
8    mf2 = new ExtendMsgFormat(ptrn2, Locale.US);
9    if (mf1.hashCode() == mf2.hashCode()) {
10     equal = true;
11   } else { equal = false; }
12   assertFalse("pattern, hashcode()", equal);
13 }
```

Fig. 3: Example test adapted from the Apache Commons Lang library [15], which fails on Line 12 due to an assertion violation.

they are slow, necessitating hybrid approaches that rely on an initial static slicing pass [14]. While static slicing is fast, the underlying analysis approaches have to be conservative, and their over-approximation in dependency calculations ends up insufficiently minimizing the test case.

One could alternatively apply dynamic slicing, e.g., using Slicer4J [16]. However, this requires a thorough instrumentation and analysis of the entire program trace, including both the executed source code and the test case to identify necessary dependencies. This process is also time-consuming, and it may not be always possible to have a complete trace, for example, when API functions with black-box code are invoked. We also note that a dynamic program slice is *not* always executable: For example, consider the program in Figure 3, which fails with an assertion violation on Line 12. Given the slicing criterion $\langle$`equal`, 12$\rangle$ (i.e., to find all statements that influence the value of the variable `equal` at Line 12), Slicer4J outputs the lines marked with ✓. Note, however, that before compiling / loading the program, both the Java compiler and JVM bytecode verifier perform an analysis pass to confirm that all variables are initialized before use. Observe also that, in the generated dynamic slice, there is an execution path from the `false`-branch of the conditional statement on Line 9 to the assertion on Line 12, corresponding to a use of the `equal` variable before initialization. The JVM bytecode verifier would therefore not allow the output slice produced by Slicer4J to be loaded and executed.

With this background, FITSLICER, the minimization algorithm that we describe in this paper *guarantees* that the generated test versions are both executable and correct. At the same time, it achieves a high degree of minimization while running in a small amount of time.

Given the test case `testExtremeValues` in Figure 1 as the input, FITSLICER constructs the program dependence graph (PDG) in Figure 4. It then performs a traditional static slicing of this graph, with respect to the failure point, i.e., Line 17, as the slicing criterion. Finally, FITSLICER outputs the minimized test version in Figure 2 which executes 180,978 Jimple statements (25% less code) and results in the same failure as the original test procedure.

One key idea to construct the PDG of Figure 4 is that we

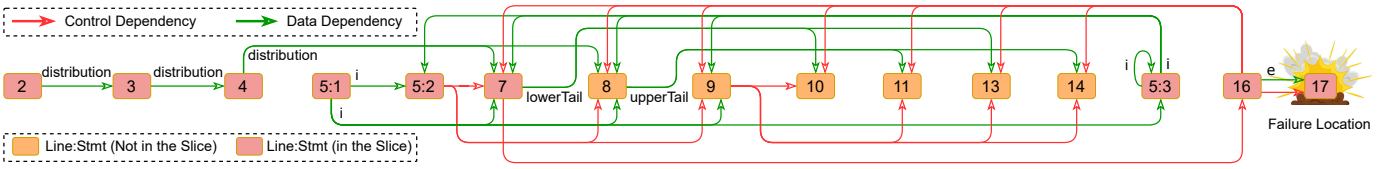

Fig. 4: Our constructed PDG representation of the test case in Figure 1.

efficiently approximate the set of variables which are likely to be redefined in the callee at the call site. For example, a conservative data flow analysis has to assume that every method invoked on a receiver object is likely to change the state of the object. This is frequently true of *setter-like* methods (defined as having a `void` return type), such as the `setMean` method on Line 3, which can be assumed to change the object state.

In contrast, notice that invoking the `cumulativeProbability` method is unlikely to affect the state of the `distribution` object. More generally, we assume that *getter-like* methods (defined as methods with a non-`void` return type) are unlikely to change the state of the receiver object, and accordingly do not add the associated data dependency edges to the PDG. Withholding these edges results in a smaller initial proposal for the sliced test.

Of course, this assumption is aggressive, and the sliced test might show a different behavior than the original test case. FITSLICER thus includes a hierarchy of increasingly conservative data flow analyzers, which allows it to start with an aggressively minimized test and roll back to a more conservative test slice so that the test behavior is preserved.

Next, observe that statically examining the source code in Figure 1 is insufficient to identify which statements in Lines 7–14 are likely to throw the exception. A conservative analysis therefore adds control dependency edges from each function invocation in the `try`-block to the corresponding `catch` statement. Once again, this would result in a larger test slice than is strictly necessary. We can instead run the test case and observe its execution flow, which reveals that the `catch` block is only triggered with an exception thrown from the method invocation on Line 7. This allows us to perform a more targeted insertion of control dependency edges into the PDG, thereby further reducing the size of the sliced test.

