Block Tests

KEVIN GUAN, Cornell University, USA
PENGYUE JIANG, Cornell University, USA
MILOS GLIGORIC, University of Texas at Austin, USA
OWOLABI LEGUNSEN, Cornell University, USA

Inline tests validate single program statements and were shown to find single-statement bugs or kill mutants that unit tests miss. Inline tests complement unit tests by enabling testing at a finer program granularity level than methods. So, inline tests can more easily find faults in target statements that unit tests do not reach, or where errors do not propagate to unit tests' oracles. But, the limitation to single statements means inline tests cannot validate data or control flow across code fragments—sequences of multiple statements in a method.

We motivate the need for testing arbitrary fragments and propose *block tests*, which generalize inline tests and validate code fragments. To motivate, we discuss six software testing needs (*e.g.*, due to increasing usage of lambdas in imperative code) for which unit tests are too coarse grained and inline tests are too fine grained. To bridge this gap, we propose syntax and semantics for specifying inputs, expected outputs, and scope of block tests. We also implement a block-test development kit (BDK) for writing and running block tests in Java.

We evaluate block tests and BDK in two ways. First, we write 1,012 block tests for 346 fragments in 146 open-source projects. Developer written unit tests do not cover 58.7% of these fragments, and automated unit-test generation does not reach 46% of them even after 30.8 CPU days. But, each block test takes us 2.2 minutes to write and 0.9 seconds to run on average. Second, we use mutation testing to evaluate the fault-finding effectiveness of block tests in fragments that unit tests cover. Block tests kill 4,418 of 9,554 mutants that survived unit tests. These results provide initial but strong evidence on block tests' feasibility and utility. We outline an agenda for future research on block testing.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: software testing

ACM Reference Format:
Kevin Guan, Pengyue Jiang, Milos Gligoric, and Owolabi Legunsen. 2026. Block Tests. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 166 (April 2026), 29 pages. https://doi.org/10.1145/3798274

1 Introduction

For over three decades, there were only three levels of granularity in the software testing pyramid [2, 5]: (i) unit tests validate units (*e.g.*, functions or methods) [6, 95]; (ii) integration tests validate interactions among units [24, 64, 83, 103]; and (iii) system tests validate end-to-end functionality [104, 109]. Tests at these three granularity levels are written and maintained separately from the code under test. So, an artificial boundary existed between code and tests.

Recently, Liu et al. [67] argued that unit tests, the lowest of these levels, are too coarse grained for many testing needs. For example, many single-statement bugs [47, 50, 91] are missed by unit tests [58]. So, Liu et al. proposed *inline tests* to validate single statements and expressions, conducted a user study, and used inline tests to find bugs that unit tests miss [67]. Liu et al. also developed

Authors' Contact Information: Kevin Guan, Cornell University, Ithaca, New York, USA, kzg5@cornell.edu; Pengyue Jiang, Cornell University, Ithaca, New York, USA, pj257@cornell.edu; Milos Gligoric, University of Texas at Austin, Austin, Texas, USA, gligoric@utexas.edu; Owolabi Legunsen, Cornell University, Ithaca, New York, USA, legunsen@cornell.edu.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART166
https://doi.org/10.1145/3798274

inline-test generation techniques and tools [44, 66, 68], showed that inline tests kill many mutants that survive unit tests [44, 66], and developed an inline testing tool [38, 69] that is now an official plugin for pytest, the most popular Python testing framework.

These results on inline tests show that testing at a finer granularity level than unit tests is feasible and complementary. Also, there is now evidence that the decades-old boundary between code and tests can be removed, since inline tests are co-located with the target statements that they validate.

```
1  for ( String line : header.split( LINE_SEPARATOR + "" ) ) {
2    if (...) continue; // more code...
3    if (...) { /* more code... */ } else {
4      String s = getCommentLinePrefix().trim();
5      blocktest().given(line,"-").given(s,"--").checkEq(line,"");
6      if ( line.startsWith( s ) ) {
7        if ( line.length() <= s.length() ) { line = ""; }
8      } else {
9        line = line.substring( s.length() );
10     } } }
```

Fig. 1. A bug [65] that unit tests miss and inline tests cannot find.

We propose *block tests*, a new test granularity level for validating arbitrary code fragments—sequences of multiple statements in methods. In a sense, block tests generalize inline tests to code fragments. To see why this generalization is needed, consider Figure 1 from the `mojohaus/license-maven-plugin` project. There, the fragment on lines 6—10 contains a bug: line 9 throws a runtime exception if `line` is shorter than `s`. Unit tests missed this bug, which is deeply nested in conditional statements. In fact, developers commented that it was hard to write a unit test to reproduce the bug, which was fixed without adding a unit test eight months after it was reported [65]. Inline tests cannot detect this bug: the fault is that the statements on lines 7 and 9 are swapped.

The block test on line 5 in Figure 1 detects this bug. There, the `blocktest()` expression declares that the statement is a block test. The two `given` expressions assign values to `line` and `s` that should be used as inputs when testing the fragment. Finally, the scope of this block test is the default one: it starts from the statement after the `blocktest()` and ends after the last statement in the block, *i.e.*, on line 10. The `checkEq` expression asserts that, after executing the fragment on lines 6–10 on these inputs, the value of `line` should be the empty string; otherwise, the block test fails. When executing the block test on line 5, the fragment on lines 6—10 is extracted and run in isolation, allowing the fragment to be tested even in the absence of unit-test coverage. During execution, the condition on line 6 evaluates to false because `line` ("-") does not start with `s` ("--"). So, the else branch on line 9 is executed. Because `line` is shorter than `s`, this execution results in a runtime exception, and the block test fails.

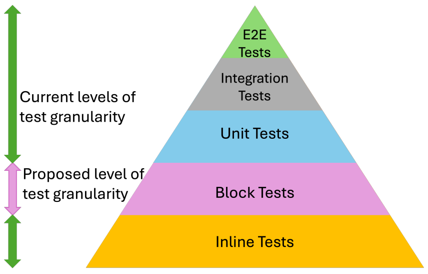
E2E Tests
Integration Tests
Unit Tests
Block Tests
Inline Tests
Current levels of test granularity
Proposed level of test granularity

Fig. 2. Testing pyramid.

Block tests bridge the gap between unit tests and inline tests. In Figure 2, block tests are between unit and inline tests in our proposed updated testing pyramid. Block tests can also address six testing needs that unit tests are too coarse grained for, but inline tests are too fine grained for:

(1) *Validating code fragments that unit tests do not reach.* Unit tests often do not reach code fragments that are deeply nested; or which require setting up complex inputs. Block tests sidestep these reachability issues by being co-located with the target fragment being validated.
(2) *Testing in recent programming language (PL) constructs.* Lambdas [3, 73], futures and promises [13, 106], Stream APIs [105], etc., are hard to validate independently with unit tests. Yet, such constructs often contain complex fragments. Block tests enable *in-situ* testing of code in these constructs.

(3) *Extending the reach of dynamic analyses.* Runtime verification (RV) [27–30, 43, 53, 54, 60, 62, 80, 112], dynamic data race prediction [36, 37, 55, 93], etc., help find software bugs in fragments. But, they are limited by unit tests' coverage. Block tests could extend the reach of these analyses.
(4) *Realizing micro execution.* Micro execution [23] is "the ability to execute any code fragment without a user-provided test driver or input data", but it works only for test input generation from binaries and requires a custom virtual machine (VM). Block tests can be seen as a form of micro execution, but for source code, and developers provide test inputs, expected outputs, oracles, and scope. Also, block tests are not limited to test generation and do not require a custom VM.
(5) *Enhancing software testing and analysis techniques.* We give two examples. The small scope of target fragments could make them excellent targets for scalable symbolic-execution-based block-test generation. Regression test selection (RTS) [19, 61, 111, 114] could be sped up by not re-running affected unit tests if changes only impact code in fragments that block tests check.
(6) *Improving the testing of isolated code snippets.* Several recent works aim to learn missing portions of isolated code snippets, *e.g.*, from StackOverflow, to run them. LExecutor [100] and Incompleter [34] are examples. Block tests can validate these snippets, which may contain security vulnerabilities [77] and where LExecutor and Incompleter often only achieve partial coverage.

This paper lays the foundations for block testing by proposing syntax and semantics for block tests in Java. We also implement our syntax and semantics in a block-test development kit (BDK) whose API can be used in any Java program. Using BDK, developers can specify the inputs, expected outputs, and scope of target fragments in block tests.

Like inline tests, block tests (i) are co-located with target fragments—BDK automates the extraction and running of each block test in an isolated environment; (ii) only run during testing and never during deployment; and (iii) work even if there are no unit tests and require only that target fragments compile (the whole project or class need not compile). BDK natively supports mocking [71, 101, 116] to enable easier setup of inputs. BDK allows writing *data flow oracles* like `checkEq` on line 5 in Figure 1 to check if data at a point in a target fragment are as expected. Users can also write *control flow oracles* to check if control reaches expected part(s) of a target fragment.

We evaluate block tests and BDK in two ways. First, we find 400 deeply nested fragments, 200 of which are multi-statement lambdas. Even though we have no domain knowledge about any of these fragments, we successfully write block tests for 346 fragments that contain 3,530 lines of code from 146 GitHub Java projects. Then, we check the coverage of these fragments by developer written unit tests or those that we generate by running Randoop [84, 85] and EvoSuite [15] for 30.8 CPU days. The average statement coverage achieved by all unit tests for 167 of these 346 fragments is 74.1%. The other 179 fragments are not covered by unit tests. On average, our block tests take only 2.2 minutes to write and 0.9 seconds to run. Also, our block tests achieve 93.2% statement coverage in these 346 fragments.

Second, we use mutation analysis to compare the fault-detection abilities of unit and block tests. In 2,528 statements that unit and block tests cover, the local scope of a block test makes it easier for errors to propagate to block tests' oracles, helping block tests kill 4,418 of 9,554 mutants that survive unit tests. Overall, block tests kill 46.2 percentage points more mutants than unit tests by covering statements that unit tests do not. So, block testing is feasible and finds faults that unit tests miss. Our evaluation uses deeply nested blocks (multi-statement sequences in imperative code) and lambdas, showing that block tests can meet testing needs (1) and (2) above, respectively.

This paper makes the following contributions:

⋆ We propose block tests, a new granularity level for software testing that allows validating arbitrary fragments—sequences of multiple statements within methods.

★ We propose BDK to support writing and running block tests in Java. BDK implements our proposed syntax and semantics of block tests.
★ The 1,012 block tests that we write as part of our evaluation are the first, and they can enable further research pending future work on automatically generating block tests.

BDK, our dataset, and all our artifacts are at https://github.com/SoftEngResearch/blocktests.

2 Examples: Features of Block Tests Inspired by our Formative Study

To motivate our design choices, we present four examples to illustrate features of block tests that we discover during our formative study and how those features work. In the formative study, we randomly select 27 GitHub Java projects, manually find fragments that are not reached by unit tests or in which developers fixed bugs, and then we attempt to write block tests for these fragments. No block test from our formative study is used in our evaluation (§4).

```
1  int sx = compute();
2  blocktest().given(input, new BinarySource(/*width*/5,/*height*/5, new byte[]{0,1,2,3})).given(sx, 5)
3    .checkFlow(IfStmt().ElseIf().Then());
4  if (sx < 0) {
5    sx += input.getWidth();
6  } else if (sx > input.getWidth()) { // fault: should be >= instead of >
7    sx %= input.getWidth();
8  }
9  blocktest().given(input, new BinarySource(5, 5, new byte[]{0, 1, 2, 3})).given(sy, 5)
10   .checkFlow(IfStmt().ElseIf().Then());
11 blocktest().given(input, new BinarySource(5, 5, new byte[]{0, 1, 2, 3})).given(sy, 0)
12   .checkFlow(IfStmt().ElseIf().ElseIf().Then());
13 if (sy < 0) {
14   sy += input.getHeight();
15 } else if (sy > input.getHeight()) { // fault: should be >= instead of >
16   sy %= input.getHeight();
17 } else if (sy == 0) {
18   System.out.println("else-if");
19 } else {
20   System.out.println("else");
21 }
```

Fig. 3. Two faults revealed by block tests' control-flow oracles in `Harium/keel` [33].

