Extracting Inline Tests from Unit Tests

Yu Liu*, Pengyu Nie*, Anna Guo*, Milos Gligoric*, Owolabi Legunsen†
*The University of Texas at Austin; †Cornell University
USA
{yuki.liu,pynie,anna.guo,gligoric}@utexas.edu,legunsen@cornell.edu

ABSTRACT

We recently proposed inline tests for validating individual program statements; they allow developers to provide test inputs, expected outputs, and test oracles immediately after a target statement. But, existing code can have many target statements. So, automatic generation of inline tests is an important next step towards increasing their adoption. We propose ExLi, the first technique for automatically generating inline tests. ExLi extracts inline tests from unit tests; it first records all variable values at a target statement while executing unit tests. Then, ExLi uses those values as test inputs and test oracles in an initial set of generated inline tests. Target statements that are executed many times could have redundant initial inline tests. So, ExLi uses a novel coverage-then-mutants based reduction process to remove redundant inline tests. We implement ExLi for Java and use it to generate inline tests for 718 target statements in 31 open-source programs. ExLi reduces 17,273 initially generated inline tests to 905 inline tests. The final set of generated inline tests kills up to 25.1% more mutants on target statements than developer written and automatically generated unit tests. That is, ExLi generates inline tests that can improve the fault-detection capability of the test suites from which they are extracted.

CCS CONCEPTS

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

KEYWORDS

Inline tests, unit tests, automatic test generation

ACM Reference Format:
Yu Liu*, Pengyu Nie*, Anna Guo*, Milos Gligoric*, Owolabi Legunsen†. 2023. Extracting Inline Tests from Unit Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598149

1 INTRODUCTION

Inline tests [46] enable developers to test individual program statements, thereby increasing the fault-detection capability of test suites. Inline tests are complementary to existing levels of test granularity—unit tests, integration tests, and end-to-end tests. Inline tests can help find single-statement bugs, which often occur [38, 39] but elude unit tests [42]. Inline tests can also provide other software engineering benefits, e.g., they document complex *target statements* and they could be easier to co-evolve with code than unit tests.

Previously, we developed two tools to provide framework-level support for inline testing. These tools make it easier for developers to write inline tests and they increase the chances for the relatively new inline testing paradigm to be adopted. One tool, pytest-inline, supports inline testing in Python [33, 47]; it is integrated with pytest, the most popular testing framework for Python [34], and has already been downloaded 2,472 times [65]. We presented the other tool for Java, ITest, in our original inline testing paper [46].

Automatic generation of inline tests is an important next step towards increasing their adoption for two reasons. First, automatic generation can reduce manual developer effort for retrofitting inline tests into existing code bases that have many target statements. Second, automatic generation can enable future inline testing research by providing more inline tests for evaluation than exist today. For example, we previously simulated runtime costs by repeatedly executing 152 manually written inline tests thousands of times [46].

We propose ExLi, *the first technique for automatically generating inline tests.* ExLi extracts inline tests from unit tests. Unit tests are an attractive source of inline tests: they are abundant in practice and they can be automatically generated [16, 62, 69]. In turn, the extracted inline tests can help find single-statement bugs that unit tests miss [42]. Extracted inline tests can also help find bugs in executed statements that are deeply-nested in conditional expressions, which can be missed by automatically generated unit tests [1].

Given the code under test (CUT), a target statement, and unit tests that cover the target statement, ExLi generates a set of inline tests for the target statement. ExLi can automatically discover the four kinds of target statements that we identified in prior work as being able to benefit from inline testing [46], and extract inline tests from the unit tests that cover them.

ExLi is agnostic to the source of unit tests; they can be manually written by developers or automatically generated by tools like Randoop [62, 69] or EvoSuite [16]. ExLi outputs a new version of the CUT in which the target statement is immediately followed by the generated inline tests. Since ExLi is a first step towards inline test generation, we assume that unit tests correctly exercise the CUT. That is, the inline tests generated by ExLi on one code version could detect regression bugs in future versions of the code.

ExLi first instruments the CUT to record all observed variable values in the target statement during unit testing. Then, the recorded values are used to automatically generate inline tests. For example, consider assignment statements. The recorded values of right-hand side variables are used as input values, and the recorded values of the left-hand side variable are used as expected values in the

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598149

generated inline test. ExLi can also generate inline tests for declarations and expressions in `if` conditions. We plan to support more locations of target statements in the future.

Inline tests are co-located with target statements, so an important concern is that readability could be degraded if too many inline tests are generated per target statement. Compilation could also fail if adding the generated inline tests causes a method's body to exceed the maximum allowable size [61]. Too many inline tests can be generated for target statements in which many sets of values are observed during unit testing. Such many-valued target statements could be covered by many unit tests, or they may be in loops. In a particularly egregious case, 14,928 sets of values were recorded for a target statement during our experiments.

To address the concern of generating too many initial inline tests per target statement, ExLi introduces a *coverage-then-mutants based* test reduction process. We consider an inline test to be redundant if it has the same fault-detection capability as other inline tests with respect to code covered and mutants killed. Code coverage [7, 21] and mutation score [36, 70] are established metrics for measuring the quality and fault-detection capability of unit tests. We adapt these two metrics to guide inline test reduction.

ExLi uses both *target coverage*—code covered while executing the target statement—and *context coverage*—code covered while executing the enclosing basic block of the target statement. ExLi also builds on existing mutation analysis tools [25, 37] but it only mutates target statements.

The coverage-then-mutants based test reduction process in ExLi works as follows. ExLi tracks the code covered in the target statement and its context during unit testing, and only records sets of values that cover code that was not covered by previously extracted sets of values. ExLi also mutates the target statement and ensures that each generated inline test kills at least one unique mutant. If no mutant is generated for a target statement, ExLi's reduction is based on coverage. But, if coverage and mutation scores are computed, reduction is based on mutation score as prior work suggests that mutation score is a more accurate metric of the fault-detection capability than coverage [74].

We implement ExLi for Java and apply it to 718 target statements in 31 open-source programs. ExLi generates an initial set of 17,273 inline tests. ExLi-UM, which uses universalmutator [25] for mutation analysis, generates a final set of 905 inline tests (reduction rate: 94.8%). ExLi-Major, which uses Major [37] for mutation analysis, generates a final set of 930 inline tests (reduction rate: 94.6%).

We also evaluate whether generated inline tests enhance the fault-detection capability of test suites from which they are extracted. We do so by performing mutation analysis only on the target statements. ExLi-UM kills 25.1% more mutants, and ExLi-Major kills 24.6% more mutants than those killed by developer written and automatically generated unit tests. Our manual inspection shows why generated inline tests can kill more mutants: the unit tests reach the target statements and infect program state, but those unit tests lack "local" oracles at the target statement. That is, errors induced by mutants do not propagate to the assertions in the unit tests, or those assertions do not check relevant parts of state.

This paper makes the following contributions:

```
1 public static final String MULTI_VALUE_DELIMITTER = ",";
2 public static final char EQ = '=';
3 public static void setAdditionalFields(String spec,GelfMsg gelfMsg){
4  if (null != spec) {
5   String[] properties = spec.split(MULTI_VALUE_DELIMITTER);
6   for (String field : properties) {
7    final int index = field.indexOf(EQ); // target statement
8    itest().given(field, "profile.requestStart.ms").given(EQ, '=')
9           .checkEq(index, -1);
10   itest().given(field, " mdcName='long']").given(EQ, '=')
11          .checkEq(index, 8);
12   if (-1 == index) { continue; }
13   ... // add field to gelfMsg
14 }}}
```

Figure 1: Target statement with ExLi-generated inline tests.

- ★ **Technique.** ExLi is the first technique for automatically generating inline tests; it extracts inline tests from unit tests.
- ★ **Reduction approach.** ExLi uses a novel inline test reduction approach that is based on both code coverage and mutation score.
- ★ **Evaluation.** ExLi's reduction strategy is effective, yielding inline tests that improve the fault-detection capability of unit test suites.
- ★ **Dataset.** ExLi generates the largest dataset of inline tests to date. ExLi and our dataset can enable future work on inline tests.

ExLi and our dataset is open-sourced at https://github.com/EngineeringSoftware/exli.

2 EXAMPLE

Figure 1 shows an example code with a target statement and inline tests that ExLi generates for that target statement after reduction. The example is simplified from mp911de/logstash-gelf [49]. Method `setAdditionalFields` splits the value stored in `spec` using `MULTI_VALUE_DELIMITTER` (",") as the delimiter, stores the results in `properties`, and adds each `field` in `properties` that contains `EQ` ("=") to `gelfMsg`. Line 7 is the target statement; it finds the index of first occurrence of `EQ` in `field`. All variables in this example have primitive or String types, but ExLi supports complex non-primitive types as well (see example in Figure 6, Section 4). A developer could use ExLi to generate inline tests for this target statement; it is in a loop and it is reached by lots of other methods.

