Automated Inline-Test Generation without Relying on Method-Level Unit Tests

Pengyue Jiang
Cornell University, Ithaca, NY, USA

Yu Liu
The University of Texas at Austin, TX, USA

Anna Guo
The University of Texas at Austin, TX, USA

Milos Gligoric
The University of Texas at Austin, TX, USA

Owolabi Legunsen
Cornell University, Ithaca, NY, USA

Abstract

Inline tests validate individual program statements and expressions, and they detect many seeded faults (i.e., mutants) that unit tests miss in these target statements. ExLi is the only automated inline-test generation technique today; it carves inline tests from method-level unit tests that are written for methods that enclose target statements. Thus, ExLi cannot work for target statements that method-level unit tests do not cover. Also, the quality of ExLi-generated inline tests depends on the quality of method-level unit tests. We propose Smack to generate inline tests without relying on method-level unit tests. Smack is inspired by how inline tests are run: each is extracted with the target statement and run independently. Smack exploits this independence. Smack first extracts each target statement into a new method. Next, Smack applies a unit-test generator to the extracted method and carves inline tests from them. Finally, Smack transplants the resulting inline tests to be right after the target statement in the original enclosing method. We evaluate Smack on the same 957 target statements in 31 open-source projects that ExLi was evaluated on. Smack generates inline tests for 277 of 312 (88.8%) statements that ExLi *cannot* handle. These inline tests kill 96.7% of 1,815 mutants in these 312 target statements. Therefore, Smack improves on ExLi by extending the reach and fault-detection ability of inline-test generation. Smack also generates inline tests for 609 of 645 (94.4%) target statements that ExLi *can* handle. Smack-generated inline tests kill 83.1% of 2,844 mutants in these 645 target statements. 147 of these killed mutants survive ExLi-generated inline tests. So, Smack is also complementary to ExLi on target statements that ExLi can handle.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Software testing, inline tests, automated test generation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.13

Supplementary Material *Software (Source Code)*: https://github.com/SoftEngResearch/smack

Funding This work is partially supported by US NSF grants CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036.

1 Introduction

Inline tests [55] were recently proposed for validating single program statements (e.g., regular expressions). Inline tests are complementary to prior levels of test granularity – unit tests [12, 81], integration tests [28, 52, 69, 99], and end-to-end tests [100, 103] – which can be too coarse grained or ill-suited to meet some modern testing needs. For example, many bugs are caused by simple mistakes on single statements [43, 45, 47], but they are often missed

© Pengyue Jiang, Yu Liu, Anna Guo, Milos Gligoric, and Owolabi Legunsen;
licensed under Creative Commons License CC-BY 4.0

40th European Conference on Object-Oriented Programming (ECOOP 2026).
Editors: Robbert Krebbers and Alexandra Silva; Article No. 13; pp. 13:1–13:32

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

by unit tests, the previous lowest test granularity level [50]. Also, programming language advances now make it possible to write complex statements (e.g., lambdas, stream API call chains) where one would previously have written a method that can be unit tested.

```
1 - m = Pattern.compile("^(0-9A-F-){36}$").matcher(orig);
2 + m = Pattern.compile("^[0-9A-F-]{36}$").matcher(orig);
3   itest().given(orig, "00000000-0000-0000-0000-000000000000").checkTrue(m.matches());
4   if (m.matches()) { ... }
```

Figure 1 An example inline test that reveals a fault.

Inline tests help meet these testing needs and helped find previously unknown single-statement bugs. For example, the *target statement* on line 1 in Figure 1 mimics a real-world bug that Liu et al. found using inline tests [55]. There, the regular expression is intended to match 36-character UUID strings like `"00000000-0000-0000-0000-000000000000"`. But, the faulty regular expression on line 1 uses `"("` and `")"` instead of `"["` and `"]"`, so it only matches `"{0-9A-F-"` followed by 36 repetitions of `"}"`. This fault was not found by method-level unit tests for over four years.

An *inline test* is "a statement that allows to provide arbitrary inputs and test oracles for checking the immediately preceding statement that is not an inline test" [55]. Inline tests run only during testing, never in deployed software. Line 3 in Figure 1 shows an example inline test; it has the following syntax: `itest()` declares a statement to be an inline test, `given()` assigns inputs to target-statement variables, and `checkTrue` is a test oracle; it checks if the regular expression in the target statement matches as expected. The oracle in this inline test fails on the original code, helping find the fault, which developers fixed as shown on line 2.

The original paper on inline tests [55] established requirements for inline-testing frameworks, proposed prototypes for two such frameworks (for Python and Java), found that inline tests incur tiny overheads, and reported on a user study. All user study participants were able to quickly and easily write inline tests. Liu et al.'s prototype for Python has been developed into a full-fledged tool, `pytest-inline` [57, 76], that was adopted as an official plugin [40] for pytest, the most popular testing framework for Python.

To increase the chance of adoption of inline tests, ExLi [54, 56] was proposed as an automated inline-test generation technique and tool for retrofitting inline tests into existing code. ExLi uses test carving [16, 24, 46, 82, 104] to obtain inline tests from method-level unit tests that cover a target statement. ExLi-generated inline tests killed about 25% more mutants than developer written or automatically generated unit tests [54], increasing the fault-detection ability of test suites from which they are extracted. The reason is that oracles in those inline tests check parts of state that unit tests' assertions do not check, or that do not propagate to those assertions.

ExLi has two main limitations. First, ExLi cannot generate inline tests for target statements that are not covered by method-level unit tests. This limitation negates an important benefit of inline tests: they can validate target statements that method-level unit tests do not cover. In fact, inline-testing tools do not run the enclosing methods of target statements. To illustrate this limitation, consider the target statement on line 7 in Figure 2, whose enclosing method is simplified from wmixvideo/nfe [97]. Method `getVerificationDigit` computes a verification digit for an access key. No developer written method-level unit test covers line 7 and all automatically generated unit tests (after three hours) use inputs on line 3 that cause the exception on line 14 to be thrown. So, ExLi cannot generate inline tests for this target statement. Second, the quality of ExLi-generated inline tests depend on the quality of method-level unit tests. Specifically, values used as inputs and expected outputs

```
1  public class MDFGenerateKey {
2   public Integer getVerificationDigit() {
3    final char[] v = generateAccessKeyWithoutVerificationDigit().toCharArray();
4    int vTmp;
5    for (int i = v.length; i > 0; i--) {
6     ...
7     vTmp = Integer.parseInt(String.valueOf(v[i - 1]));
8     itest().given(i, 1).given(v, new char[]{'7','/',';','Y','Y','1'}).checkEq(vTmp, 7);
9     ... }
10  ... }
11
12   private String generateAccessKeyWithoutVerificationDigit() {
13     if (StringUtils.isBlank(mdfe.getInfo().getIdentification().getNumericCode())) {
14       throw new IllegalStateException("Numeric code must be present to generate the
            access key");
15     }
16     return StringUtils.leftPad(mdfe.getInfo()/*long chain*/);
17   }
18 }
```

Figure 2 A target statement (line 7) that ExLi *cannot* handle but Smack can (line 8).

in ExLi-generated inline tests are those observed at the target statement while running method-level unit tests. Such values may be insufficient for detecting seeded faults (i.e., mutants) on target statements.

We propose Smack to generate inline tests without relying on method-level unit tests. Smack is based on an idea – *extract-test-transplant* – that is inspired by how inline tests are run. Inline-testing tools like `pytest-inline` [57] for Python or ITest for Java [55] (i) extract a new method containing only a target statement and its inline tests; (ii) run the extracted method and its inline tests in an environment that is independent of the target statement's original enclosing method; and (iii) report the inline tests' pass/fail outcomes.

Smack works in four steps to generate inline tests from two inputs: a target statement and a unit-test generator. First, Smack extracts a method that contains only the target statement and any imports needed to run the target statement. Second, Smack runs the unit-test generator to obtain unit tests for the extracted method. Third, Smack carves inline tests from the execution of the generated unit tests. Finally, Smack "transplants" the generated inline tests to be after the target statement in its original (not extracted) enclosing method. Our current Smack implementation uses Randoop [70, 71, 96] and EvoSuite [18, 21, 23, 83] as unit-test generators. The inline test on line 8 in Figure 2 is generated by Smack. § 3 discusses Smack in more details.

We evaluate Smack and compare it with ExLi in three ways, using the same 957 target statements from 31 open-source projects that Liu et al. evaluated ExLi on [54]:

1. Smack generates inline tests for 277 of 312 (88.8%) target statements that ExLi *cannot* handle (because no method-level unit test covers them). So, Smack extends the reach of inline-test generation to statements that are not covered by method-level unit tests.

2. Smack generates inline tests for 609 of 645 (94.4%) target statements that ExLi *can* handle (because they are covered by method-level unit tests). So, Smack generalizes to target statements that are covered by method-level unit tests as well.

3. Smack achieves a mutation score (a percentage measure of seeded faults detected) of 96.7% on the 312 target statements that ExLi cannot handle. So, Smack also improves on ExLi in terms of fault-detection ability. We also compare Smack's fault-detection ability with ExLi's on the 609 target statements that they *both* handle, to compare mutation scores of inline-test generation with (like ExLi does) vs. without (like Smack does) relying on

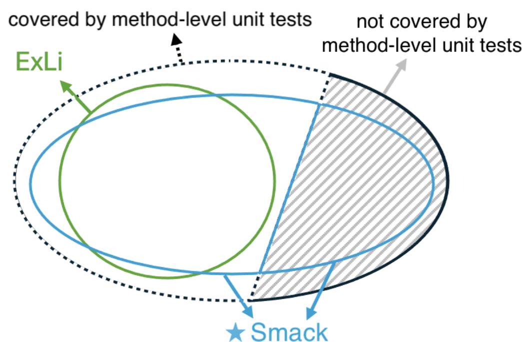

Figure 3 The gap in inline-test generation that SMACK fills (not drawn to scale).

method-level unit tests. SMACK's mutation score is 83.1%, compared to 87.7% when running EXLI with developer written *and* automatically generated method-level unit tests. Also, SMACK kills 147 mutants that survive EXLI-generated inline tests, while EXLI kills 222 mutants that survive SMACK-generated inline tests. Thus, SMACK is complementary to EXLI on target statements that EXLI can handle.

Overall, SMACK meets an important need in inline-test generation, as depicted in Figure 3 (not drawn to scale). There, the striped portion represents target statements that only SMACK can handle today (because they are not covered by method-level unit tests). The plain portion of Figure 3 represents target statements that EXLI can handle, but to which SMACK generalizes and provides complementary fault-detection ability.

This paper makes the following contributions:

- **Technique.** SMACK is the first technique that generates inline tests without relying on method-level unit tests. SMACK decouples inline-test generation from the problem of writing or generating high-quality method-level unit tests.
- **Evaluation.** SMACK generates inline tests that kill most mutants on statements that are not covered by method-level unit tests. Also, SMACK and EXLI kill different sets of mutants on statements that are covered by method-level unit tests. So, SMACK is complementary to EXLI.
- **Prototype and dataset.** Our SMACK prototype is publicly available for future research to build on. We also show SMACK's utility by adding 2,108 new inline tests that it generates to the growing publicly-available corpus of inline tests.

SMACK and our dataset are at `https://github.com/SoftEngResearch/smack`; that link also contains an appendix with additional details on our experiments.

