Inline Tests

Yu Liu
UT Austin, USA
yuki.liu@utexas.edu

Pengyu Nie
UT Austin, USA
pynie@utexas.edu

Owolabi Legunsen
Cornell University, USA
legunsen@cornell.edu

Milos Gligoric
UT Austin, USA
gligoric@utexas.edu

ABSTRACT

Unit tests are widely used to check source code quality, but they can be too coarse-grained or ill-suited for testing individual program statements. We introduce *inline tests* to make it easier to check for faults in statements. We motivate inline tests through several language features and a common testing scenario in which inline tests could be beneficial. For example, inline tests can allow a developer to test a regular expression in place. We also define language-agnostic requirements for inline testing frameworks. Lastly, we implement I-Test, the first inline testing framework. I-Test works for Python and Java, and it satisfies most of the requirements. We evaluate I-Test on open-source projects by using it to test 144 statements in 31 Python programs and 37 Java programs. We also perform a user study. All nine user study participants say that inline tests are easy to write and that inline testing is beneficial. The cost of running inline tests is negligible, at 0.007x–0.014x, and our inline tests helped find two faults that have been fixed by the developers.

CCS CONCEPTS

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

KEYWORDS

inline tests, software testing

ACM Reference Format:
Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline Tests. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556952

1 INTRODUCTION

Testing is essential for checking code quality during software development. Today, testing frameworks only support three levels of test granularity—unit testing, integration testing and end-to-end testing. These levels, shown in the top three layers of Figure 1 (known as the test pyramid), reflect developer testing needs. Developers write unit tests to check the correctness of logical units of functionality, e.g., methods or functions [15, 77]. Integration tests are used to check that logical units interact correctly [32, 54, 69, 95]. Developers use end-to-end tests to check if code runs correctly in its operating environment, and if functional and non-functional requirements are being met [96, 100].

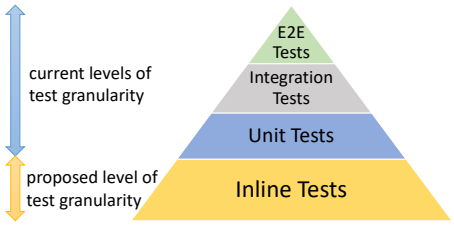


Figure 1: Testing pyramid.

Unfortunately, there is little support for developer testing needs below the unit-test level. Yet, developers may want to test individual statements for at least four reasons:

(1) Single-statement bugs occur frequently [38, 39], but unit tests rarely fail on commits that introduce single-statement bugs [47].
(2) The statement to be checked, i.e., the *target statement*, may be buried deeply inside complicated program logic.
(3) Developers may want to check and better comprehend harder-to-understand traditional programming language features like regular expressions (regexes) [16, 17, 42, 62, 101], bit manipulation [4, 51], and string manipulation [20, 46, 74].
(4) Recent language features, e.g., Java's stream API [18], allow writing complex program logic in one statement where one would previously have written a method that can be unit tested.

Due to the lack of direct support for statement-level testing, developers often resort to wasteful or *ad hoc* manual approaches. We briefly mention three of them here and describe them and others in Section 2. First, in the commonly-practiced "`printf` debugging" [5, 10, 31, 36, 55, 73], developers wastefully add and then remove print statements to visually check correctness at specific program points. Second, if the target statement is in privately accessible code, some developers violate core software engineering principles to enable checking them with unit tests. For example, `google/guava` [30] developers use the "@VisibleForTesting" annotation to expose non-public variables or methods for unit testing [70, 71]. Lastly, developers lose productivity when they repeatedly use any of the many third-party websites [14, 19, 94] or in-IDE pop-ups like the one in IntelliJ [37] to test regexes.

We argue that there is a need for specialized support to allow testing individual statements "in place". A simple approach is to first extract the target statement into a method by itself and then write a unit test for the extracted method. Doing so would not be effective for three reasons. First, to correctly set up the right state for testing, developers may have to duplicate code from the method that contains the target statement in the test for the extracted method. Second, if there are many target statements, extracting each one can devolve into a hard-to-maintain "one unit test per statement" scenario. Finally, programs may become harder to comprehend if one has to look up method bodies to understand individual statements.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556952

We introduce *inline tests*, a new kind of tests that makes it easier to check individual program statements. An inline test is a statement that allows to provide arbitrary inputs and test oracles for checking the immediately preceding statement that is not an inline test. Inline tests can be viewed as a way to bring the power of unit tests to the statement level. Structurally, inline tests add a new level of granularity below unit tests to the testing pyramid in Figure 1.

Inline tests could provide software development benefits beyond testing. For example, prior work showed that tests and code do not usually co-evolve gracefully [9]. Unlike unit tests, inline tests are co-located in the same file as target statements. So, inline tests could be easier to co-evolve with code. Prior work also showed that test coverage can stay stable over time because existing tests cover newly-added code [59]. Inline tests can help find faults in newly-added code. The inputs and expected outputs in inline tests are a form of documentation and they could improve code comprehension. Also, inline tests could improve developer productivity by being more durable and less wasteful than "`printf` debugging".

Inline tests are different from the `assert` construct that many programming languages provide, e.g., [68, 91]. `Assert` statements can enable *production-time enforcement* of conditions on program state at given code locations without requiring developer-provided inputs. For example, an `assert` can be used to ensure that a variable is in range, or that a method's return value is not `null`. Differently, inline tests require developer-provided inputs and oracles, and they only enable *test-time checking* of individual statements.

We implement I-Test, the first inline testing framework. Our starting point is to define language-agnostic requirements for inline testing frameworks (Section 3.1). For example, it should *not* be possible to use inline tests in place of unit tests or debuggers. The requirements that we define provide a basis for I-Test and they can provide guidance for the development of future inline testing frameworks. Our current I-Test implementation supports inline testing for Python and Java, and it satisfies most of the requirements.

We evaluate I-Test on open-source projects by using it to test 144 statements in 31 Python programs and 37 Java programs. We perform a user study to assess how easy it is to write inline tests, and to obtain feedback about inline testing. Lastly, we measure the runtime cost of inline tests. All nine user participants who completed the study say that inline tests are easy to write, needing an average of 2.5 minutes to write each inline test, and that inline testing is beneficial. Inline tests incur negligible cost, at 0.007x for Python and 0.014x for Java on average, and our inline tests helped find two new faults that have been fixed by developers after we reported the bugs. These results show the promise of inline tests.