Line 8 is one of the two inline tests that ExLi generates. All inline tests have three parts. First, the "Declare" part—`itest()`—marks the current statement as an inline test. Second, the "Assign" part—`given(field, "profile.requestStart.ms").given(EQ, '=')`—provides inputs to the inline test. Third, the "Assert" part—`checkEq(index, -1)`—specifies a test oracle, including an expected output. In Figure 1, given the inputs for `field` and `EQ`, the `index` variable computed by the target statement should be −1 for the inline test on line 8 to pass.

The example target statement is executed 2,413 times with 215 unique sets of values during unit testing. But, directly generating 215 inline tests to check one statement could be an overkill for two reasons. First, many of the 215 sets of values are redundant because they exercise the target statement in the same way. So, using them all is wasteful. Second, adding 215 inline tests for this target statement will likely make the code harder to read and maintain. So, ExLi must reduce the number of generated inline tests by eliminating redundancy. ExLi's coverage-then-mutants based

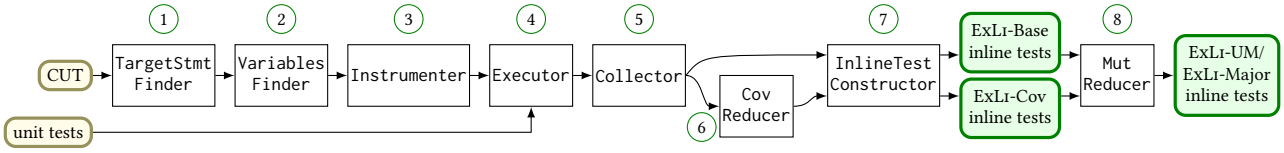


Figure 2: The steps in ExLi's workflow.

reduction process reduces those 215 inline tests to the two shown in Figure 1, without loss in fault-detection capability.

3 TECHNIQUE

Figure 2 shows ExLi's procedure for generating inline tests. The inputs are the CUT (required), the unit tests (required), and line numbers of the target statements (optional, not shown). ExLi outputs the generated inline tests after the coverage-then-mutants based reduction. ExLi also produces two intermediate outputs for evaluation and debugging purposes: ExLi-Base inline tests without any reduction; and ExLi-Cov inline tests with reduction based only on code coverage but not mutation score.

3.1 Finding and Analyzing Target Statements

The first two steps of ExLi's workflow are for finding and analyzing the target statements. In step ①, `TargetStmtFinder` parses the abstract syntax tree (AST) of the CUT and extracts the target statements. If developers provided the optional input of line numbers of the target statements, ExLi will skip this step and directly use the developer-specified target statements. Then, in step ②, `VariablesFinder` identifies the variables used in each target statement, which will be the input or output variables in the generated inline tests. For example, `VariablesFinder` should identify three variables for the target statement in Figure 1: two input variables, `field` and `EQ`, and one output variable `index`.

3.2 Generating Inline Tests

We here describe steps ③, ④, ⑤, and ⑦, which generate ExLi-Base inline tests without performing reduction.

First, the `Instrumenter` (step ③) adds code *before* each target statement to collect the values of input variables and *after* each target statement to collect the values of output variables. Figure 3 shows how we instrument the code in Figure 1: `collectInputs` (line 7) is added before the target statement to collect the values of `field` and `EQ`, and `collectOutputs` (line 9) is added after the target statement to collect the value of `index`. Other code added by `Instrumenter` for test reduction will be described in Section 3.3.

Then, the `Executor` (step ④) runs unit tests on the instrumented code, and the `Collector` stores in memory the *unique* sets of values observed during unit testing (step ⑤).

Using the collected sets of values, `InlineTestConstructor` (step ⑦) synthesizes inline tests. To do so, the value collected for each input variable is used in `given(...)` calls; these calls can be chained. That is, the inline test will assign each value to the corresponding input variable when testing the target statement. Then, the value collected for each output variable is used in a `check_eq(...)` construct. That is, inline tests check that the resulting value in

```
1 public static void setAdditionalFields(String spec,GelfMsg gelfMsg){
2   if (null != spec) {
3     String[] properties = spec.split(MULTI_VALUE_DELIMITTER);
4     for (String field : properties) {
5       try {
6         collectCov(); // cov1
7         collectInputs(field, EQ);
8         final int index = field.indexOf(EQ); // target statement
9         collectOutputs(index);
10        collectCov(); // cov2
11        if (-1 == index) { continue; }
12        ... // add field to gelfMsg
13      } finally { collectCov(); } // cov3
14 }}}
```

Figure 3: Example showing ExLi's instrumentation.

each output variable after executing the target statement using the assigned input values equals outputs recorded during unit testing.

The `InlineTestConstructor` also edits the CUT to insert constructed inline tests right after the target statement. After that, ExLi uses ITest [46] (our inline testing tool for Java) to run each generated inline test. If any inline test fails, ExLi filters it out: the failing inline test is removed from the CUT. Such failing inline tests are due to the target statement using inputs other than the input variables (e.g., a static variable used in a method invoked from the target statement) that is not collected by ExLi; future work can explore storing such inputs from the global program state.

3.3 Coverage-then-Mutants Based Reduction

ExLi-Base generates an inline test for each unique set of values collected during executing unit tests. But, too many sets of values could be collected for some target statements even if we only keep unique sets of values (Section 1). We observe in our experiments that many sets of values are redundant with respect to one another: they have similar fault-detection capability and exercise the target statement in the same way. (Recall that, from a unit testing point of view, the sets of values that ExLi collects are intermediate values.)

To avoid generating redundant inline tests, ExLi uses a novel *coverage-then-mutants based* test reduction process: reducing the inline tests (or sets of values, if reducing before constructing inline tests) that have redundant fault-detection capability, using both code coverage [7, 21] and mutation score [36, 70] as metrics for fault-detection capability.

3.3.1 Reduction by Code Coverage. ExLi collects code coverage using JaCoCo [56], a widely-used code coverage tool for Java. To fit the inline testing scenario, ExLi considers two kinds of code coverage: *target coverage*, the coverage collected while executing the target statement; and *context coverage*, the coverage after executing the target statement while executing the *context* of the target statement. The context of a target statement is defined as

Algorithm 1 CovReducer

Global var: `tgtStmtToCovered`: mapping from target statement to the set of lines covered by the target statement's collected values
Inputs: `cov1`, `cov2`, `cov3`: code coverage information for the current set of values; $\ell 0$: target statement's lineno
Outputs: `true` if the set of values should be kept, `false` otherwise

```
1:  procedure shouldKeepValues(cov1, cov2, cov3, ℓ0)
2:    tgtCovChanged ← covChanged(cov1, cov2, ℓ0)
3:    ctxCovChanged ← covChanged(cov2, cov3, ℓ0)
4:    return tgtCovChanged ∨ ctxCovChanged
5:  procedure covChanged(cov, cov′, ℓ0)
6:    change ← false
7:    for ℓ ∈ cov′.keys() do
8:      if ℓ ∉ cov ∨ cov[ℓ] < cov′[ℓ] then        ▷ line ℓ coverage changed
9:        if ℓ ∉ tgtStmtToCovered[ℓ0] then
                                  ▷ line ℓ not covered by ℓ0's collected values
10:         change ← true
11:         tgtStmtToCovered[ℓ0] ← tgtStmtToCovered[ℓ0] ∪ {ℓ}
12:   return change
```

code between the target statement and the end of its enclosing basic block. For example, for the target statement in Figure 1 (line 7), its enclosing basic block is the `for` loop from lines 6 to 14, and its context is the code from lines 12 to 14. Using context coverage in addition to target coverage makes reduction more accurate. The target coverage alone may not provide enough information to distinguish non-redundant inline tests. For example, the inline tests at line 8 and line 10 in Figure 1, which have different fault-detection capability, have the same target coverage, but they have different context coverage because only the first inline test covers the `then` branch of the `if` statement in the context at line 12.

To collect target coverage and context coverage, `Instrumenter` (step ③) adds code to collect code coverage at three points, see the `collectCov` calls in Figure 3: (1) the instruction-level coverage just before the target statement (line 6, `cov1`), (2) the instruction-level coverage right after the target statement (line 10, `cov2`) and (3) the instruction-level coverage at the end of target statement's enclosing basic block (line 13, `cov3`). Then, `CovReducer` (step ⑥) processes each collected set of values and instruction-level coverage information. Only sets of values that increase either target coverage or context coverage of the corresponding target statement are kept and sent to `InlineTestConstructor`.