Example 1 (control-flow oracles and type resolution from context). BDK supports assertions on how control flows in the target fragment, and type resolution on input variables. Consider the fragment in Figure 3, which contains two bugs in a class from the `Harium/keel` project for image transformation. In Figure 3, when `sx` or `sy` is greater than or equal to the width or height of the input image, respectively, the values of these variables should be adjusted using the modulo of the width or height of the input image. But, lines 6 and 15 use > instead of >=, so `sx` and `sy` are adjusted incorrectly when `sx` or `sy` is equal to the width or height, respectively. Unit tests miss these bugs and inline tests are not applicable, but block tests detect them easily. The block test on lines 2–3 uses `checkFlow`, a control-flow oracle, to validate the fragment on lines 4–7. That block test has a default scope that ends on line 7, which is the final use of `sx`. Before executing the fragment, the test uses `given` expressions to assign `new BinarySource(5, 5, new byte[]{0, 1, 2, 3})` to `input` and 5 to `sx`. Observe that the `int` type of `sx` is not specified in the `given()` expression, so BDK resolves that type from the context of the fragment. The `checkFlow` oracle asserts that the `else-if` block on line 7 must be reached for the given inputs. Here, `IfStmt().ElseIf().Then()` denotes that execution reaches the `if` statement, then the `else-if`, and then the block in the `then` branch of that `else-if`. The block test on lines 2–3 detects the fault on line 6: `sx` is equal to `input.getWidth()` (both are 5) on line 4, so control never reaches the `else-if` on line 7, causing the `checkFlow` assertion to fail. The block test on lines 9–10 detects the fault on line 15 in a similar manner. Lines 11–12 show another block test, but with a different control-flow oracle. The control-flow oracle on line 12 checks if line 18, which performs no computation, is reached. Data-flow oracles cannot perform such checks, but they can be used together with control-flow oracles in the same block test.

```
1 // @blocktest().given(jsonish, "hi<s\\cript>test</script>").given(i, 2).checkEq(lc1, 's').checkEq(lc2, 'c').checkEq(lc3, 'r');
2 blocktest().given(jsonish, "hi<s\\cript>test</script>").given(i, 2).checkEq("lc1", 's')
3   .checkEq("lc2", 'c').checkEq("lc3", 'r');
4 int la = i + 1;
5 int c1AndDelta = unescapedChar(jsonish, la); char c1 = (char) c1AndDelta;
6 la += c1AndDelta >>> 16;
7 long c2AndDelta = unescapedChar(jsonish, la); char c2 = (char) c2AndDelta;
8 la += c2AndDelta >>> 16;
9 long c3AndEnd = unescapedChar(jsonish, la); char c3 = (char) c3AndEnd;
10 char lc1 = (char) (c1 | 32); char lc2 = (char) (c2 | 32); char lc3 = (char) (c3 | 32);
11 blocktest().given(jsonish, "hi<s\\cript>test</script>").given(i, 2).checkEq(lc1, 's')
12   .checkEq(lc2, 'c').checkEq(lc3, 'r').start(FIRST_BLOCK);
```

Fig. 4. Illustrating three ways of referencing undeclared or un-initialized variables in block tests.

Example 2 (referencing undeclared or un-initialized variables). Many block tests would not compile without support for allowing them to reference undeclared or un-initialized target-fragment variables. The reason is that Java does not allow local variables to be referenced before they are declared and initialized. For example, the block test on lines 2–3 in Figure 4 is for the target fragment on lines 4 to 10; it uses string `jsonish` and integer `i` as input and checks that output values in `lc1`, `lc2`, and `lc3` are as expected. Placing a block test before line 4 that directly checks the values in these output variables (as we did in all prior examples) will cause compilation to fail. The reason is that these variables are only declared and assigned in the last statement of the target fragment (on line 10). BDK supports three features to address this issue: users can (i) write block tests in code comments (*e.g.*, as shown on line 1); (ii) refer to undeclared or un-initialized variables as strings (*e.g.*, `"lc1"` on line 2); and (iii) declare block tests at the end of the target fragment, *e.g.*, the `start()` expression on line 12 means that the target fragment is the first basic block *before* the block test. In Figure 4, all three block tests validate the target fragment on lines 4–10 for illustration purposes only. Each test assigns the string value `"hi<s\\cript>test</script>"` to `jsonish` and value `2` to `i`, and then checks that after executing the fragment, `lc1` equals ‘s’, `lc2` equals ‘c’, and `lc3` equals ‘r’.

```
boolean found = false;
blocktest().given(vCardType, new AdrType("foo")).given(label, new LabelType("baz")).given(found,
  false).setup("((AdrType) vCardType).setLabel(new LabelType(\"qux\"));").checkFalse(found);
blocktest().given(vCardType, new AdrType("foo")).given(label, new LabelType("bar")).given(found,
  false).setup(() -> {
    ((AdrType) vCardType).setLabel(new LabelType("bar"));
}).checkTrue(found);
AdrType adr = (AdrType)vCardType;
if(adr.hasLabel()) {
  if(label.equals(adr.getLabel())) {
    found = true;
    break;
  }
}
```

Fig. 5. The `setup()` expression provides test fixture functionality in block tests.

Example 3 (fixtures and preventing value inference from context). BDK supports using fixtures [25, 78] for performing additional setup on input objects. Also, BDK prevents value inference (re-using values of variables from the target fragment's context) by default. To see an example of fixture usage, consider the example in Figure 5. There, both block tests target the fragment on lines 8–14 and assign a new `AdrType("foo")` object to `vCardType`. Although `vCardType` is used on line 8, `AdrType` does not have a constructor that takes a `Label`. Yet, setting the `Label` of `vCardType` is important for block testing this fragment, since the `if` statement on line 9 evaluates to `false` if the `Label` is not set. BDK's `setup` expression can be used to perform such extra setup in two ways. In this example, `setLabel()` is called to set the `Label` on a newly instantiated `AdrType` either via a string containing `setup` expressions (*e.g.*, on line 3) or via a lambda function (*e.g.*, on line 6). Putting expressions in a string can prevent compilation errors because lambdas in Java cannot capture variables that are not `final` or effectively `final`. Both block tests also assign `found` to value `false` (via `given`), even though `found` is already defined on line 1. BDK does not automatically infer values for variables assigned outside the target fragment, because changes to non-target fragments can then affect inferred values and make block tests more prone to break as software evolves. The block test on line 2 constructs a new `Label` from "baz" and sets it as the label of `vCardType`, whereas the block test on line 4 sets `vCardType`'s label to a `Label` constructed from "bar". Finally, the first block test asserts that after executing the fragment on these inputs, the value of `found` is `false` (using `checkFalse`), whereas the second block test asserts that `found` is `true` (using `checkTrue`).

```
1 lambdatest().args(new File(System.getProperty("user.dir"))).checkReturnTrue();
2 lambdatest().args(new File(System.getProperty("user.dir") + "/target/gen-src/annotations"))
3   .checkReturnFalse();
4 catalogs.addAll(Arrays.stream(catalogFiles)
5   .filter(File::isDirectory)
6   .filter((File catalog) -> {
7     final File[] schemas = catalog.listFiles(new DirectoryFilter(DEFAULT_LOCALE));
8     return schemas != null && schemas.length > 0;
9   })
10  .map(File::getName)
11  .collect(Collectors.toList())
12 );
```

Fig. 6. Block tests can validate return statements, lambdas, and their mixture in other complex PL constructs.

Example 4 (special support for validating lambdas). BDK provides special syntax for validating lambdas, which often contain complex code and are within methods. For example, the fragment on lines 4–12 in Figure 6 mixes Java's Stream API with lambdas and has computations inside a return statement to filter directories from a list (line 8). To support block testing in the presence of such complex usage of relatively newer PL features, BDK provides a dedicated `lambdatest` construct that allows developers to directly validate lambdas. The input variables in a `lambdatest` can be explicitly provided. For example, the block tests on lines 1–3 use BDK's `args` expression to specify objects that should be assigned to the `catalog` input variable on line 6. Also, to handle complex return statements within lambda expressions (*e.g.* line 8), the block test on line 1 uses the `checkReturnTrue` expression to assert that the lambda returns `true`. Lastly, the block test on lines 2–3 asserts that the lambda returns `false` using the `checkReturnFalse` expression.

3 Framework

First, §3.1 presents our design choices in BDK and their rationale. Then, §3.2 presents syntax and semantics of block tests as implemented in BDK, and proof sketches for some safety properties. Finally, in §3.3, we briefly describe our BDK implementation and its current limitations.

⟨Prog⟩	::=	⟨Fns⟩ ⟨Block⟩
⟨Fns⟩	::=	⟨Fn⟩ ⟨Fns⟩ | ϵ
⟨Fn⟩	::=	⟨Identifier⟩ **( )** ⟨Exp⟩
⟨Block⟩	::=	**start** ⟨Stmts⟩ **end**
⟨Stmts⟩	::=	⟨Stmt⟩ ⟨Stmts⟩ | ϵ
⟨Stmt⟩	::=	⟨Assignment⟩ | ⟨Block⟩ | ⟨If⟩ | ⟨Assert⟩ | ⟨BlockTest⟩
⟨Assignment⟩	::=	⟨Identifier⟩ = ⟨Exp⟩
⟨If⟩	::=	**if (** ⟨Exp⟩ ⟨Op⟩ ⟨Exp⟩ **)** ⟨Block⟩ **else** ⟨Block⟩
⟨Assert⟩	::=	**assert (** ⟨Exp⟩ **,** ⟨Exp⟩ **) ;**
⟨Op⟩	::=	== | < | **!=**
⟨Exp⟩	::=	⟨Identifier⟩ | ⟨Number⟩ | ⟨FnCall⟩
⟨FnCall⟩	::=	⟨Identifier⟩ **( )**
⟨BlockTest⟩	::=	**blocktest** ⟨Setup⟩ ⟨Oracles⟩ **begin ;**
⟨Setup⟩	::=	⟨InputAssignment⟩ ⟨Setup⟩ | ⟨FnMock⟩ ⟨Setup⟩ | ϵ
⟨InputAssignment⟩	::=	**given (** ⟨Identifier⟩ **,** ⟨Number⟩ **) ;** | **setup (** ⟨Block⟩ **) ;**
⟨FnMock⟩	::=	**mock (** ⟨Identifier⟩ **,** ⟨Number⟩ **) ;**
⟨Oracles⟩	::=	⟨Oracle⟩ ⟨Oracles⟩ | ϵ
⟨Oracle⟩	::=	**check (** ⟨Identifier⟩ **,** ⟨Number⟩ **) ;**

Fig. 7. Grammar of a featherweight programming language that we use to illustrate the semantics of block tests. Non-terminals in the host language are in ⟨⟩, *e.g.*, ⟨Prog⟩, while non-terminals that we introduce for block tests are in underlined ⟨⟩, *e.g.*, ⟨BlockTest⟩. Similarly, terminals in the host language are bold, *e.g.*, **start**, while terminals that we introduce for block tests are bold and underlined, *e.g.*, **blocktest**.

3.1 Design Choices and Their Rationale

The examples that we presented so far have described and provided rationale for five design choices that we make in BDK: (i) support for both data- and control-flow oracles; (ii) support for referencing undeclared or un-initialized variables; (iii) resolving types of input variables from context and preventing inference of their values; (iv) support for using fixtures and mocks; and (v) special syntax for validating lambdas. (Mocks differ from fixtures; the former replace whole method call chains with values). These choices are initial, based on our formative study (§2) and prior work on inline tests. Future work can expand or adapt them for other programming languages or language features. Here, we describe an additional design choice and its rationale:

Complementary to Existing Code and Test Granularities. Block tests' executions should not interfere with those of test granularity levels (*e.g.*, unit and inline tests), since block tests are intended to complement and not replace these other granularity levels. Also, although block tests are defined within the code under test (there is no code-test boundary as in unit tests), block tests should only run during testing; their presence must not affect the behavior of deployed programs.

3.2 Syntax and Semantics for Block Tests in Java

We first describe syntax of block tests, as embedded into the Java programming language (§3.2.1). Then, we give formal semantics of block tests (§3.2.2) and proof sketches of some properties (§3.2.3).