2 Examples and Further Motivation

§ 2.1 introduces EXLI, whose limitation inspired SMACK. Next, § 2.2 illustrates SMACK. § 2.3 further motivates SMACK via an experiment in which we run Randoop [71] and EvoSuite [21] to generate more method-level unit tests for extending EXLI's reach. § 2.4 discusses examples of target statements that remain uncovered by automatically generated method-level unit tests after hundreds of hours of generation, using various seeds. Lastly, § 2.5 introduces mutation testing, which we use to evaluate inline tests' fault-detection ability.

```
1 public class MDFGenerateKey_7 {
2   public void targetStmtGenerated(char[] v, int i) {
3     int vTmp;
4     vTmp = Integer.parseInt(String.valueOf(v[i - 1]));
5   }
6 }
```

Figure 4 Extracted class for the target statement in Figure 2.

2.1 ExLi: The only prior automated inline-test generation technique

ExLi takes a target statement and (developer written or automatically generated) method-level unit tests that cover that statement. Then, ExLi generates inline tests for the given target statement in four steps. First, ExLi records all observed values of input and output variables at the target statement while running the given method-level unit tests. Too many values of input and output variables may be recorded, e.g., if a target statement is covered by many unit tests or is in a loop. ExLi's second step is to reduce the set of recorded values by eliminating redundant inputs based on coverage. Third, to preserve the bug-finding ability of the original recorded set of values, ExLi adds to the initial reduced set any recorded set of values from the original set that kills a mutant that survives the initial reduced set. Finally, ExLi transforms the final set of input and output variable values into inline tests right after the given target statement.

2.2 Smack by example

Figure 2 shows a target statement on line 7 and a Smack-generated inline test on line 8. ExLi cannot handle this target statement because no developer written or automatically generated method-level unit test for `getVerificationDigit` covers line 7.

We now illustrate Smack's four main steps that enable it to generate inline tests without relying on method-level unit tests for the enclosing method of the target statement. First, Smack extracts the target statement on line 7 in Figure 2 into a method in a new class. Figure 4 shows such a class, `MDFGenerateKey_7`, that is named after the original class and the line number of the target statement in that original class. The extracted `targetStmtGenerated` method's arguments are the input variables `v` and `i` on the target statement. To ensure compilation, Smack also extracts the declaration of output variable `vTmp` from the original method to the extracted `targetStmtGenerated` method in `MDFGenerateKey_7` (line 3).

Second, Smack runs a unit-test generator on the extracted method. Since the extracted method has a simpler control flow than the original enclosing method (e.g., `getVerificationDigit` in Figure 2), unit-test generation will cover the target statement in the extracted method. Whereas all values generated using ExLi failed to reach the target statement due to the exception on line 14, Smack-generated variable values will not cause that exception to be thrown. Some values (e.g., a single-element `char` array for `v` and 2 for `i`) that Smack uses may cause `ArrayIndexOutOfBoundsException` to be thrown instead, but these can be valuable if they are converted into exception-expecting inline tests. Third, Smack generates inline tests using the input and output values observed on the target statement when running the generated unit tests. Finally, Smack "transplants" the generated inline tests to be right after the target statement in its original enclosing method, e.g., line 8 in Figure 2.

Internally, Smack uses a reduction step to eliminate generated inline tests that are redundant to others with respect to mutants killed on the target statement. This step is needed because Smack can initially generate too many inline tests for a target statement

```
1  public void export(ITextNode[] nodes, String mlang, String lang, Status[] states)
2      throws ExportException {
3    ...
4    try (OutputStream os = outputStreamFactory.createOutputStream(outputFile);
5         BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(os, "UTF-8"))) {
6      synchronized (this) {
7        ...
8        for (ITextNode node : nodes) {
9          IValueNode v = node.getValueNode(lang);
10         if (v != null) {
11           if (states == null || TremaCoreUtil.containsStatus(v.getStatus(), states)) {
12             ... // "key" is defined here
13             if (value != null) {
14               String f = resolveIOSPlaceholders(value);
15               String r = String.format("<string name=\"%s\">%s</string>", key, f);
16               bw.write(r);
17               bw.write("\n");
18             }
19           }
20         }
21       }
22       ...
23     }
24     bw.flush();
25   } catch (IOException e) {
26     throw new ExportException("Could not store properties:" + e.getMessage());
27   }
28   ...
29 }
```

Figure 5 A target statement in `trema−core` that method-level unit tests do not cover.

whose SMACK-extracted class (e.g., Figure 4) is covered by many generated unit tests. In Figure 2, SMACK keeps only one inline test after reduction because the other inline tests cannot kill a mutant that changes `v[i-1]` to `v[i]` on the target statement, but the inline test on line 8 can kill this mutant and all others on the target statement.

2.3 Further empirical motivation for Smack

We check if varying seeds used by Randoop and EvoSuite, and running these unit-test generators for longer periods, could help EXLI generate inline tests for more target statements. To do so, we configure EXLI to use five non-default seeds. Then, we run Randoop using a three-hour timeout for each project, and EvoSuite using a two-minute timeout for each class containing target statements, for a total of 363.5 CPU hours (i.e., over 15 CPU days).

Taking a union of generated unit tests from both tools, 431 target statements are still not covered. Even after combining existing developer written unit tests with these automatically generated method-level unit tests, 312 target statements remain uncovered. We manually confirm that these 312 statements are hard to reach by automated means, as opposed to being unreachable. Also, 293 of these 312 statements (i.e., 93.9%) are in a method that developer written or automatically generated method-level unit tests partially cover.

These empirical results suggest that varying seeds in Randoop and EvoSuite, and running them longer does not suffice to cover many target statements, further motivating the need for a new approach for generating inline tests without relying on method-level unit tests.

```java
protected String getTagValues(Item item) {
  String value = defaultValue;
  Document document = item.getDescriptorDom();
  if (document != null) {
    String srcVal = XmlUtils.selectSingleNodeValue(document.getRootElement(), srcField);
    if (StringUtils.isNotEmpty(srcVal)) {
      for (Map.Entry<String, String> entry : valueMapping.entrySet()) {
        if (srcVal.matches(entry.getKey())) {
          value = entry.getValue();
          break;
        }
      }
    }
  }
  return value;
}
```

Figure 6 A target statement in `craftercms/core` that method-level unit tests do not cover.

2.4 Examples showing the limits of relying on method-level unit tests

In Figure 5, the target statement on line 15 is intended to write `key` and a formatted `String f` (derived from `value`) into an XML tag. This target statement is in a three-level deep nested `if` statements inside a `for` loop, which is in a `try-catch` block. So, it is hard for Randoop and EvoSuite to set up the complex input needed to reach line 15. Also, no developer written test covers this target statement, suggesting that developers find it hard to write method-level unit tests that reach it. But, Smack's extract-test-transplant technique allows it to more easily generate inline tests for that target statement (not shown in Figure 5).

Line 8 in Figure 6 shows another example target statement that ExLi cannot handle. Similar to Figure 5, this target statement is deeply nested: it is in an `if` statement in a loop that is nested in two other `if` statements. Randoop and EvoSuite do not create a `Document` object (line 3) that is valid and non-empty (line 6). The developer written tests also do not create such an object, so they do not reach the inner-most `if` statement containing the target statement. However, Smack generates 2 inline tests for this target statement (not shown).

2.5 A brief background on mutation testing

Inline tests were proposed only recently, so there is a lack of reasonably-sized datasets of real-world bugs that inline tests find. Therefore, in our evaluation of fault-detection ability of inline tests, we artificially seed faults in target statements using mutation testing. Mutants are programs obtained by introducing small syntactic variations to the original program. Invalid mutants are mutants that are syntactically invalid and thus cannot be executed. Mutators are rules that generate mutants, for instance, rules for transforming binary operators. An example mutant that universalmutator– the mutation testing tool that we use in this paper – generates for the target statement in Figure 4 is:

```java
  // Original Statement
  vTmp = Integer.parseInt(String.valueOf(v[i - 1]));
  // Mutated Statement
  vTmp = Integer.parseInt(String.valueOf(v[i - (-1)]));
```

The hypothesis in mutation testing is that tests that are effective at detecting mutants should detect real faults as well [44, 74]. If a passing test fails on the mutant, that mutant is said to be *killed* by the test. Otherwise, the mutant is said to *survive*. The mutation score is the ratio of mutants killed to the total number of mutants.

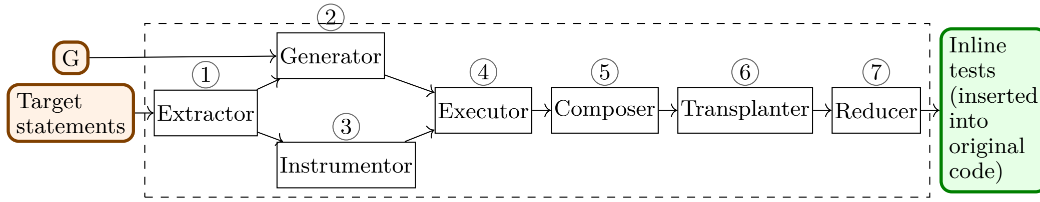

Figure 7 An overview of SMACK and its components.

Algorithm 1 The steps in SMACK's Extractor.

1: **procedure** EXTRACTOR(S)
2: $targets \leftarrow \emptyset$
3: **for** each s in S **do**
4: $c^s \leftarrow$ NEWCLASS()
5: $args \leftarrow$ BUILDARGS(s.code)
6: $imports \leftarrow$ EXTRACTDEPS(s)
7: $m^s, \delta^s \leftarrow$ MAKEMETHOD($args$, RW(s))
8: $c^s \leftarrow$ ADDTOCLASS(c^s, $imports$, m^s)
9: $targets \leftarrow targets \cup (c^s, \delta^s, s)$
10: **return** $targets$

3 Technique and Implementation

We describe SMACK (§ 3.1) and its implementation for Java (§ 3.2).

3.1 Smack

Figure 7 is an overview of SMACK, which takes two inputs: a set S of target statements and unit-test generator `G`. Each target statement $s \in S$ is associated with metadata, including source file (s.file), line number (s.loc), and code (s.code). SMACK's outputs are updated source files in the code under test (CUT), where each target statement is immediately followed by a (possibly empty) set of generated inline tests. Next, we describe each SMACK component in the order shown in Figure 7.

① Extractor. This component extracts target statements from their original enclosing method (e.g., `getVerificationDigit()` in Figure 2) into a new method (e.g., `targetStmtGenerated` in Figure 4) in a new class (e.g., `MDFGenerateKey_7` in Figure 4). Figure 1 shows the Extractor's steps. For each target statement s, the Extractor creates a new class c^s containing only method m^s, whose arguments are the input variables used in s.code. The argument types are obtained via analysis of s.file and its dependencies. The body of m^s is the output of rw(s) – line 7 in Figure 1; rw applies rewrite rules in Table 1 to increase the chance that the extracted code compiles. But, the Extractor cannot rewrite statements that call methods that return `void`. Also, return statements are not extracted: unit tests are better suited for testing them. In principle, the Extractor can rewrite all other statements, but Table 1 shows what kinds of rewriting SMACK currently supports. The Extractor also records δ^s, the modifications that rw makes to s, so that the Transplanter can later reverse them. In sum, the Extractor takes a set S of target statements and produces the set E^1:

$$\{(c^s, \delta^s, s) \mid s \in S\} \quad (1)$$

Table 1 A list of rewrite rules that the Extractor component uses to transform target statements.