The shouldKeepValues procedure in Algorithm 1 describes how `CovReducer` computes the target coverage and context coverage and decides when to keep a set of values. The inputs are code coverage information `cov1`, `cov2`, `cov3`, and target statement $\ell 0$. `CovReducer` uses a global map, `tgtStmtToCovered`, to store the code coverage metric: the lines of code covered by the collected sets of values (which is initialized to empty) of each target statement. shouldKeepValues checks if the target coverage changed (line 2) and if the context coverage changed (line 3) and returns `true` if either changed. covChanged compares the code coverage at two points, and checks if the later one has covered any line not covered by the former one (line 8) and that line was not covered by previously collected values (line 9). If so, covChanged updates `tgtStmtToCovered` and returns `true`. The instruction-level coverage reported by JaCoCo is a mapping from line number to the count of instructions on that line being covered. So, line 8 considers a line's coverage as changed if its instruction counts changed (from zero to non-zero; or, from non-zero to a larger value for ternary operators or Boolean expressions).

3.3.2 Reduction by Mutation Score. Mutation score is an established measure of the fault-detection capability of tests [36, 70]; it is the ratio of mutants killed by the unit tests (i.e., that cause the tests to fail) to the total number of mutants. Mutants are typically small syntactic modifications to the CUT that simulate seeded faults. ExLi uses two popular mutation generators for Java: universalmutator [25] and Major [37]. ExLi uses all mutation operators in the two generators, but it only mutates target statements. To do so, we specify line numbers to mutate (for universalmutator) or filter out mutants that are not for the target statements (for Major).

`MutReducer` (step ⑧) performs reduction by mutation score, given the ExLi-Base inline tests without reduction and ExLi-Cov inline tests after reduction by code coverage. Note that the mutant generator may fail to generate mutants for some target statements (9.6% for universalmutator, 8.9% for Major), in which case mutation score cannot be computed, and `MutReducer` will directly output the ExLi-Cov inline tests for those target statements. For all other target statements, `MutReducer` further reduces the coverage-reduced inline tests by mutation score, which prior work suggests measures fault-detection capability more accurately than coverage [74].

`MutReducer` first executes the ExLi-Base and ExLi-Cov inline tests on the mutants and maps each inline test to mutants that it kills. Then, `MutReducer` uses the Greedy test-suite reduction algorithm [87] (used in prior work [74, 76, 78]), using the mapping of ExLi-Cov inline tests to killed mutants, to reduce ExLi-Cov inline tests that kill the same mutants. Each inline test in the reduced set kills at least one unique mutant. Finally, if ExLi-Base inline tests kill any mutant that is not killed by the reduced ExLi-Cov inline tests, then reduction by coverage results in a loss in mutation score. So, `MutReducer` adds one ExLi-Base inline test that killed that mutant to the reduced inline tests to remedy this loss.

We refer to the final set of inline tests after `MutReducer` as ExLi-UM or ExLi-Major, when using universalmutator or Major as the mutant generator, respectively. So, the final set of inline tests preserves fault-detection capability, as measured by mutation score, compared to ExLi-Base inline tests before reduction.

Remark 1. Conceptually, ExLi could directly use test-suite reduction with respect to mutants on the target statement to reduce the collected sets of values. Instead, we make the design choice to first use reduction by code coverage for three reasons. First, using mutants for minimization requires to first generate inline tests for all the collected sets of values. It is not always possible to do so due to limits on method sizes [61]. Second, using reduction by code coverage has the benefit that we can use mutation testing as a sanity check of the fault-detection capability of the reduced set of inline tests. There would be no automated sanity check if mutation testing is used initially. Lastly, ExLi will need to preserve all inline tests for target statements in which no mutant is created. So, if ExLi only uses reduction by mutation score and if a frequently covered target statement has no mutants, then readability may degrade because too many inline tests are generated.

Table 1: API used to filter statements.

Type	API
Regex	Matcher.matches(), Matcher.find(), Matcher.group()
String	String.split(), String.substring(), String.indexOf(), String.format(), String.replace()
Bit	», «, &, |, ^, ~, &=, |=, ^=, »=, «=
Stream	Stream.of(), *.stream()

Remark 2. Implicitly, generating inline tests from unit tests induces a trade-off space among the competing goals of good readability, high coverage, and high fault-detection capability. Since inline tests are co-located with the CUT, fewer inline tests will likely lead to better readability, but at the cost of possibly lower coverage or lower fault-detection capability. We design ExLi to have high readability and high fault-detection capability at the cost of possible loss in the code coverage of the target statement or its context. Specifically, reduction by mutation score is not guaranteed to preserve the code coverage achieved by ExLi-Cov inline tests. We optimize for code maintenance settings where high readability with high fault-detection capability is likely preferable to poor readability. ExLi can be configured to optimize differently along the trade-off space. Also, now that ExLi can generate many more inline tests than previously possible, future work can more easily perform user studies of developers' trade-off preferences.

4 IMPLEMENTATION

We describe our ExLi implementation, using the same step numbers as in Section 3 to make our descriptions easier to follow.

① **Find target statements**. ExLi currently supports finding the same four kinds of Java target statements as in our prior work [46]: regular expressions, string manipulation, bit manipulation, and stream processing. Given a kind of target statement, `TargetStmtFinder` searches for target statements that use APIs that are commonly used in the kinds of statements of interest. Table 1 lists the APIs that ExLi searches for. Unlike our earlier ITest prototype that searches program text, ExLi improves accuracy by parsing the AST (using JavaParser [35]) to find target statements.

② **Identify variables**. `VariablesFinder` parses the AST of a given target statement (using JavaParser [35]) to identify its free variables, i.e., not including the variables whose scope is fully contained by the target statement. For example, in the following target statement, `str` and `list` are free variables, but `item` is not:

```
String str = list.stream().map(item -> item.replace("a",
    "b")).collect(Collectors.joining(","));
```

An array indexing expression, e.g., `arr[i]`, is also treated as a single variable, because inline tests may only need to assign to, or check certain elements of the array.

③ **Instrument CUT**. `Instrumenter` is implemented using the ASM library [6]. ExLi currently supports instrumenting target statements at three syntactic locations:

- *Condition of an* `if` *statement.* Figure 4 shows an example from `json-schema-validator` [57]. Line 7 is the target statement; it checks if `value` matches a pattern. `Instrumenter` adds code before the `if` statement (line 6) to collect input variables, at the beginning of the `then` branch (line 8) to collect true as the value of the output variable—the result of evaluating a conditional expression, and at the beginning of the `else` branch (line 15) to collect false as the value of the output variable.
- *Declaration statement.* `Instrumenter` adds code before the target statement to collect right-hand side variable values and after the target statement to collect left-hand side variable values.
- *Assignment statement.* `Instrumenter` adds code to collect left- and right-hand side variable values before the target statement and to collect left-hand side variable values after the target statement. Left-hand side variables are collected both before and after the target statement, because they may be both input and output variables in compound assignment statements like `a += 1`.

Moreover, `Instrumenter` handles the following special cases:

- If there is an increment/decrement expression in an array index, `Instrumenter` rewrites the array-indexing expression such that the correct element is collected. For example, in Figure 5, the output variable on line 6 is `mOutBuffer[ptr++]`, but its value is collected on line 7 as `mOutBuffer[ptr - 1]` because `ptr` would be incremented after executing the target statement.
- Some target statements are in `if` blocks that have jump (`return`, `break`, `continue`, `throw`, etc.) instructions in the `then` and `else` branches. To avoid compilation error (unreachable code) that would occur if `Instrumenter` adds code to the end of blocks in

```
1  public String[] match(String value) { ...
2    for (int i = 0; i < patterns.length; i++) {
3      try {
4        Matcher matcher = patterns[i].matcher(value);
5        collectCov(); // cov1
6        collectInputs(matcher);
7        if (matcher.matches()) { // target statement
8          collectOutputCond(true);
9          collectCov(); // cov2
10         int count = matcher.groupCount();
11         String[] groups = new String[count];
12         for (int j = 0; j < count; j++)
13           groups[j] = matcher.group(j + 1);
14         return groups;
15       } else { collectOutputCond(false); }
16     } finally { collectCov(); } // cov3
17   }
18   return null; }
```

Figure 4: Example of ExLi instrumenting a target statement at a condition of an `if` statement.

```
1  public void write(int c) throws IOException {...
2    if (c < 0x800) {
3      try {
4        collectCov(); // cov1
5        collectInputs(ptr, c);
6        mOutBuffer[ptr++] = (byte) (0xc0 | (c >> 6)); // target statement
7        collectOutputs(mOutBuffer[ptr-1]);
8        // wrong: collectOutputs(mOutBuffer[ptr]);
9        collectCov(); // cov2
10       ...
11     } finally { collectCov(); } // cov3
12   } ... }
```

Figure 5: Example of ExLi instrumenting a target statement with an increment expression in an array index.

```
1 public CompiledTemplate compile(IdentifiableStringTemplateSource
2    templateSource) throws TemplateException {
3  String id = templateSource.getId().replace('/', ';'); // target statement
4  itest().given(templateSource, "25.xml")
5         .checkEq(id, ";root;body@;folder;descriptor.txt");
6  String source = templateSource.getSource();
7  StringTemplateSource currentTemplateSource =
8    (StringTemplateSource) templateLoader.findTemplateSource(id);
9  ... }
```

(a) An inline test with an object that is serialized to an XML file.

```
1 <org.craftercms.core.util.template.impl.IdentifiableStringTemplateSource>
2  <id>/root/body@/folder/descriptor.txt</id>
3  <source>${body}</source>
4 </org.craftercms.core.util.template.impl.IdentifiableStringTemplateSource>
```

(b) The contents that are serialized to an XML file.