In this way, FITSLICER uses a hybrid multi-level slicing approach which combines information from static and dynamic analysis in order to minimize the size of the sliced test case. In our experiments with the Defects4J benchmarks in Section IV, we observe that this results in an average reduction of 40% in test size, while only requiring an average of 6 seconds to perform test minimization. We present the overall workflow in Figure 5, which we will formally describe in the next section.

III. FITSLICER DESIGN

We now describe the main workflow of FITSLICER as shown in Figure 5. Given a program and a failing test case, FITSLICER first instruments the test case and executes it in order to capture the test execution flow. Next, FITSLICER identifies the failure location as the last statement executed within the test procedure and calculates a series of backward slices with respect to each operand variable v in the failure point (see Algorithm 1). The union of the statements within these slices is returned as the minimized version of the test case, which *by definition* captures all statements that *might* affect the value of a used variable at the failure location. The minimized test version is then verified to ensure that it triggers the same failure as the original test.

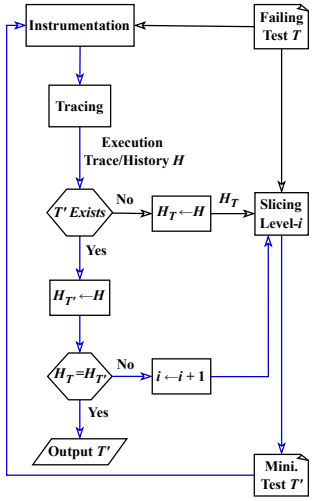

Fig. 5: FITSLICER workflow. The blue edges show the slicing cycle that is triggered after generating the first minimized version T' of the test T using ℓ_0. $H_T = H_{T'}$ denotes identical failure contexts, including the last executed statement and stack trace, for the two executions.

As discussed in Section II, we start by deliberately ignoring data flow paths deemed unlikely to be feasible due to standard coding conventions. This allows us to initially derive a fairly small test version. If the new test version fails to pass the verification step, then a more conservative slicing approach is adopted on demand and this cycle repeats until a correct executable test slice is obtained. In the following sections, we elaborate on different slicing levels and the backward slice computation of Algorithm 2.

1) Data dependency analysis: Conventional slicing relies on the program dependence graph (PDG) [17], which is in turn constructed by combining the data dependence graph (DDG) and the control dependence graph (CDG).

We start by describing how the underlying DDG is constructed from reaching definitions and DEF-USE chains. Note that calculating inter-procedural reaching definitions is not straightforward and a precise analysis can be expensive, specifically in the presence of reference variables such as objects and arrays whose manipulation in one procedure might affect other procedures that use the same variables. This forces us to only perform intra-procedural analysis of the test code.

The state of an object *may* change when (*a*) one of its methods is called, or (*b*) it is passed as an argument to other procedures. The latter also holds for other reference variables, such as arrays. Our first innovation thus involves a new way of analyzing data flow, which allows us to approximate the effect

TABLE I: Defined and used variables in 3-address statements. $A = \bigcup_{i=1}^{n}\{a_i\}$ is the set of all variables passed as callee arguments and $R = \{a_i \in A \mid a_i \text{ is a reference variable}\}$.

Stmt Type	Stmt Format	DEF(ℓ_0)	DEF(ℓ_1)	USE
Object/Array Instantiation	t = new Class$(a_1, \ldots, a_n)$ t = new Type$[x]$	$\{t\} \cup R$ $\{t\}$	$\{t\} \cup R$ $\{t\}$	A $\{x\}$
Field Load/Store	$t = u$.field u.field $= t$	$\{t\}$ $\{u\}$	$\{t\}$ $\{u\}$	$\{u\}$ $\{u, t\}$
Array Load/Store	$t = u[x]$ $u[x] = t$	$\{t\}$ $\{u\}$	$\{t\}$ $\{u\}$	$\{u, x\}$ $\{u, x, t\}$
Assignment	$t = u$ $t = u_1$ binop u_2	$\{t\}$ $\{t\}$	$\{t\}$ $\{t\}$	$\{u\}$ $\{u_1, u_2\}$
Return	return t return	$\varnothing$ $\varnothing$	$\varnothing$ $\varnothing$	$\{t\}$ $\varnothing$
If Goto	if u_1 binop u_2 goto L goto L	$\varnothing$ $\varnothing$	$\varnothing$ $\varnothing$	$\{u_1, u_2\}$ $\varnothing$
Throw Catch	throw t catch e	$\varnothing$ $\{e\}$	$\varnothing$ $\{e\}$	$\{t\}$ $\varnothing$
Method Invocation	$t = u$.method$(a_1, \ldots, a_n)$ u.method$(a_1, \ldots, a_n)$	$\{t\}$ $\{u\} \cup R$	$\{t, u\} \cup R$ $\{u\} \cup R$	$\{u\} \cup A$ $\{u\} \cup A$
Function Invocation	t = function$(a_1, \ldots, a_n)$ function$(a_1, \ldots, a_n)$	$\{t\}$ R	$\{t\} \cup R$ R	A A

of callee procedures on caller variables at invocation sites.