3.2.1 Syntax. In Figure 7, we introduce a Featherweight programming language to show how block tests can be embedded in a programming language (subsequently referred to as the "host" language), and to aid the presentation of their semantics. There, terminals in the host language are in **bold**, while terminals that we introduce to illustrate block tests are **bold and underlined**. Also, non-terminals in the host language are in ⟨⟩ *without* underlines, *e.g.*, ⟨Prog⟩, while non-terminals that we introduce to illustrate block tests are in ⟨⟩ and underlined, *e.g.*, ⟨BlockTest⟩.

$$\frac{\langle \text{fns}, \sigma\rangle \Downarrow \langle \sigma'\rangle \ \langle \text{blk}, \sigma'\rangle \Downarrow \langle \sigma''\rangle}{\langle \text{fns blk}, \underline{\text{init}}()\rangle \Downarrow \langle \sigma''\rangle}\ \text{prog (1)} \qquad \frac{}{\langle \epsilon, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{epsilon (2)}$$

$$\frac{\langle \text{fn}, \sigma\rangle \Downarrow \langle \sigma'\rangle \ \langle \text{fns}, \sigma'\rangle \Downarrow \langle \sigma''\rangle}{\langle \text{fn fns}, \sigma\rangle \Downarrow \langle \sigma''\rangle}\ \text{fns (3)} \qquad \frac{\mathit{Funcs}' = \sigma.\mathit{Funcs}[\text{f}/\text{e}]}{\langle \text{f() e}, \sigma\rangle \Downarrow \langle \sigma[\mathit{Funcs}/\mathit{Funcs}']\rangle}\ \text{fn (4)}$$

$$\frac{\langle \textbf{start}, \sigma\rangle \Downarrow \langle \sigma'\rangle \langle \text{stmts}, \sigma'\rangle \Downarrow \langle \sigma''\rangle \langle \textbf{end}, \sigma''\rangle \Downarrow \langle \sigma'''\rangle}{\langle \textbf{start}\ \text{stmts}\ \textbf{end}, \sigma\rangle \Downarrow \langle \sigma'''\rangle}\ \text{block (5)} \qquad \frac{\langle \text{stmt}, \sigma\rangle \Downarrow \langle \sigma'\rangle \ \langle \text{stmts}, \sigma'\rangle \Downarrow \langle \sigma''\rangle}{\langle \text{stmt stmts}, \sigma\rangle \Downarrow \langle \sigma''\rangle}\ \text{stmts (6)}$$

$$\frac{\langle \sigma.\mathit{Funcs}[\text{f}], \sigma\rangle \Downarrow \text{v}}{\langle \text{f}(), \sigma\rangle \Downarrow \text{v}}\ \text{fn-call (7)} \qquad \frac{}{\langle \text{x}, \sigma\rangle \Downarrow \sigma.\mathit{Mem}[\text{x}]}\ \text{variable (8)} \qquad \frac{}{\langle \text{v}, \sigma\rangle \Downarrow \text{v}}\ \text{literal (9)}$$

$$\frac{\langle \text{e1}, \sigma\rangle \Downarrow \text{v1} \quad \langle \text{e2}, \sigma\rangle \Downarrow \text{v2}}{\langle \text{e1} < \text{e2}, \sigma\rangle \Downarrow \text{v1} <_{int} \text{v2}}\ \text{lt (10)} \qquad \frac{\langle \text{e1}, \sigma\rangle \Downarrow \text{v1} \quad \langle \text{e2}, \sigma\rangle \Downarrow \text{v2}}{\langle \text{e1} == \text{e2}, \sigma\rangle \Downarrow \text{v1} ==_{int} \text{v2}}\ \text{eq (11)} \qquad \frac{\langle \text{e1}, \sigma\rangle \Downarrow \text{v1} \quad \langle \text{e2}, \sigma\rangle \Downarrow \text{v2}}{\langle \text{e1}! = \text{e2}, \sigma\rangle \Downarrow \text{v1}! =_{int} \text{v2}}\ \text{neq (12)}$$

Fig. 8. The semantics for both normal program runs (*NORMAL*) and block test runs (*TEST*, for the *IN* mode). The rules in this figure follow behavior commonly seen in well-known programming languages.

Each program (⟨Prog⟩) in this language has one or more function declarations followed by a block to execute. A function (⟨Fn⟩) defines an expression to be executed when the function is called. A block (⟨Block⟩), which has start and end markers, is a sequence of statements. A statement (⟨Stmt⟩) can be an assignment (⟨Assignment⟩), a block, if (⟨If⟩), or assert (⟨Assert⟩). An expression (⟨Exp⟩) is either an identifier (⟨Identifier⟩), a number (⟨Number⟩), or a function call (⟨FnCall⟩).

We extend the host language with the ⟨BlockTest⟩ non-terminal. A block test starts with the keyword **blocktest**, followed by a setup (⟨Setup⟩) to initialize the state before running the test, and test oracles (⟨Oracles⟩). A block test declaration ends with keyword **begin**. The block test itself, as we show in our semantics, terminates at the end of a fragment. A test can have variable assignments or execute the extra code block ⟨InputAssignment⟩ during setup. Setup can use mocks (⟨FnMock⟩), *i.e.*, replacement values for previously defined functions. Lastly, each oracle (⟨Oracle⟩) defines a variable's expected value at the end of a fragment and checks the variable against observed output.

3.2.2 Semantics. We provide big-step structural operational semantics for block tests [46, 87]. We define the configuration as $\langle \mathit{Code}, \sigma\rangle$, where the first element is the code to be executed and the second element (σ) is the program state. The state is a named tuple: $(\mathit{Mode}, \mathit{Mem}, \mathit{Asserts}, \mathit{Funcs}, \mathit{Scope})$.

- *Mode* indicates if execution is outside (*OUT*) or inside (*IN*) a block test.
- *Mem* maps variables to values.
- *Asserts* is a block that contains dynamically created assert statements.
- *Funcs* maps function names to expressions that they evaluate.
- *Scope* is an integer indicating the nesting level of blocks.

Funcs defines three functions that we use in the semantics rules: (i) zero "zeros out" the program state, *i.e.*, set σ to (unchanged, empty map, empty block, unchanged, 0); (ii) append($\mathit{stmt}, \mathit{blk}$) appends statement *stmt* to block *blk* and returns a new block; and (iii) init creates an empty state, but that initial state differs between two kinds of runs (*NORMAL* and *TEST*) as we describe later.

We give semantics for two kinds of runs: (a) normal program run (*NORMAL*), *i.e.*, running host code while skipping block tests; and (b) block test run (*TEST*), which finds and runs block tests.

(a) Normal Program Runs (*NORMAL*). Semantics of normal program runs (*i.e.*, running host code while skipping block tests) are given by the rules in Figures 8 and 9. Function init creates the following initial state in this mode: (*IN*, empty map, empty block, empty map, 0). Statements that are unrelated to block tests have their standard meaning, *e.g.*, as defined by rule (10, lt), (11, eq), etc.

$$\frac{\langle \mathrm{e}, \sigma\rangle \Downarrow \mathrm{v} \quad Mem' = \sigma.Mem[\mathrm{x}/\mathrm{v}]}{\langle \mathrm{x} = \mathrm{e}, \sigma\rangle \Downarrow \langle \sigma[Mem/Mem']\rangle}\ \text{assignment (13)}$$

$$\frac{\langle \mathrm{e}, \sigma\rangle \Downarrow true \quad \langle \mathrm{blk1}, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \textbf{if}\ (\mathrm{e})\ \mathrm{blk1}\ \textbf{else}\ \mathrm{blk2}, \sigma\rangle \Downarrow \langle \sigma'\rangle}\ \text{if-true (14)} \qquad \frac{\langle \mathrm{e}, \sigma\rangle \Downarrow false \quad \langle \mathrm{blk2}, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \textbf{if}\ (\mathrm{e})\ \mathrm{blk1}\ \textbf{else}\ \mathrm{blk2}, \sigma\rangle \Downarrow \langle \sigma'\rangle}\ \text{if-false (15)}$$

$$\frac{\langle \mathrm{e1}, \sigma\rangle \Downarrow \mathrm{v1} \quad \langle \mathrm{e2}, \sigma\rangle \Downarrow \mathrm{v2} \quad \mathrm{v1} = \mathrm{v2}}{\langle \textbf{assert}(\mathrm{e1}, \mathrm{e2}), \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{assert (16)} \qquad \frac{Scope' = \sigma[Scope] + 1}{\langle \textbf{start}, \sigma\rangle \Downarrow \langle \sigma[Scope/Scope']\rangle}\ \text{start-block (17)}$$

$$\frac{\sigma.Scope = \mathrm{k} \quad \mathrm{k} > 0}{\langle \textbf{end}, \sigma\rangle \Downarrow \langle \sigma[Scope/\mathrm{k}-1]\rangle}\ \text{end-block (18)} \qquad \frac{\langle \sigma.Asserts, \sigma\rangle \Downarrow \langle \sigma\rangle \quad \sigma.\mathrm{Scope} = 0}{\langle \textbf{end}, \sigma\rangle \Downarrow \langle \sigma[Mode/OUT, Scope/0]\rangle}\ \text{end-zero-block (19)}$$

$$\frac{}{\langle \underline{\textbf{blocktest}}, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{blocktest (20)} \qquad \frac{}{\langle \underline{\textbf{begin}}, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{begin (21)} \qquad \frac{}{\langle \underline{\textbf{given}}(\mathrm{x}, \mathrm{v}), \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{given (22)}$$

$$\frac{}{\langle \underline{\textbf{setup}}(\mathrm{blk}), \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{setup (23)} \qquad \frac{}{\langle \underline{\textbf{mock}}(\mathrm{f}, \mathrm{e}), \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{mock (24)} \qquad \frac{}{\langle \underline{\textbf{check}}(\mathrm{x}, \mathrm{v}), \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{oracle (25)}$$

Fig. 9. The semantics for both normal program runs (*NORMAL*) and block test runs (*TEST*, for the *IN* mode). Note that rules (20, in:blocktest)-(25, in:oracle) make no changes to the state, *i.e.*, block tests are ignored in *NORMAL* runs and nested block tests are skipped in *TEST* runs.

$$\frac{}{\langle \mathrm{x} = \mathrm{e}, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{assignment (26)} \qquad \frac{\langle \mathrm{blk1}, \sigma\rangle \Downarrow \langle \sigma'\rangle \ \langle \mathrm{blk2}, \sigma'\rangle \Downarrow \langle \sigma''\rangle}{\langle \textbf{if}\ (\mathrm{e})\ \mathrm{blk1}\ \textbf{else}\ \mathrm{blk2}, \sigma\rangle \quad \Downarrow \quad \langle \sigma''\rangle}\ \text{if (27)}$$

$$\frac{}{\langle \textbf{assert}(\mathrm{e1}, \mathrm{e2}), \sigma \Downarrow \langle \sigma\rangle}\ \text{assert (28)} \qquad \frac{}{\langle \textbf{start}, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{start (29)} \qquad \frac{}{\langle \textbf{end}, \sigma\rangle \Downarrow \langle \sigma\rangle}\ \text{end (30)}$$

$$\frac{\sigma' = \underline{\mathrm{zero}}(\sigma)}{\langle \underline{\textbf{blocktest}}, \sigma\rangle \Downarrow \langle \sigma'\rangle}\ \text{blocktest (31)} \qquad \frac{}{\langle \underline{\textbf{begin}}, \sigma\rangle \Downarrow \langle \sigma[Mode/IN]\rangle}\ \text{begin (32)}$$

$$\frac{Mem' = \sigma.Mem[\mathrm{x}/\mathrm{v}]}{\langle \underline{\textbf{given}}(\mathrm{x}, \mathrm{v}), \sigma\rangle \Downarrow \langle \sigma[Mem/Mem']\rangle}\ \text{given (33)} \qquad \frac{\langle \mathrm{blk}, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \underline{\textbf{setup}}(\mathrm{blk}), \sigma\rangle \Downarrow \langle \sigma'\rangle}\ \text{setup (34)}$$

$$\frac{Funcs' = \sigma.Funcs[\mathrm{f}/\mathrm{e}]}{\langle \underline{\textbf{mock}}(\mathrm{f}, \mathrm{e}), \sigma\rangle \Downarrow \langle \sigma[Funcs/Funcs']\rangle}\ \text{mock (35)} \qquad \frac{Asserts' = \underline{\mathrm{append}}(\sigma.Asserts, \textbf{assert}(\mathrm{x}, \mathrm{v}))}{\langle \underline{\textbf{check}}(\mathrm{x}, \mathrm{v}), \sigma\rangle \Downarrow \langle \sigma[Asserts/Asserts']\rangle}\ \text{oracle (36)}$$

Fig. 10. The semantics for the block test runs (*TEST* for the *OUT* mode). These rules define abstract execution of the program structure without evaluating any expression. The goal of this abstract execution is to find all block tests regardless of their location in the program. For example, (27, out:if) triggers the execution of both branches, but ignores the conditional expression.

Constructs related to block tests, *e.g.*, **blocktest** (20, in:blocktest), **begin** (21, in:begin), **given** (22, in:given), **setup** (23, in:setup), **mock** (24, in:mock), **oracle** (25, in:oracle), have no impact as they are skipped in this kind of run. Note that the rule (19, in:end-zero-block), which ends the fragment when scope has a value of zero is never used in this kind of run, *i.e.*, each **start** has a matching **end**.
(b) Block Test Runs (*TEST*). The semantics of block test runs are given using rules in Figures 8, 9, and 10. Function init creates initial state $\sigma = (OUT, \text{empty map}, \text{empty block}, \text{empty map}, 0)$. The execution starts in *OUT* (*i.e.*, outside block tests) and then switches to *IN* whenever a block test starts until the end of the block test when it changes to *OUT* again.