Figure 6: An inline test that saves an object in an XML file.

such branches, `Instrumenter` always wraps the parent node of the target statement in the AST in a `try` block. If the target statement's parent node is a constructor body whose first statement is a constructor call (e.g., `super()` or `this()`), ExLi excludes such constructor calls from the `try` block to avoid compilation error (super/this has to be the first statement).

④ **Execute unit tests and** ⑤ **collect values**. `Executor` runs unit tests on the instrumented CUT and the `Collector` stores the values of input and output variables that are observed during execution. ExLi is agnostic to the source of unit tests; they can be manually written or automatically generated. We currently use Randoop [62, 69] and EvoSuite [16] for automatic unit test generation; future work can investigate other test generators.

When the variable whose value is to be collected is of a primitive type, a wrapper type for a primitive type, a `String`, or an array of these types, `Collector` directly stores the collected values (which will be used on the constructed code for the inline test). Otherwise, `Collector` uses XStream [50] to serialize the values, which will be deserialized in future executions of the generated inline test. This support for complex non-primitive types was not available in our earlier Inline Test prototype and is added in this work.

Figure 6 shows an example inline test using XStream to support complex non-primitive types, from craftercms/core [81]. Line 3 is the target statement; it replaces "/" in `templateSource`'s `id` with ";". Line 4 is an inline test that ExLi generates. The variable being assigned, `templateSource`, is of a complex non-primitive type `IdentifiableStringTemplateSource`, whose value is serialized into "`25.xml`" (Figure 6b).

⑥ **Reduce by code coverage**. `CovReducer` reduces redundancy among collected sets of variable values that cover a target statement in the same way. We set JaCoCo [56], the code coverage tool used by ExLi, to instrument and collect all classes in the current project and dependency libraries, including the Java standard library. However, some classes in the Java standard library (e.g., `java.lang.String`) are loaded during JaCoCo initialization and are thus not instrumented. To avoid missing coverage information in such classes, especially for string-related and regex-related target statements, our implementation uses wrapper classes that we write for `java.lang.String` and `java.util.Matcher` so that the method calls of these classes can be instrumented. It is necessary to wrap `java.util.Matcher` because some `java.lang.String` methods that are used by our evaluation subjects depend on it.

Table 2: Projects used in our evaluation.

PID	project	SHA	LOC
P1	AquaticInformatics/aquarius-sdk-java	8f4edb9	21,634
P2	Asana/java-asana	52fef9b	5,572
P3	awslabs/amazon-sqs-java-extended-client-lib	58fed25	1,288
P4	Bernardo-MG/maven-site-fixer	60244c0	1,689
P5	Bernardo-MG/velocity-config-tool	26226f5	358
P6	craftercms/core	4d394a9	10,233
P7	CycloneDX/cyclonedx-core-java	d933705	6,011
P8	finos/messageml-utils	b4c75c6	21,765
P9	fleipold/jproc	b872abf	1,189
P10	hyperledger/fabric-sdk-java	da35400	33,677
P11	jenkinsci/email-ext-plugin	699277c	13,190
P12	jkuhnert/ognl	5c30e1e	18,190
P13	jscep/jscep	b20e944	6,310
P14	lamarios/sherdog-parser	aa6806a	1,546
P15	liquibase/liquibase-oracle	6ab7dea	7,170
P16	maxmind/geoip-api-java	1030316	11,526
P17	medcl/elasticsearch-analysis-pinyin	01dda56	2,169
P18	mojohaus/build-helper-maven-plugin	f1fac8c	2,424
P19	mojohaus/properties-maven-plugin	6cf7c2b	891
P20	mp911de/logstash-gelf	66debd8	13,130
P21	mpatric/mp3agic	407f7a9	9,907
P22	netceteragroup/trema-core	fa9f76d	3,285
P23	phax/ph-pdf-layout	f2d7b98	14,408
P24	ralscha/extclassgenerator	40ad147	6,271
P25	red6/pdfcompare	1259ef2	4,213
P26	restfb/restfb	35a34dd	42,022
P27	steveash/jopenfst	14c4a1d	5,180
P28	TNG/property-loader	928f414	1,860
P29	uwolfer/gerrit-rest-java-client	a0bf7cc	14,594
P30	visenze/visearch-sdk-java	0efcda3	7,643
P31	wmixvideo/nfe	1ccdba7	133,698
Total			423,043
Avg			13,646.5

⑦ **Construct inline tests**. `InlineTestConstructor` creates the inline tests at the AST level with the help of JavaParser [35].

⑧ **Reduce by mutation score**. `MutReducer` performs mutation analysis, using universalmutator [25] and Major [37], and test-suite reduction, using an existing implementation [73], to further reduce the generated inline tests. The test-suite reduction implementation [73] supports four algorithms: Greedy [87], GE, and GRE [9], as well as HGS [31]. We found that the four algorithms always result in the same number of inline tests in the reduced set (but different inline tests are selected) in our experiments, thus we set Greedy as the default algorithm.

5 EVALUATION

We answer the following research questions:
RQ1: How many inline tests does ExLi generate *before* reduction?
RQ2: How many inline tests does ExLi generate *after* reduction?
RQ3: How effective are the generated inline tests in terms of fault-detection capability, compared with unit tests?
RQ4: What is the runtime cost of ExLi?
Experimental environment. We run all experiments on a machine with Intel Core i7-11700K @ 3.60GHz (8 cores, 16 threads) CPU, 64 GB RAM, Ubuntu 20.04, Java 8, and Maven 3.8.6.

5.1 Curating an Evaluation Dataset

We start with a large set of projects from our recent work on learning to complete unit tests [58]. That prior work used different experimental requirements than this work to filter projects. So, we start from the original unfiltered set containing 1,535 Java projects

Table 3: Statistics about unit tests used in this paper.

PID	Dev				Randoop				EvoSuite			
	#tests	T[s]	L[%]	B[%]	#tests	T[s]	L[%]	B[%]	#tests	T[s]	L[%]	B[%]
P1	165	2.3	1	50	8,728	12.2	67	43	167	5.7	9	51
P2	67	1.9	24	79	1,476	7.7	89	36	1,040	10.8	90	41
P3	36	3.3	69	63	16,400	20.4	18	7	3	4.3	12	3
P4	73	3.5	88	84	2,098	7.7	24	8	62	4.0	38	44
P5	15	4.7	100	100	18,927	17.4	24	7	11	3.0	37	28
P6	63	7.6	52	47	3,741	10.2	40	23	396	10.6	23	19
P7	371	6.6	67	37	3,286	17.3	55	28	37	5.0	3	3
P8	1,170	5.3	89	81	2,886	12.5	44	27	1,221	34.0	55	43
P9	38	14.1	89	89	4,867	8.7	31	23	39	3.0	24	20
P10	430	215.2	12	9	8,697	18.2	25	20	77	38.0	1	0
P11	334	435.0	66	54	7,032	29.6	23	11	9	11.0	1	0
P12	939	10.8	70	61	494	7.7	29	17	1,905	8.3	44	35
P13	210	38.3	80	73	1,412	8.2	32	29	104	5.1	12	10
P14	12	24.1	68	52	1,212	220.4	73	43	70	14.5	49	28
P15	140	3.3	37	9	11,098	14.6	67	49	72	5.8	12	12
P16	11	2.5	22	5	10,869	11.8	17	4	18	2.9	11	0
P17	20	3.1	78	76	7,341	12.1	35	24	144	215.6	81	76
P18	55	4.2	14	7	19,884	20.6	31	23	45	3.6	11	8
P19	10	3.7	30	22	2,159	7.9	36	32	20	3.2	7	5
P20	269	9.2	78	70	11,467	12.7	53	30	81	5.2	4	8
P21	495	2.7	88	68	10,147	11.8	68	49	1,257	5.6	81	70
P22	60	3.5	72	61	4,332	8.9	44	31	98	4.6	20	16
P23	99	5.6	70	58	2,708	10.7	27	18	45	7.5	3	2
P24	99	3.4	78	70	763	5.3	24	11	176	5.9	49	41
P25	73	10.3	43	37	2,968	10.4	36	29	126	5.2	20	16
P26	1,273	21.0	59	75	7,100	23.6	68	30	442	16.1	12	12
P27	88	1.9	84	74	7,843	12.4	36	33	75	3.7	12	8
P28	105	2.8	85	91	3,421	6.5	74	54	113	3.6	78	68
P29	244	3.6	51	35	10,961	10.9	53	34	435	7.8	24	16
P30	151	3.9	75	68	3,496	134.1	73	51	15	3.1	2	0
P31	3,600	3.6	32	13	17,451	21.7	49	14	2,287	24.3	20	13
Total	10,715	861.0	N/A	N/A	215,264	734.3	N/A	N/A	10,590	481.2	N/A	N/A
Avg	345.6	27.8	57.2	50.6	6,944.0	23.7	44.0	27.0	341.6	15.5	27.3	22.5