To calculate reaching definitions, we add receiver objects to the DEF set at call sites as well as the objects and other reference variables, including arrays, which are passed as arguments in call statements. This allows an intra-procedural DEF-USE chain analysis to consider that the new versions of these variables are used in the caller after returning from the callee, thereby preserving call statements which may be necessary to retain test behavior. We call this approach Slicing Level-1 (ℓ_1), and describe its associated definitions in Table I.

Of course, as discussed in Section II, the primary drawback of this approach is that it is extremely conservative. This results in large slices and is ineffective in eliminating irrelevant invocation statements from failing traces.

Our next observation is that even though invoking a method of an object can change its attributes, not all methods are equally likely to result in a state change. In particular, we note that *getters* simply return the result of stateless calculations, and do not frequently manipulate object attributes. In general, we believe that this is also frequently true of the more general class of *getter-like* methods, which we define as methods with a non-`void` return type. Therefore, we exclude receiver objects and argument variables from the DEF sets of such invocations in our more aggressive slicing strategy, which we call Slicing Level-0 (ℓ_0). Compare the DEF(ℓ_0) and DEF(ℓ_1) calculations for invocation statements in Table I.

On the other hand, we note that *setter-like* methods (constructors and methods with a `void` return type) are more likely to change the state of the receiver object, or of reference variables passed as input parameters. We retain the conservative DEF set calculations for such invocations even in Slicing Level-0.

We start Algorithm 2 by constructing the augmented control flow graph (ACFG) using the construction described by Choi and Ferrante [18]. Besides the edges in a regular CFG, the ACFG also contains a series of additional edges from each

FitS	S4J
✓	✓
✓	✓
✓	✓
✓	✓
✗	✓
✗	✓
✗	✗
✓	✓
✓	✗
✗	✗
✗	✗

```
1  public void testRemoveRowByKey() {
2    KeyedObjects2D data = new KeyedObjects2D();
3    data.setObject("Obj1", "R1", "C1");
4    data.setObject("Obj2", "R2", "C2");
5    data.removeRow("R2");
6    assertEquals(1, data.getRowCount());
7    assertEquals("Obj1", data.getObject(0, 0));
8    boolean pass = false;
9    try { data.removeRow("XXX"); }
10   catch (UnknownKeyException e) {
11       pass = true;
12   } assertTrue(pass);
13 }
```

Fig. 6: Example test from the JFreeChart library [22], failing on Line 9 due to an unexpected `IndexOutOfBounds` exception.

`goto` statement to its fall-through instruction. We then compute reaching definitions and construct the DDG by using the traditional iterative data flow analysis algorithm [19] and with respect to ACFG edges and the DEF/USE sets in Table I.

2) Control dependency analysis: Our second innovation is in CDG construction. We focus on two important aspects of our algorithm. First, as part of its type checking process, the JVM bytecode verifier confirms the type of elements on the operand stack. Ensuring that the sliced code is type-safe therefore requires careful consideration of `try-catch` blocks, *even if* an exception was not encountered at runtime. Second, static analysis of exceptional control dependencies results in large PDGs with numerous unnecessary edges, which compromises the effectiveness of test slicing. We employ a novel combination of static and dynamic analysis to precisely calculate the necessary CDG edges around `try-catch` blocks.

Algorithm 2 starts with an initial CDG on Line 3 which is constructed using the traditional approach by considering branching instructions (e.g., `if` statements) and computing dominance frontiers on the reverse CFG [20].

Now consider the test case in Figure 6 which fails with an unexpected `IndexOutOfBoundsException` on Line 9. This exception is uncaught, and immediately leads to a test crash. Naive backward slicing from this point would therefore conclude that the exception handler on Lines 10–12 was unnecessary and eliminate it from the minimized test. However, this exception handler is required for popping an operand of type `UnknownKeyException` from the JVM stack. Eliminating it causes the JVM verifier to detect an inconsistency and refuse to even load the new test for execution [21].