The execution starts using rules defined in Figures 8 and 10 (*OUT* mode). The execution is *abstract*, *i.e.*, it explores the program structure without evaluating any expression along the way and without changing the state of the program. Only when the beginning of a block test is reached in rule (31, out:blocktest) is the program state impacted. Rule (31, out:blocktest) zeros the state to prepare it for the execution of the block test. The key thing to note here is that zeroing state sets the *Scope* to zero, which never happens during normal program runs. As part of the block test definitions, there are rules for memory initialization (33, out:given) and (34, out:setup), mocking of functions (35, out:mock), and preparing oracles (36, out:oracle). The definition of a block test ends with rule $\underline{\textbf{begin}}$ (32, out:begin), which changes the execution mode to *IN*, thereby setting the rules for executing the block test as those in Figure 9.

As we stated, execution inside the block tests (Figure 9) follows the same semantics as the execution of normal program runs (*NORMAL*). That means that most statements have their traditional semantics (Figure 8) and we ignore nested block tests.

The key step of the *IN* mode is the end of the block (19, in:end-zero-block). Namely, once the end of the fragment in which the block test is defined has been reached, all oracles that were created while executing the test are checked. Also, this rule switches back to the *OUT* mode, thus continuing the abstract execution of the program using rules in Figure 10.

3.2.3 Safety Properties. We now state and sketch proofs for a few properties which ensure that: (i) block tests are properly isolated and well-behaved with respect to scope management; (ii) execution in *OUT* mode is independent of concrete values in *Mem*; and (iii) if a block test completes successfully, all oracles were evaluated and they passed.

Property 1 (Block Test Bracketing). *If the execution of a complete* `blocktest()` *statement starts in mode OUT with $Scope = 0$ and the execution does not get stuck (*i.e.*, there is always a rule to apply), then the final configuration has $Scope = 0$ and $Mode = OUT$.*

Proof Sketch. Consider the execution sequence of a block test:

(1) **Initial state**: σ_0 with $\sigma_0.Mode = OUT$ and $\sigma_0.Scope = 0$.

(2) **Apply rule (31,** out:blocktest**)**:

$$\langle \underline{\textbf{blocktest}}, \sigma_0 \rangle \Downarrow \langle \sigma_1 \rangle \text{ where } \sigma_1 = \underline{\text{zero}}(\sigma_0)$$

By definition of $\underline{\text{zero}}$: $\sigma_1.Scope = 0$.

(3) **Setup phase**: Applications of rules (33, out:given), (34, out:setup), (35, out:mock), and (36, out:oracle) do not modify *Scope*. Thus, after setup: $\sigma_{\text{setup}}.Scope = 0$.

(4) **Apply rule (32,** out:begin**)**:

$$\langle \underline{\textbf{begin}}, \sigma_{\text{setup}} \rangle \Downarrow \langle \sigma_{\text{in}} \rangle \text{ where } \sigma_{\text{in}} = \sigma_{\text{setup}}[Mode/IN]$$

The *Scope* is unchanged: $\sigma_{\text{in}}.Scope = 0$, but the mode changes: $\sigma_{\text{in}}.Mode = IN$.

(5) **Fragment execution**: The target fragment executes. During execution, most semantic rules (such as (13, in:assignment), (14, in:if-true), (15, in:if-false), (7, fn-call), (8, variable), (9, literal), and (16, in:assert)) do not modify *Scope*. Only the following rules affect *Scope*:

- Rule (17, in:start-block) increments *Scope* by 1 for nested blocks.
- Rule (18, in:end-block) decrements *Scope* by 1 (only when $Scope > 0$) for nested blocks.
- These rules balance out for well-formed code with properly nested blocks.

(6) **Block test termination**: Eventually, the outermost `end` is reached with $Scope = 0$.

(7) **Apply rule (19,** in:end-zero-block**)**:

$$\langle \textbf{end}, \sigma_{\text{final}} \rangle \Downarrow \langle \sigma_{\text{out}} \rangle \text{ where } \sigma_{\text{out}} = \sigma_{\text{final}}[Mode/OUT, Scope/0]$$

Therefore: $\sigma_{\text{out}}.Scope = 0$ and $\sigma_{\text{out}}.Mode = OUT$.

Well-formedness assumption: We assume the fragment has properly balanced `start`/`end` pairs. In well-formed code, every `start` (incrementing $Scope$) has a matching `end` (decrementing $Scope$). When the fragment's last `end` is reached, all nested blocks have closed, and $Scope$ returns to 0. □

Corollary (Test Isolation). *Block tests do not interfere with each other's scope state.*

Proof Sketch. By the Block Test Bracketing property, each block test starts and ends with $Scope = 0$. Combined with the $\underline{\text{zero}}$ function in rule (31, out:blocktest), each new block test begins with a fresh $Scope = 0$ regardless of the state left behind by previous tests. (We cover other parts of the state in the next property.) □

Property 2 (Memory Abstraction in OUT Mode). *Code executing in OUT mode does not observe concrete values in Mem that were modified during IN mode execution. More precisely, after transitioning from IN to OUT mode via rule (19,* in:end-zero-block*), subsequent execution in OUT mode is independent of the concrete values in Mem.*

Proof Sketch. We show that OUT mode execution is abstract and does not read memory values produced by IN mode execution.

(1) OUT **mode rules that are abstract**: In OUT mode, most rules do not read from Mem:
- Rule (26, out:assignment): Performs no operation; does not evaluate the expression or read from Mem.
- Rule (27, out:if): Explores both branches regardless of the condition; does not evaluate the conditional expression or read from Mem.
- Rule (28, out:assert): Performs no operation; does not evaluate the assertion or read from Mem.
- Rules (29, out:start) and (30, out:end): Performs no operations.

(2) OUT **mode rules that write to** Mem: The only OUT mode rules that modify Mem are for setting up the state that the *next* block test uses:
- Rule (33, out:given): Writes test input values to Mem for the upcoming block test.
- Rule (34, out:setup): Executes setup code that may modify Mem for the upcoming block test.

Both rules execute *before* the block test enters IN mode via rule (32, out:begin).

(3) **Memory reads occur only in** IN **mode**: Expression evaluation rules (7, fn-call), (8, variable), (9, literal), comparison operators that read from Mem are only used during IN mode execution when the target fragment actually runs with concrete semantics.

(4) **Memory reset between tests**: After rule (19, in:end-zero-block) returns execution to OUT mode, the next `blocktest()` invocation triggers $\underline{\text{zero}}(\sigma)$, which resets Mem to an empty map, discarding any values from the previous test's IN mode execution.

Conclusion: Memory modifications during IN mode (fragment execution) are confined to that block test and do not affect the abstract program execution in OUT mode. The separation between IN mode (concrete execution) and OUT mode (abstract execution) ensures that block tests do not interfere with abstract program exploration. □

Property 3 (Oracle Completeness). *If the execution of a complete* `blocktest()` *statement starts in mode OUT with $Scope = 0$ and completes successfully (i.e., reaches a final configuration with $Mode = OUT$ and $Scope = 0$), then all oracles created in $Asserts$ during the block test's execution were evaluated and they passed.*

Proof Sketch. We show that successful test completion implies that all oracles were checked.

(1) **Oracle collection**: During the test setup phase in OUT mode, each `check(x, v)` statement applies rule (36, out:oracle), which appends a block test assertion to $Asserts$.

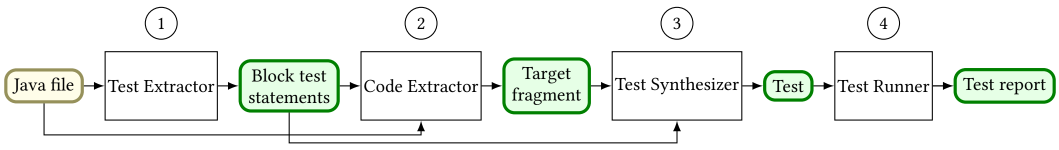

Fig. 11. The main components in BDK's architecture.

(2) **Transition to *IN* mode**: Rule (32, out:begin) transitions to *IN* mode without modifying *Asserts*.

(3) **Fragment execution**: The target fragment executes in *IN* mode. Any check statements inside the fragment are ignored by rule (25, in:oracle).

(4) **Oracle evaluation**: When the outermost end is reached with $Scope = 0$, rule (19, in:end-zero-block) applies. This rule has a premise that requires evaluating all oracles in *Asserts* using rule (16, in:assert).

(5) **Failure implies stuck execution**: If any oracle fails, rule (16, in:assert) cannot apply, the premise of rule (19, in:end-zero-block) is not satisfied, and execution gets stuck.

Conclusion: Since successful completion requires reaching $Mode = OUT$, rule (19, in:end-zero-block) must have applied, meaning that all oracles were evaluated and they passed. □

3.3 BDK: Implementation and Current Limitations

§3.3.1 first describes how BDK works end to end. Then, §3.3.2 describes current BDK limitations.

```
int sx = compute();
blocktest("test1").given(input, new BinarySource(/*width*/5, /*height*/5, new byte[]{0, 1, 2, 3}))
  .given(sx, 5).checkFlow(IfStmt().ElseIf().Then()).checkFalse(sx == 5).end(FIRST_ASSIGN_BLOCK);
// @blocktest("test2").given(...).checkFalse(sx == 5);
blocktest("test3").given(...).checkFalse(sx == 5).end(FIRST_BLOCK);
blocktest("test4").given(...).checkFalse(sx == 5).end(FIRST_BLOCK, 2);
blocktest("test5").given(...).checkFalse(sx == 5).end(FIRST_THROW);
blocktest("test6").given(...).checkFalse(sx == 5);
if (sx < 0) { // test1–test6, and test9 start here
  sx += input.getWidth();
} else if (sx > input.getWidth()) { // fault: should be >= instead of >
  sx %= input.getWidth();
} // test1, test3 end here
if (input.getWidth() < 0) throw new Exception(); // test4 and test5 end here, test8 and test10 start here
sx += 1; // test6 ends here
blocktest("test6").end();
if (input.getWidth() > 100) { // test7 starts here
  sx *= input.getWidth();
} // test2, test7–test10 end here
blocktest("test7").given(...).checkFalse(sx == 5).start(FIRST_BLOCK);
blocktest("test8").given(...).checkFalse(sx == 5).start(FIRST_BLOCK, 2);
blocktest("test9").given(...).checkFalse(sx == 5).start(FIRST_BLOCK, 3);
blocktest("test10").given(...).checkFalse(sx == 5).start(FIRST_THROW);
int sy = compute();
```

Fig. 12. Artificially expanding code in Example 1 (§2) to illustrate more details about BDK.

3.3.1 Implementation. Figure 11 shows the architecture of our BDK implementation in terms of how it works end to end to run block tests for a target fragment. We use Figure 12 as a running example; it contains code similar to Figure 3, but expanded artificially to explain how scope in block tests works. In Figure 12, we use BDK's optional syntax for uniquely identifying block tests

Table 1. Target-fragment scopes for each block test in Figure 12, as determined by Code Extractor.

	Starts	Ends	Reason
`test1`	9	13	This `if` block is the first block containing an assignment to `sx`.
`test2`	9	19	Does not have an `end()`; default to the last reference to `sx`.
`test3`	9	13	Ends after the first block following the block test statement.
`test4`	9	14	Ends after the second block following the block test statement.
`test5`	9	14	Ends after the first throw statement.
`test6`	9	15	Ends at the line where `end()` is used with a matching block-test name.
`test7`	17	19	Starts just before the first block preceding the block test statement, counting from bottom to top.
`test8`	14	19	Starts just before the second block preceding the block test statement, counting from bottom to top.
`test9`	9	19	Starts just before the third block preceding the block test statement, counting from bottom to top.
`test10`	14	19	Starts just before the first throw statement, counting from bottom to top.

to name them as `"test1"`, `"test2"`, ..., `"test10"`. Given a Java file (*e.g.*, a file containing the fragment in Figure 12), BDK's four main components work as follows:
① *The Test Extractor* uses JavaParser [39] to find block test statements as those starting with `blocktest()` or `lambdatest()` declarations. In Figure 11, Test Extractor identifies lines 2–8 and lines 20–23 as block test statements. If users provide block tests in comments (for reasons explained in Example 2 of §2), Test Extractor also obtains block test statements as those that start with `//@blocktest()` or `//@lambdatest()` (*e.g.*, `"test2"` on line 4 in Figure 12).
② *The Code Extractor* extracts the target fragment from its enclosing method using the input Java file and block test statements extracted by the Test Extractor. The Code Extractor first determines the scope (start and end) of the target fragment. To do so, the Code Extractor checks for the `start(..)` and `end(..)` API calls in block test statements. If neither API call is present (*e.g.*, in `test2` on line 4 and `test6` on line 8), Code Extractor starts the fragment at the first statement after the block test statement which is not a block test statement (*e.g.*, line 9). Code Extractor then searches downward until it encounters either an explicit fragment-end declaration `blocktest(name).end()` (*e.g.*, line 15 for `test6`) or the last occurrence of a variable referenced in the block test's assertion (*e.g.*, line 19 for `test2`, where variable `sx` is used for the last time). If the `end(..)` but not the `start(..)` API call is present (*e.g.*, in `test1` on lines 2–3), Code Extractor determines the scope as starting from the first non-block test statement immediately following the current block test statement and determines the end based on the required[1] argument of `end()`. For example, `FIRST_ASSIGN_BLOCK` means that the fragment ends after the end of the first block containing assignments to all variables referenced in the block test's assertions, `FIRST_BLOCK` ends after the first basic block, and `FIRST_THROW` ends after the first `throw` statement.