The main contributions of this paper include:

- ⋆ **Idea.** We introduce inline tests, the benefits that they provide, and requirements for testing frameworks that support them.
- ⋆ **Framework.** We implement I-Test, the first inline testing framework. I-Test works for Python and Java.
- ⋆ **User study.** We evaluate programmer perceptions about inline testing, and obtain feedback about their inline testing needs.
- ⋆ **Performance evaluation.** We measure runtime costs of I-Test using 152 inline tests that we write in 68 open-source projects.

Our code and data is publicly available at https://github.com/EngineeringSoftware/inlinetest.

```
1  def parse_diff(diff: str) -> Diff:
2   ...
3   nm = re.match(r'^--- (?:(?:/dev/null)|(?:a/(.*)))$', line)
4   Here().given(line,
          '--- a/python/regex.py').check_true(nm).check_eq(nm.groups(),
          ('python/regex.py',))
5   if nm:
6    name, = nm.groups()
```

Figure 2: Regex in Python code, and an inline test in blue.

2 MOTIVATION AND EXAMPLES

We motivate inline tests by showing examples of some programming language (PL) features and one common testing scenario for which inline tests could be beneficial. For each, we discuss problems that developers face due to the lack of direct support for statement-level testing, and show example inline tests that can help.

2.1 An Example Inline Test

We start by illustrating what inline tests look like because we show several of them in this section, before the I-Test API is described (Section 3.4). Consider this inline test that we write for a target statement in `apprenticeharper/DeDRM_tools` [86]; that target statement is shown and described in Figure 5:

```
Here() .given(dt, (1980, 1, 25, 17, 13, 14)) .check_eq(dosdate, 57)
Declare          Assign                      Assert
```

The "Declare" portion tells the inline testing framework to process the statement as an inline test. The "Assign" portion allows the developer to provide test inputs to the inline test. In this case, (`1980`, `1`, `25`, `17`, `13`, `14`) is to be used as the value of the `dt` variable that is in the target statement. Finally, the "Assert" portion allows the developer to specify a test oracle. In this case, given the test input for `dt`, the `dosdate` variable that is being computed in the target statement should equal 57 for the inline test to pass.

2.2 PL Features That Inline Tests Help Check

Regular expressions (regexes). Prior work showed that regexes are widely used, but they are difficult for developers to understand and to use correctly [12, 16, 17, 62]. So, inline tests can allow developers to check what regexes do, and to test them in place. Consider the Python code fragment in Figure 2, which is simplified from `pytorch/pytorch` [43]. The regex on line 3 is a search pattern that starts with "--- " and ends with the non-capturing group "/dev/null" or "a/(.*)". A matched string is assigned to the `name` variable.

Directly checking what the regex on line 3 matches, or testing that it is correct, is difficult without support for statement-level testing. Three unit tests check `parse_diff` (written in a different file and executed using `pytest` [41]), but they mock the `parse_diff` inputs and do not directly test the regex. In fact, we are not aware of an easy way to directly unit-test the regex on line 3 with `pytest`.

In practice, a main way of checking regexes is to use regex-checking websites [14, 19, 94]. Figure 3a shows one such website. One could also use in-IDE pop-ups like the one in Figure 3b for IntelliJ [37]. These websites and in-IDE pop-ups strengthen our argument for statement-level testing in four ways. First, the existence and usage of these websites or pop-ups show that developers have a need to directly test regexes. Second, these websites and

(a) A regex-checking website [19] **(b) IntelliJ pop-up for checking regexes [37]**

Figure 3: Screenshots of an online website and an in-IDE pop-up for checking regexes.

```
1 orig = os.path.splitext(os.path.basename(infile))[0]
2 if (re.match('^B[A-Z0-9]{9}(_EBOK|_EBSP|_sample)?$', orig) or
3-re.match('^{0-9A-F-}{36}$', orig)
4+re.match('^[0-9A-F-]{36}$', orig)
5 ):
6    # Kindle for PC / Mac / Android / Fire / iOS
7    Here().given(orig,
           '0123456789ABCDEF0123456789ABCDEF0123').check_true(Group(1))
8    clean_title = cleanup_name(book.getBookTitle())
```

Figure 4: Fix for faulty regex that an inline test helped find.

```
1  def FileHeader(self):
2   dt = self.date_time
3   dosdate = (dt[0] - 1980) << 9 | dt[1] << 5 | dt[2]
4   Here().given(dt, (1980, 1, 25, 17, 13, 14)).check_eq(dosdate, 57)
5   dostime = dt[3] << 11 | dt[4] << 5 | (dt[5] // 2)
6   Here().given(dt, (1980, 1, 25, 17, 13, 14)).check_eq(dostime, 35239)
7   if self.flag_bits & 0x08:
8    # Set these to zero because we write them after the file data
9    CRC = compress_size = file_size = 0
```

Figure 5: Bit manipulation in Python, and inline tests in blue.

```
1  public static int executeSqlScript(Context context, Database db, String
       assetFilename, boolean transactional)
2        throws IOException {
3     byte[] bytes = readAsset(context, assetFilename);
4     String sql = new String(bytes, "UTF-8");
5     String[] lines = sql.split(";(\\s)*[\n\r]");
6     new Here().given(sql, "CREATE TABLE MINIMAL_ENTITY (_id INTEGER PRIMARY
          KEY);\nINSERT INTO MINIMAL_ENTITY VALUES (1);\nINSERT INTO
          MINIMAL_ENTITY \nVALUES (2);"
7    .checkEq(lines.length, 3);
8     int count;
9     if (transactional) {
10       count = executeSqlStatementsInTx(db, lines);
11    }
12    ...
13    return count;
14 }
```

Figure 6: String manipulation in Java, and inline test in blue.

pop-ups are not connected to the target statement(s), so developers cannot easily specify where in the code the checks should be performed, what kind of oracles should be used, and what the expected outcome should be. Third, each time developers leave their development environment to use websites or pop-ups, they mentally switch context and may lose productivity as a result [48, 49]. Lastly, knowledge gained from using websites and pop-ups may not be documented, so (other) developers in the same organization may later wastefully re-check the same regex.

Line 4 in Figure 2 shows how inline tests can be used to directly test a regex. There, a developer specifies an input and an expected output. Then a framework like I-Test can run the inline test to provide feedback on what the regex does. Using inline tests as shown in Figure 2 mitigates the aforementioned problems of using regex-checking websites and in-IDE pop-ups: developers have more control to specify how to test the target statement, they do not have to leave their development environment to perform checks, and inline tests self-document knowledge about regexes.