Therefore, we statically create *backward* edges from every `catch` statement to each statement within its corresponding `try` block (see Lines 4–6 in Algorithm 2). This forces the slicer to also include the `catch` statement as part of the minimized test, when statements from its associated `try` block exist in the slice, thereby ensuring the well-formedness of the output test.

Notably, one of our baselines, Slicer4J, does not include this well-formedness check. As a result, its output slice is not always executable by the JVM.

Next, every exception that is *actually* triggered results in a transfer of control from the exception-triggering instruction to

Algorithm 1: SLICING(T, H, ℓ), where T is a failing test case, $H = \langle t_1, t_2, \cdots t_n \rangle$ is the test execution history in which every t_i is an instance of a program statement s_i, and ℓ is the slicing level.

1 $t_n := H$.lastStmt, $s_n :=$ STMT(t_n), $T' := \{\}$
2 For each variable $v \in$ USE(s_n):
$T' := T' \cup$ BACKWARDSLICE$(T, H, \ell, \langle s_n, v \rangle)$
3 Return T' as the minimized test case.

the corresponding `catch` statement and subsequent execution of its block. A naive static analysis approach would therefore include an edge from every potentially exception-throwing statement within a `try` block to the corresponding `catch`. Unsurprisingly, this would result in an increase in CDG size and reduce the effectiveness of the minimization procedure.

In order to more precisely calculate these control dependencies, we instead incorporate dynamic information collected during test execution. In particular, we only focus on exception handlers that were actually executed, and for each such `catch` statement t_i, we create an edge in the CDG from the last executed statement s_{i-1} to s_i. We also create additional edges to each statement within a `catch` block from its corresponding `catch` statement. With these edges, including a statement from a `catch` block in the slice also forces the inclusion of both the `catch` statement and the exception-throwing statement into the sliced test. See Lines 7–11 of Algorithm 2.

3) Static test slicing: As mentioned earlier in Section II, the sliced test must be validated by the JVM bytecode verifier before it can be loaded and executed. Since the bytecode verifier relies on static analysis, even non-executable portions of the slice must follow type safety and control flow integrity rules. In our context, the most notable rule disallows the use of uninitialized variables along execution paths [21]. Note that in contrast to dynamic slicing, static slicing, which is adopted in Lines 12–17 of Algorithm 2, naturally results in code that conforms to these constraints.

However, one might worry about the execution overhead of using static slices compared to those produced using dynamic approaches. We will argue that this overhead is often negligible.

Consider a test fragment which is not executed at runtime as its controlling statement evaluates to `false`. If statements from this fragment are added into the slice at some point, then the branching instruction which controls their execution will be added as well. Now when this slice is executed, note that the branching statement again evaluates to `false`, and thus the execution overhead of a static slice against a dynamic slice in this example is simply the evaluation of a branch condition.

We show in our evaluation that despite using static slicing, FITSLICER-generated tests are smaller in size and execute less code compared to those produced by Slicer4J. We believe that this is because Slicer4J relies on conservative user-supplied summaries of the behavior of third-party library functions. Also note that FITSLICER treats the source code as a black-box and only relies on execution flow within the test procedure to perform its slicing. In contrast, dynamic slicers, e.g., Slicer4J, require a complete execution history. This causes FITSLICER to produce its output slices approximately $6\times$ faster than Slicer4J.

Algorithm 2: BACKWARDSLICE$(T, H, \ell, \langle s, v \rangle)$, where T is a failing test with the execution history H, ℓ is the slicing level, and $\langle s, v \rangle$ is the slicing criterion (slicing *wrt* variable v at statement s).

1 $ACFG :=$ AugmentedCFG(T)
2 $DDG :=$ ComputeDDG$(ACFG, \ell)$
3 $CDG :=$ RevDomFront$(ACFG)$
4 For each block $B \in$ CATCHBLOCKS with its corresponding try block B':
5 For each statement $s' \in B'.body$:
6 $CDG := CDG \cup \{B.first \rightarrow s'\}$
7 For each pair of consecutive statements $\langle t_{i-1}, t_i \rangle \in H$:
8 If t_i is an instance of catch statement:
9 $CDG := CDG \cup \{s_{i-1} \rightarrow s_i\}$
10 For each block $B \in$ CATCHBLOCKS and statement $s' \in B.body$:
11 $CDG := CDG \cup \{B.first \rightarrow s'\}$
12 $W := \{s' | s$ is control dependent on s'or data dependent *wrt* variable $v\}$
13 $workSet := W$, $S := \{\}$
14 While $(workSet \neq \emptyset)$ do:
15 $n :=$ remove$(workSet)$ and $S := S \cup \{n\}$
16 For each edge $n' \rightarrow n \in (DDG \cup CDG)$ if $n' \notin S$:
17 $workSet := workSet \cup \{n'\}$
18 Return the backward slice S.

IV. EVALUATION

Our evaluation sought to answer the following questions:

RQ1. How effective is FITSLICER in minimizing failing tests and how does it compare to other slicing approaches?