that use Maven, have no compilation error, and have appropriate licenses. To simplify our experiments, we select the subset of 1,209 single-module projects. From these, we select the 128 actively-maintained projects that have commits after January 1, 2022, to facilitate future work on integrating the generated inline tests into these projects. Next, we filter out projects in which developer written unit tests fail (84 remain), in which JaCoCo fails (73 remain), and in which Randoop or EvoSuite fails (48 remain).

On these remaining 48 projects, we use ExLi to find target statements and generate inline tests. We filter out 6 projects that do not have the kinds of target statement that we look for [46]; one project where all target statements are not covered by any unit test; and one project for which ExLi does not generate any passing inline test. We also filter out 8 projects where ExLi's instrumentation clashes with the projects' instrumentation for other purposes, and one project where developer written tests take more than one hour.

We use the remaining 31 projects as our evaluation subjects. Table 2 shows the PIDs and names of these projects, the SHA that we use, and total lines of Java code.

Figure 7 shows statistics about the number of target statements in the 31 projects. ExLi initially finds 1,104 target statements (84 for regular expression, 745 for string manipulation, 241 for bit manipulation, and 34 for stream operations). Of these, 820 target statements are covered by at least one unit test (532 are covered by at least one developer written unit test, 491 are covered by at least one Randoop-generated unit test, and 613 are covered by at least one EvoSuite-generated unit test). After removing failing inline

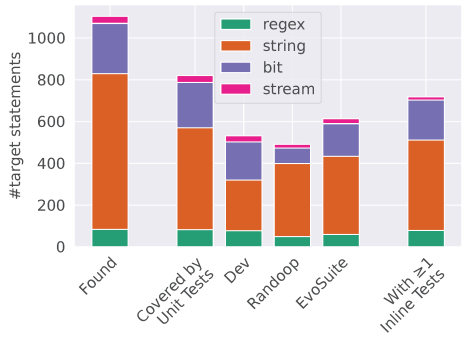


Figure 7: No. of target statements that we find for four kinds of APIs, covered by (all, developer written, Randoop, and EvoSuite) unit tests, and where ExLi generates inline tests.

tests and corresponding target statements, ExLi generates inline tests for 718 target statements (79 for regular expression, 432 for string manipulation, 192 for bit manipulation, and 15 for stream operations); we use them in the rest of our evaluation.

5.2 Extracting Inline Tests

First, we run Randoop and EvoSuite to obtain automatically generated unit tests for each project in our dataset. We run Randoop with a time limit of 10 minutes to generate unit tests for each project (as suggested by the Randoop user manual [84]); we set other options to default values. We run EvoSuite with a time limit of 120 seconds (as

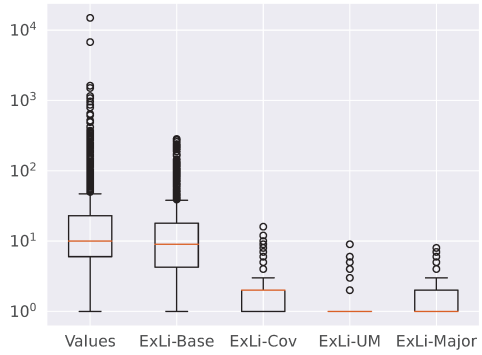
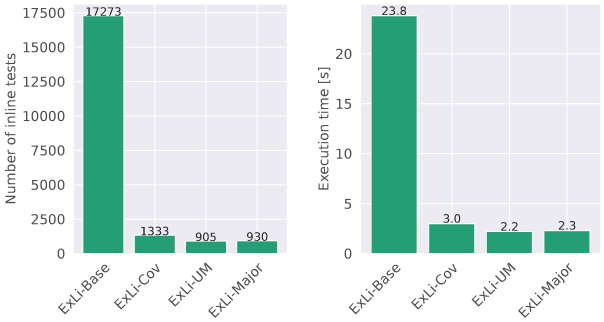


Figure 8: Distribution of inline tests per target statement.



(a) Number of inline tests. **(b) Execution time.**

Figure 9: Number and execution time of inline tests extracted by ExLi with different levels of reduction.

suggested by the configuration in the recent SBST competition [71]) for each class with at least one target statement.

Table 3 shows the statistics about the unit tests: number of test methods (**#tests**), test-running time (**T[s]**), line coverage (**L[%]**), and branch coverage (**B[%]**). Note that EvoSuite's line and branch coverage for some projects are low, because it is setup to only generate unit tests for classes with target statements which may be a small number of classes with few lines and branches.

Next, we run ExLi to extract inline tests from unit tests. We compile and run developer written, Randoop-generated, and EvoSuite-generated tests separately to allow flexible set up of different environments for each source of unit tests. We run developer written and Randoop-generated tests using Maven, but we run EvoSuite-generated tests with a JUnit runner. EvoSuite puts generated tests in customized runners that cause problems with Maven.

When performing coverage-based reduction, ExLi supports saving the code coverage information at the end of previous run and loading it at the beginning of the next run. For example, the extraction of inline tests from Randoop-generated unit tests could reuse coverage information collected from developer written unit tests. Similarly, extraction from EvoSuite-generated unit tests could reuse coverage information collected from developer written and Randoop-generated unit tests.

For each source of unit tests, we set an upper limit for the number of inline tests generated per target statement to 100, to avoid excessive disk space consumption in corner cases (especially when not performing reduction). With three sources of tests, our upper limit for inline tests generated per target statement is 300.

We compare the four sets of inline tests generated by ExLi as intermediate or final results (also see workflow in Figure 2): **ExLi-Base** without reduction, **ExLi-Cov** with only reduction by code coverage, **ExLi-UM** with coverage-then-mutants based reduction using universalmutator, and **ExLi-Major** with coverage-then-mutants based reduction using Major.

Figure 8 shows the distribution of generated inline tests per target statement. We also include the number of unique sets of variable values collected during execution of unit tests (denoted as **Values**), to show the number of inline tests that ExLi would generate without setting the 300 upper limit. The average number of inline tests per target statement for Values, ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major are 88.9, 24.1, 1.9, 1.3, and 1.3, respectively. The medians for Values, ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major are 10.0, 9.0, 2.0, 1.0, and 1.0, respectively.

The distribution of the number of inline tests per target statement for Values is long-tailed, which justifies our decision to set an upper limit of number of inline tests to prevent issues in corner cases. We observe that 95% of target statements are not affected by the limit of 300 inline tests per target statement. That is, the number of inline tests per target statement at the 95th percentile is 225.8.

> **Answer to RQ1**. ExLi could generate an average of 88.9 inline tests per target statement if recording all values during execution. Limiting to at most 300 per target statement and removing the failing ones, ExLi generates 24.1 inline tests before reduction per target statement on average.

Figure 9 shows the number of inline tests and their execution time (note that we did not include compilation time here). To evaluate the effectiveness of ExLi's reduction, we consider ExLi-Base as the baseline before reduction; it generates 17,273 inline tests that take 23.8 seconds to execute.

ExLi's coverage-based reduction (ExLi-Cov) reduces the number of inline tests to 1,333 (reduction rate: 92.3%) and the time to 3.0 seconds (reduction rate: 87.4%). Then, when performing mutation-based reduction using universalmutator (ExLi-UM), the number of inline tests is further reduced to 905 (cumulative reduction rate: 94.8%) and the time to 2.2 seconds (cumulative reduction rate: 90.8%). When using Major (ExLi-Major), the number of inline tests is further reduced to 930 (cumulative reduction rate: 94.6%) and the time to 2.3 seconds (cumulative reduction rate: 90.2%). The reduction rate of ExLi-UM and ExLi-Major with respect to ExLi-Cov is 32.1% and 30.2% in terms of number of inline tests, and 27.1% and 22.2% in terms of execution time, respectively.