We showcase an additional benefit of using inline tests to check regexes: it helped us find a fault. Figure 4 shows a fix that we report to developers of a project in our evaluation, who have since accepted our pull request[1]. The goal of the faulty regex on line 3 is to match valid string representations of 36-digit hexadecimal numbers or "-", but it wrongly matches "{0-9A-F-" followed by 36 repetitions of "}". The inline test on line 7 helped us find this fault. The inline test input (provided using I-Test's `given` function) is a string that represents a 36-digit hexadecimal number. `Group` is an I-Test construct for automatically matching conditional expressions in `if` or `while` statement headers; it accepts a zero-based index that represents the position of a condition in the header. So, `Group(1)` matches the second conditional expression in the `if` statement in Figure 4, i.e., `re.match('^{0-9A-F-}{36}$', orig)`. We expected the matched condition to be `True`, but it was `False` and the inline test failed. Our fix is on line 4. In sum, an inline test was useful for reducing the burden of setting up and writing a unit test for this regex without the need to first perform some throw-away refactoring to extract the regex from the conditional expression.

Bit manipulation. Figure 5 shows a simplified code fragment from `apprenticeharper/DeDRM_tools` [86]. Line 3 parses the year, month, and day into a 32-bit DOS date. Line 5 uses the hour, minute, and second to compute a 32-bit DOS time. The `FileHeader` function that contains the fragment in Figure 5 has many other statements that we elide, and it can be unit tested to check that it constructs correct headers. However, it is hard to directly test lines 3 and 5 without first extracting these statements into separate functions. Also, bit manipulation is fast but it may be hard to understand. With the inline tests on lines 4 and 6, we are able to directly check the code, and the inputs and expected outputs in those inline tests document what the target statements compute.

String manipulation. Figure 6 shows simplified code in a method from `greenrobot/GreenDAO` [33]. Line 5 uses a regex to tokenize a string. The result of line 5 is subsequently used to query a database on line 10, so a developer may want to check that the split is correct. Although there is a unit test for this function, it only indirectly checks line 5 together with the logic that is implemented in lines 8 to 12. The inline test on line 6 directly tests line 5.

[1] https://github.com/noDRM/DeDRM_tools/commit/012ff533ab6ba6920813284a4eb7

```
1   ...
2 - elif ch < ' ' or ch == 0x7F:
3 + elif ch < ' ' or ord(ch) == 0x7F:
4     out.write('\\x')
5     out.write(hexdigits[(ord(ch) >> 4) & 0x000F])
6 -   Here().given(ch, 0x7F).check_eq((ord(ch)>>4)&0x000F, 0x07)
7 +   Here().given(ch, chr(0x7F)).check_eq((ord(ch)>>4)&0x000F, 0x07)
8     out.write(hexdigits[ord(ch) & 0x000F])
```

Figure 7: Inline test helped find string manipulation fault.

```
1  private CalculatedQueryOperation unwrapFromAlias(CallExpression call) {
2   List<Expression> children = call.getChildren();
3   List<String> aliases =
4    children.subList(1, children.size())
5    .stream()
6    .map(alias -> ExpressionUtils.extractValue(alias, String.class)
7     .orElseThrow(() -> new ValidationException("Unexpected: " + alias)))
8    .collect(toList());
9    new Here().given(children, Arrays.asList(new Expression[]{new
           SqlCallExpression("SELECT MIN(Price) AS SmallestPrice FROM
           Products; "), new SqlCallExpression("SELECT COUNT(ProductID) FROM
           Products;")})).checkEq(aliases, Arrays.asList("SELECT
           COUNT(ProductID) FROM Products;"));
10   CallExpression tc = (CallExpression) children.get(0);
11   return createFunctionCall(tc, aliases, tc.getResolvedChildren());
12  }
```

Figure 8: Java code using stream, and an inline test in blue.

Using an inline test to check statements that manipulate strings also helped us find a fault, which we show together with the fix in Figure 7. Specifically, the condition on line 2 is faulty because it directly compares a string with an integer. So, the inline test on line 6 fails with the message, "TypeError: ord() expected string of length 1, but int found". Changing the condition to be as shown on line 3 fixes the fault and the developers have accepted our pull request[2]. Line 7 is our updated inline test after our fix. No unit test covers this function, but other functions can call it in production.

Stream. The target statement on lines 3 to 8 in Figure 8 uses Java's stream API; it is from `apache/flink` [23] and it extracts the values of an expression's children to a list. Using unit tests to check whether the `aliases` variable is computed correctly will require using sophisticated Java features like reflection [60] (the target statement is in a private method). Moreover, a unit test cannot help to directly check `aliases`; only the value computed on line 11 is returned. Lastly, the `unwrapFromAlias` method is not directly tested by any unit test but it is called by methods in other classes. The inline test on line 9 directly tests the target statement. Also, given the complexity of the statement on lines 3 to 8, a developer who is new to `apache/flink` is likely to be better able to understand the code with the inline test than they would do without it.

2.3 A Common Scenario: "`printf` debugging"

Developers commonly perform "`printf` debugging", in which they temporarily add print statements so that they can visually check whether correct values are being computed at the target statement. Then, after some time, they remove these print statements.

One indication of "`printf` debugging" popularity can be seen by searching for "remove debug" on GitHub or by going to [26]. (We found 3,344,094 matching commits in May 2022, but we did not look through them all to see if they are all about "`printf` debugging".) GitHub commits likely underestimate "`printf` debugging" popularity; developers may clean the print statements before committing their code. Dedicated utilities like git –remove–debug [11] and others [13, 40, 58] clean up after "`printf` debugging". Figure 9 shows a GitHub commit[3] that cleaned up after "`printf` debugging" a complex statement in a private method. Researchers found many reasons why developers do "`printf` debugging": lack of familiarity with debuggers [10], lack of platform-specific debuggers [5, 36], perceived speed [73] and simplicity [55] of "`printf` debugging", the inability of debuggers to handle parallel PL constructs [31], etc.

```
1 private List<Field> getNestedField(...) {
2  if (subField.isAnnotationPresent(Indexed.class)) {
3 - System.out.println(">>> Found Indexed SUBFIELD....");
4   boolean sfIsTagField = ((subField
5    .isAnnotationPresent(Indexed.class)
6    && ((CharSequence.class.isAssignableFrom(subField.getType())
7     || (subField.getType() == Boolean.class)
8      || (maybeCollectionType.isPresent()
9       && (CharSequence.class
10       .isAssignableFrom(maybeCollectionType.get())
11      || (maybeCollectionType.get() == Boolean.class)))))));
12 - System.out.println(">>> sfIsTagField ==> " + sfIsTagField);
13  new Here().given(subField, new Object() {@Indexed CharSequence f;}
14    .getClass().getDeclaredField("f"))
15   .checkEq(sfIsTagField, true);
16   ...
17 }}}
```

Figure 9: How inline tests can help Java "`printf` debugging".