AST Node	Rule	Example
The `this` expression	Remove `this`	`this.var` → `this_var`
Static nested class	Import the class	`fields.stream().map(SubField::new)` → remove and add an import: `import Field.SubField;`
Checked exception	Add a `throws` clause	`void targetStmtGenerated()` → `void targetStmtGenerated() throws XXException`
Generic type	Change to a type to which it resolves, or `Object`	`T a = b;` → `Object a = b;`
Missing declaration	Add declaration before the target statement	`a = b;` → `int a; a = b;`
Multiple assignments	Declare each variable separately	`a = b = c;` → `int a; int b; a = b = c;`
If condition	Extract the target expression into an assignment statement with a variable declaration	`if (a > b) {...}` → `boolean res = a > b;`

Algorithm 2 The steps in SMACK's Instrumentor component.

1: **procedure** INSTRUMENTOR($\mathsf{c^s}, \mathsf{s}$)
2: $\overline{\mathsf{c^s}} \leftarrow$ COPY($\mathsf{c^s}$)
3: s.varsBefore ← COLLECTVARS(GETRHS(s))
4: **if** HASCOMPOUNDASSIGNMENTS(s) **then** ▷ e.g., $x += 1;$
5: s.varsBefore ← s.varsBefore ∪ COLLECTVARS(GETLHS(s))
6: s.varsAfter ← COLLECTVARS(GETLHS(s))
7: **for each** v **in** s.varsBefore **do**
8: ADDLOGBEFORE($\overline{\mathsf{c^s}}, \mathsf{s}, v$)
9: **for each** v **in** s.varsAfter **do**
10: ADDLOGAFTER($\overline{\mathsf{c^s}}, \mathsf{s}, v$)
11: **return** $\overline{\mathsf{c^s}}$

② **Generator.** This component uses the given generators to obtain unit tests for each class $\mathsf{c^s}$ that Extractor produces. More precisely, the Generator takes E^1 and a unit-test generator, and produces set E^2, defined as follows:

$$\{(\mathsf{c^s}, \delta^{\mathsf{s}}, \mathsf{s}, T^{\mathsf{s}}) \mid (\mathsf{c^s}, \delta^{\mathsf{s}}, \mathsf{s}) \in E^1\} \tag{2}$$

where T^{s} (a set of unit tests) is the result of running a test generator G on a class produced by the Extractor, i.e., $G(\mathsf{c^s})$.

③ **Instrumentor.** To find values that should be used as inline-test inputs (via the `given` construct) and expected output (via the `checkEq` construct), this component instruments each extracted class $\mathsf{c^s}$ to collect runtime values of target-statement variables *before* and *after* as inputs and expected outputs, respectively, while running tests output by the Generator. The output of this component is a set, E^3:

$$\{(\overline{\mathsf{c^s}}, \delta^{\mathsf{s}}, \mathsf{s}, T^{\mathsf{s}}) \mid (\mathsf{c^s}, \delta^{\mathsf{s}}, \mathsf{s}, T^{\mathsf{s}}) \in E^2\} \tag{3}$$

```
1 public class MDFGenerateKey_7 {
2     public void targetStmtGenerated(char[] v, int i) {
3         int vTmp;
4         collectVars("v", v);
5         collectVars("i", i);
6         vTmp = Integer.parseInt(String.valueOf(v[i - 1]));
7         collectVars("vTmp", vTmp);
8     }
9 }
```

Figure 8 Instrumentor output for extracted class in Figure 4.

where $\overline{\mathsf{c}^{\mathsf{s}}}$ is an instrumented version of c^{s}. Figure 2 shows how instrumentation inserted by the Instrumentor records values that are later used to compose inline tests; it records values of all variables in the extracted method's argument list before executing the target statement, and values of variables being assigned in the target statement (left-hand side of assignments) after executing the target statement (lines 3-6 in Figure 2). Some special cases (not shown in Figure 2) include:

- The instrumentor does not log values of target-statement scoped variables, e.g., `x` in `l.stream().map(x -> x + 1).collect(Collectors.toList());` `x` is invisible outside the lambda expression so extracting it would cause compilation errors.
- The Instrumentor logs return values of methods that a target statement calls, e.g., for `m1(arg)`, the Instrumentor captures and associates the method call with a new variable `m1_arg` whose type is the return type of `m1()`. Eventually during inline-test execution, `m1_arg` replaces every instance of `m1(arg)` to avoid dependency on the original method call `m1()`, making the inline test self contained.

Figure 8 shows the instrumented version of the extracted code from Figure 4. Note: Instrumentor and Generator run in parallel in SMACK, but we describe them in sequence to ease presentation.

④ Executor. This component runs unit tests produced by the Generator and collects data from instrumented code points. More formally, it outputs the set E^4:

$$\{(\overline{\mathsf{c}^{\mathsf{s}}}, \delta^{\mathsf{s}}, \mathsf{s}, D^{\mathsf{s}}) \mid (\overline{\mathsf{c}^{\mathsf{s}}}, \delta^{\mathsf{s}}, \mathsf{s}, T^{\mathsf{s}}) \in E^3\} \tag{4}$$

where D^{s} is data collected by running test $\tau \in T^{\mathsf{s}}$ on instrumented class $\overline{\mathsf{c}^{\mathsf{s}}}$. D^{s} maps a unit test's name to data collected at each instrumentation point ($\tau \rightarrow data$), where *data* contains values collected for each variable v before ($D^{\mathsf{s}}[\tau][v]$`.before`) and after ($D^{\mathsf{s}}[\tau][v]$`.after`) target statement s.

⑤ Composer. This component composes inline tests from collected data D^{s} from the Executor and outputs E^5:

$$\{(\overline{\mathsf{c}^{\mathsf{s}}}, \delta^{\mathsf{s}}, \mathsf{s}, I^{\mathsf{s}}) | (\overline{\mathsf{c}^{\mathsf{s}}}, \delta^{\mathsf{s}}, \mathsf{s}, D^{\mathsf{s}}) \in E^4\} \tag{5}$$

where I^{s} is a set of inline tests, one set per $\tau \in D^{\mathsf{s}}$`.keys`.

⑥ Transplanter. This component inserts generated inline tests in E^5 to be right after the corresponding target statements in the original enclosing method of the target statement. The Transplanter also checks that inserted inline tests pass in the original CUT. An inline test that fails – e.g., due to non-determinism – is discarded. Note: the Transplanter uses δ^{s} and rw^{-1} (the inverse of the Extractor's rw procedure) to fit inline tests into the CUT. The Transplanter also uses s.file and s.loc to find information on where to modify the CUT.

⑦ Reducer. This final component produces a smaller set of generated inline tests by removing redundant ones. Inserting too many inline tests into the CUT reduces readability and can lead to compilation failure if doing so exceeds the maximum allowable method size [67]. The Reducer uses a coverage-then-mutants approach [54]. The idea is to first reduce the number of inline tests by eliminating those that are redundant based on instruction coverage (of the target statement), and then adding back inline test that kills mutants that survive the initially reduced set. Doing so ideally ensures that the final reduced set has the same bug-finding ability as the original set.

3.2 Implementation

We briefly describe our implementation of each SMACK component.

① Extractor. SMACK uses JavaParser [41] to extract each target statement into a new method and, subsequently, a class that imports all needed packages. A mapping from target statement to its line number in the CUT, and the Extractor's modifications are also stored for later use in transplanting generated inline tests to the CUT.

② Generator. SMACK applies two unit-test generators – Randoop [70] and EvoSuite [21] – to the extracted methods. Other test generators can be added in the future.

③ Instrumentor. SMACK uses JavaParser for source-level instrumentation of extracted methods. That instrumentation logs input and output values at each target statement. Those values are subsequently used to compose inline tests. SMACK logs primitive or String-typed target statement variables as is. For other non-primitive types, SMACK uses XStream [98] to serialize objects into XML, persists the serialized objects to disk, and logs the file path so that inline tests can deserialize those objects at runtime.

④ Executor. SMACK uses JUnit to run generated unit tests; other frameworks like TestNG can be supported in the future.

⑤ Composer. SMACK uses JavaParser to compose inline tests, using the input and output values recorded at a target statement while executing the instrumented extracted method.

⑥ Transplanter. SMACK again uses JavaParser to insert generated inline tests to be after the target statement in the original enclosing method. SMACK first uses the mapping stored by the Extractor to locate the target statement in the CUT. Then, SMACK inserts nodes for the inline tests to be right after nodes that represent the target statement in the abstract syntax tree, which is then converted to a Java file.

⑦ Reducer. First, SMACK uses JaCoCo [62] to collect the instruction coverage of all generated inline tests I^{s} for each target statement. Then, an inline test $\tau \in I^{\mathsf{s}}$ is added to an initially empty reduced set T' if τ covers instructions that those currently in T' do not cover. Then, SMACK uses universalmutator [29] to generate mutants on the target statement. If an inline test $\tau' \in I^{\mathsf{s}} \wedge \tau' \notin T'$ kills a mutant that survives T', then τ' is added to the final reduced set T. But, as we discuss in § 6, the order in which inline tests are considered by the Reducer affects the minimality of T.

4 Evaluation

We answer the following research questions:

RQ1: How well does Smack work on target statements that ExLi *cannot* handle?

RQ2: How well does Smack work on target statements that ExLi *can* handle?

RQ3: What is the fault-detection capability of Smack-generated inline tests?

RQ4: How effective is Smack's reduction strategy and what are its runtime costs?

RQ1 investigates how well Smack extends inline-test generation to target statements that are not covered by method-level unit tests (so ExLi cannot work for them). RQ2 compares Smack's inline-test generation ability with ExLi's on target statements that method-level unit tests cover (so ExLi can work for them). RQ3 uses mutation testing to measure the fault-detection ability of Smack-generated inline tests on target statements that ExLi *cannot* handle, and to compare the fault-detection capability of inline tests generated by Smack and ExLi on target statements that ExLi *can* handle. Finally, RQ4 provides metrics about Smack's approach for reducing the number of inline tests that are inserted after a target statement in the code under test (CUT), and metrics about Smack's runtime costs.

4.1 Experiment Setup

Projects used in evaluation. Table 2 shows the projects that we evaluate; they are all those that Liu et al. evaluated ExLi on [54, 56]. These projects (1) are single-module Maven projects; (2) are actively maintained and changed since 2022; (3) do not have failing unit tests when run alone, with JaCoCo, Randoop, or EvoSuite; (4) contain a statement in one of the four statement categories – regular expressions, string manipulation, bit manipulation, and stream operation – that Liu et al. focused on; and (5) have at least one inline test generated by ExLi. Table 2 shows an identifier (PID) for subsequent reference, project URL, the commit SHA used, number of lines of code (computed using `cloc` [1]), total number of target statements evaluated (All), the number of those target statements that are not covered (UC) and covered (C) by developer written or automatically generated method-level unit tests. The last row shows sums across all projects.

Breakdown of evaluated target statements. We evaluate 957 target statements from these 31 projects ("Σ" row of the "All" column in Table 2). The 312 in the "Σ" row of the "UC" column are those that developer written or automatically generated method-level unit tests *do not* cover (§ 2.3). We subsequently call these 312 "uncovered statements" or "Uncovered312". Among the 312, 3 use regular expressions, 284 do string manipulation, 9 do bit manipulation, and 16 call stream APIs. These are the four kinds of statements that all prior work on inline tests target for Java. Dually, the 645 target statements in the "Σ" row of the "C" column are those covered by method-level unit tests. We subsequently refer to these 645 as "covered statements" or "Covered645". Of these, 361 are covered by developer written method-level unit tests, 424 by Randoop-generated method-level unit tests, and 483 by EvoSuite-generated method-level unit tests. The kinds of these 645 statements are: 81 regular expressions, 458 string manipulation, 88 bit manipulation, and 18 stream operations.