Comparing ExLi-UM and ExLi-Major, we observe that using universalmutator achieves higher reduction than using Major. Our inspections showed that universalmutator generates more mutants than Major (3,784 vs. 2,388 mutants), and that mutants generated by Major tend to be generic (e.g., changing right hand side of an assignment to null) compared to the ones generated by universalmutator. Future work can explore improving the quality of the generated mutants, e.g., by using mutation operators that are designed for the

Table 4: Mutation analysis evaluation results. P15 is excluded because no mutant was generated for it.

PID	#stmts mutated	#mutants	Dev		Randoop		EvoSuite		ExLi-Base		ExLi-Cov		ExLi-UM		ExLi-Major	
			#tests	M[%]	#tests	M[%]	#tests	M[%]	#tests	M[%]	#tests	M[%]	#tests	M[%]	#tests	M[%]
P1	3	10	165	60.0	8,728	30.0	167	30.0	16	100.0	4	100.0	4	100.0	4	100.0
P2	244	494	67	8.1	1,476	7.3	1,040	10.7	6,287	100.0	486	100.0	313	100.0	319	99.8
P3	2	10	36	80.0	16,400	0.0	3	0.0	5	80.0	3	80.0	2	80.0	3	70.0
P4	2	18	73	83.3	2,098	0.0	62	0.0	10	83.3	3	83.3	2	83.3	2	72.2
P5	1	19	15	57.9	18,927	0.0	11	0.0	39	57.9	1	36.8	1	57.9	1	36.8
P6	13	44	63	86.4	3,741	18.2	396	100.0	555	77.3	26	75.0	14	77.3	12	59.1
P7	2	2	371	50.0	3,286	100.0	37	0.0	10	100.0	3	100.0	3	100.0	3	100.0
P8	11	47	1,170	83.0	2,886	10.6	1,221	48.9	98	89.4	15	76.6	11	89.4	11	85.1
P9	2	2	38	100.0	4,867	50.0	39	100.0	42	100.0	3	100.0	3	100.0	2	100.0
P10	16	75	430	77.3	8,697	13.3	77	2.7	455	82.7	33	82.7	22	82.7	23	80.0
P11	8	25	334	68.0	7,032	0.0	9	0.0	321	96.0	17	84.0	10	96.0	17	84.0
P12	130	1,434	939	57.7	494	8.0	1,905	33.6	2,313	69.6	244	67.0	156	69.6	176	67.4
P13	3	5	210	60.0	1,412	40.0	104	100.0	53	100.0	5	100.0	6	100.0	5	100.0
P14	2	5	12	60.0	1,212	0.0	70	0.0	21	100.0	4	100.0	2	100.0	3	100.0
P16	17	241	11	60.2	10,869	2.9	18	0.0	298	80.9	27	74.3	22	80.9	19	80.5
P17	6	42	20	64.3	7,341	19.0	144	28.6	72	76.2	10	61.9	9	76.2	9	57.1
P18	12	52	55	96.2	19,884	67.3	45	21.2	300	96.2	16	96.2	15	96.2	16	96.2
P19	7	34	10	73.5	2,159	0.0	20	55.9	292	76.5	19	67.6	9	76.5	7	67.6
P20	34	229	269	38.4	11,467	100.0	81	31.0	850	83.8	54	69.9	36	83.8	37	80.8
P21	32	497	495	85.3	10,147	47.9	1,257	88.3	889	81.7	57	53.3	38	81.7	40	78.1
P22	4	10	60	100.0	4,332	30.0	98	30.0	42	90.0	11	60.0	5	90.0	9	70.0
P23	5	42	99	23.8	2,708	59.5	45	38.1	249	100.0	8	81.0	7	100.0	8	100.0
P24	2	3	99	100.0	763	33.3	176	100.0	19	100.0	4	100.0	1	100.0	3	100.0
P25	5	25	73	92.0	2,968	0.0	126	100.0	55	92.0	11	92.0	6	92.0	5	92.0
P26	18	97	1,273	97.9	7,100	100.0	442	83.5	249	70.1	30	69.1	22	70.1	19	64.9
P27	3	31	88	22.6	7,843	0.0	75	19.4	11	90.3	4	51.6	3	90.3	3	90.3
P28	5	19	105	84.2	3,421	5.3	113	5.3	114	73.7	12	73.7	8	73.7	6	73.7
P29	10	66	244	42.4	10,961	47.0	435	100.0	487	93.9	18	92.4	14	93.9	16	89.4
P30	4	12	151	33.3	3,496	100.0	15	100.0	46	100.0	9	100.0	5	100.0	5	100.0
P31	46	194	3,600	90.7	17,451	53.1	2,287	87.6	1,016	96.9	78	84.0	59	96.9	51	92.3
Total	649	3,784	10,575	N/A	204,166	N/A	10,518	N/A	15,214	N/A	1,215	N/A	808	N/A	834	N/A
Avg	21.6	126.1	352.5	67.9	6,805.5	31.4	350.6	43.8	507.1	87.9	40.5	80.4	26.9	87.9	27.8	82.9

four kinds of target statements, to further improve the effectiveness of ExLi's mutation-based reduction.

Answer to RQ2. ExLi's coverage-then-mutants based reduction can effectively reduce all generated inline tests by 94.8% (with universalmutator) or 94.6% (with Major), resulting in an average of 1.3 inline tests per target statement.

5.3 Performing Mutation Analysis

Mutation testing is widely used to evaluate the quality of test suites [11, 66]. In this section, we perform mutation analysis using the mutants for the target statements generated by universalmutator. We reuse the same mutants that universalmutator generated during step ⑨ in Section 4 for reducing inline tests. We report results based on the 649 target statements that have non-stillborn mutants, and compare the mutation scores of inline tests generated by ExLi against unit tests. Note that universalmutator did not generate any mutant for any target statement in liquibase/liquibase-oracle (P15), so we excluded it from the mutation analysis evaluation.

Table 4 shows the number of tests and mutation scores of developer written, Randoop-generated, and EvoSuite-generated unit tests, and ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major inline tests. Note that the mutation scores of ExLi-UM and ExLi-Base are always the same by design, because during the mutation-based reduction, ExLi adds any inline test from ExLi-Base that kills a mutant that survives ExLi-Cov inline tests. The average mutation score of ExLi-Base is 87.9%, which is much higher than the mutation score of developer written (67.9%), Randoop-generated (31.4%), and

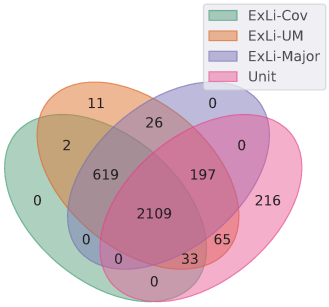


Figure 10: Sets of mutants killed by inline tests and unit tests.

EvoSuite-generated (43.8%) unit tests. These scores are computed only on the target statement. ExLi-Cov achieves 80.4%, slightly lower than ExLi-Base, but higher than the mutation score of unit tests. By performing additional mutation-based reduction, ExLi-UM fully recovers the mutation score to 87.9%, and ExLi-Major improves the mutation score to 82.9%. The difference between ExLi-UM and ExLi-Major is small, and suggests that the two mutation generation tools are quite similar (see also reports in prior work [25]).

Figure 10 shows a Venn diagram illustrating the overlap among the sets of mutants killed by all unit tests and inline tests from ExLi-Cov, ExLi-UM (which is the same as ExLi-Base), and ExLi-Major. All inline tests and unit tests kill 3,278 mutants in total. 2,404 mutants are killed by both inline tests and unit tests. The set of mutants killed by ExLi-Major inline tests is a subset of the set of

mutants killed by ExLi-UM inline tests, but the difference is small: ExLi-UM inline tests kills 111 or 3.8% more mutants than ExLi-Major inline tests. Compared with ExLi-UM inline tests, ExLi-Cov inline tests miss 299 mutants (9.1% of all killed mutants). Compared with unit tests, ExLi-UM inline tests miss 216 mutants (6.6% of all killed mutants). This is because unit tests can check global program state (e.g., fields) that is modified by the target statement, but inline tests currently cannot; future extensions of inline tests can address this limitation. But, ExLi-UM kills 658 more mutants than unit tests (20.1% of all killed mutants or 25.1% of mutants killed by unit tests).