We do not claim that inline tests could replace "`printf` debugging". The many reasons for the longevity and popularity of "`printf` debugging" suggests that there is no silver bullet. However, inline tests can help to reduce some of the wastefulness of adding and then removing print statements during "`printf` debugging". Specifically developers could use inline tests to persist knowledge that they gain during "`printf` debugging". For example, line 13 to line 15 in Figure 9 shows how one could migrate the print statements from "`printf` debugging" into inline tests.

3 THE I-TEST FRAMEWORK

We start with a list of language-agnostic requirements for inline testing frameworks. Then, we give an overview of I-Test, the inline testing framework that we implement in this paper. Lastly, we introduce I-Test's API, and describe our current implementation.

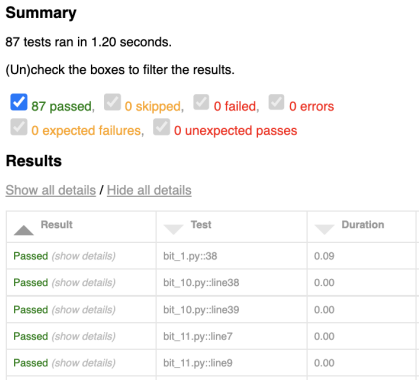
3.1 Inline Testing Framework Requirements

Section 2 motivated the need for inline tests. We now turn to the question, *what are the requirements for inline testing frameworks?* Answering it helps to (1) distinguish inline testing from existing forms of testing, (2) provide a road map for inline testing development, and (3) provide a basis for evaluating I-Test. Inline testing frameworks should meet this minimum set of requirements:

(1) Inline tests are *not* replacements for unit tests or debuggers. ✓
(2) An inline test should only check one target statement. ✓
(3) Multiple inline tests can check the same target statement. ✓
(4) An inline test should allow developers to provide multiple values for a variable in the target statement. ✓

[2]https://github.com/python/cpython/commit/5535f3f745761e53a6ff941b8ef74b5ce

[3]https://github.com/redis/redis-om-spring/commit/f808c9b3a0c72d22c14221e37228a389a3ff139d



Summary

87 tests ran in 1.20 seconds.

(Un)check the boxes to filter the results.

87 passed, 0 skipped, 0 failed, 0 errors
0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration
Passed *(show details)*	bit_1.py::38	0.09
Passed *(show details)*	bit_10.py::line38	0.00
Passed *(show details)*	bit_10.py::line39	0.00
Passed *(show details)*	bit_11.py::line7	0.00
Passed *(show details)*	bit_11.py::line9	0.00

Figure 10: Test report in HTML format.

(5) Inline tests should be easy for developers to write and run using similar idioms as those they already use, to ease adoption. ✓*
(6) Inline testing frameworks should be easy to integrate with testing frameworks and IDEs that developers use. ✓*
(7) To aid readability, when integrated with IDEs, inline testing frameworks should hide inline tests by default, and allow developers to hide or view inline tests as needed. ✗
(8) It should be possible to enable inline tests during testing and to disable them in production. ✓
(9) When *enabled*, the runtime cost of inline tests should be low. ✓
(10) When *disabled*, inline tests should have negligible overhead. ✓
(11) It should be possible for developers to run subsets of all inline tests—developers often perform manual test selection [29]. ✓*
(12) It should be possible to run inline tests in parallel. ✗
(13) It should be possible to write inline tests for target statements that invoke methods or functions whose arguments need initialization. ✗
(14) It should be possible to write inline tests for expressions in branch conditions, without requiring developers to copy those expressions into the inline test. ✓*

I-Test currently meets requirements marked as ✓; it only partially supports those marked ✓*and it does not support those marked ✗. The ✓* in requirements 11 and 14 means that our current Python implementation satisfies the requirement but our current Java implementation does not. Other ✓* marks mean that we partially satisfy the requirement in Python and Java.

These requirements that we enumerate are *initial*, based on our understanding so far, and they are likely incomplete. Our goal for providing them is to bootstrap the development of inline testing and to aid better community understanding of inline testing.

3.2 Overview of the I-Test Framework

I-Test is our inline testing framework that provides developers with support for statement-level testing. I-Test's API provides three kinds of methods that allow developers to (1) declare an inline test, (2) provide input values that should be assigned to the variables in the target statement during testing, and (3) specify test oracles. If developers write multiple inline tests, they can run the inline tests separately or in a batch. We started integrating I-Test with two popular unit testing frameworks—`JUnit` and `pytest`—and it also generates test reports. Figure 10 is an example test report generated by I-Test, based on the `pytest-html` plugin [75] that it uses. Inline tests must be the next statements after a target statement that is being tested. Since inline tests are co-located with code, I-Test provides facilities for turning off the execution of inline tests in production environments. When inline testing is turned off, the inline tests are still in the code but running the code should incur negligible runtime overhead.

3.3 I-Test Development Process and API

To ground I-Test in likely developer needs, we focus our current implementation on selected kinds of statements from open-source projects. Based on our own programming experience, these kinds of statements could benefit from inline testing. We described some of these kinds of statements in Section 2.2, but we focus our implementation on five of them: regexes, string manipulation, bit manipulation, stream API usage, and collection handling code.

One challenge is to better understand the API that I-Test should provide to support statement-level testing for the kinds of statements that we focus on. To address this challenge, we collect examples of these kinds of statements from open-source, manually inspect them, and iteratively refine our I-Test API. Specifically, we first collect Java and Python projects from GitHub. Then, we filter out projects that do not contain the kinds of statements that we focus on. Lastly, we find examples from those that remain and we use them to guide our API design. We next describe our example collection process, and provide more details on the current API.

3.3.1 Example Collection Process. We are interested in target statements that are in possibly complicated code blocks, such that the target statement may be difficult to test directly with unit tests. (See Section 1 for a discussion of the pitfalls of extracting individual statements into methods or functions for the sole purpose of enabling unit testing.) We look for Java and Python statements with regular expressions, as well as those that manipulate strings and bits. We also look for statements that use the stream API in Java and those that manipulate collections in Python.