Randoop and EvoSuite configurations. During inline-test generation with Smack and ExLi, we run Randoop for 100 seconds for every non-test class in the project and every extracted class to generate unit tests for each project (per the Randoop user manual [96]);

Table 2 Projects that we use in our evaluation. **UC**: not covered by developer written or automatically generated method-level unit tests; **C**: covered by method-level unit tests. Project names are clickable and linked to their GitHub repositories.

PID	Project	SHA	LOC	# Target stmts		
				All	UC	C
P1	AquaticInformatics/aquarius-sdk-java	8f4edb9	21,634	3	0	3
P2	Asana/java-asana	52fef9b	5,572	247	1	246
P3	awslabs/amazon-sqs-java-extended-client-lib	58fed25	1,288	4	0	4
P4	Bernardo-MG/maven-site-fixer	60244c0	1,689	2	0	2
P5	Bernardo-MG/velocity-config-tool	26226f5	358	3	0	3
P6	craftercms/core	4d394a9	10,233	24	2	22
P7	CycloneDX/cyclonedx-core-java	d933705	6,011	4	0	4
P8	finos/messageml-utils	b4c75c6	21,765	40	14	26
P9	fleipold/jproc	b872abf	1,189	2	0	2
P10	hyperledger/fabric-sdk-java	da35400	33,677	25	5	20
P11	jenkinsci/email-ext-plugin	699277c	13,190	14	4	10
P12	jkuhnert/ognl	5c30e1e	18,190	42	10	32
P13	jscep/jscep	b20e944	6,310	4	1	3
P14	lamarios/sherdog-parser	aa6806a	1,546	9	4	5
P15	liquibase/liquibase-oracle	6ab7dea	7,170	3	0	3
P16	maxmind/geoip-api-java	1030316	11,526	21	4	17
P17	medcl/elasticsearch-analysis-pinyin	01dda56	2,169	9	2	7
P18	mojohaus/build-helper-maven-plugin	f1fac8c	2,424	27	14	13
P19	mojohaus/properties-maven-plugin	6cf7c2b	891	11	0	11
P20	mp911de/logstash-gelf	66debd8	13,130	71	18	53
P21	mpatric/mp3agic	407f7a9	9,907	38	0	38
P22	netceteragroup/trema-core	fa9f76d	3,285	10	1	9
P23	phax/ph-pdf-layout	f2d7b98	14,408	6	0	6
P24	ralscha/extclassgenerator	40ad147	6,271	8	6	2
P25	red6/pdfcompare	1259ef2	4,213	9	4	5
P26	restfb/restfb	35a34dd	42,022	28	7	21
P27	steveash/jopenfst	14c4a1d	5,180	7	0	7
P28	TNG/property-loader	928f414	1,860	9	2	7
P29	uwolfer/gerrit-rest-java-client	a0bf7cc	14,594	18	8	10
P30	visenze/visearch-sdk-java	0efcda3	7,643	4	0	4
P31	wmixvideo/nfe	1ccdba7	133,698	255	205	50
$\sum$	-	-	423,043	957	312	645

we set `seed` to 42, `usethreads` to true, and all other options as default. We run EvoSuite for 120 seconds (per a recent SBST competition [83]) per extracted class. We also set the seed to 42. We subsequently denote the Randoop-based approach as Smack-R and the EvoSuite-based approach as Smack-E.

Mutation testing tools. We use universalmutator [14, 29] because it generates mutants at the source code level, allowing us to filter mutants by target statement. Also universalmutator was used in all prior work on inline-test generation [54, 56].

Experimental environment. We run all experiments on an Intel Xeon w9-3475X @ 2.20GHz (36 cores, 72 threads) CPU, 125 GB RAM, Ubuntu 20.04, Java 8, and Maven 3.8.6.

Table 3 SMACK's inline-test generation results on UNCOVERED312 *before* reduction. "Contributions of unit-test generators" shows the number of target statements for which SMACK generates inline tests using Randoop alone, EvoSuite alone, or Randoop *and* EvoSuite ("Union"). "None" shows the number of target statements for which SMACK generates no inline test.

PID	Contributions of unit-test generators				# Inline tests	
	Randoop	EvoSuite	Union	None	Randoop	EvoSuite
P2	0	0	0	1	0	0
P6	1	2	2	0	1	3
P8	8	8	8	6	141	34
P10	3	3	3	2	25	9
P11	1	1	2	2	1	1
P12	4	8	8	2	10	19
P13	0	0	0	1	0	0
P14	1	1	1	3	4	5
P16	4	4	4	0	404	41
P17	0	2	2	0	0	2
P18	14	14	14	0	252	68
P20	3	17	17	1	260	36
P22	1	1	1	0	101	10
P24	4	4	4	2	354	18
P25	2	1	2	2	4	8
P26	0	0	0	7	0	0
P28	1	1	1	1	81	9
P29	2	3	3	5	47	13
P31	195	205	205	0	1,676	318
$\sum$	244	275	277	35	3,361	594

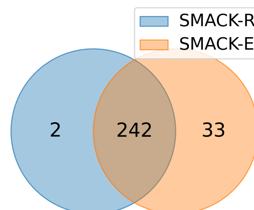

Figure 9 Contributions of inline tests for UNCOVERED312 from SMACK-R and SMACK-E.

4.2 RQ1: Smack on target statements that ExLi *cannot* handle

Table 3 shows SMACK's inline-test generation results on UNCOVERED312, the set of 312 evaluated target statements (from 19 projects) that EXLI cannot generate inline tests for because they are not covered by method-level unit tests. For each project, "Contributions of unit-test generators" shows the number of target statements for which SMACK generates inline tests using Randoop alone, EvoSuite alone, or Randoop *and* EvoSuite ("Union"). We also show the number of target statements for which SMACK generates no inline test ("None"). Figure 9 shows the contributions of unit-test generators in generating inline tests for UNCOVERED312 from SMACK-R and SMACK-E. Randoop helps SMACK generate inline tests for 2 uncovered statements that EvoSuite does not help with. Also, EvoSuite helps generate inline tests for 33 uncovered statements that Randoop does not help with. So, both generators help. We manually find two reasons why SMACK does not generate inline tests for 35 statements (the "Σ" row of the "None" column in Table 3):

1. Randoop and EvoSuite generate no unit test for extracted methods in 13 cases. Of these, method extraction failed for five because JavaParser encounters type resolution errors. SMACK extracts methods for the other eight but no unit test was generated for them because, although SMACK supports non-primitive types (§ 3.2), the objects in these cases are complex. For example, consider this target statement in `ralscha_extclassgenerator`:

```
String n = StringUtils.uncapitalize(method.getName().substring(3));
```

The `method` variable is a `java.lang.reflect.Method`, a class in Java's reflection API, so it is hard for Randoop and EvoSuite to generate objects of this class from scratch. Randoop and EvoSuite utilize reflection to generate other objects, but objects in reflection API (e.g., of type `Method`, `Field`, etc.) are not used as target objects to construct test oracles.

2. Inline tests fail in 23 cases due to non-deterministic (flaky) assertions. For ten of these, the inline tests' assertions fail because serialized objects' classes use default hash codes that depend on specific memory addresses. For instance, consider this target statement from `velocity-config-tool`:

```
cFile = String.valueOf(currentFileObj);
```

The `cFile` object is assigned the String value of `currentFileObj`. But, since `currentFileObj` is of type `Object`, `String.valueOf` returns the hash code of `currentFileObj`, which is obtained from execution-specific memory addresses. Internally, non-JDK objects (such as `currentFileObj`) are serialized into XML files (such as `0.xml`), and are deserialized at runtime. One generated failing inline test is:

```
itest().given(currentFileObj, "0.xml").checkEq(cFile, "java.lang.Object@7eaebfdb");
```

This inline test fails because the memory address of `cFile` used in the `checkEq` expression changed in the next execution, causing the assertion to fail. For two other statements, the generated inline tests are flaky because they extract dynamic information from online webpages. So, they non-deterministically pass or fail as the webpages' contents change. Consider this target statement from `lamarios/sherdogparser`:

```
doc = ParserUtils.parseDocument(String.format(url, page));
```

An inline test generated for this statement uses a Google search page as `url`. Generated inline tests for this target statement check if `doc` objects in two accesses are identical, so they fail. Future work can reduce flakiness by using mocks. For other statements in this category, generated inline tests fail because XStream cannot deserialize objects due to type resolution errors.

The "# Inline tests" column in Table 3 shows passing inline tests that SMACK generates *before reduction*. RQ4 (Section 4.5) shows that SMACK's reduction step reduces the average number of generated inline tests per target statement from 16 to 2.2.

> RQ1 shows that SMACK effectively generates inline tests for uncovered statements. SMACK generates passing inline tests for 277, or 88.8% of statements in UNCOVERED312.

Table 4 SMACK's inline-test generation results on COVERED645 *before* reduction. "Contributions of unit-test generators" shows the number of target statements for which SMACK generates inline tests using Randoop alone, EvoSuite alone, or Randoop *and* EvoSuite ("Union"). "None" shows the number of target statements for which SMACK generates no inline test.

PID	Contributions of unit-test generators				# Inline tests	
	Randoop	EvoSuite	Union	None	Randoop	EvoSuite
P1	1	2	2	1	1	2
P2	246	246	246	0	6,133	1,445
P3	4	4	4	0	149	11
P4	2	2	2	0	4	9
P5	3	2	3	0	37	5
P6	16	22	22	0	1,020	81
P7	2	2	2	2	14	6
P8	17	21	21	5	610	115
P9	1	2	2	0	101	9
P10	12	15	16	4	100	77
P11	8	0	8	2	722	0
P12	14	31	31	1	22	228
P13	2	2	2	1	17	2
P14	4	5	5	0	167	13
P15	3	3	3	0	77	20
P16	17	0	17	0	1,448	0
P17	6	7	7	0	256	15
P18	0	4	4	9	0	11
P19	10	11	11	0	65	25
P20	42	53	53	0	3,338	252
P21	38	38	38	0	2,483	252
P22	8	9	9	0	606	45
P23	6	6	6	0	263	19
P24	2	2	2	0	148	6
P25	1	2	2	3	9	12
P26	14	16	16	5	119	56
P27	7	7	7	0	534	51
P28	5	7	7	0	179	16
P29	5	9	9	1	112	46
P30	4	4	4	0	104	34
P31	19	48	48	2	237	256
$\sum$	519	582	609	36	19,075	3,119

4.3 RQ2: Smack on target statements that ExLi *can* handle

We compare EXLI and SMACK on target statements that both techniques can handle (these statements are covered by method-level unit tests). Table 4 shows SMACK's results on COVERED645 (645 target statements that are covered by method-level unit tests; they are from all 31 projects). The sum ("Σ") of the "Union" column under "Contributions of unit-test generators" shows that SMACK generates inline tests for 609 of these 645 statements. Also, Σ in the "None" column shows that SMACK does not generate inline tests for 36 (or 5.6%) of these 645 statements. Our manual analysis shows two reasons why SMACK did not generate any inline tests for these 36 statements:

```
1  // Source code
2  class ServiceDiscovery {
3   private String parseEndpoint(Protocol.Peer peerRet) throws
        InvalidProtocolBufferRuntimeException {
4    ...
5    if (name != null) {
6     if (asLocalhost) {
7      /*target statement; line 1267 in original code*/
8      point = "localhost" + name.substring(name.lastIndexOf(':'));
9      } else { point = name.toLowerCase().trim(); }// to ease comparison
10   } ... return point;
11  }
12 }
13
14 // Extracted class for the target statement on line 8
15 class ServiceDiscovery_1267 {
16  public void targetStmtGen(String name) {
17   String point = "localhost" + name.substring(name.lastIndexOf(':'));
18  }
19 }
20
21 // Randoop-generated test; we shorten fully-qualified names
22 @Test public void test02968() throws Throwable {
23  ...
24  ServiceDiscovery_1267 serviceDiscovery_1267_0 = new ServiceDiscovery_1267();
25  // This exception was thrown during ... test generation
26  try {
27   serviceDiscovery_1267_0.targetStmtGen("protos.Endorser");
28   fail("Expected ... StringIndexOutOfBoundsException; ...");
29  } catch (java.lang.StringIndexOutOfBoundsException e) {
30   // Expected exception.
31  }
32 }
```

Figure 10 Illustrating why Smack fails when generated unit tests only check exceptional behavior.