RQ2. How useful are the minimized test versions generated by FITSLICER for the downstream task of fault localization?

RQ3. How well does each slicing level preserve test behavior?

1) Implementation and experimental setup: All analysis in our implementation was performed over Jimple IR obtained using the Soot framework [12]. We also conducted all our experiments on a workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 22.04.

2) Benchmarks: We evaluated our approach on failing tests of 15 projects from the Defects4J (v3.0.1) dataset [10]. Table II provides detailed information about these projects. These are the projects from the latest version of Defects4J for which ground truth fault locations, which we rely on to answer **RQ2**, have been established in prior work [2]. Among the total 1,197 failure-triggering test cases of these 15 projects, we evaluated FITSLICER on 1,141 test cases. Since we rely on the failure location, e.g., failing assertion, crash site, as the criterion by which to perform slicing, we exclude tests that fail during the initialization portion of the test suite before the body of the test case is even executed. This includes 35 test cases in total. We also rule out failing tests which involve "*errors of omission*",

TABLE II: Subject software for test case minimization.

Project	#Faulty Versions	LOC	#Failing Tests
Chart	26	96K	92
Cli	39	4K	66
Codec	18	7K	43
Collections	28	65K	38
Compress	47	9K	72
Csv	16	2K	24
Gson	18	14K	34
JacksonCore	26	22K	53
JacksonDatabind	110	4K	130
JacksonXml	6	9K	12
Jsoup	93	8K	145
Lang	61	22K	120
Math	106	85K	176
Mockito	38	11K	118
Time	26	28K	74
Total	658	386K	1,197

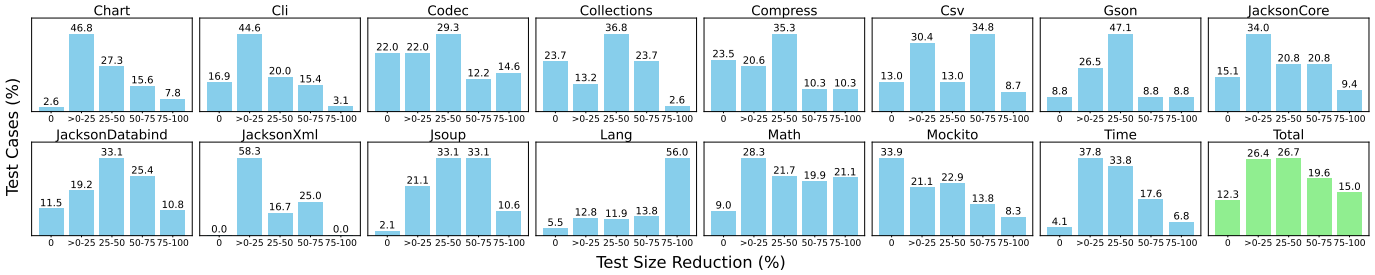

Fig. 7: Breakdown of test size reduction achieved by FITSLICER across the Defects4J dataset. Each bar represents the portion of tests which experience a size reduction in the specified range, and the zero-labeled bar indicates tests with no size reduction.

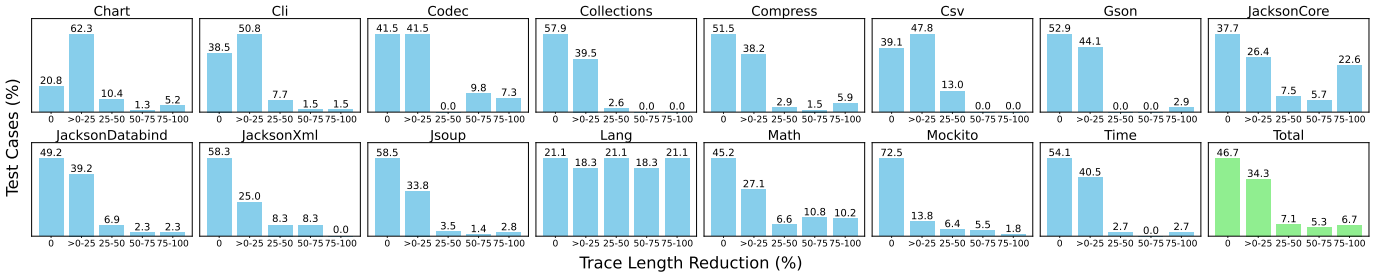

Fig. 8: Breakdown of trace length reduction achieved by FITSLICER across the Defects4J dataset. Each bar shows the portion of tests whose minimized versions generate shorter traces than the original tests with a length reduction in the specified range.