We perform keyword search (such as "re.match" and "re.split" for Python regular expressions) among the 100 top-starred Java and Python projects on GitHub (a total of 200 projects). All keywords that we use for each language and the number of matches that we find are provided in the data package for this paper. We manually inspect metadata for these projects and remove those that are about tutorials, e.g., interview questions. We then use the remaining 83 Java projects and 91 Python projects. For each project that remains, we select examples and manually inspect them for suitability to help guide our API design.

To make our manual check easier, we make our keyword search return five lines of leading and trailing context for each match. We then manually check whether the matched lines are for the kinds of target statements that we focus on. We filter out cases where keywords only appear in comments or in which we deem the code too simple to warrant an inline test, e.g., for keyword "split" we find `String[] errorMessageSplit = e.getMessage().split(" ");`. We also filter out keyword searches that yield false positives. For example, we search for >> as the right shift operator in bit manipulation but sometimes match the closing tag of a parameterized generic

Table 1: No. of examples and the inline tests that we write to guide API design. PL= programming language, # Projs= no. of projects, # Examples= no. of examples, # Target stmts= no. of target statements, and # Inline tests= no. of inline tests.

PL	# Projs	# Examples	# Target stmts	# Inline tests
Python	31	50	80	87
Java	37	50	64	65

Table 2: Breakdown of the inline tests that we write.

Kind	PL	# Projs	# Examples	# Target stmts	# Inline tests
Regex	Python	15	17	19	22
	Java	15	17	17	17
String	Python	13	14	30	32
	Java	15	15	20	20
Bit	Python	15	15	26	27
	Java	16	16	25	26
Collection	Python	4	4	5	6
Stream	Java	2	2	2	2

type, e.g., <String , Box<Integer>>. Among the rest, for each kind of target statement per project, we extract an example which is the first snippet with a target statement that can be tested at the statement level. Finally, based on randomly extracted 50 examples of Python and 50 examples of Java, we design the I-Test API.

3.3.2 Corpus. Data about the selected examples that we base our design of I-Test API on are shown in Table 1. For Python, we write 87 inline tests for 80 statements in 50 examples from 31 projects. For Java, we write 65 inline tests for 64 statements in 50 examples from 37 projects. There are sometimes multiple target statements in some examples, and we sometimes write multiple inline tests for a target statement.

Table 2 shows a breakdown of the number of inline tests that we write for each kind of target statement. Columns represent the kind of target statement, the PL, the number of projects, the number of examples, the number of target statements, and the number of inline tests. We write at least one inline test per target statement. There are fewer numbers in the "Collection" row because although operations on collections, like list comprehension or sorting, look complicated, some developers may want to test them and others may not. Our user study proves this variation in preferences (Section 5).

3.4 The I-Test API

We design the I-Test API to have three components, based on what they allow developers to do:

(1) Declare and initialize an inline test. This API component signals to the I-Test framework to process a statement as an inline test and allows users to optionally specify a name for the inline test. If a test name is not specified, I-Test defaults to using a name which is the concatenation of the current file name and the line number of the inline test. This component comprises the `Here()` and `Here(test_name = "")` functions in Python and the `Here()` and `Here(testName)` methods in Java. With these `Here()` functions or methods, users can also provide optional parameters for customizing inline test execution. These parameters include those that (1) set the number of times to re-rerun an inline test; (2) disable the inline test so that it is not executed (similar to the `@Ignore` annotation in `JUnit`); (3) indicate that sets of values can be used to parameterize

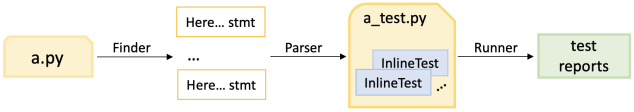


Figure 11: Workflow of I-Test for Python.

an inline test; and (4) tag inline tests so that users can filter out those that they do not want to run (similar to the `@Tag` annotation in `JUnit` [88]). We plan to implement support for other features in this I-Test API component, including allowing users to specify that an inline test should be run conditionally.

(2) Provide test inputs. Developers can use this API component to initialize variables in the target statement to desired test input values. The rationale is that, to directly test a target statement, I-Test has to be able to re-initialize the variables in that statement to the values that should be used for testing. In Python and Java, this API component is the `given(variable, value)` function or method. I-Test assigns `value` to `variable` only while running the inline test. Two input-related needs may arise during inline testing: a target statement may have multiple variables, or a developer may want to test a target statement using multiple values of the same variable. To address the first need, I-Test allows chaining `given(...)` calls. To address the second need, I-Test allows to provide a list of values in each `given(...)` call if `Here(parameterized = True)` is used. This feature is similar to parameterized unit tests [92, 93].

(3) Specify test oracles. This API component allows developers to make assertions on the results of running the inline test. Driven by the examples that we base our design on, I-Test supports checking equality of two expressions with `check_eq(expr1, expr2)`, checking whether a condition holds or not with `check_true(expr)` and `check_false(expr)`. The last two are for convenience; they are equivalent to `check_eq(expr, True)` and `check_eq(expr, False)`, respectively. In Java, we support oracles with the same functionality but they have camel-case naming. Unit testing frameworks typically support more kinds of assertions. As I-Test grows, we may need to add more kinds of assertions. These three suffice to check the target statements in our corpus (Section 3.3.2).

Even though we base our design on selected examples from open source, we are encouraged that our API design resulted in components that should be familiar to developers who already know how to write unit tests. The API is also the same for Java and Python. Even if small tweaks are needed to support other programming languages, current evidence suggests that the same inline testing API components may be useful more broadly.

3.5 I-Test Implementation

Figure 11 shows the workflow of I-Test for Python; it is similar for Java. Given a source file, `Finder` searches for statements that start with `Here` calls. `Parser` traverses the AST of the source file to discover the target statement. `Parser` also uses the output of `Finder` to reconstruct assignments and assertions and to collect each inline test into a new source file that can be executed. Moreover, `Parser` copies the import statements used by the target statement and the inline test to the new source file; thus, execution of this new source file only requires the packages used by the target statement and the inline test. Finally, `Runner` executes the inline test files and generates test reports like the one shown in Figure 10.

Python. We implement I-Test as a standalone Python library, which can be run from the command line; we also integrate I-Test into `pytest`. I-Test uses the Python AST library [85] to parse the source code, extract the tested statement, process the input assignments and assertions, compose an executable test, and execute the inline test in the name space of the module in which tested statement exists. More precisely, I-Test uses the visitor design pattern to detect inline test initialization and to find target statements. Oracles are implemented on top of the `assert` construct in Python. If an assertion fails, the resulting error message shows the line number of the failing inline test, and its observed and expected outputs. We integrate I-Test as a plugin into `pytest` to reuse the various testing options that `pytest` provides and to generate test reports.