We manually inspect surviving mutants that lead to loss of mutation scores when ExLi-Cov is compared with ExLi-Base. So far, we found two limitations of ExLi that lead to such intermediate losses. (1) There are multiple clauses in an `if` condition, but the mutation operator only modifies one of them. This limitation occurs because, unlike *pytest-inline*, ITest does not yet support testing individual clauses in a condition. This limitation will go away as ITest matures. (2) Multiple sets of values can kill a mutant but they all cover the target statement and its context in the same way as a chosen set of values that cannot kill the mutant. This is a limitation of reduction by coverage as we discussed in Section 3.

Observe from Figure 10 that inline tests and unit tests are complementary in terms of their fault-detection capability on the target statements. So, inline tests can enhance the fault-detection capability of the unit test suites from which they are extracted. To understand why some mutants on target statements can be killed by inline tests but not by the unit tests, we manually inspected 63 randomly sampled mutants from the 658. We found two reasons: (1) unit tests lack good assertions to kill the mutants, i.e., the mutant could be killed if we add assertions to the unit tests (77.8% of cases); (2) the mutant does not change program state that propagates to unit tests, i.e., it only changes local variables or control flow but not the return value or global variables, but inline tests' "local" assertions kill such mutants (22.2% of the cases).

Answer to RQ3. Inline tests complement the fault-detection capability of unit tests on the target statements. ExLi-UM and ExLi-Major generate inline tests with average mutation scores of 87.9% and 82.9%, respectively, which are higher than the mutation scores on the target statements of unit tests written by developers (67.9%), and those generated by Randoop (31.4%) and EvoSuite (43.8%).

5.4 Measuring ExLi's Runtime Cost

Generating inline tests with ExLi-UM and ExLi-Major takes, on average across projects, 1,053.7s and 949.9s, respectively. (We omit compilation time of the mutants; it is an offline process and is currently slow because we recompile per mutant. Future work can optimize this process by compiling in parallel or by using incremental compilation.) The breakdown of the average runtime is: 67.0s for running unit tests, 598.2s for recording variable values, coverage-based reduction, and generating inline tests, and 388.5s (universalmutator) or 284.7s (Major) for mutation-based reduction.

We are very encouraged by these early results on runtime costs, especially when compared with our estimated amount of time that it would take developers to write all 905–930 inline tests that ExLi generates. Our prior user study [46] showed that participants spent around 6.3 minutes (378s) to understand and write inline tests for each target statement in Python. Assume that the times to understand target statements and write inline tests is uniformly distributed and are the same for Java and Python. Then, on average, participants would have needed 271,404s to write inline tests for all 718 target statements that we use.

Answer to RQ4. Running ExLi-UM/ExLi-Major takes 949.9s to 1,053.7s on average per project, excluding mutant compilation times. Our estimates, based on our prior user study, suggests that these average times is evidence that ExLi can reduce manual effort for writing inline tests.

6 DISCUSSION

Usage modes. The inline tests that ExLi generates can help find regressions in future versions of the code, and there is need for future work on co-evolving inline tests with code. But, ExLi can also help find bugs in the current program versions if developers inspect the generated tests. By inspecting inline tests in prior work [46], we found two bugs that have now been fixed by the developers.

Limitations. (1) ExLi uses coverage of the target statement and its context for initially reducing the set of inline tests. Flaky tests [3, 26, 41, 51, 64, 75] can cause coverage to fluctuate. We do not control for flaky tests in the unit tests that ExLi uses. (2) Extracted inline tests may be flaky and fail if the expected output in the oracles that are generated depend on data that may change, e.g., current date or device configuration. (3) When potential inputs cause the target statement or its context to throw an exception, ExLi does not use such values to construct inline tests because ITest [46] does not yet support using expected exceptions as test oracles. (4) We do not evaluate the extracted inline tests with developers of the open-source projects that we evaluate. But, we have initial confidence from our prior user study, which showed that participants find inline tests useful. We plan to communicate more with open-source developers in the future, especially as ITest [46] matures.

Threats to validity. Our code to instrument target statements, collect coverage rates, and perform reduction could contain bugs. To mitigate this threat, at least two co-authors review the code, and multiple authors inspect the results. Our findings could be limited to projects that we evaluate and their unit tests. To mitigate this threat, we used open-source projects with various characteristics and used automatically generated unit tests. The ideas in ExLi are general but our results may not generalize to other programming languages. We plan to use our *pytest-inline* tool [47] as a basis for a tool that extracts inline tests from Python unit tests.

Future work. We plan to (1) support generation of inline tests for target statements in other program locations than the three that ExLi supports (`if` conditions, assignment statements, and declarations); (2) support other kinds of target statements than the four that our inline testing research so far considered; (3) generate inline tests for other programming languages; and (4) investigate regression test selection (RTS) for inline tests, borrowing from our work on RTS for unit tests [19, 20, 23, 27, 43, 44, 48, 77, 90].

7 RELATED WORK

Single-statement bugs and inline tests. Inline tests are partly motivated by recent work [38, 39, 42, 68] showing that many bugs are caused by faults in single statements, and that unit tests miss such bugs. We used inline tests to find single-statement bugs [46], and ExLi could help find more in the future. The ManySStuBs4J [39] dataset contains single-statement bugs that are curated by statically analyzing open-source Java projects and their version histories. As the ManySStuBs4J dataset evolves to capture more recent versions of those projects, it can be a benchmark for evaluating the bug-detection capability of inline tests. We do not use ManySStuBs4J because (1) the filtering process that was followed to curate the dataset resulted in many false positives during our initial search for target statements; (2) the commits used in the dataset are from before 2019, so we had trouble running the unit tests in some projects.

"ppx inline tests" [79] and the inline tests in this paper [46] share a name and the characteristic that they are co-located with code. But, "ppx inline tests" check the correctness of functions instead of single statements. Xiong et al. [86] propose inner oracles: assertions declared in unit tests to check internal states. Inline tests allow specifying both oracles and test inputs to check single statements.

Automatic test generation. Automatically generation of tests is a popular research topic and many test generation techniques have been proposed for Java [2, 8, 16, 18, 22, 59, 62, 72]. But, ExLi is the first automatic generation technique for inline tests. Elbaum et al.'s technique [14] extracts unit tests from system tests. ExLi is similar in spirit—it also extracts lower granularity tests from higher granularity tests—but differs in the granularity levels that it targets. Also, unlike Elbaum et al.'s technique, ExLi further reduces generated inline tests.

Test suite reduction/minimization. Yoo and Harman [87] present a survey on test suite minimization. Zhang et al. [89] study the effectiveness of test suite reduction techniques. Test-suite reduction techniques include those that use (1) Greedy algorithms [10, 80], (2) heuristics [9, 31], and (3) integer programming [32, 45]. We use a recent implementation of the Greedy algorithm [73] to further reduce inline tests that ExLi generates.

Shi et al. [74] found that techniques based on statement coverage reduce test-suite sizes by 62.9% but lose 20.5% in killed mutants. Conversely, techniques based on killed mutants have no loss in killed mutants but have test-suites that are 10.9 percentage points larger than those produced by coverage-based minimization, on average. Shi et al.'s study gives more confidence in preservation of fault-detection capability in ExLi reduction based on killed mutants.

Noemmer and Haas [60] recently compare test suite minimization techniques on open-source projects and find that, on average, test suites reduce by 70% while losing 12.5% of the fault-detection capability. Our results show that traditional test suite minimization reduces generated inline tests by 32.1% and ExLi preserves fault-detection capability.

Using coverage as feedback in automated testing. Coverage was used as feedback for test generation [16, 17, 54] and test-suite reduction [30, 40, 53]. We use a combined change of coverage rate of target statements and their enclosing basic blocks.

Assertion/Invariant generation. Program assertions/invariants are useful for checking the correctness of program states. Inline tests are similar to assert statements: both are co-located with program statements and they can be turned off in production. But, inline tests are different: they allow to provide arbitrary inputs, expected outputs, and oracles for testing statements. Further, assert statements only run if they are in code covered by unit tests, but inline tests run in a different context even if the target statement is not covered by unit tests. Lastly, existing inline testing frameworks provide features that are typically not supported in assert statements: parameterized tests, repeating test runs (helpful to see if inline tests are flaky), grouping tests, and running tests in parallel.

There have been many techniques for automatically generating assertions and invariants, including those that (1) infer invariants from runtime information [5, 12, 15]; (2) generate assertions from comments and documentation [4, 24, 55]; and (3) learn assertions from code [13, 28, 58, 85, 88]. ExLi is most similar to approaches in the first category, as it extracts inline tests from runtime information. But, ExLi additionally (1) uses the collected information to construct inputs, expected outputs, and oracles for the generated inline tests; and (2) reduces the set of generated inline tests.

Mutation testing. Mutation testing is a technique for evaluating the effectiveness of test suites [29, 63, 67]. Popular mutant generators for Java include universalmutator [25], Major [82], PIT [83], and MuJava [52]. ExLi uses the first two tools which perform mutation on the source code level, thus allowing filtering mutants for the target statements. But, future work can explore integrating other mutation tools with ExLi.