mainly in the form of expected exceptions that are not actually raised. In these cases, the failure does not manifest within the test procedure and graceful return from the test is implicitly interpreted as a failure. There are 21 such tests in total.

A. *RQ1: The Effectiveness of* FITSLICER

We quantify the effectiveness of our slicing approach on test case simplification using two measures: (*a*) the reduction in test size, and (*b*) the reduction in length of the failing traces.

1) Reduction in test size: To specify how well FITSLICER reduces the size of the failing test cases, we simply counted the number of Jimple statements inside the bodies of the original tests and their simplified versions. For each test case, we then computed the size reduction achieved through test minimization. Figure 7 summarizes these results.

Observe that a significant portion of the test cases which belong to the projects such as Lang, Math, and Csv experience notable size reduction (i.e., $\geq$ 50%) when they are minimized using FITSLICER. Overall, FITSLICER could reduce the size of 62% and 35% of the failing tests in our overall dataset by at least 25% and 50%, respectively.

2) Reduction in trace length: To measure the effect of FITSLICER on trace reduction, we started by instrumenting the source code and the test cases of each faulty version in the benchmark set. We then ran all the failure-triggering tests for each benchmark and collected their execution traces. Next, we used FITSLICER across all these failing tests to create their minimized versions. Once again, by running these new tests, we collected execution traces and calculated the reduction in trace length for each test. Figure 8 aggregates these results.

Notice that, in contrast to Figure 7, Figure 8 appears to show a smaller effect of FITSLICER in reducing trace lengths. However, this should not be surprising: many test cases fail at statements that reside early in their bodies. For such cases, the original trace is already heavily abbreviated, since a large portion of the test body remained *unexecuted*. Even though FITSLICER is successfully able to discard large portions of the test body, its effect on trace length of these cases appears to be smaller. Specifically, our measurement shows that only 21% of the tests with early failures (i.e., fail on the first 5 statements) experience trace reduction after minimization, while this number is 66% for the tests which fail later in their body.

Nevertheless, observe that the length of the execution traces significantly drops in projects such as Lang and JacksonCore. Overall, we observe a trace length reduction in 53% of all test cases. On average, these minimized tests execute 24% less code than the original ones. Additionally, in 23% of these cases, we observe a trace length reduction of at least 50%.

3) Comparison to baselines: We compare the performance of FITSLICER to two baselines: (*a*) Leitner et al.'s [14] static slicing, and (*b*) Slicer4J [16] which performs dynamic slicing of Java code. Leitner et al.'s slicing approach conservatively assume that receiver objects are redefined at *all* call statements regardless of the invocation type. This analysis is still unsound and ignores possible updates to arguments passed by reference at invocation sites. While this work does not consider control structures in tests, we combined the proposed DDG calculation with conventional CDG [20] to implement our baseline.

Figure 9 summarizes the results of our comparison with Baseline (*a*) on all test cases across the entire dataset. Observe

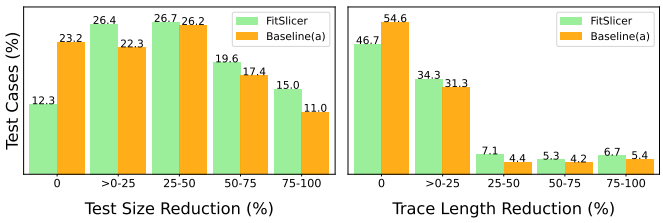

Fig. 9: A dataset-wide comparison of the test size and the trace length reductions achieved by FITSLICER and [14] respectively.

TABLE III: Performance of Slicer4J compared to FITSLICER for minimizing the set of 60 selected tests with the largest size.

Crash / Timeout	Incorrect	|S4J| > |FitS|	|S4J| = |FitS|	|S4J| < |FitS|
14 (23%)	32 (53%)	6 (10%)	7 (12%)	1 (2%)

that the baseline technique fails to minimize 23% of the failing tests, while this drops to only 12% when we use FITSLICER.