1. Neither Randoop nor EvoSuite generates unit tests for corresponding extracted methods for 31 target statements because of exceptional behavior. The target statement, extracted class, and generated unit test in Figure 10 illustrates the way exceptional behavior prevents inline-test generation. There, Smack extracts the target statement on line 8 into the class on lines 15-19 and Randoop generates the test on lines 22-32 for that class. That target statement computes the substring of `name` starting from the last index of colon in `name`. However, all Randoop-generated tests use values of `name` with no colon. So, they all throw exceptions and Smack cannot collect all values from which to compose inline tests.

2. We filter out statements where Smack only generates failing inline tests: 1 for Randoop and 4 for EvoSuite have the same problem of non-determinism as those described in § 4.2.

> We conclude from RQ2 that Smack's inline-test generation is not limited to target statements that are not covered by method-level unit tests. Specifically, Smack generates inline tests for 94.4% of statements in Covered645 that method-level unit tests cover.

4.4 RQ3: Fault-detection ability of Smack-generated inline tests

We use mutation testing [33, 58, 72–74] to evaluate the fault-detection ability of Smack in Uncovered312 and to compare the fault-detection ability of Smack-generated against unit tests and ExLi-generated inline tests in Covered645.

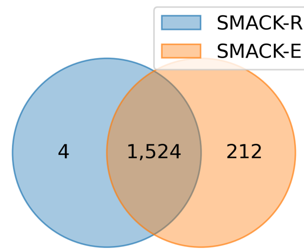

Figure 11 Mutants killed by SMACK-generated inline tests using Randoop (SMACK-R) or EvoSuite (SMACK-E).

Mutants. Among UNCOVERED312, universalmutator generates 1,815 mutants for 258 statements. Among COVERED645, universalmutator generates 2,844 mutants for 534 statements. For the other 54 statements in UNCOVERED312 and 111 statements in COVERED645, universalmutator only generates invalid mutants or has no applicable mutator.

Smack on statements that are not covered by method-level unit tests. Table 5 shows the mutation analysis results for UNCOVERED312. There, "Contributions of unit-test generators" shows if Randoop (SMACK-R), EvoSuite (SMACK-E) or both (Union) are used. "# Stmts Run Pass" are statements for which SMACK generates passing inline tests (last row of Table 3); "# Stmts w/ mutants" are statements with mutants; "# Killed" are killed mutants; "# Surviving" are surviving mutants; and "% Mutation score" is computed as (# Killed × 100) ÷ (# Killed + # Surviving).

Mutation scores on UNCOVERED312 are high, suggesting that SMACK-generated inline tests are effective for detecting faults on statements that are not covered by method-level unit tests. Overall, we find that SMACK-E is generally more effective at killing mutants than SMACK-R, but both kill mutants that the other does not. Figure 11 shows the sets of mutants killed in statements where Randoop and EvoSuite generate inline tests: 4 mutants are killed when using only Randoop (SMACK-R) compared with 212 mutants when using only EvoSuite (SMACK-E). Of the 212 mutants killed by only SMACK-E, SMACK-R has no unit tests that can help generate inline tests that kill 97. The remaining 115 mutants are killed by SMACK-E when using specific characters or strings that SMACK-R does not use. For example, a mutant that is only killed by SMACK-E modifies the original string manipulation statement from `corrected = text.replace('/', '-')...` to `corrected = text.replace('*', '-')...`. SMACK-R inline tests only have `text` with simple values such as only alphabetic characters, but SMACK-E inline tests use a broader range of values for `text`, and one of them contains $*$, which helps kill this mutant.

SMACK-R kills 4 mutants that are not killed by SMACK-E because SMACK-R uses a broader range of integers as start and stop indices in the `substring(start, end)` method. These mutants simply switch the arguments of `substring`, and can only be killed when the `start` and `stop` indices are not equal. In more detail, SMACK-E outperforms SMACK-R in killing mutants involving `indexOf` and `startsWith` – it uses the caller as an input. Conversely, SMACK-R is more effective in detecting mutations involving `substring` because it uses diverse integers for the start and stop indices. These mutants often survive SMACK-E, which frequently uses identical start and stop indices. This analysis highlights the complementary strengths of using Randoop and EvoSuite in SMACK. Also, the mutation scores after the first round of reduction (not shown in our tables, 91.1%) is 5.6 percentage points lower than that of "Union" in Table 5, so SMACK's two-round reduction is beneficial.

Table 5 "# Stmts Run Pass"are statements for which SMACK generates passing inline tests; "# Stmts w/ mutants" are statements with mutants; "# Killed" are killed mutants; "# Surviving" are surviving mutants; and "% Mutation score" is (# Killed × 100) ÷ (# Killed + # Surviving).

Contributions of unit-test generators	# Stmts Run Pass	# Stmts w/ mutants	# Killed	# Surviving	% Mutation score
SMACK-R	244	228	1,530	285	84.3
SMACK-E	275	256	1,749	66	96.4
Union	277	258	1,755	60	96.7

Table 6 "# Stmts Run Pass"are statements for which SMACK generates passing inline tests; "# Stmts w/ mutants" are statements with mutants; "# Killed" are killed mutants; "# Surviving" are surviving mutants; and "% Mutation score" is (# Killed × 100) ÷ (# Killed + # Surviving).

Contributions of unit-test generators	# Stmts Run Pass	# Stmts w/ mutants	# Killed	# Surviving	% Mutation score
SMACK-R	519	452	1,731	1,113	60.9
SMACK-E	582	534	2,287	557	80.4
Union	609	534	2,362	482	83.1
ExLi	645	562	2,495	349	87.7
UnitTests	-	562	2,149	695	75.6

Comparing Smack and ExLi on target statements that method-level unit tests cover. Table 6 shows mutation testing results on COVERED645. There, mutation scores are not as high as those for UNCOVERED312 (Table 5). Comparing the "Union" and "EXLI" rows in Table 6 shows that SMACK's mutation score (83.1%) is lower than EXLI's (87.7%) for all 645 statements. Also, comparing the "Union" and "UnitTests" rows shows that SMACK's mutation score (83.1%) is better than that of using only method-level unit tests (75.6%). However, on all 957 evaluated target statements in this paper, SMACK-generated inline tests achieve a mutation score of 88.4%, compared to 53.6% for EXLI-generated ones, and 46.1% when using only method-level unit tests.

Figure 12 compares the sets of mutants killed by SMACK with Randoop only (SMACK-R), SMACK with EvoSuite only (SMACK-E), and EXLI (which uses developer written and automatically generated method-level unit tests), on 2,697 mutants in target statements that are covered by manually-written or automatically-generated tests. We find again that (1) using SMACK with Randoop and EvoSuite is better than using either alone; and (2) SMACK and EXLI are complementary for mutant killing, so future work should investigate using them together more efficiently.

Analysis of killed mutants in Covered645: Smack-R vs. Smack-E. Figure 12 shows that SMACK-R and SMACK-E kill mutants that the other cannot. SMACK-R kills 75 mutants that SMACK-E does not (regions H, J, K, I), and SMACK-E kills 622 mutants that SMACK-R does not (regions D, F, G, E). We next discuss examples of mutants killed only by SMACK-R or SMACK-E. Figure 13 shows a mutant [19] killed only by SMACK-R. There, line 3 is the target statement and line 4 is the inline test. The mutant changes the second `!=` (line 1) to `<` (line 2), so its condition, `minValue != null && minValue.indexOf(".") < -1`, is false. Killing the mutant requires `minValue` to contain `"."` and `maxValue` to be null or not contain `"."`. But, the inline tests generated by SMACK-E use no such values.

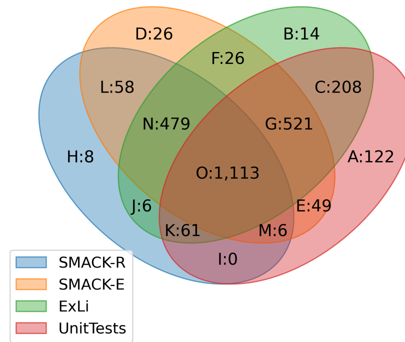

Figure 12 Mutants killed by SMACK (using Randoop and EvoSuite) and EXLI on COVERED645.

```
1 if (minValue != null && minValue.indexOf(".") != -1 // original code
2 if (minValue != null && minValue.indexOf(".") < -1 // mutated code
3     || maxValue != null && maxValue.indexOf(".") != -1) {
4     itest().given(minValue, "Ext.define").given(maxValue, "//").checkTrue(group());
5     ...
6 }
```

Figure 13 Mutant killed only by SMACK-R-generated inline tests.

```
1 if ( !line.isEmpty() &&
2     !line.replace( " ", "" ).startsWith( "#" ) ) // original code
3     !line.replace( "", " " ).startsWith( "#" ) ) // mutated code
4 {
5     itest().given(line, "#1'2Bs^$CTiD^").checkFalse(group());
6 }
```

Figure 14 Mutant killed only by SMACK-E-generated inline tests.

```
1 body = "{" + (castExpression != null ? castExpression : "") + pre + setterCode + ";}";
2
3 if (body.indexOf("..") >= 0) { // original code
4 if (body.indexOf("..") > 0) { // mutated code
5     itest().given(body, "..").checkTrue(group());
6     body = body.replaceAll("\\.\\.", ".");
```

Figure 15 Mutant killed only by SMACK-E-generated inline tests.

Conversely, Figure 14 shows a mutant [75] that is killed only by SMACK-E. There, line 2 is the target statement and line 5 is the inline test. The mutant swaps the arguments of `replace(" ", "")`. To kill this mutant, `line` must start with zero or more leading whitespaces followed by `#`. But, SMACK-R-generated inline tests do not use such values. These mutation analyses show that SMACK-R and SMACK-E contribute complementary inline tests with respect to fault-detection ability on target statements.

Analysis of killed mutants in Covered645: Smack vs. ExLi and method-level unit tests. SMACK and EXLI are complementary as well. Figure 12 shows that SMACK kills 147 mutants that EXLI does not (regions D, L, H, E, M, I), and EXLI kills 222 mutants that SMACK cannot kill (regions B, C). We next discuss examples of mutants that are only killed by SMACK or EXLI. Figure 15 shows a mutant [66] killed only by SMACK. There, line 3 is the target statement and line 5 is the inline test. EXLI-generated inline tests for this target statement do not use a string starting with `".."`. Conversely, Figure 16 shows a mutant [5] that is killed only by EXLI. There, line 4 is the target statement and line 6 is the inline test that kills

```
1 int result;
2 long temp;
3 temp = Double.doubleToLongBits(weight);
4 result = (int) (temp ^ (temp >>> 32)); // original code
5 result = (int) (temp ^ (temp >>> (32 - 1))); // mutated code
6 itest().given(temp,4609434218613702656L).checkEq(result,1073217536);
```

Figure 16 Mutant killed only by ExLi-generated inline tests.

```
1 if ((dboptions & GEOIP_MEMORY_CACHE) == 1) {
2     ...
3 } else if ((dboptions & GEOIP_INDEX_CACHE) != 0) { // original code
4   if ((dboptions & GEOIP_INDEX_CACHE) != 0) { // mutated code
5     ...
6 }
```

Figure 17 Mutant killed only by method-level unit tests.

the mutant because the `temp` value has 19 digits. But, `temp` values used by Smack contain at most 4 digits, making `temp ^ (temp >>> 32)` and `temp ^ (temp >>> (32 - 1))` to always be equal to `temp`. These two examples demonstrate that Smack and ExLi generate inline tests that are complementary with respect to fault-detection ability. That is, the union of inline tests that they generate kill more mutants than when they are used separately.