If the `start(..)` but not the `end(..)` API call is present, it means that the target fragment precedes the block test statement. So, the Code Extractor ends the fragment immediately before the block test statement and determines the start of that fragment based on the required argument to `start()`. Those arguments are syntactically the same as the ones for the `end()` API call, but semantically, they cause the Code Extractor to search backwards for the beginning of the target fragment. BDK prevents the use of both `start()` and `end()` API calls in the same block test statement. Table 1 shows the scope for all block tests in Figure 12, in terms of the start and end of target fragments, along with the reason why the scope is resolved as stated.

[1]`end()` does not require an argument when used in an explicit fragment-end declaration, *e.g.* `blocktest(name).end()`, but requires an explicitly provided name for the test.

```
1 @Test public void test1() {
2   // blocktest("test1").given(input, new BinarySource(5, 5, new byte[]{0, 1, 2, 3})).given(sx, 5).checkFlow(IfStmt().ElseIf().
        Then()).checkFalse(sx == 5).end(FIRST_ASSIGN_BLOCK);
3   /* >>> Setup Starts <<< */
4   BinarySource input = new BinarySource(5, 5, new byte[]{0, 1, 2, 3}); // provided in given expression
5   int sx = 5; // provided in given expression
6   boolean[] _blocktest_checkFlow_1 = new boolean[1]; // generated from checkFlow expression
7   /* >>> Setup Ends, Target Fragment Starts <<< */
8   if (sx < 0) {
9     sx += input.getWidth();
10  } else if (sx > input.getWidth()) { // fault: should be >= instead of >
11    _blocktest_checkFlow_1[0] = true;
12    sx %= input.getWidth();
13  }
14  /* >>> Target Fragment Ends, Assertions Start <<< */
15  assertFalse(sx == 5); // generated from checkFalse expression
16  for (int i = 0; i < _blocktest_checkFlow_1.length; i++) { // generated from checkFlow expression
17    assertTrue("checkFlow(IfStmt().ElseIf().Then()) fails at step "+i, _blocktest_checkFlow_1[i]);
18  }
19  /* >>> Assertions End <<< */
20 }
```

Fig. 13. An example unit test that is synthesized “behind the scenes” to run a block test.

③ *The Test Synthesizer* generates code for, and compiles the block test using the block test statement from the Test Extractor and the fragment from Code Extractor. To ensure compilation, the Test Synthesizer (i) initializes input variables as assigned in any `given()` or `args()` expressions; (ii) generates initialization code (using any `setup()` expressions); (iii) performs mocking (using any `mock()` expressions); (iv) translates control flow oracles (using any `checkFlow()` expressions) into a list of Boolean flags, where each flag is to be set at runtime if control flows to a desired branch; and (v) connects expected exceptions (using any `expect()` expressions) to the testing framework’s annotation, *e.g.*, `@Test(expected = NullPointerException.class)` for JUnit.
Developers need not see code that Test Synthesizer generates, but we show and discuss it next, to give readers more insights into how BDK works. Figure 13 shows the generated code for `test1`, which is defined on lines 2–3 in Figure 12. In Figure 13, the declarations of `input` and `sx` on lines 4 and 5 are generated from the corresponding `given` expressions. The code on lines 8 and 13 show the target fragment extracted by Code Extractor; note the inserted code to set the `checkFlow`-related flag on line 11. Finally, the assertions on lines 15 and 17 are synthesized from `test1`’s `checkFalse()` and `checkFlow()` expressions. The assertion on line 17 is in a loop: if any `checkFlow`-related flags have not been set, it means that control has not flowed as expected and the block test fails. In this case, the block test reports an error message showing the first unset flag to aid debugging.
④ *The Test Runner* takes compiled block test code from the Test Synthesizer, executes them, and produces a test report showing whether block tests pass or fail. As shown in Figure 13, our BDK implementation currently produces a unit test behind the scenes. Our rationale for doing so is twofold. First, BDK reuses publicly available and mature unit-testing infrastructure. Second, users can see the output of block and unit tests in the same place (for emphasis, we repeat that users need not see synthesized block-test code).

BDK’s Type Resolution. As discussed in Example 1 (§2), users do not need to explicitly specify types of input variables when assigning input values using `given()` expressions. In such cases, the Test Synthesizer component automatically resolves types using JavaParser’s source-level type resolution (for variables declared in project code) and bytecode-level resolution (for variables declared in Java and third-party libraries). If type resolution fails (typically due to the use of

Table 2. BDK's current rules for predicting variable types from assigned values, with examples.

Rule	Example value(s)	Example type(s)
Literal	`"foo"`, `1`, `1000000L`, `true`	`String`, `int`, `long`, `boolean`
Collection	`new ArrayList<>(Arrays.asList(1, 2, 3, 4))`	`ArrayList<Integer>`
Array	`new int[]{1, 2}`, `new boolean[]{true, false}`	`int[]`, `boolean[]`
Enum	`Direction.LEFT`	`Direction`
Class	`Example.class`	`Class`
Object Constructor	`new BinarySource(1, 2)`	`BinarySource`
Casting	`(Foo) Bar.get()`	`Foo`
Method call	`Paths.get("src")`	`Path`
Builder pattern	`new Baz.Builder().setX(1).build()`	`Baz`

generics or untyped lambda parameters), the Test Synthesizer falls back to resolving the variable's type based on its assigned value. Table 2 summarizes the extensible set of rules currently used by the Test Synthesizer for type resolution from values. For example, line 2 in Figure 12 assigns a new `BinarySource` object to the variable `input` (`given(input, new BinarySource(...))`). If JavaParser cannot resolve the type of `input`, the Test Synthesizer applies the Object Constructor rule and resolves `BinarySource` as the type of `input`, based on the assigned value `new BinarySource(...)`. Since target fragments may contain method calls, mocked method return types can also be resolved. The context here is that for non-static methods (*e.g.*, `methodCall()` or `this.methodCall()`), users must mock the return values; static methods are invoked. When mocking non-static method return values, if users do not provide the type, then the Test Synthesizer can resolve the type.

Correspondence with Semantics. The Test Synthesizer generates separate code for each `blocktest()` or `lambdatest()` statement. Doing so allows the Test Synthesizer to execute each block test in isolation. All parts of state needed to run each block test are set at the beginning of the code generated by the Test Synthesizer. Also, BDK does not infer variable values from the target fragment's context. Lastly, users must specify the values of global variables to be used during the execution of each block test. These four choices (isolation, setup, no value inference, and global-variable specification) ensure that execution of block tests does not interfere with global state. So, BDK satisfies Test Isolation and Memory Abstraction properties (§3.2.2) in *OUT* mode. When a block test begins its execution, the `begin` rule (32, out:begin, §3.2.2) is applied, switching execution mode to *IN*. The `given` assignments, `setup` code, `mock` expressions, and the target fragment are run in order. Then, all oracles are evaluated. If all oracles pass (by checking that observed values of all variables in data-flow oracles are as expected and that control flows along specified paths), the block test finishes and the `end` rule (19, in:end-zero-block, §3.2.2) is applied, returning the mode to *OUT*. The next block test is then executed in a fresh state (*i.e.*, state zeros out again). If a test fails, an error is reported and execution terminates. So, BDK satisfies the Block Test Bracketing and Oracle Completeness properties. Lastly, to prevent block tests or any method that they call from running in deployed software, the Test Synthesizer also strips block test statements from the bytecode (*i.e.*, `.class` file) corresponding to the input Java file. So, constructs related to block tests are skipped during *NORMAL* runs, as described in §3.2.2.

3.3.2 Current BDK Limitations. While writing block tests for open-source projects, we discovered three limitations of BDK. First, although BDK allows validating the return values from a fragment, one can currently only do so if the values of all return values in that target fragment have the same declared type. Second, some target fragments require test fixtures (via `setup()` expressions), but the compiler sometimes requires variables in lambdas to be declared as `final` or to be effectively `final` (*i.e.*, the variable's value cannot change after the first assignment). In such cases, a block test with a `setup()` expression may not compile. Avoiding such compilation errors is another reason

Table 3. Summary statistics on 165 GitHub projects that we evaluate: no. of unit tests (#tests), unit-test time in seconds (t), lines of code (SLOC), % statement coverage (cov^s), % branch coverage (cov^b), no. of GitHub commits (#SHAs), years since first commit (age), and no. of stars (#★).

	#tests	t	SLOC	cov^s	cov^b	#SHAs	age	#★
Mean	284.9	17.3	20,734.3	41	34.3	587.1	10.2	247.5
Med	36	2.7	9,246	40.6	32.4	217	9	35
Min	1	1.3	271	0	0	3	3	0
Max	17,874	1,465.4	2.4×10^5	97.2	92.7	7,790	27	5,201
Sum	47,013	2,857.1	3.4×10^6	n/a	n/a	n/a	n/a	40,835

BDK allows writing block tests in comments. Third, BDK does not support two kinds of method invocations in target fragments: (i) a method invoked in the target fragment cannot belong to the same class as the fragment unless it is static or invoked through another object (*i.e.*, the target object cannot be `this`); and (ii) static methods invoked in the target fragment must be public. Note that users can use mocks to supply return values for such unsupported method calls. These three limitations are engineering issues that we plan to address in future work.

4 Evaluation

We organize our evaluation of block tests' feasibility and usefulness around four research questions:

RQ1. How feasible are block tests for validating complex and arbitrary code fragments in blocks (multi-statement sequences in imperative code) or multi-statement lambdas?

RQ2. How costly is it to write block tests and run them with BDK?

RQ3. How do block tests compare with unit tests in terms of reaching code in target fragments?

RQ4. How effective are block tests for finding bugs in target fragments, compared to unit tests?

In RQ1, we evaluate the extent to which block tests and their features (as supported in BDK) can validate a diverse set of target fragments beyond those fragments in our formative study (§2). RQ2 reports on our human time to write block tests and the machine time to run block tests using BDK. RQ3 compares coverage of target fragments achieved by block tests, developer written unit tests, automatically generated unit tests, and the combination of developer written and automatically generated unit tests. Lastly, RQ4 measures BDK's bug-finding ability using mutation analysis and compares the results with those of unit tests on the target fragments.

Evaluation subjects. We target 400 fragments (200 blocks and 200 multi-line lambdas) from 165 open-source projects. To select these projects, we sample a subset of Maven-based Java projects from prior work [28, 82] on testing, where all unit tests pass. We are not the developers of any of these projects. Table 3 shows summary statistics on these 165 projects; the table's caption explains the column headers. The minimum number of unit tests per project is one, reflecting the fact that we do not discriminate among projects based on number of unit tests. The minimum statement coverage shows up as zero due to rounding, but the actual minimum statement coverage is 0.03%. For the project with 0% branch coverage, none of its unit tests covers any branch (*e.g.*, the code reached by the unit tests does not contain any `if` or `switch` statements). To select target blocks, we first compute the cyclomatic complexity [8, 74] of all methods in all 165 projects. Then, we rank the methods in decreasing order of cyclomatic complexity. From these methods, we manually and randomly extract blocks until we reach 200. To increase diversity, we try not to select all blocks in a method. Rather, we sample a subset of blocks (up to five) from each method. To select lambdas, we collect all lambdas from all projects, sort them by the number of branching conditions that must be satisfied to reach each lambda, and choose the top 200 with more than one statement. Using branching conditions increases the chance to find fragments that unit tests do not reach.