Overall, FITSLICER raises the number of test cases which experience more than 50% size reduction by 22% compared to the baseline. In addition, FITSLICER yields a 36% increase in tests whose minimized versions generate traces with more than 25% reduction in length over the baseline technique.

Although Slicer4J also relies on the Jimple IR, its implementation uses a different version of the Soot framework, resulting in minor discrepancies in the bytecode-to-Jimple translation. This requires manual inspection to map statements in our test slices to their counterparts in the output slice of Slicer4J, which in turn confines the feasibility of an automated comparison. We therefore limited our comparison to the 60 test cases with the largest size and present the results in Table III. We believe that these test cases have the greatest potential for size reduction, and provide a meaningful basis for evaluating the effectiveness of each slicing approach in test simplification.

Across 60 tests, FITSLICER reduces the size of these tests and their trace length on average by 66% and 36%, respectively. Slicer4J however fails to produce correct executable slices for a substantial portion of tests, i.e., 76%. The incorrect slices either make use of undefined variables, or they involve missing control constructs which causes the output slice to trigger a different failure or to be non-executable. Also, for 10% of the tests, Slicer4J creates test slices of larger size than FITSLICER.

B. *RQ2: The Usefulness of* FITSLICER *for Fault Localization*

To evaluate the effect of test minimization on the debugging process, we compare the performance of four spectrum-based fault localization approaches (i.e., DStar [3], Tarantula [5], Ochiai [23], Op2 [4]) under two different scenarios: First, by using the original test cases and second, by using the minimized test versions. This comparison allows us to measure how test case simplification influences the accuracy of fault localization techniques. Table IV summarizes the results of this experiment. Each row in this table indicates the improvement in the accuracy of an SBFL formula under the second scenario.

Observe that these minimized tests allow DStar, the most effective SBFL approach in the literature [24], [25], to achieve a 17% lower EXAM score and identify 11% more true faults within its *first* prediction for the most suspicious line of code.

TABLE IV: Improvement in the accuracy of SBFL techniques when using minimized test versions instead of the original tests. We perform our evaluation on the entire dataset from Table II.

Approach	ΔTop-1	ΔTop-3	ΔTop-5	ΔTop-10	ΔEXAM
DStar	11%	6%	7%	6%	-17%
Ochiai	6%	6%	6%	5%	-14%
Op2	12%	5%	7%	5%	-9%
Tarantula	8%	7%	6%	5%	-16%

TABLE V: Fraction of test cases in which each slicing mode is necessary in order to preserve the original test behavior.

FITSLICER	Slicing Level	Tests
	ℓ_0 (Aggressive)	93.2%
	ℓ_1 (Conservative)	1.1%
	ℓ_2 (No slicing)	5.7%

Baseline (*a*)	Slicing Mode	Tests
	Normal	82.4%
	No slicing	17.6%

C. *RQ3: Effect of Slicing Levels on Test Behavior*

We measured and summarized the fraction of test cases whose behavior are preserved by each slicing level in Table V.

Observe that more than 93% of the test cases can be minimized using the under-approximated DEF-USE relations while creating the DDG. Also, notice that falling back to ℓ_1 only manages to preserve test behavior on an *additional* 1.1% of all test cases, thereby validating our hypotheses about the state update behavior of getter- and setter-like methods.

In contrast, observe that the slicing algorithm of Baseline (*a*) fails to produce correct minimized test versions for 17.6% of the test cases. In these situations, the technique is forced to back off and simply reproduce the entire unminimized test case. As a comparison, this eventuality only occurs for 5.7% of the test cases processed by FITSLICER. We believe that this is because, in addition to the receiver objects, we also consider potential redefinition of reference arguments which are passed at call sites. This enhances the soundness of our data flow analysis and therefore the DDG construction.

Finally, note that ℓ_2 (i.e., no slicing) may be triggered when the test contains alias-inducing statements where operations on one object implicitly affect the state of another object. Since our alias analysis is limited, we may mistakenly disregard data flow paths in such cases. These missing edges in the DDG might result in the omission of statements that are required to preserve test behavior in slicing ℓ_0 and ℓ_1. We also observed time-sensitive test cases whose successful execution relies on `Thread.sleep`. Our data flow analysis may classify these invocations as unnecessary when they are not reachable from the failure point through either data or control dependency edges. Although such method calls do not affect data dependencies, their omission may influence scheduling and the sequence of operations, thereby creating invalid ℓ_0 and ℓ_1 test slices.