Figure 17 shows a mutant [39] that is killed by method-level unit tests but not Smack or ExLi. There, line 3 is the target statement, and we elide the unit test. The mutant changes `else if` to `if`; it is not killed by inline tests because validating logic in multi-statement `if` blocks is out of scope for inline tests. The recently proposed block tests [30] would be more applicable in this case.

> We conclude from RQ3 that (i) Smack's default generators yield complementary fault-detection ability: using Randoop and EvoSuite together helps kill more mutants than each one alone; and (ii) Smack and ExLi yield complementary fault-detection capability: using Smack and ExLi together helps kill more mutants than using them separately.

4.5 RQ4: Reduction Rates and Runtime Cost

In this section, we evaluate Smack's reduction rates and runtime costs on Uncovered312 and Covered645. Too many inline tests affect readability. So, Smack's inline test reduction rate has practical importance. Similarly, low runtime costs will help wider adoption.

Reduction rates. Smack's two-round reduction aims to reduce the set of inline tests that are inserted after the target statement in the CUT, without losing the fault-detection ability of the whole set of generated inline tests. The first round starts from R0, the initial set of all generated inline tests, removes those that are redundant based on coverage, and produces the initially reduced set, R1. Then, the second round further reduces R1 by removing those that are redundant based on mutants killed, producing an intermediate set. To preserve the fault-detection ability of R0, Smack produces the final reduced set, i.e., R2, by adding to the intermediate set any inline test from R0 that kills mutants that survive the intermediate set.

The first two rows in Table 7 show reduction rates on the Uncovered312 dataset. There, we see that Smack achieves high reduction rates on Uncovered312. With Randoop (Smack-R row), coverage-based reduction in R1 yields 250 of 3,361 R0 inline tests. After mutation-based reduction and adding back some R0 inline tests to preserve the mutation

Table 7 SMACK's reduction rates, with SMACK-R and SMACK-E, on UNCOVERED312 and COVERED645. **RR(%)** is R2 reduction rate – the percentage of inline tests that are removed. **UR2**, Unified R2, is the number of R2 inline tests after reduction with both generators. **URR(%)** is the reduction rate of Unified R2.

Dataset	generator	# Stmts	R0	R1	R2	RR(%)	UR2	URR(%)
UNCOVERED312	SMACK-R	244	3,361	250	337	90.0	634	84.0
	SMACK-E	275	594	274	348	41.4		
COVERED645	SMACK-R	519	19,075	518	625	96.7	1,054	95.3
	SMACK-E	582	3,119	580	798	74.4		

score of R0, the number of inline tests in R2 becomes 337. With EvoSuite (SMACK-E row), the numbers of inline tests in R0, R1, and R2 are 594, 274, and 348, respectively. Computing reduction rates as $((R0 - R2)/R0)$, the reduction rates on UNCOVERED312 are 90.0% with Randoop and 41.4% with EvoSuite.

The third and fourth rows in Table 7 show the reduction rates for the COVERED645. With Randoop ("COVERED645" , "SMACK-R" row), coverage-based reduction in R1 leaves 518 of 19,075 R0 inline tests. After mutation-based reduction and adding back some R0 inline tests to preserve the fault-detection ability of R0, the number of R2 inline tests becomes 625. With EvoSuite ("COVERED645", "SMACK-E" row), the numbers of inline tests in R0, R1, and R2 are 3,119, 580, and 798, respectively. These reduction rates are 96.7% with Randoop and 74.4% with EvoSuite.

In addition, to reduce the set of inline tests per unit-test generator, SMACK also supports unified reduction, which further reduces the set of R2 inline tests from all generators based on mutation testing. The rightmost two columns in Table 7 show the results of unified reduction. The "UR2" column shows the number of R2 inline tests after unified reduction, with 634 and 1,054 inline tests for UNCOVERED312 and COVERED645, respectively. The "URR(%)" column shows the reduction rate of Unified R2, with 84.0% and 95.3% for UNCOVERED312 and COVERED645, respectively. This translates to an average of 2 inline tests per target statement on UNCOVERED312 and 1.6 inline tests per target statement on COVERED645.

Are Smack's times feasible in practice? We compare SMACK times with the only known manual inline-test writing times from Liu et al.'s user study [55]. In that study, participants on average take 150 seconds to understand a target statement, write 1.7 inline tests per target statement, and spend 210 seconds to write an inline test. Summing these numbers, it took study participants about 507 seconds to understand and write inline tests per target statement. Table 8 shows different views of SMACK's runtime costs. The top part shows the runtime costs per inline test, per target statement, and per project. Comparing manual writing times with SMACK's average time per target statement (Table 8 row $\boldsymbol{\mu_s}$**(s)**), SMACK-R has similar costs as manually writing per inline tests, while SMACK-E is much faster. Furthermore, the user-study times that we compare against do not include the time for evaluating mutation-killing effectiveness of manually written inline tests. The slower SMACK-R takes, on average, 661 and 464.3 seconds to generate inline tests for each target statement in UNCOVERED312 and COVERED645, respectively. The faster SMACK-E takes 48.8 and 37 seconds to generate inline tests for each uncovered statement and covered statement, respectively. On a per inline test basis, this cost is even lower. The slower SMACK-R takes 478.6 and 385.5 seconds to generate inline tests for each target statement in UNCOVERED312 and COVERED645, respectively. The faster SMACK-E takes 38.6 and 27 seconds to generate inline tests for each uncovered statement and covered statement, respectively.

Table 8 Breakdown of SMACK's end-to-end times by granularity and stage. The top part shows times per inline test ($\boldsymbol{\mu_{it}}$**(s)**), per target statement ($\boldsymbol{\mu_{s}}$**(s)**), and per project ($\boldsymbol{\mu_{p}}$**(s)**). The bottom part shows the breakdown of total runtime (Σ(s) in seconds, or Σ(h) in hours): extraction (Extract(s)), generation (Gen(s)), executing unit tests (Exec UT(s)), executing R0 inline tests (Exec IT(s)), and mutation testing (Mutate(s)).

	Uncovered312		**Covered645**	
	Smack-R	**Smack-E**	**Smack-R**	**Smack-E**
$\boldsymbol{\mu_{it}}$**(s)**	478.6	38.6	385.5	27
$\boldsymbol{\mu_{s}}$**(s)**	661	48.8	464.3	37
$\boldsymbol{\mu_{p}}$**(s)**	8488.8	706.6	7772.5	694.7
Σ(s)	161,287.5	13,424.7	240,946.7	21,536
Σ(h)	44.8	3.7	66.9	6
Extract(s)	161.1	161.7	335.8	343.4
Gen(s)	149,772.2	3,402.7	224,823.5	8,980.5
Exec UT(s)	2,934.5	858.6	5,749.6	2,162.2
Exec IT(s)	144.8	137.7	374.8	261.9
Mutate(s)	8,274.9	8,864.0	9,663.0	9,788.0

Considering that inline tests are relatively new, SMACK is valuable for reducing manual effort and opens new opportunities for larger-scale research on inline tests. We measure the times while running all the steps in sequence, but the tasks in each step (e.g., mutation testing) can be parallelized. We leave such optimizations as future work.

End-to-end runtime costs. The bottom part of Table 8 shows the breakdown of end-to-end time of running SMACK on UNCOVERED312 and COVERED645 (including time to run R0 inline tests and perform mutation analysis). All times except the inline tests execution time are one-time costs. Overall, the results show that the time to run SMACK on UNCOVERED312 is manageable, but future research should look into speeding it up some more. Specifically, it takes 44.8 hours for SMACK-R and 3.7 hours for SMACK-E to generate unit tests for UNCOVERED312 (Table 8, column UNCOVERED312). The breakdown of these times are as follows. SMACK-R takes 161.1s to extract target statements, 41.6 hours to generate unit tests (an inherent cost of Randoop, not SMACK), 2,934.5s to execute the generated unit tests, 144.8s to execute R0 inline tests, and 2.3 hours to perform mutation testing. For SMACK-E, the extraction, generation, execution, R0 inline-test execution, and mutation testing times are 161.7s, 0.9 hours, 858.6s, 137.7s, and 2.5 hours, respectively. Figure 18 displays the runtime breakdown of SMACK on UNCOVERED312 and COVERED645 visually. There, it can be seen that the end-to-end times are dominated by unit-test generation and mutation testing time. Table 9 shows the runtime statistics across all evaluated projects for different sets of target statements (in seconds), along with number of target statements.

Looking at row Gen(s), we observe a huge difference in the generation times of SMACK-R and SMACK-E. This is due to how these unit-test generators work – Randoop generates unit tests for the entire project [96], while EvoSuite only generates unit tests for the class enclosing the target statement. Randoop takes time that is proportional to the number of classes in a project, while EvoSuite takes time that is proportional to the number of target statements. So, Randoop spends more time than EvoSuite to generate unit tests. Yet, SMACK-R and SMACK-E spend similar time on execute generated inline tests and mutation testing. Prior work [55] shows that one needs to inject three orders of magnitude more inline tests than what is injected into the CUT to trigger pronounced differences in inline-test execution time.

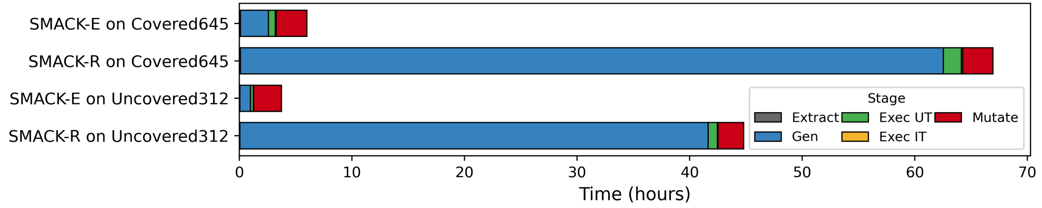

Figure 18 Time breakdown of Smack on Uncovered312 and Covered645.

Table 9 Summary runtime statistics across all evaluated projects for different sets of target statements (in seconds).

Dataset	Generator	Sum	Mean	Median	Max	Min
Uncovered312	#Stmts	312	16	4	205	1
	Smack-R	161,287.3	8,488.8	10,940.9	18,686.7	1,526.2
	Smack-E	13,424.6	706.6	150.1	8,860.1	16.6
Covered645	#Stmts	645	21	7	246	2
	Smack-R	240,946.6	7,772.5	8,303.6	13,293.9	595.6
	Smack-E	21,536.1	694.7	225.4	2,897.6	27.4
All	#Stmts	957	31	10	255	2
	Smack-R	402,234.0	8,044.7	9,236.7	18,686.7	595.6
	Smack-E	34,960.7	699.2	199.0	8,860.1	16.6

But, Smack-R generates only about 6 times more inline tests than Smack-E.