Table 4. Summary statistics on complexity metrics for target fragments that we evaluate. *Conditional statements*: no. of levels of conditional statements in which the target fragment is nested. *Cyclomatic complexity*: McCabe's cyclomatic complexity of the method containing the target fragment. *Method size*: no. of lines of code in the method containing the target fragment.

	Conditional statements	Cyclomatic complexity	Method size
Mean	3.6	160.8	376.7
Med	3	106	210
Min	0	1	11
Max	33	4,319	8,827

Table 4 shows summary statistics on complexity metrics for target fragments that we evaluate. These 200 blocks and 200 lambdas are at level 3.6 of nested conditionals on average (max: level 33), they are in methods with average McCabe's cyclomatic complexity of 160.8 (max: 4,319) and those methods have 376.7 lines of code on average (max: 8,827 lines of code).

Running Experiments. We use BDK's Maven extension to integrate BDK in the evaluated projects and write scripts to analyze the results. All experiments are run in Docker containers; our artifacts contain our Docker file and instructions on how to use it. Timed experiments are run on an Intel® Xeon® w9-3475X (72 threads) CPU, 128 GB RAM, Ubuntu 24.04, Java 8, and Maven 3.8.9.

4.1 RQ1: Feasibility of Block Testing in Open-Source Projects

We attempt to manually write block tests for all 400 fragments (blocks and lambdas) that we select as described earlier. We impose a time limit of one hour to do so per fragment. If the total time to understand a target fragment and write block tests for it exceeds this limit, we stop and mark the fragment as being hard to validate with block tests. For each fragment, we aim to achieve 100% statement coverage. Many other criteria exist beyond statement coverage [1, 35, 110]. We focus on statement coverage for simplicity. We successfully write block tests for 346 fragments—178 (of 200) blocks and 168 (of 200) lambdas. These 346 fragments are from 231 files in 146 projects, and contain a total of 3,530 lines of code.

Table 5. 54 fragments that we fail to write block tests for.

Reason	Block	Lambda	Total
Requires domain knowledge	15	18	33
Uses generics for non-input variables	0	1	1
Conditional return statement	1	1	2
Protected class or member	3	4	7
Non-static methods or variables	3	8	11

Table 5 summarizes why we are unable to write block tests for 54 (of 400) fragments. There, the first row has nothing to do with BDK: we fail to write tests for 33 fragments because we lack domain knowledge to set up block test inputs. For example, block testing a fragment may require a complex input object that we do not know how to instantiate. We anticipate that lack of domain knowledge (the main reason we fail to write block tests) would be less of an issue for developers, who are more familiar with their code. The last four rows in Table 5 represent failures caused by the current limitations of BDK (described in §3.3.2). Specifically, BDK does not currently support (i) the use of generic types in target fragments when they are not input variables (1 case); (ii) having return values with different declared types in the target fragment (2 cases); and (iii) using non-public (7 cases) or non-static (11 cases) methods or classes in target fragments.

Summary Statistics on Block Tests That We Write. In total, we write 1,012 block tests (638 for blocks and 374 for lambdas). Among the 346 target fragments, we write only one block test for 99

Table 6. Our cost of manually writing block tests for fragments using BDK (time in seconds).

	Understand	Write	Total
Mean	256.9	135.2	392.1
Med	157.5	128	289.5
Min	0	15	15
Max	2,275	773	2,756
Sum	88,899	46,765	135,664

Table 7. Cost of running BDK with no duplicates and with block tests duplicated 10 to 1,000 times.

#	Σ Time (s)	Avg. Time (s)
1,012	915.5	0.9
10,120	1,027.6	0.1
101,200	1,910	0.02
1,012,000	11,442.3	0.01

(28.6%). On average, each target fragment has 2.9 block tests. The target fragment with the maximum number of 24 block tests contains a very long `if` statement with many `else if` branches [72]. We analyze BDK's features and APIs that we use in writing all these block tests. Among the 1,012 block tests, 43 use the `checkFlow` control flow oracle and 64 are exceptional behavior block tests that use the `expect()` oracle. The remaining block tests use only data flow oracles—1,521 data flow oracles to be precise. Qualitatively, we find that control flow oracles are particularly helpful for validating target fragments whose behaviors are difficult to validate using oracles that compare data values. For example, code that prints error messages to standard output cannot be easily validated using data flow oracles. But, a control flow oracle can validate that the code containing a print statement is reached.

Among the 346 fragments that we successfully write block tests for, only 17 (4.9%) fragments require us to manually specify input variable types. In these 17 fragments, we provide types for 21 block tests (2% of all 1,012 block tests). This finding shows that: (i) BDK's type resolution succeeds for majority of fragments even without analyzing bytecode, and (ii) most fragments that require manual type specification result from JavaParser returning an incorrect variable type, so BDK does not fall back to resolving the type based on value because JavaParser has already resolved one (albeit wrongly). The small number of such fragments gives us some confidence in BDK's type resolution to be effective for many open-source projects.

We conclude from our ability to write all these block tests for a diverse set of target fragments that block tests are feasible for validating multi-statement sequences in methods. Note again that many target fragments that we write block tests for are complex and not reached by unit tests.

4.2 RQ2: Costs of Writing and Running Block Tests

Process. We evaluate the costs to write and run block tests using BDK. To do so, we measure the time to (i) gain basic understanding of target fragments, (ii) write block tests using BDK, and (iii) run block tests using BDK. The first two steps incur human time to write block tests, while the third incurs machine time to run block tests. To measure test-writing time, we start a timer immediately after opening the file containing the target fragment. We then spend time to understand the fragment, stopping the timer once we have sufficient understanding and record the time elapsed as the time to understand that fragment. Next, we restart the timer and start editing the fragment to add block tests. During this process, we may run the block tests iteratively to ensure that their oracles work as expected, revising the tests as needed until all block tests pass. We stop the timer after all block tests that we write pass and record the time elapsed as the time to write block tests for that fragment.

Results: Time to Write Block Tests. Table 6 shows the arithmetic mean (Mean), median (Med), minimum (Min), maximum (Max), and total (Sum) time to understand target fragments (Understand), write block tests (Write), and the combined time (Total). We compute these summary statistics per project (*i.e.*, values in the same row in Table 6 may correspond to different projects). In total,

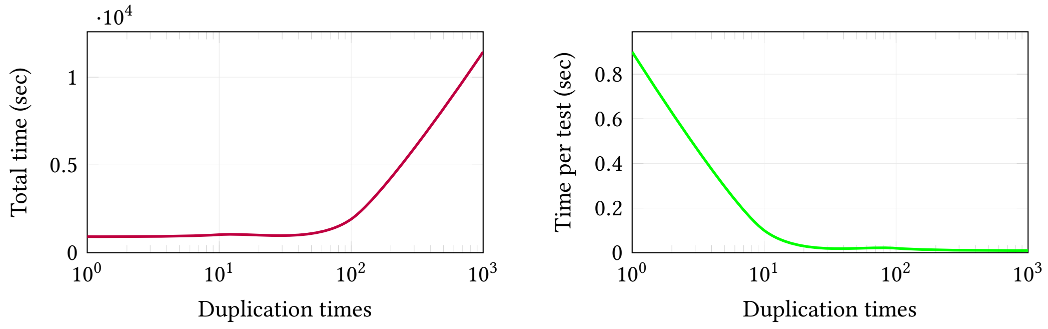

Fig. 14. Duplication times vs. total time (left) and average time (right).

we spend 37.7 person hours to write 1,012 block tests for 346 fragments: 24.7 hours to understand target fragments and 12.9 hours to write block tests. Excluding time to understand the fragments, each block test only takes 46 seconds to write on average; that average per block test is 2.2 minutes when time to understand fragments is included. Note that per-test times are not shown in Table 6; that table presents times that we aggregate across each fragment.

Understanding the fragments takes a big portion of our total human time. Developers are unlikely to need that much time for their own code. Anecdotally, it took us much less time to understand these fragments than it typically takes us to understand whole methods. Notably, one fragment takes only 15 seconds to write a block test for, because we were already familiar with the fragment while writing tests for another fragment in the same class. So, we do not need additional time for understanding the fragment (*i.e.*, the time to understand it was zero). However, the maximum time to write a block test for a fragment is 773 seconds because even after understanding the fragment, it was still hard for us to determine correct expected values in our block test oracles. In such cases, we rerun the tests multiple times with different expected values until we find valid ones. So, we also expect developers to spend less time than us when writing block tests, after gaining familiarity with BDK's APIs.

Results: Time to Run Block Tests. Table 7 shows the time BDK takes to run block tests. When each block test is run only once, the average time to run each block test is only 0.9 seconds. To simulate an evaluation of how well BDK may scale as the number of tests increases, and since ours are only the first set of block tests, we duplicate each block test 10, 100, and 1,000 times. After duplication, the average time to run each block test decreases to 0.01 seconds. Figure 14 shows how the total and average block-test running times change as the number of tests increases. We observe that for many fragments, the time for Test Runner in step ④ in Figure 11 to execute tests remains constant, likely due to just-in-time compilation, which Liu et al. also noted for inline tests [67]. But, for six fragments, the total running time grows linearly after duplicating 100 times. The reason is that each block test runs in a fixed amount of time regardless of the total number of tests. We also find that the decrease in average time per block test between no duplication and duplicating 100 times occurs because the time required to process the block test statements in input Java files (steps ①, ②, and ③ in Figure 11) is mostly constant. So the cost of parsing Java files and extracting block tests is amortized across multiple executions when we perform duplication.

Results: Overhead of Block Tests. BDK strips out block-test code before compilation. So, there should be zero overhead in production. Next, we evaluate the overhead of retaining block tests in the code when the option to strip out block-test code is manually disabled. We run all 47,013 unit tests in all projects while block tests are in the code (block-testing mode) and without block tests in the code (non-block-testing mode). Testing mode took 2,714.1 seconds (summed across all projects), while non-testing mode took 2,706.2 seconds. So, the overhead seems negligible.

Table 8. Coverage statistics for developer written unit tests ("Dev tests"), developer written and automatically generated unit tests ("Dev and auto tests") for 326 fragments, and block tests for 346 fragments. Statement % and Branch % represent the percentage of statement coverage and branch coverage, respectively.

Source	Counts			Statement %		Branch %	
	Fully covered	Partially covered	Not covered	Mean	Med	Mean	Med
Dev tests	63	60	203	28.2	0	27.2	0
Dev and auto tests	85	82	159	38.8	4	44	33.3
Block tests	308	38	0	93.2	100	97.6	100

We conclude from our results on the costs of writing and running block tests that block testing is not just feasible, it can also be efficient and fit well in typical software development workflows. Also, the overhead of having block-test code in production is negligible.

4.3 RQ3: Comparing Coverage of Target Fragments by Unit and Block Tests

We do not select target fragments based on their coverage (or lack thereof) by unit tests. So, here we use JaCoCo [81] to measure the statement and branch coverage of block tests in the target fragments and compare these coverage metrics with those of developer written and automatically generated unit tests.

Process. In JaCoCo, a line of code is fully covered, partially covered (if only one outgoing edge from a branch is taken), not covered, or empty. Empty means that a line contains a continuation of the statement in the preceding line or does not contain a statement (*e.g.*, a closing brace). We ignore empty. We say a *fragment* is (i) fully covered if all its non-empty lines are fully covered; (ii) not covered if none of its non-empty lines are covered; and (iii) partially covered if it does not fall into any of the previous two cases. We compute statement coverage of a block as the ratio of its partially or fully covered lines to the total number of non-empty lines. For branch coverage, we compute the ratio of the number of covered branches to the total number of branches. We use Randoop [85, 88] and EvoSuite [15] to automatically generate additional unit tests for each fragment. We run EvoSuite only on the class containing the target fragment, but we run Randoop for the whole project based on the Randoop manual [89]. We use timeouts of 1 minute per class for EvoSuite and 1 hour per project for Randoop. After unit-test generation, we merge developer written and automatically generated unit tests and use JaCoCo to collect coverage.