8 CONCLUSION

In this paper, we presented ExLi, a technique for automatically generating inline tests with coverage-then-mutants based test reduction. The coverage-based reduction is based on context-aware coverage feedback, and the mutation-based reduction is based on killed mutants. We evaluate ExLi on 31 open-source Java projects and find that ExLi generates between 905 (when using universalmutator to reduce tests) and 930 (when using Major to reduce tests) inline tests for 718 target statements. ExLi reduces initially generated inline tests by more than 94%. ExLi enables developers to enhance the fault-detection capability of their test suites by easily obtaining and adding inline tests.

ACKNOWLEDGMENTS

We thank Fred Schneider, August Shi, Ayaka Yorihiro, Zhiqiang Zang, and the anonymous reviewers for their comments and feedback. Some of this research was sponsored by the Army Research Office and was accomplished under Cooperative Agreement Number W911NF-19-2-0333. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. This work is also partially supported by a Google Faculty Research Award and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2019277, CCF-2045596, CCF-2107291, and CCF-2217696.

REFERENCES

[1] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *ICSE-SEIP*. 263–272.

[2] Andrea Arcuri and Gordon Fraser. 2016. Java Enterprise Edition support in search-based JUnit test generation. In *SSBSE*. 3–17.

[3] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*. 433–444.

[4] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA*. 242–253.

[5] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *ISSTA*. 169–180.

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems* 30, 19 (2002).

[7] Xia Cai and Michael R Lyu. 2005. The effect of code coverage on fault detection under different testing profiles. In *A-MOST*. 1–7.

[8] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded exhaustive test-input generation on GPUs. *PACMPL* 1, OOPSLA (2017), 1–25.

[9] Tsong Yueh Chen and Man Fai Lau. 1970. 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. 1998. A simulation study on some heuristics for test suite reduction. *IST* 40, 13 (1998), 777–787.

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

[12] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*. 281–290.

[13] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. TOGA: A neural method for test oracle generation. In *ICSE*. 2130–2141.

[14] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *FSE*. 253–264.

[15] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *SCP* 69, 1-3 (2007), 35–45.

[16] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: Automatic test suite generation for object-oriented software. In *FSE*. 416–419.

[17] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *TSE* 39, 2 (2012), 276–291.

[18] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. 2013. JST: An automatic test generation tool for industrial Java applications with strings. *ICSE*, 992–1001.

[19] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *ICSE-Demo*. 713–716.

[20] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*. 211–222.

[21] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *ISSTA*. 302–313.

[22] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *ICSE*. 225–234.

[23] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An empirical evaluation and comparison of manual and automated test selection. In *ASE*. 361–372.

[24] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA*. 213–224.

[25] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In *ICSE-Demo*. 25–28.

[26] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *ICST*. 993–997.

[27] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*. 112–122.

[28] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. 2017. Assertion generation through active learning. In *ICFEM*. 174–191.

[29] Farah Hariri, August Shi, Owolabi Legunsen, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. 2018. Approximate transformations as mutation operators. In *ICST*. 285–296.

[30] Preethi Harris and Raju Nedunchezhian. 2015. A greedy approach for coverage-based test suite reduction. *IAJIT* 12 (2015), 17–23.

[31] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A methodology for controlling the size of a test suite. *TOSEM* 2, 3 (1993), 270–285.

[32] Joshua Hartmann and Dave J Robson. 1989. Revalidation during the software maintenance phase. In *ICSM*. 70–80.

[33] Inline Testing Team 2023. pytest-inline GitHub Page. https://github.com/pytest-dev/pytest-inline.

[34] Inline Testing Team 2023. pytest-inline on PyPi. https://pypi.org/project/pytest-inline.

[35] JavaParser Team. 2023. JavaParser. https://github.com/javaparser/javaparser.

[36] Dennis Jeffrey and Neelam Gupta. 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. *TSE* 33, 2 (2007), 108–123.

[37] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA-Demo*. 433–436.

[38] Arthur V Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. 2021. PySStuBs: Characterizing single-statement bugs in popular open-source Python projects. In *MSR*. 520–524.

[39] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? The ManySStuBs4J dataset. In *MSR*. 573–577.

[40] Saif Ur Rehman Khan, Sai Peck Lee, Reza Meimandi Parizi, and Manzoor Elahi. 2014. A code coverage-based test suite reduction and prioritization framework. *WICT*, 229–234.

[41] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. *ICST*, 312–322.

[42] Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. 2021. How effective is continuous integration in indicating single-statement bugs?. In *MSR*. 500–504.

[43] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *FSE*. 583–594.

[44] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE-Demo*. 949–954.

[45] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *ICSE*. 1039–1049.

[46] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline tests. In *ASE*. 1–13.

[47] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. pytest-inline: An inline testing tool for Python. In *ICSE-Demo*. to appear.

[48] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *ISSTA*. To appear.

[49] LogstashGelf 2022. Mp911de Logstash Gelf. https://github.com/mp911de/logstash-gelf.

[50] LogstashGelf 2022. XStream developer. https://x-stream.github.io/index.html.

[51] Qingzhou Luo, Lamyaa Eloussi, Farah Hariri, and Darko Marinov. 2014. An empirical analysis of flaky tests. *FSE*, 643–653.

[52] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An automated class mutation system. *STVR* 15, 2 (2005), 97–133.

[53] Alessandro Marchetto, Giuseppe Scanniello, and Angelo Susi. 2019. Combining code and requirements coverage with execution cost for test suite reduction. *TSE* 45 (2019), 363–390.

[54] Phil McMinn. 2004. Search-based software test data generation: A survey. *STVR* 14, 2 (2004), 105–156.

[55] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise oracles from structured natural language specifications. In *ICSE*. 188–199.

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

[57] Networknt 2022. JSON Schema Validator. https://github.com/networknt/json-schema-validator.

[58] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *ICSE*. 1–12.

[59] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying execution of imperative generators and declarative specifications. *PACMPL* 4, OOPSLA (2020).

[60] Raphael Noemmer and Roman Haas. 2019. An evaluation of test suite minimization techniques. In *SWQD*. 51–66.

[61] Oracle. 2022. Chapter 4. The class file format. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.3.

[62] Carlos Pacheco and Michael D Ernst. 2007. Randoop: Feedback-directed random testing for Java. In *OOPSLA*. 815–816.

[63] Mike Papadakis, Marinos Kintis, Jie M. Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter six - mutation testing advances: An analysis and survey. *ADV Computers* 112 (2019), 275–378.

[64] Owain Parry, Gregory M. Kapfhammer, Michael C Hilton, and Phil McMinn. 2022. A survey of flaky tests. *TOSEM* 31 (2022), 17:1–17:74.

[65] PePy Team. 2023. pytest-inline downloads. https://pepy.tech/project/pytest-inline.
[66] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical mutation testing at scale: A view from Google. *TSE* 48, 10 (2021), 3900–3912.
[67] Goran R. Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices? *ICSE*, 910–921.
[68] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: Mining single statement bugs at massive scale. In *MSR*. 418–422.
[69] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE*. 23–32.
[70] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*. 34–43.
[71] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. 2022. EvoSuite at the SBST 2022 tool competition. In *SBST*. 33–34.
[72] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In *ASE*. 201–211.
[73] August Shi. 2023. Collection of scripts to conduct test-suite reduction. https://github.com/august782/testsuite-reduction.
[74] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *FSE*. 246–256.
[75] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. 80–90.
[76] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*. 84–94.
[77] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. In *OOPSLA*. 187:1–187:29.
[78] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *FSE*. 237–247.
[79] Jane Street. 2023. Inline tests. https://github.com/janestreet/ppx_inline_test.
[80] Sriraman Tallam and Neelam Gupta. 2005. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes* 31, 1 (2005), 35–42.
[81] Crafter Core Team. 2023. Crafter CMS Core. https://github.com/craftercms/core.
[82] Major Team. 2023. Major mutation framework. https://mutation-testing.org/.
[83] Pitest Team. 2022. PIT - Mutation Testing for Java. https://pitest.org/.
[84] Randoop Team. 2023. Randoop Manual. https://randoop.github.io/randoop/manual/.
[85] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE*. 1398–1409.
[86] Yingfei Xiong, Dan Hao, Lu Zhang, Tao Zhu, Muyao Zhu, and Tian Lan. 2015. Inner oracles: Input-specific assertions on internal states. In *FSE*. 902–905.
[87] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *STVR* 22, 2 (2012), 67–120.
[88] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *ICSE*. 163–174.
[89] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An empirical study of JUnit test-suite reduction. In *ISSRE*. 170–179.
[90] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*. 430–441.