In Covered645, the distribution of runtime costs for different stages are similar to those in Uncovered312 (Table 8, column Covered645). It takes 66.9 hours for Randoop and 6 hours for EvoSuite to generate unit tests for this dataset. The breakdown of these times are as follows. With Randoop, Smack takes 335.8 seconds to extract target statements, 62.5 hours to generate unit tests, 5,749.6 seconds to execute the generated unit tests, 374.8 seconds to execute R0 inline tests, and 2.7 hours to perform mutation testing. With EvoSuite, the extraction, generation, unit test execution, R0 inline tests execution, and mutation testing times are 343.4 seconds, 2.5 hours, 2,162.2 seconds, 261.9 seconds, and 2.7 hours, respectively.

RQ4 shows that (i) Smack's reduction step significantly reduces the number of generated inline tests, which mitigates the potential impact on code readability; (ii) Smack's runtime costs are manageable and are similar to manual writing times, with the additional benefit of full automation and fault-detection capability from mutation testing.

5 Discussion

Impact of inline tests on code readability. As Liu et al. noted, Inline tests could degrade code readability (since they are co-located with the target statements that they validate) [55]. This concern is a subject of ongoing research, and seems to be a cost for improving test suites' fault-detection capability, even in target statements that method-level unit tests do not cover. Future work can investigate in-IDE support for automatically hiding inline tests by default. Developers can then view inline tests only when they need to do so.

```
1 public UrlResource(String url) throws MojoExecutionException {
2  if (url.startsWith(CLASSPATH_PREFIX)){
3   String resource = url.substring(CLASSPATH_PREFIX.length());
4   if (resource.startsWith(SLASH_PREFIX)){
5    resource = resource.substring(1);
6    itest().given(resource, "ReadPropertiesMojo_382").checkEq(resource, "
         eadPropertiesMojo_382");
7   } ...
8  } ...
9 }
```

Figure 19 Example showing a case where SMACK generates inline tests without taking context into account. SMACK is unaware that lines 2 and 4 limit values used in the target statement.

An alternative to transplanting? Instead of transplanting the generated inline tests, SMACK could leave the extracted class (e.g., Figure 4) in the CUT and refactor the original code by inserting a call to a method in the extracted class. Doing so has three drawbacks. First, leaving in the extracted class could be more burdensome on developers, who would then have to inspect three things: the refactored original code, the extracted class, and the generated inline tests. With transplanting, developers only have to inspect the inline tests. Second, leaving in many extracted classes could bloat the source code. Finally, leaving extracted classes violates the inline-testing goal to validate target statements in place.

Qualitative analysis of inline tests from ExLi and Smack. Syntactically, our manual inspection has so far not revealed any differences between inline tests generated by EXLI and SMACK. The differences come down to values that they use as test inputs and expected outputs. Since EXLI extracts those values from developer written unit tests, the values that it uses tend to be based on domain knowledge. On the other hand, values used in SMACK-generated inline tests are based on internal workings of unit-test generators. So, on statements that are covered by automatically generated method-level unit tests, values used in SMACK-generated inline tests can help trigger corner cases that developers forget to test for, but values from developer written unit tests are more likely to *not* trigger exceptions when running inline tests.

5.1 Limitations and Future Work

Generation without context. SMACK does not take into account useful information from the context of the target statement in the original enclosing method. For example, a conditional in that context might limit the set of values that should be used in the target statement. We use Figure 19 to illustrate this limitation. There, line 2 and line 4 restrict the inputs that can be used on the target statement on line 5 to only strings with prefix `CLASSPATH_PREFIX` (“`classpath`:”), followed by `SLASH_PREFIX` (“/”). By extracting only the target statement for inline-test generation, SMACK becomes unaware of this context. So, SMACK uses a string that does not satisfy the contextual requirements. The resulting inline test passes, kills mutants, but such input are unlikely to be realistic. There is value in generating inline tests with and without the context of the original enclosing method. We will explore how to capture more of that context for inline-test generation in the future. For example, it may be possible to extract a backward-slice from the target statement and use that slice as part of inline-test generation. Or we could collect invariants [17] for the surrounding method and use them as constraints during inline-test generation.

Programming language. We only implement SMACK for Java, to enable comparison with EXLI, which was also only implemented in and evaluated on Java. Also, the availability of robust unit-test generation tools – Randoop and EvoSuite– provide an extra motivation to use Java. We plan to explore inline-test generation for other languages, such as Python (for which there is already an inline testing tool [57]).

Supporting exceptional inline tests. Currently, SMACK cannot generate inline tests if all the tests generated by test generators throw exceptions. However, there are no theoretical limitations that prevent inline tests from expecting exceptions. Future work can explore how to support exception checking in ITEST and include tests that expect exceptions in the pipeline of automatic inline test generation.

6 Related Work

Single statement bugs. Prior work [43, 45, 47, 50, 77] found that many bugs are caused by faults in single statements, and that those faults are often missed by unit tests. This finding partially motivated the development of inline tests, and SMACK.

Automated test generation. Many automated test generation techniques were proposed, e.g., [2, 8, 13, 15, 25, 26, 64, 85]. Random testing [32], a black-box testing technique, generates unit tests by randomly selecting inputs from the input domain of the program under test. Randoop [70] is a popular tool that uses a feedback-directed random approach to generate unit tests. Search-based techniques, e.g., [4, 60], are alternatives to random approaches; they are white-box techniques that generate unit tests by searching for tests that satisfy a criterion. One notable search-based tool is EvoSuite [21], which focuses on optimizing coverage [78] or mutation scores [23]. SMACK uses Randoop and EvoSuite as generators to obtain unit tests from which inline tests are carved. Beyond that usage, our work is orthogonal to all prior test generation approaches: we focus on inline-test generation.

Large language models (LLMs). LLMs have been recently used to generate unit tests [37, 51, 63, 84, 102]. For example, TestPilot [84] generated tests and fixed failed tests by re-prompting the model with the failed tests and their exceptions. LLMs could be a unit-test generator. Future work can study the feasibility of fine-tuning LLMs to generate inline tests.

Asserts. Inline tests differ in at least three ways from assert statements [3, 7, 27, 38, 79, 101], which many programming languages support [6, 68, 90, 92, 95]. First, `asserts` do not allow providing arbitrary inputs and oracles for a statement. Second, unlike inline tests, `asserts` can run in production. Lastly, `asserts` check global program state at a code location, but inline tests are more local and test the input-output behavior of one statement.

Program synthesis. Program synthesis [31, 61] generates programs from specifications or input/output examples. LooPy [20, 48] allows developers to interactively synthesize program blocks. Future work could develop IDE plugins to enable interactive synthesis of inline tests, e.g., based on recent work on automatic test completion [63]. Doing so could be a valuable way to bring developers into the inline-test generation loop.

Test suite reduction/minimization. Test-suite reduction techniques [11, 22, 42, 49, 59, 65, 80, 87–89, 105, 106] find a minimal subset of a test suite that preserves some measure of test effectiveness, e.g., fault-detection capability or coverage. Some of those test-suite

reduction techniques use (1) greedy algorithms [10, 34, 91], (2) heuristics [9, 35], or (3) integer programming [36, 53]. SMACK uses a Greedy algorithm implementation [86] to reduce generated inline tests, while aiming to preserve mutation scores on the target statement. However, our implementation does not guarantee a minimal reduced set. For instance, if two inline tests are given to a reduction algorithm in an order such that the latter kills all mutants killed by the former, then both will be preserved. But if this order is reversed, then only one inline test will be preserved, because preserving only the second inline test kills all mutants that the first one kills. However, given that SMACK generates few inline tests per target statement on average after reduction, alternative reduction algorithms may not bring noticeable further reduction in many cases.

Mutation testing. Mutation testing is often used to evaluate test-suite effectiveness [33, 72, 74]. Popular Java mutation testing tools include universalmutator [29], Major [93], PIT [94], and MuJava [58]. Our SMACK implementation uses universalmutator for source code level mutation, thus allowing us to more easily select mutants on the target statements. Liu et al. [54] compared universalmutator and Major, and found that there is a negligible advantage of using the latter instead of the former for inline-test generation.

7 Conclusion

SMACK automatically generates inline tests even for statements that developer written unit tests do not cover and automatically generated unit tests do not reach. SMACK first extracts a target statement to a new and independent method. Then, SMACK uses existing unit-test generators to obtain unit tests for the extracted method. Inline tests are carved from the generated unit tests and then automatically transplanted to immediately follow the target statement in the original code. SMACK generates inline tests for 277 (88.8%) of 312 statements that are not covered by manually written or automatically generated method-level unit tests. Those generated inline tests kill 96.7% of 1,815 mutants that we seed on these target statements. SMACK introduces a flexible framework in which various unit test generators and test reduction techniques are integrated. Our results show that SMACK improves the fault-detection ability of existing test suites.

References

1 AlDanial. Cloc: Count Lines of Code. `https://github.com/AlDanial/cloc`.
2 M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *ICSE-SEIP*, pages 263–272, 2017. `doi:10.1109/icse-seip.2017.27`.
3 Mauricio Finavaro Aniche, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. What Do the Asserts in a Unit Test Tell Us About Code Quality? A Study on Open Source and Industrial Projects. In *CSMR*, pages 111–120, 2013. `doi:10.1109/csmr.2013.21`.
4 Andrea Arcuri and Gordon Fraser. Java enterprise edition support in search-based junit test generation. In *SSBSE*, pages 3–17, 2016. `doi:10.1007/978-3-319-47106-8_1`.
5 Steve Ash. Steveash jopenfst. `https://github.com/steveash/jopenfst/tree/14c4a1d/src/main/java/com/github/steveash/jopenfst/MutableArc.java#L124`, 2023.
6 Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java With Assertions. In *RV*, pages 103–117, 2001. `doi:10.1016/S1571-0661(04)00247-6`.
7 Kevin Boos, Chien-Liang Fok, Christine Julien, and Miryung Kim. Brace: An assertion framework for debugging cyber-physical systems. In *ICSE*, pages 1341–1344, 2012. `doi:10.1109/icse.2012.6227084`.

8 Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. Bounded Exhaustive Test-Input Generation on GPUs. *Proc. ACM Program. Lang.*, 1(OOPSLA):1–25, 2017. doi:10.1145/3133918.

9 Tsong Yueh Chen and Man Fai Lau. Heuristics towards the optimization of the size of a test suite. *WIT Transactions on Information and Communication Technologies*, 14, 1970.

10 Tsong Yueh Chen and Man Fai Lau. A simulation study on some heuristics for test suite reduction. *IST*, 40(13):777–787, 1998. doi:10.1016/s0950-5849(98)00094-9.

11 Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *ASE*, pages 237–249, 2020. doi:10.1145/3324884.3416667.

12 Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *ISSRE*, pages 201–211, 2014. doi:10.1109/issre.2014.11.

13 Matthew Davis, Sangheon Choi, Sam Estep, Brad Myers, and Joshua Sunshine. NaNofuzz: A usable tool for automatic test generation. In *FSE*, pages 1114–1126, 2023. doi:10.1145/3611643.3616327.

14 Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. In *FSE*, pages 654–674, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643756.

15 Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing probabilistic programming systems. In *FSE*, pages 574–586, 2018. doi:10.1145/3236024.3236057.

16 Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *TSE*, 35(1):29–45, 2008. doi:10.1109/TSE.2008.103.

17 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007. doi:10.1016/J.SCICO.2007.01.015.

18 Evosuite. http://www.evosuite.org.

19 Ralscha ExtClassGenerator. Ralscha extclassgenerator. https://github.com/ralscha/extclassgenerator/blob/40ad1471284143896aa0e3b4aad8135392befe9c/src/main/java/ch/rasc/extclassgenerator/validation/AbstractValidation.java#L257, 2023.

20 Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Loopy: Interactive program synthesis with control structures. *OOPSLA*, 5:1–29, 2021. doi:10.1145/3485530.