Results. Table 8 shows coverage statistics for developer written unit tests, and of their combination with automatically generated unit tests. We are only able to collect coverage for developer tests in 326 of 346 fragments. JaCoCo failures prevent us from collecting coverage on the rest. We use JaCoCo despite these failures because it is a state-of-the-art coverage collection tool for Java, is integrated into popular IDEs [41], and similar issues were reported in prior work [28]. 203 of these 326 fragments are not covered by any developer written unit test. The mean statement and branch coverage achieved by developer written unit tests in these fragments are only 28.2% and 27.2%, respectively. At least half of the fragments are not covered (the median statement coverage is 0%). Even after 30.8 CPU days of automatic unit-test generation for 326 fragments (average: 2.2 CPU hours per fragment), statement coverage improves only marginally. More concerning, EvoSuite and Randoop crashed when generating tests for 34 fragments, which reside in complex classes. The average statement coverage increased only by 10.6 percentage points, and the median slightly improved by 4 percentage points. Automatically generated unit tests improve average branch coverage by 16.8 percentage points, and improve median branch coverage by 33.3 percentage points. Some fragments have higher statement coverage when automatically generated unit tests are

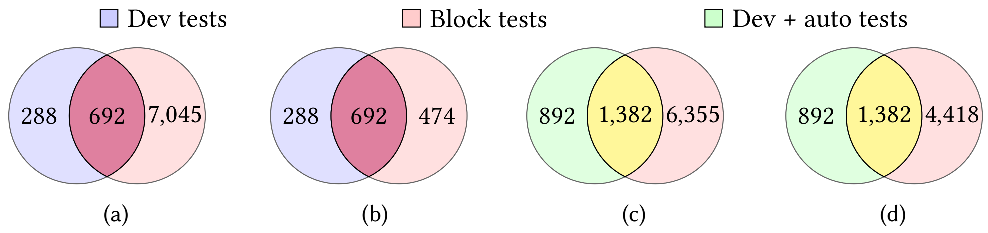

Fig. 15. Left to right; mutants killed by (a) developer written unit tests ("Dev tests") vs. block tests for all fragments, and (b) Dev tests vs. block tests for fragments covered by at least one Dev test, (c) Dev tests and automatically generated unit tests ("Dev + auto tests") vs. block tests for all fragments, and (d) Dev + auto tests vs. block tests for fragments covered by at least one Dev test or automatically generated unit tests.

combined with developer written ones, but around half of the fragments still have no coverage with unit tests. The last row in Table 8 shows coverage statistics for block tests: an average statement coverage of 93.2%, with fragments that achieve full statement coverage. Branch coverage shows a similar pattern, with an average branch coverage of 97.6% and 318 fragments achieving full branch coverage. The minimum statement and branch coverage achieved by all unit tests are both 0%. For block tests, the minimum statement and branch coverage are 4.2% and 10%, respectively. Compared to the high average time to generate unit tests, the time to manually write block tests is low (392.1 seconds) and produces much higher statement coverage in these fragments.

We make two conclusions from these coverage experiments. First, these results strongly support our claim that block tests can meet existing needs that unit tests do not meet. Second, many of these fragments being lambdas also provides initial strong evidence that block tests can help validate code in the presence of complexity arising out of the use of newer programming language features.

4.4 RQ4: Bug-finding Effectiveness

Lastly, we turn to the questions of (i) whether block tests' increased coverage of target fragments provides bug-finding benefits; and (ii) how those bug-finding benefits compare with those of unit tests in parts of these evaluated fragments that unit and block tests cover.

Process. We use mutation analysis as a proxy for evaluating bug-finding effectiveness [45]. Specifically, we apply the default set of mutators (changing operators or operands, adding or removing statements, etc.) in `universalmutator` [26] to each statement in the target fragment. Then, we keep only mutants that compile, resulting in 11,828 mutants for 295 fragments (average: 40 mutants per fragment). Next, we record which mutants are killed by developer written unit tests, automatically generated unit tests, and block tests. Finally, we report our qualitative findings based on our manual inspection of 104 mutants: 52 killed only by block tests and 52 killed only by unit tests.

Results. Of all 11,828 mutants, 980 are killed by only developer written tests, 2,274 mutants are killed by combining developer written and automatically generated unit tests, and 7,737 mutants are killed by block tests.

Figure 15 compares the sets of mutants killed by developer written unit tests, automatically generated unit tests, and block tests. Venn diagram regions show number of mutants killed by each kind of tests. Considering all fragments (Figure 15a), block tests kill many more mutants than developer written unit tests. Specifically, block tests cover more statements and therefore kill 6,757 more mutants than developer written unit tests in these fragments. For statements covered by both kinds of tests (Figure 15b), block tests kill about 1.6x as many mutants as developer written unit tests. Comparing the set of mutants killed by developer written + automatically generated unit tests with that of block tests (Figure 15c) shows that automatically generation helps unit tests kill more mutants than developer written unit tests alone, but the number of killed mutants is still lower than that of block tests. Lastly, block tests kill 5x as many mutants as developer written and automatically generated unit tests for the fragments that unit and block tests cover (Figure 15d). These findings suggest that factors besides coverage contribute to block tests' bug-finding effectiveness.

Reasons (beyond coverage) why block tests kill more mutants than unit tests. Figure 16 shows an example mutant that is killed by a block test but not by any unit test. In the original program, the fragment first sets `writeOptimal` to `true`, and then modifies its value based on `variant`, `csm` and `numBands`. This mutant is obtained by deleting a statement on line 4, which assigns `true` to `writeOptimal`. The block test on lines 1-3 checks that by the end of the fragment, the value of `writeOptimal` is `true`. `writeOptimal` originally has value `false` (via the `given` expression). When the block test is run on this mutant, line 4 is skipped and the condition on line 5 is checked. Since the block test assigns `variant` to `PPM_ASCII` (via the `given` expression), `variant` is not `PPM_RAW` so the condition on line 5 is `false`, skipping the rest of the fragment, making `writeOptimal`'s value to be `false`. Thus, the block test fails and the mutant is killed. Unit tests achieve full statement coverage but not full branch coverage for this fragment. But, these unit tests do not kill this mutant because `writeOptimal` is neither a parameter nor an output of the enclosing method, and the erroneous value of `writeOptimal` goes undetected. Block tests, on the other hand, have very local scope, so they can specify local oracles to check the value of `writeOptimal` before it is used in other parts of the program. So, block tests enable errors to be propagated and revealed more easily. We observe similar patterns in 12 out of 52 mutants that are killed by block tests but not unit tests. In the other 40 cases, unit tests lack coverage for the mutated lines in the fragment, so they cannot kill these mutants.

```
1  blocktest().given(writeOptimal, false).given(numBands,5)
2       .given(variant, PPM_ASCII)
3       .given(csm, null).checkTrue(writeOptimal);
4  /*writeOptimal = true;*/  // mutation: delete this statement
5  if(variant == PPM_RAW) {
6      int[] bandOffsets = csm.getBandOffsets();
7      for(int b = 0; b < numBands; b++) {
8          if(bandOffsets[b] != b) {
9              writeOptimal = false;
10             break;
11         }
12     }
13 }
```

Fig. 16. An example mutant from `jai-imageio/jai-imageio-core` that is killed by block test but not by unit tests. The mutation on line 4 is to delete a statement in the fragment under test.

Reason why unit tests kill mutants that survived block tests. We find two reasons why block tests did not kill some mutants that unit tests kill. First, in majority of these cases (50 of 52), extra block tests are needed to achieve full branch coverage for the fragment, in addition to our manually written block tests that target full statement coverage. The other two inspected fragments are in a loop. Block tests written inside a loop are not able to test beyond an iteration of the loop. Figure 17 shows an example mutant killed by unit tests but not by

```
1  while (!terminate) {
2      ... // more code...
3      blocktest().given(scheme,"file").checkEq(state,PUS.FH);
4      blocktest().given(scheme,"http").checkEq(state,PUS.AIS);
5      if ("file".equals(scheme)) {
6          state = PUS.FH;
7      } else {
8          state = PUS.AIS;
9          continue;  // mutation: add a continue statement
10     }
11     ... // more code...
12 }
```

Fig. 17. A mutant from `smola/galimatias` block tests did not kill. Line 9 is introduced by adding a `continue`.

block tests from project smola/galimatias. The mutant adds a continue statement to line 9. The enclosing while loop on line 1 implements the core logic of a character-by-character URL parser. This mutant is not killed by the block tests (on lines 3 to 4) written for this fragment because those block tests only test the fragment from lines 5 to 10 but nothing beyond. Unit tests, on the other hand, are able to kill this mutant because the effect of the mutation caused the URL processed with multiple loop iterations to differ from what is specified in a unit test oracle.

5 Discussion

Future Work. We outline an agenda for future research on block tests, beyond those in §3.3.2.
Automated generation of block tests. The manual approach for writing block tests that we take in this paper is meant to show that they are feasible. But, to accelerate research on block tests, future work must investigate automated generation of block tests to reduce the tedium of manually writing them and to scale block tests to more projects. Many techniques were proposed for automatic unit test generation [15, 22, 84, 85, 94, 102] and two have been proposed for inline tests [44, 66, 68]. So a technique to generate block tests automatically will likely also be valuable. Future research can explore whether symbolic execution [108] (given the small scope of block tests) and/or carving block tests from higher-granularity tests [7, 9, 10, 17, 48, 66] can work well for automatically generating block tests. To improve the quality of automatically generated tests, future work can also explore techniques for automatically mocking non-static methods and inferring appropriate and realistic values for mocks and global variables.
User study. We did not conduct a user study in this paper because prior work on the lower-granularity test level of inline tests [67] showed that participants are willing to adopt finer-granularity test granularity. Given the structural similarity of block and inline tests, we conjecture that users will also find block tests useful, especially if they are already familiar with unit tests. So, this paper has focused on establishing foundations for block tests. But, since block tests and inline tests differ in both granularity and API, future work should conduct a user study to evaluate participants' perceptions of block tests.
IDE support. Like inline tests, the presence of block tests in code can affect readability. To mitigate this issue, as part of future work, we plan to provide IDE support in BDK that hides block tests from view until developers want to read them. As part of this IDE support, BDK can also integrate with IDE refactoring engines [14, 40, 42, 57] so that when developers refactor their code using the IDE, relevant parts of the corresponding block tests are also automatically updated.
Research on other testing needs that motivate block tests. We provide evidence that block tests can meet two of six testing needs that we use to motivate this new test granularity level (§1). But, there is still plenty of research to be done on truly realizing the use of block tests for addressing all these needs. Automatic generation of block tests will be a logical next step that can enable several research directions on using block tests to meet these other needs.
Coverage beyond statements and branches. Our evaluation currently relies on statement and branch coverage to determine whether code is executed. However, many other criteria exist beyond statement and branch coverage [35, 110], such as condition or instruction coverage. Future work can investigate how block tests can further enhance the satisfaction of these other criteria as well.
Improved type resolution. Type resolution enables developers to define input variables without explicitly specifying their types. But, BDK's current type resolution sometimes fails (in 4.9% of fragments, or 2% of block tests), as discussed in §4.1. Prior work showed that machine learning can help type resolution [79, 86] for Python, so work can explore this direction to improve BDK's type resolution. Also, BDK's current project-defined type resolution already operates at the source and bytecode levels to accommodate scenarios where full projects do not compile. So, future work

could leverage semantic information from the full project to further improve the accuracy of type resolution when complete build information is available. For example, bytecode analysis could enable more robust type resolution.

Block tests vs. refactoring code into finer-grained units. While block tests may be unnecessary if developers refactor code into finer-grained units that are easier to unit test, such refactoring may have a greater negative impact on readability than having block tests in the code. Specifically, extracting multiple target fragments from an original method into several new methods solely for testing purposes (*i.e.*, not as part of intended refactoring) forces developers to navigate multiple methods to understand the same business logic. Prior research showed that avoiding this kind of negative impact on readability is one reason why developers write lambda expressions in existing methods rather than writing new methods [73]. So, block tests provide a valuable alternative.

Threats to validity. Our results may not generalize beyond the projects that we evaluate. To address this threat, we use fragments from open-source projects that were used in prior work on testing and pick diverse set of fragments from those projects. Another threat is that the coverage result of automatically generated unit tests may not always be reproducible due to differences in physical machines. To deal with this threat, we set a sufficiently long timeout for unit-test generation to aid reproducibility. We also repeat our test generation experiments multiple times. Our BDK implementation may have bugs. To mitigate this threat, multiple co-authors reviewed the code and a different co-author than those who wrote the code performed code review and checked the documentation of BDK's APIs to check for inconsistency. Finally, the idea of block tests may not extend to all programming languages, and even when it does, it may not provide the same benefits that we see for Java. To mitigate this threat, we investigated block tests for a popular programming language that bears similarity to several other C-like languages. We also showed that block tests can validate code written in imperative style, functional style, and their mixture.