Java. We use Javaparser [87] to manipulate Java AST. Java I-Test additionally infers variable types in `given` calls using a symbol table that it maintains. For example, in `given(a, 1)`, it looks up the type that `a` was declared with in the program. We support two compilation modes for Java inline tests. The first (guard mode) keeps the inline test in the resulting bytecode and uses a flag to skip or run the inline test. The second (delete mode) discards the inline tests from the bytecode. We also support two ways to run inline tests in Java. The first generates an *ad hoc* class for each source file, where each inline test is converted to a method and a main method is added to run all the inline tests. The second produces a JUnit test class for the given file, where each inline test is converted to a test method, which can be executed using the JUnit runner.

4 PERFORMANCE EVALUATION FOR I-TEST

We answer these research questions to assess inline testing costs:

RQ1: How long does it take to run inline tests?

RQ2: What is the runtime overhead when inline tests are *enabled* during the execution of existing unit tests?

RQ3: What is the runtime overhead when inline tests are *disabled* during the execution of existing unit tests?

We measure the times for answering these questions using the inline tests from the 100 examples that we write (Section 3.3.2). We also duplicate each of these inline tests 10, 100, and 1000 times, so that we can simulate the costs as the number of inline tests grows. We evaluate RQ2 and RQ3 on 21 projects in our corpus where we could run the unit tests.

4.1 Experimental Setup

Standalone experiments. To run the inline tests in an example, I-Test does not need all code elements (class, method, or field) in that example. Rather, it only needs code elements used by the target statement and the inline test. For example, the code fragment in Figure 6 has classes `Context` and `Database` in the method signature. But, the inline test there does not need these classes; it only needs the `String` class from the standard library and the `Here` class in I-Test. On the contrary, running a unit test for the same example requires loading all the classes. So, I-Test can run all 152 inline tests under the standalone mode without setting up the environments needed to run unit tests. For Python, we run the inline tests in each example using the I-Test plugin that we integrate into `pytest`. For Java, we run the inline tests in each example by using I-Test to produce an ad-hoc class and then invoke its main method.

Integrated experiments. To measure the runtime overhead of inline tests, we need to run them together with unit tests using the runtime environment specified by each project. We write inline tests directly in the projects from which we extract the examples. But, we face difficulties in setting up some runtime environments or in running unit tests. So, we perform the experiments for answering RQ2 and RQ3 on a subset of 21 projects. Below, we discuss the difficulties that we face for Python and Java, respectively.

I-Test for Python relies on `pytest` to run inline tests. Of 31 Python projects in our corpus, we could not setup the appropriate `pytest` runtime environment for 2: `keras-team/keras` uses the bazel build system which requires additional time to setup; and `kovidgoyal/kitty` mixes C++ with Python code, leading to problems with importing C++ code into `pytest` using a `pyi` interface file. Of the other 29 projects, 5 have no unit tests. We confirm absence of unit tests by (1) checking the README.md and CONTRIBUTING.md files which contain instructions for setting up the projects; (2) inspecting the Continuous Integration logs, if any; and (3) searching for `*test*.py` in the repositories. 5 projects do not use `pytest` to run unit tests. Lastly, another 5 projects have many unit tests that consistently fail. If a project manifests less than 10 flaky unit tests [8, 44, 45, 57, 80] that can be skipped without causing more failures, we run the remaining unit tests in that project. We run inline tests and unit tests for the remaining 14 projects (first column of Table 4a).

For Java, we use I-Test to generate ad-hoc classes for the integrated examples, and compile the generated classes together with the other source code in the project. Of 37 projects in our corpus, 10 have compilation failures (before integrating any inline test) and 3 have no unit tests. We confirm that these projects have no unit tests and handle flaky tests similarly as we did for Python. If running unit tests across a multi-module project fails, we retry running only the unit tests in the sub-modules that we write inline tests for (and refrain from using the project in our experiments if there are still too many failures). We run inline tests and unit tests for the remaining 7 projects, shown in the first column of Table 4c.

Duplicating inline tests. Since we are the first to explore inline tests, the number of inline tests we have written for each project is often not as much as the number of unit tests that a project typically has. In the future, about equal or even more inline tests than unit tests may be written. To simulate the performance of I-Test in such scenario with the corpus we currently have, we experiment with duplicating each inline test 10, 100, and 1000 times. When duplicating inline tests 1000 times, two Java projects (`alibaba/fastjson` and `apache/kafka`) do not compile because the size of the bytecode in the method containing the target statement exceeded the allowable limit in Java [67]. So, we exclude these two projects (only when duplicating 1000 times).

Experimental procedure and environment. We run inline tests and unit tests four times. The first run is for warm-up, and we average the times for the last three runs. We run experiments on a machine with Intel Core i7-11700K @ 3.60GHz (8 cores, 16 threads) CPU, 64 GB RAM, and Ubuntu 20.04. We use Java 8 and Python 3.9 in the standalone experiments, and use the software versions required by each project in the integrated experiments.

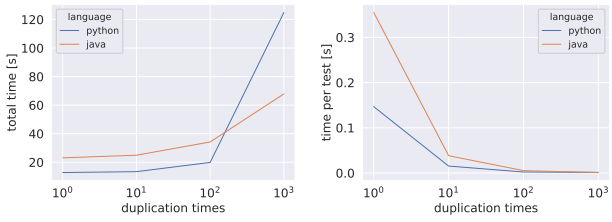
Table 3: Results of standalone experiments. Dup = duplication count, #IT= total no. of inline tests, T_{IT}[s]= total inline tests running time, t_{IT}[s]= inline test running time.

(a) Python

Dup	#IT	T_{IT}[s]	t_{IT}[s]
x1	87	12.78	0.147
x10	870	13.41	0.015
x100	8,700	19.86	0.002
x1000	87,000	124.92	0.001

(b) Java

Dup	#IT	T_{IT}[s]	t_{IT}[s]
x1	65	23.08	0.355
x10	650	24.92	0.038
x100	6,500	34.21	0.005
x1000	65,000	67.87	0.001



(a) total time **(b) per-test time**

Figure 12: Line plots of duplication times vs. total/per-test time when running inline tests standalone.

4.2 Results

RQ1: cost of running only inline tests. Table 3 shows the results of running Python and Java inline tests in the standalone mode. Without duplicating the inline tests in each example, the average time for running each inline test is 0.147s for Python and 0.355s for Java. As we duplicate the inline tests in each example, the average time for running each inline test reduces to 0.001s for Python and 0.001s for Java. There could be two reasons. First, the cost of reading a file and extracting inline tests is amortized. Second, repeatedly executing the same inline test is faster.