21 Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *FSE*, pages 416–419, 2011. doi:10.1145/2025113.2025179.

22 Gordon Fraser and Andrea Arcuri. Whole test suite generation. *TSE*, 39(2):276–291, 2012. doi:10.1109/tse.2012.14.

23 Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *ESE*, 20:783–812, 2015. doi:10.1007/s10664-013-9299-z.

24 Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev, and Mattia Fazzini. Action-based test carving for Android apps. In *ICSTW*, pages 107–116, 2023. doi:10.1109/icstw58534.2023.00032.

25 Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: An automatic test generation tool for industrial Java applications with strings. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 992–1001, 2013. doi:10.1109/icse.2013.6606649.

26 Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test Generation Through Programming in UDITA. In *ICSE*, pages 225–234, 2010. doi:10.1145/1806799.1806835.

27 Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *DFT*, pages 581–588, 2003. doi:10.1109/DFTVS.2003.1250158.

28 Mark Grechanik and Gurudev Devanla. Generating integration tests automatically using frequent patterns of method execution sequences. In *SEKE*, pages 209–280, 2019. doi:10.18293/seke2019-001.

29 Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *ICSE-Demo*, pages 25–28, 2018. doi:10.1145/3183440.3183485.

30 Kevin Guan, Pengyue Jiang, Milos Gligoric, and Owolabi Legunsen. Block tests. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA):2044–2072, 2026.

31 Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. doi:10.1561/2500000010.

32 Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994. doi:10.1017/9781108974073.015.

33 Farah Hariri, August Shi, Owolabi Legunsen, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. Approximate transformations as mutation operators. In *ICST*, pages 285–296, 2018. doi:10.1109/icst.2018.00036.

34 Preethi Harris and Raju Nedunchezhian. A greedy approach for coverage-based test suite reduction. *IAJIT*, 12:17–23, 2015. URL: https://www.ccis2k.org/iajit/PDF/vol.12,no.1/5246.pdf.

35 M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *TOSEM*, 2(3):270–285, 1993. doi:10.1109/icsm.1990.131378.

36 Joshua Hartmann and Dave J Robson. Revalidation during the software maintenance phase. In *ICSM*, pages 70–80, 1989. doi:10.1109/icsm.1989.65195.

37 Sepehr Hashtroudi, Jiho Shin, Hadi Hemmati, and Song Wang. Domain adaptation for deep unit test case generation, 2023. doi:10.48550/arXiv.2308.08033.

38 Charles Antony Richard Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003. doi:10.1142/9781848162914_0005.

39 Maxmind Inc. Maxmind geoip-api-java. https://github.com/maxmind/geoip-api-java/blob/1030316a1e9fe5ccce02446fe65b2cd800d7eeed/src/main/java/com/maxmind/geoip/LookupService.java#L939, 2023.

40 Pytest-Inline on PyPi. https://pypi.org/project/pytest-inline.

41 JavaParser Team. JavaParser. https://github.com/javaparser/javaparser, 2023.

42 Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *TSE*, 33(2):108–123, 2007. doi:10.1109/tse.2007.18.

43 Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. In *MSR*, pages 563–575, 2023. doi:10.1109/msr59073.2023.00082.

44 René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014. doi:10.1145/2635868.2635929.

45 Arthur V Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. PySStuBs: Characterizing Single-Statement Bugs in Popular Open-Source Python Projects. In *MSR*, pages 520–524, 2021. doi:10.1109/msr52588.2021.00066.

46 Alexander Kampmann and Andreas Zeller. Carving parameterized unit tests. In *ICSE Companion*, pages 248–249, 2019. doi:10.1109/icse-companion.2019.00098.

47 Rafael-Michael Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *MSR*, pages 573–577, 2020. doi:10.1145/3379597.3387491.

48 Tomer Katz and Hila Peleg. Programming-by-example with nested examples. In *VL/HCC*, pages 280–282, 2023. doi:10.1109/vl-hcc57772.2023.00053.

49 Saif Ur Rehman Khan, Sai Peck Lee, Reza Meimandi Parizi, and Manzoor Elahi. A code coverage-based test suite reduction and prioritization framework. *WICT*, pages 229–234, 2014. doi:10.1109/wict.2014.7076910.

50 Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. How effective is continuous integration in indicating single-statement bugs? In *MSR*, pages 500–504, 2021. doi:10.1109/msr52588.2021.00062.

51 Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *ICSE*, pages 919–931, 2023. doi:10.1109/icse48619.2023.00085.

52 Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *ICSM*, pages 290–301, 1990. doi:10.1109/icsm.1990.131377.

53 Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *ICSE*, pages 1039–1049, 2018. doi:10.1145/3180155.3180174.

54 Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. Extracting inline tests from unit tests. In *ISSTA*, pages 1458–1470, 2023. doi:10.1145/3597926.3598149.

55 Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. Inline tests. In *ASE*, pages 1–13, 2022. doi:10.1145/3551349.3556952.

56 Yu Liu, Aditya Thimmaiah, Owolabi Legunsen, and Milos Gligoric. ExLi: An Inline-Test Generation Tool for Java. In *FSE (Tool Demo Track)*, pages 1–5, 2024.

57 Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. Pytest-Inline: An Inline Testing Tool for Python. In *ICSE-Demo*, pages 1–4, 2023. doi:10.1109/icse-companion58688.2023.00046.

58 Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system. *STVR*, 15(2):97–133, 2005. doi:10.1002/stvr.308.

59 Alessandro Marchetto, Giuseppe Scanniello, and Angelo Susi. Combining code and requirements coverage with execution cost for test suite reduction. *TSE*, 45:363–390, 2019. doi:10.1109/tse.2017.2777831.

60 Phil McMinn. Search-based software test data generation: A survey. *STVR*, 14(2):105–156, 2004. doi:10.1002/stvr.294.

61 Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013. doi:10.5555/3042817.3042840.

62 Mountainminds GmbH & Co. KG and Contributors. JaCoCo - Java Code Coverage Library. https://www.jacoco.org/jacoco, 2023.

63 Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *ICSE*, pages 2111–2123, 2023. doi:10.1109/icse48619.2023.00178.

64 Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. Unifying execution of imperative generators and declarative specifications. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. doi:10.1145/3428285.

65 Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In *SWQD*, pages 51–66, 2019. doi:10.1007/978-3-030-35510-4_4.

66 Jkuhnert Ognl. Jkuhnert ognl. https://github.com/jkuhnert/ognl/tree/5c30e1e/src/main/java/ognl/enhance/ExpressionCompiler.java#L639, 2023.

67 Oracle. Chapter 4. the class file format. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.3, 2022.

68 Oracle. Programming with assertions. https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html, 2022.

69 Alessandro Orso. Integration testing of object-oriented software. *Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano*, pages 1–119, 1998.

70 Carlos Pacheco and Michael D Ernst. Randoop: Feedback-Directed Random Testing for Java. In *OOPSLA*, pages 815–816, 2007. doi:10.1145/1297846.1297902.

71 Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007. doi:10.1109/icse.2007.37.

72 Mike Papadakis, Marinos Kintis, Jie M. Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *ADV Computers*, 112:275–378, 2019. doi:10.1016/bs.adcom.2018.03.015.

73 Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical Mutation Testing at Scale: A View From Google. *TSE*, 48(10):3900–3912, 2021. doi:10.1109/tse.2021.3107634.

74 Goran R. Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does Mutation Testing Improve Testing Practices? In *ICSE*, pages 910–921, 2021. doi:10.1109/icse43902.2021.00087.

75 Mojohaus project. Mojohaus Build Helper Maven Plugin. https://github.com/mojohaus/build-helper-maven-plugin/tree/f1fac8c/src/main/java/org/codehaus/mojo/buildhelper/ReserveListenerPortMojo.java#L416, 2023.

76 Inline Testing Team, 2023. https://github.com/pytest-dev/pytest-inline.

77 Cedric Richter and Heike Wehrheim. Tssb-3m: Mining single statement bugs at massive scale. In *MSR*, pages 418–422, 2022. doi:10.1145/3524842.3528505.

78 José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *SSBSE*, pages 93–108, 2015. doi:10.1007/978-3-319-22183-0_7.

79 David S. Rosenblum. A practical approach to programming with assertions. *TSE*, 21(1):19–31, 1995. doi:10.1109/32.341844.

80 Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998. doi:10.1109/icsm.1998.738487.

81 Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006. doi:10.1109/ms.2006.91.

82 David Saff, Shay Artzi, Jeff H Perkins, and Michael D Ernst. Automatic test factoring for Java. In *ASE*, pages 114–123, 2005. doi:10.1145/1101908.1101927.

83 Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. EvoSuite at the SBST 2022 Tool Competition. In *SBST*, pages 33–34, 2022. doi:10.1145/3526072.3527526.

84 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model. In *arXiv*, pages 1–14, 2023. doi:10.48550/arXiv.2302.06527.

85 Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In *ASE*, pages 201–211, 2015. doi:10.1109/ase.2015.86.

86 August Shi. Collection of scripts to conduct test-suite reduction. https://github.com/august782/testsuite-reduction, 2023.

87 August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014. doi:10.1145/2635868.2635921.

88 August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. Evaluating test-suite reduction in real software evolution. In *ISSTA*, pages 84–94, 2018. doi:10.1145/3213846.3213875.

89 August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *FSE*, pages 237–247, 2015. doi:10.1145/2786805.2786878.

90 Volker Stolz and Eric Bodden. Temporal Assertions Using AspectJ. *ENTCS*, 144(4):109–124, 2006. doi:10.1016/j.entcs.2006.02.007.

91 Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005. doi:10.1145/1108768.1108802.

92 Richard N Taylor. Assertions in programming languages. *SIGPLAN Notices*, 15(1):105–114, 1980. doi:10.1145/954127.954139.

93 Major Team. Major mutation framework. https://mutation-testing.org/, 2023.

94 Pitest Team. PIT - mutation testing for Java. https://pitest.org/, 2022.

95 Python 3.10.5 Documentation Team. Simple statements. https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement, 2022.

96 Randoop Team. Randoop manual. https://randoop.github.io/randoop/manual/, 2023.

97 Wmixvideo Team. Wmixvideo nfe. https://github.com/wmixvideo/nfe/, 2023.

98 XStream Team. Xstream. https://x-stream.github.io/, 2024.

99 Fabian Trautsch. *An Analysis of the Differences Between Unit and Integration Tests*. PhD thesis, Georg-August University, 2019. doi:10.53846/goediss-7393.

100 W.T. Tsai, Xiaoying Bai, R. Paul, Weiguang Shao, and V. Agarwal. End-to-end integration testing design. In *COMPSAC*, pages 166–171, 2001. doi:10.1109/cmpsac.2001.960613.

101 Jeffrey M Voas and Keith W Miller. Putting assertions in their place. In *ISSRE*, pages 152–157, 1994. doi:10.1109/issre.1994.341367.

102 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software Testing With Large Language Models: Survey, Landscape, and Vision. *TSE*, 50(4):911–936, 2024. doi:10.1109/TSE.2024.3368208.

103 Rahulkrishna Yandrapally and Ali Mesbah. Mutation analysis for assessing end-to-end web tests. In *ICSME*, pages 183–194, 2021. doi:10.26226/morressier.613b5417842293c031b5b5c3.

104 Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. Carving UI tests to generate API tests and API specification. In *ICSE*, pages 1971–1982, 2023. doi:10.1109/icse48619.2023.00167.

105 Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012. doi:10.1002/stv.430.

106 Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An Empirical Study of JUnit Test-Suite Reduction. In *ISSRE*, pages 170–179, 2011. doi:10.1109/issre.2011.26.