6 Related Work

Testing code fragments. The work most closely related to ours is Liu et al.'s inline tests [67]. Like block tests, inline tests are fine grained tests; they allow developers to write tests at a granularity level that is below unit tests. However, inline tests are limited to testing individual statements, whereas block tests are more general and target the validation of multi-statement fragments in a method. Godefroid proposed the idea of micro execution [23], a technique that can execute any code fragment without user inputs. However, Micro Execution requires using a custom virtual machine, is designed for test generation campaigns, and works only on binaries. In contrast, BDK is more flexible, allows user (*i.e.*, developer) input, works on source code and requires no custom virtual machine. Other researchers have also explored finer-grained code elements during testing. Galindo et al. propose method-specific test generation to test individual methods [16]. Method-specific test generation enables test generation tools to produce tests with human-understandable names and can even help improve test coverage. Unlike other automatic unit test generation techniques, such as Randoop [84, 85] and EvoSuite [15], which generally generate tests for entire classes or projects, Galindo et al. focus on generating tests for individual methods. While their approach targets individual methods, block tests are more fine-grained and focus on fragments in a method.

Intramorphic testing. Rigger and Su introduce intramorphic testing [92], a technique designed to address the test oracle problem. This approach modifies the system under test so that, for a given input and the original system's output, an input-specific oracle for the output of the modified system can be derived. One then checks whether this expected relationship holds. Like block tests, intramorphic testing provides a novel way to exercise code under test. However, block tests allow developers to test specific fragments using test oracles, whereas intramorphic testing is applied in settings where a test oracle is difficult to obtain and is not focused on testing fragments.

Live programming. Live programming [11, 12, 32, 49, 51, 52, 56, 63, 75, 76, 90, 96, 107] enables developers to visualize a program's runtime behavior as the code is being developed. Similar to live programming, block testing helps developers detect bugs early, before the program is fully implemented. But, live programming requires a continuous evaluation environment to provide immediate feedback. By providing program inputs, developers can see the program's outputs updated in real time and visually check whether its behavior matches their expectation while writing code. Block tests are more focused on testing existing code and require a test oracle, whereas in live programming, developers typically manually or visually judge whether outputs are as expected. That is, there is no automated oracle. As with block testing, high-quality inputs are also crucial for live programming. Prior work explored techniques for generating useful input values for incomplete loops using loop seeds [63]. Developers can manually provide loop seeds—values used to initiate hypothetical loop iterations—to help visualize the behavior of those iterations. Other work investigated techniques that automatically synthesize fragments based on user-defined inputs, outputs [12], and scope [52]. Building on these approaches, future work on block tests can explore how to generate high-quality block tests for synthesized loop fragments and how to synthesize fragments from block test statements.

Regression testing. The goal of regression testing is to detect bugs introduced by code changes [4, 18–21, 31, 59, 61, 70, 97–99, 113–115]. Similar to regression testing, the goal of block tests is to find bugs during software testing. But traditionally, regression testing is done at the unit-test level. So future work can investigate regression testing with block tests, since our mutation analysis shows that block tests can help find faults that unit tests miss.

7 Conclusions

Block tests introduce a new level of test granularity that are complementary to existing test granularity levels: inline, unit, integration, and system tests. The goal of block tests is to allow for easier validation of code fragments, *i.e.*, multi-statement sequences in methods. We motivate the need for block tests by discussing several testing needs that current levels of test granularity are either too coarse grained or fine grained. Our implementation of BDK, the first block-testing framework, supports features identified in our formative study. More importantly, we use our BDK to evaluate the feasibility, coverage benefits, efficiency, and bug-finding benefits of block tests on a diverse set of 346 fragments from 146 open-source projects. Results show that block tests are broadly applicable and fast to write and run. Also, block tests more easily reach fragments that unit tests do not reach, and kill many mutants that survive unit tests. Finally, we highlight many exciting research problems that should be tackled as part of work to make block testing more widely used among software engineers.

Acknowledgments

We thank Saikat Dutta, Shinhae Kim, Yu Liu, David Langus Rodriguez, and the anonymous reviewers for their help, comments, and feedback. This work is partially supported by the US NSF under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036.

References

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press.

[2] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Simon and Schuster.

[3] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2018. Java & lambda: a featherweight story. *LICS* 14.

[4] Lingchao Chen and Lingming Zhang. 2018. Speeding up mutation testing via regression test selection: An extensive study. In *ICST.*

[5] Mike Cohn. 2010. *Succeeding with agile: software development using Scrum.* Pearson Education.

[6] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *ISSRE.*

[7] Amirhossein Deljouyi and Andy Zaidman. 2023. Generating understandable unit tests through end-to-end test scenario carving. In *SCAM.*

[8] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic complexity. *IEEE software* 33, 6.

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

[10] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Matthew Jorde. 2008. Carving and replaying differential unit test cases from system test cases. *TSE* 35, 1.

[11] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: interactive program synthesis with control structures. In *OOPSLA.*

[12] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-step live programming by example. In *UIST.*

[13] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a promising future. In *COORDINATION.*

[14] Stephen R Foster, William G Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *ICSE.*

[15] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *FSE.*

[16] Carlos Galindo, Manuel Gregorio, and Josep Silva. 2024. Addressing EvoSuite's Limitations: Method-Specific Test Case Generation and Execution in Controlled Environments. In *VALID.*

[17] Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev, and Mattia Fazzini. 2023. Action-based test carving for android apps. In *ICSTW.*

[18] Milos Gligoric. 2015. *Regression Test Selection: Theory and Practice.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA.

[19] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *ICSE Demo.*

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

[21] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression Test Selection for Distributed Software Histories. In *CAV.*

[22] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *POPL.*

[23] Patrice Godefroid. 2014. Micro execution. In *ICSE.*

[24] Mark Grechanik and Gurudev Devanla. 2019. Generating integration tests automatically using frequent patterns of method execution sequences. In *SEKE.*

[25] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *ICST.*

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

[27] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster explicit-trace monitoring-oriented programming for runtime verification of software tests. In *OOPSLA.*

[28] Kevin Guan and Owolabi Legunsen. 2024. An In-depth Study of Runtime Verification Overheads during Software Testing. In *ISSTA.*

[29] Kevin Guan and Owolabi Legunsen. 2025. Instrumentation-Driven Evolution-Aware Runtime Verification. In *ICSE.*

[30] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *FSE Demo.*

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

[32] Christopher Michael Hancock. 2003. *Real-time programming and the big ideas of computational literacy.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[33] Harium 2018. Bug report. https://github.com/Harium/keel/commit/e4e6a302eb6b7dcf6dab5d1167cf6ceef836d4db.

[34] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2024. Feedback-directed partial execution. In *ISSTA*.

[35] Hadi Hemmati. 2015. How effective are code coverage criteria?. In *QRS*.

[36] Jeff Huang, Qingzhou Luo, and Grigore Roşu. 2015. GPredict: Generic Predictive Concurrency Analysis. In *ICSE*.

[37] Jeff Huang, Patrick O'Neil Meredith, and Grigore Roşu. 2014. Maximal sound predictive race detection with control flow abstraction. In *PLDI*.

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

[39] JavaParser Team 2025. JavaParser - Home. https://javaparser.org.

[40] JetBrains IntelliJ 2026. Code refactoring. https://www.jetbrains.com/help/idea/refactoring-source-code.html.

[41] JetBrains IntelliJ 2026. IntelliJ IDEA can use JaCoCo to collect coverage. https://www.jetbrains.com/help/idea/code-coverage.html.

[42] JetBrains Research 2023. RefactorInsight Plugin. https://github.com/JetBrains-Research/RefactorInsight.

[43] Pengyue Jiang, Kevin Guan, Mahdi Khosravi, Moustafa Ismail, Marcelo d'Amorim, and Owolabi Legunsen. 2026. Fine-Grained Analyses for Evolution-Aware Runtime Verification. In *ICSE*.

[44] Pengyue Jiang, Yu Liu, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2026. Automated Inline-Test Generation without Relying on Method-Level Unit Tests. In *ECOOP*.

[45] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*.

[46] Gilles Kahn. 1987. Natural semantics. In *STACS*.

[47] 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*.

[48] Alexander Kampmann and Andreas Zeller. 2019. Carving parameterized unit tests. In *ICSECompanion*.

[49] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *UIST*.

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

[51] Saketh Kasibatla and Alessandro Warth. 2017. Seymour: Live Programming for the Classroom. In *LIVE*, Vol. 2017.

[52] Tomer Katz and Hila Peleg. 2025. ScooPy: Enhancing Program Synthesis with Nested Example Specifications. In *Onward*.

[53] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2025. Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring. In *ASE*.

[54] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2026. Valg: A Fast Reinforcement Learning-Based Runtime Verification Tool for Java. In *ICSE Demo*.

[55] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *PLDI*.

[56] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *VLHCC*.

[57] Zarina Kurbatova, Vladimir Kovalenko, Ioana Savu, Bob Brockbernd, Dan Andreescu, Matei Anton, Roman Venediktov, Elena Tikhomirova, and Timofey Bryksin. 2021. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. In *ASE*.

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

[59] 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*.

[60] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE*.

[61] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE Demo*.

[62] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*.

[63] Sorin Lerner. 2020. Focused live programming with loop seeds. In *UIST*.

[64] Hareton K.N. Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *ICSM*.

[65] LicenceMavenPlugin 2016. Fix for String index out of range: -1. https://github.com/mojohaus/license-maven-plugin/pull/28.

[66] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *ISSTA*.

[67] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline tests. In *ASE*.

[68] Yu Liu, Aditya Thimmaiah, Owolabi Legunsen, and Milos Gligoric. 2024. ExLi: An Inline-Test Generation Tool for Java. In *FSE Demo*.

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

[70] 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*.

[71] Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-testing: unit testing with mock objects. *Extreme programming examined*.

[72] maxmind/geoip-api-java 2026. A block with many conditonal statements. https://github.com/maxmind/geoip-api-java/blob/0ed8a38981512667533d47204486aae0278a1bc8/src/main/java/com/maxmind/geoip/timeZone.java.

[73] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. In *OOPSLA*.

[74] Thomas J McCabe. 1976. A complexity measure. *TSE* 4.

[75] Sean McDirmid. 2007. Living it up with a live programming language. *ACM SIGPLAN Notices* 42, 10.

[76] Sean McDirmid. 2013. Usable live programming. In *Onward*.

[77] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in Java: challenges and vulnerabilities. In *ICSE*.

[78] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[79] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for Python. In *ICSE*.

[80] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.

[81] Mountainminds GmbH & Co. KG 2009. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.

[82] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *ICSE*.

[83] Alessandro Orso. 1998. Integration testing of object-oriented software. Citeseer.

[84] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *OOSPLA Companion*.

[85] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*.

[86] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for Python. In *ICSE*.

[87] G. D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report FN-19. DAIMI, Aarhus University.

[88] M. Pradel and T. R. Gross. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*.

[89] Randoop Team 2026. Randoop. https://randoop.github.io/randoop/.

[90] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *arXiv preprint arXiv:1902.00549*.

[91] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: Mining single statement bugs at massive scale. In *MSR*.

[92] Manuel Rigger and Zhendong Su. 2022. Intramorphic testing: A new approach to the test oracle problem. In *Onward*.

[93] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: efficient predictive race detection. In *PLDI*.

[94] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *SSBSE*.

[95] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4.

[96] Mark Santolucito, William T Hallahan, and Ruzica Piskac. 2019. Live programming by example. In *CHIEA*.

[97] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. In *OOPSLA*.

[98] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *FSE*.

[99] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *ISSRE*.

[100] Beatriz Souza and Michael Pradel. 2023. Lexecutor: Learning-guided execution. In *FSE*.

[101] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? an empirical study on mocking practices. In *MSR*.

[102] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.

[103] Fabian Trautsch. 2019. *An analysis of the differences between unit and integration tests*. Ph. D. Dissertation.

[104] W.T. Tsai, Xiaoying Bai, R. Paul, Weiguang Shao, and V. Agarwal. 2001. End-to-end integration testing design. In *COMPSAC*.

[105] Raoul-Gabriel Urma, Alan Mycroft, and Mario Fusco. 2018. *Modern Java in Action: Lambdas, streams, functional and reactive programming*. Simon and Schuster.

[106] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe futures for Java. In *OOPSLA*.

[107] Eric M Wilcox, J William Atwood, Margaret M Burnett, Jonathan J Cadiz, and Curtis R Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *CHI*.

[108] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*.

[109] Rahulkrishna Yandrapally and Ali Mesbah. 2021. Mutation analysis for assessing end-to-end web tests. In *ICSME*.

[110] Qian Yang, J Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *AST*.

[111] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2.

[112] Ayaka Yorihiro, Pengyue Jiang, Valeria Marques, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven Plugin for Evolution-Aware Runtime Verification. In *RV*.

[113] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and combining analysis-based and learning-based regression test selection. In *AST*.

[114] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE*.

[115] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*.

[116] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. Mocksniffer: Characterizing and recommending mocking decisions for unit tests. In *ASE*.

Received 2025-10-10; accepted 2026-02-17