Figure 12 shows how total and per-test execution time scale as the number of inline tests grows. There, the total time for running inline tests stays almost constant when duplicating the inline tests 10 or 100 times (corresponding to around 10 and 100 inline tests per file), and starts to grow dramatically when duplicating 1000 times. The Java version of I-Test shows better scalability than the Python-version, as it is slower initially but faster when duplicating 1000 times, probably due to just-in-time compilation.

RQ2: overhead of running unit tests with inline tests *enabled*. Table 4 shows the results of running Python and Java inline tests after integrating with the open-source projects and their unit tests. There, the O_{ITE} columns show the overhead when inline tests are enabled and executed during the execution of existing unit tests. Overall, without duplicating inline tests (tables 4a and 4c), the overhead of running inline tests is negligible compared to unit tests, and is 0.007x for Python and 0.014x for Java. This observation holds when duplicating inline tests (tables 4b and 4d); for example, when duplicating inline tests 1000 times, which brings the number of inline tests similar to the number of unit tests, the overhead is 0.088x for Python and 0.008x for Java. Negligible overhead may be due to inline tests running much faster than unit tests.

RQ3: overhead of running unit tests with inline tests *disabled*. The O_{ITD} columns in Table 4 show the overhead when inline tests are disabled during the execution of existing unit tests. The inline tests are not executed, but having them in the code base may require unit tests to execute additional no-op statements. Nevertheless, we found such overhead to be negligible, even when duplicating the inline tests for 10–1000 times; the negative close-to-zero overhead numbers (e.g., -0.001x for Python when not duplicating inline tests) are likely due to nondeterminism during execution.

5 USER STUDY

The goals of our study are to evaluate the ease with which developers learn and use I-Test, and to obtain their perception about inline testing or how I-Test can be improved.

5.1 Study Design

We ask participants to complete three activities: (1) a short tutorial to learn about inline testing and I-Test (expected duration: 20 minutes), (2) four testing tasks in which they write inline tests for four specified target statements (expected duration: 10 minutes per task), and (3) a questionnaire with six questions (unspecified duration). We suggest a one-hour time limit, but results show that most participants finish faster. We write scripts to process the responses, and manually check the correctness of participants' inline tests.

We only use I-Test for Python in our user study to keep participants focused on inline testing and not on switching between programming languages. We plan to do a user study of I-Test for Java (and other programming languages) in the future. A sample user study (without responses) is in our GitHub repository. We briefly describe the activities that participants undertake.

(1) Tutorial. We provide an overview of I-Test's API (Section 3.4). Then, we ask each participant to run a provided script to setup the environment. Finally, we illustrate I-Test using three examples. The first example is a toy "hello world" example; the other two are examples from our corpus. Each example contains a code snippet, specifies a target statement or two together with one or two inline tests per target statement. We also describe I-Test's API and instructions for running the inline tests.

(2) Using inline tests. We ask participants to write and run inline tests for four examples from our corpus. For each example, we present the participant with the code snippet (without our inline tests) and specify a target statement. Then, we ask participants to write one or more inline tests for the target statement. We also ask participants to ensure that their inline tests pass. Finally, we ask participants to separately report the time taken to understand the target statement and the time taken to write all inline tests.

(3) Survey. We ask participants to fill a questionnaire, to record their experiences with I-Test and their feedback. Specifically, we ask participants to (a) rate the difficulty of learning I-Test's API and of writing inline tests, (b) report their number of years of general and Python programming experience (to understand if expertise impacts their experiences), (c) say whether they think writing inline tests is beneficial for each of the four tasks compared with unit tests (they can optionally justify their "yes" or "no" responses), (d) comment on how to improve I-Test.

Participants. Our valid user study participants are six graduate students and two undergraduate students from our institutions and one professional software engineer. We start with 13 participants. Two participants partake in a pilot study, but we discard their

Table 4: Results of integrated experiments. Proj= project name, Dup = duplication times, #UT= total no. of unit tests, #IT= total no. of inline tests, t_{UT}[s]= time to run each unit test, t_{IT}[s]= time to run each inline test, T_{ITE} [s]= total time to run unit tests with inline tests enabled, t_{ITE}[s]= time to run each unit test with inline tests enabled, O_{ITE}= overhead of running unit tests with inline tests enabled, T_{ITD}[s]= total time to run unit tests with inline tests disabled, t_{ITD}[s]= time to run each unit test with inline tests disabled, O_{ITD}= overhead of running unit tests with inline tests disabled.

(a) Python

Proj	#UT	#IT	T_{UT} [s]	t_{UT}[s]	T_{ITE} [s]	t_{ITE}[s]	O_{ITE}	T_{ITD}[s]	t_{ITD}[s]	O_{ITD}
RaRe-Technologies/gensim	968	2	225.92	0.233	226.90	0.234	0.004	226.35	0.234	0.002
Textualize/rich	622	2	3.71	0.006	3.94	0.006	0.063	3.72	0.006	0.002
bokeh/bokeh	8,616	8	49.63	0.006	50.91	0.006	0.026	50.13	0.006	0.010
chubin/cheat.sh	1	3	0.34	0.337	0.74	0.186	1.204	0.33	0.334	-0.010
davidsandberg/facenet	3	1	0.97	0.323	1.83	0.458	0.888	0.98	0.325	0.006
geekcomputers/Python	1	4	0.17	0.169	0.38	0.075	1.217	0.18	0.179	0.058
google-research/bert	15	1	2.05	0.137	2.69	0.168	0.314	2.07	0.138	0.011
joke2k/faker	1,596	4	16.73	0.010	16.91	0.011	0.011	16.64	0.010	-0.006
mitmproxy/mitmproxy	1,287	1	7.50	0.006	7.85	0.006	0.046	7.45	0.006	-0.007
numpy/numpy	19,644	2	147.82	0.008	145.88	0.007	-0.013	145.36	0.007	-0.017
pandas-dev/pandas	147,307	2	278.43	0.002	279.81	0.002	0.005	278.88	0.002	0.002
psf/black	236	1	6.96	0.029	7.29	0.031	0.048	7.02	0.030	0.009
pypa/pipenv	106	1	3.63	0.034	4.17	0.039	0.151	3.64	0.034	0.003
scrapy/scrapy	2,246	2	130.07	0.058	130.93	0.058	0.007	130.42	0.058	0.003
avg	13,046.29	2.43	62.42	0.005	62.87	0.005	0.007	62.37	0.005	-0.001
Σ	182,648	34	873.93	N/A	880.24	N/A	N/A	873.16	N/A	N/A