V. RELATED WORK

Zeller's delta debugging technique [13] is perhaps the earliest work on automated test case simplification, which uses an

iterative "*divide-and-test*" process to reduce failing inputs to their minimal forms. Leitner et al. [14] later combined delta debugging with static program slicing to accelerate test minimization. Their original research was in the context of test inputs that were automatically generated by AutoTest [26], which were straight-line, branch-free pieces of code. As a result, our primary worry, namely preserving executability of test cases, was not a consideration in their work.

Later developments in delta debugging include GTR [27], ProbDD [28], and WDD [29], which incorporate abstract syntax trees, probabilistic models, and weights of different program elements to accelerate the pruning process, respectively.

Instead of individual test inputs, a large body of research has focused on minimizing test *suites* to accelerate regression testing [30]. Most prior work views this problem as selecting a representative subset of tests which adequately examine the program behaviors [31]. In this context, Vahabzadeh et al.'s Testler [32] is notable as it identifies redundancies within individual tests to enable finer-grained test suite minimization. However, the primary focus of these techniques is in preserving coverage, rather than the faulty behavior of failing tests.

VI. Conclusion

We presented a *convention-aware* approach to slicing programs and investigated its application in minimizing failing tests. By paying attention to the type safety and control flow integrity requirements of the JVM bytecode verifier, our technique, FitSlicer, was able to effectively minimize Java tests while preserving original test behaviors. We also showed the effectiveness of these minimized test versions on the accuracy of spectrum-based fault localization techniques. FitSlicer is publicly available at github.com/SaraBaradaran/FitSlicer.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This research was supported in part by the U.S. National Science Foundation (NSF) under grant 2146518.

References

[1] Y. Wu, Y. Liu, Y. Yin, M. Zeng, Z. Ye, X. Zhang, Y. Xiong, and L. Zhang, "Smartfl: Semantics based probabilistic fault localization," *IEEE Transactions on Software Engineering*, 2025.

[2] S. Baradaran, Y. Huang, W. Le, and M. Raghothaman, "Prosecutor: Bayesian counterfactual fault localization," 2026. [Online]. Available: https://raw.githack.com/SaraBaradaran/SaraBaradaran.github.io/master/Prosecutor.pdf

[3] E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, 2014.

[4] L. Naish, H. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology*, 2011.

[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 273–282.

[6] S. Baradaran, M. Heidari, A. Kamali, and M. Mouzarani, "A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes," *International Journal of Information Security*, pp. 1277–1290, 2023.

[7] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," *arXiv preprint arXiv:2303.18184*, 2023.

[8] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Journal of Software: Testing, Verification and Reliability*, pp. 605–628, 2015.

[9] S. Wu, Z. Li, Y. Liu, X. Chen, and M. Li, "Gmbfl: Optimizing mutation-based fault localization via graph representation," in *IEEE International Conference on Software Maintenance and Evolution*, 2023, pp. 245–257.

[10] R. Just, D. Jalali, and M. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[11] "Apache Commons Math java library." [Online]. Available: https://commons.apache.org/proper/commons-math

[12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, p. 13.

[13] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, pp. 183–200, 2002.

[14] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 417–420.

[15] "Apache Commons Lang java library." [Online]. Available: https://commons.apache.org/proper/commons-lang

[16] K. Ahmed, M. Lis, and J. Rubin, "Slicer4j: a dynamic slicer for java," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1570–1574.

[17] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGSOFT Softw. Eng. Notes*, p. 177–184, 1984.

[18] J.-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements," *ACM Transactions on Programming Languages and Systems*, pp. 1097–1113, 1994.

[19] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973, pp. 194–206.

[20] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, pp. 451–490, 1991.

[21] "Java virtual machine class file format." [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[22] "Jfreechart." [Online]. Available: https://www.jfree.org/jfreechart

[23] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, pp. 1780–1792, 2009.

[24] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, 2021.

[25] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, pp. 332–347, 2021.

[26] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, "Automatic testing of object-oriented software," in *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2007, p. 114–129.

[27] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 861–871.

[28] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 881–892.

[29] X. Zhou, Z. Xu, M. Zhang, Y. Tian, and C. Sun, "Wdd: Weighted delta debugging," *arXiv preprint arXiv:2411.19410*, 2024.

[30] S. Rizwan, M. S. Ali Sobuj, and M. R. Akhond, "A survey on software test case minimization," in *Proceedings of the 14th International Conference on Contemporary Computing*, 2022, p. 679–684.

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, pp. 67–120, 2012.

[32] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering*, 2018, pp. 210–221.