(b) Python, with duplicating inline tests

Dup	#UT	#IT	T_{UT} [s]	t_{UT}[s]	T_{ITE} [s]	t_{ITE}[s]	O_{ITE}	T_{ITD}[s]	t_{ITD}[s]	O_{ITD}
x1	182,648	34	873.93	0.005	880.24	0.005	0.007	873.16	0.005	-0.001
x10	182,647	340	871.73	0.005	922.03	0.005	0.058	914.68	0.005	0.049
x100	182,648	3,400	876.13	0.005	884.16	0.005	0.009	873.65	0.005	-0.003
x1000	182,647	34,000	872.59	0.005	949.02	0.004	0.088	889.00	0.005	0.019

(c) Java

Proj	#UT	#IT	T_{UT} [s]	t_{UT}[s]	T_{ITE} [s]	t_{ITE}[s]	O_{ITE}	T_{ITD}[s]	t_{ITD}[s]	O_{ITD}
alibaba/fastjson	5,022	2	44.99	0.009	45.59	0.009	0.013	44.86	0.009	-0.003
alibaba/nacos	971	1	249.45	0.257	250.67	0.258	0.005	249.93	0.257	0.002
apache/dubbo	3,180	1	678.86	0.213	680.26	0.214	0.002	679.43	0.214	0.001
apache/kafka	221	1	9.84	0.045	10.76	0.048	0.094	10.09	0.046	0.026
apache/shardingsphere	44	2	5.03	0.114	5.75	0.125	0.143	5.04	0.115	0.002
jenkinsci/jenkins	32	2	4.67	0.146	5.29	0.156	0.132	4.64	0.145	-0.007
skylot/jadx	709	1	66.57	0.094	76.21	0.107	0.145	75.47	0.106	0.134
avg	1,454.14	1.43	151.34	0.104	153.50	0.105	0.014	152.78	0.105	0.009
Σ	10,179	10	1,059.41	N/A	1,074.53	N/A	N/A	1,069.47	N/A	N/A

(d) Java, with duplicating inline tests

Dup	#UT	#IT	T_{UT} [s]	t_{UT}[s]	T_{ITE} [s]	t_{ITE}[s]	O_{ITE}	T_{ITD}[s]	t_{ITD}[s]	O_{ITD}
x1	10,179	10	1,059.41	0.104	1,074.53	0.105	0.014	1,069.47	0.105	0.009
x10	10,179	100	1,059.36	0.104	1,065.38	0.104	0.006	1,060.47	0.104	0.001
x100	10,179	1,000	1,059.11	0.104	1,073.50	0.096	0.014	1,068.44	0.105	0.009
x1000	4,936	7,000	1,004.24	0.203	1,012.16	0.085	0.008	1,008.55	0.204	0.004

responses after using those responses to refine the user study. We then send the study to the other participants in batches of five and six. No participant is a co-author of this paper, and we confirm that none of them contributes to the open-source projects being tested. We got nine valid responses; participants report an average 6.1 years (median: 6.0 years) of programming experience. On a scale of 1 to 5, with 1 being novice and 5 being expert, participants self-rate their Python expertise as 3.4 on average (median: 3.0).

Inline tests vs. unit tests. We did not ask user study participants to write unit tests or to directly compare them with inline tests for the testing tasks. Rather, we only ask for anecdotal comparisons of inline tests and unit tests in the questionnaire. We chose this study design for three reasons. First, setting up the unit testing environment per project is hard (even for us) and differs across projects. So, asking participants to set up environments before writing unit tests could be a source of bias. Second, providing a Docker image (or similar) could induce bias—installing and running Docker containers could be hard for participants who are unfamiliar with Docker. Lastly, we do not assume familiarity with `pytest`, which participants would need to write unit tests in Python. To

work around these three problems, we provide participants with a script that sets up a minimal Python runtime environment for inline tests. It takes only about one minute to run the script.

5.2 User Study Results

Quantitative analysis. Our user study results are shown in Table 5, grouped by the four tasks. For each task, we show the average time (in minutes) spent by each participant on understanding the target statement, writing all inline tests, and writing each inline test. We also show the number of inline tests that participants write, the number of participants for whom all inline tests pass, and the number of participants who answer "yes" to "writing inline tests is beneficial compared with just writing unit tests". On a scale of 1 to 5 (1 being very difficult and 5 being very easy), participants rank the difficulty of learning I-Test as 4.2 (median: 4.0) and rank the difficulty of writing inline tests as 4.1 (median: 4.0). On average, participants write 1.7 inline tests (median: 1.7) per task, and spend 2.5 (median: 2.6) minutes to understand a target statement and 3.5 (median: 3.6) minutes to write an inline test.

Qualitative analysis. All participants found inline tests to be beneficial for some of the tasks. In fact, for all four tasks, most participants think that writing inline tests is beneficial, and all participants agree that inline tests are beneficial for Task 4. The one participant who said that inline testing is not beneficial for Task 1 preferred to extract the target statement into a function and then write unit tests. So, while they did not use inline testing for this task, they still found it important to test the target statement. For Task 2, the one participant who did not find inline testing beneficial said that they think that the target statement is too trivial to test. Lastly, the four participants who did not find inline testing useful for Task 3 provide two kinds of reasons: (1) the variable in the target statement is being returned from the function, so a unit test would suffice (two participants); and (2) the target statement performs sorting, which is easy to understand and does not warrant inline testing (two participants). The variance in perceptions on Tasks 1, Task 2, and Task 3, plus the different reasons given by participants who think that a target statement does not warrant an inline test shows that developers will likely use inline tests in different ways.

Participants provide feedback on how to further improve I-Test, including by (a) minimizing the long stack traces that are shown when inline tests fail (*"The stack trace you get when a test fails is quite long, but this is an easy fix"*); (b) allowing inline tests to use symbolic variables (*"Having tests with symbolic values, meaning that you don't provide values for inputs"*); (c) providing other methods in the API that allow writing other kinds of oracles beyond equality checks (*"Other kinds of checks besides equality"*); (d) supporting parameterized inline tests, which we have now implemented (*"I would like shortcut for checking for multiple inputs"*).

Participants also share feedback on using I-Test. A participant liked having inline tests in addition to unit tests: *"it is quite useful to have an inline testing option available. Unit testing and inline testing don't have to be exclusionary, there are some situations where one might be preferable but having both as an option is nice"*. Another participant commented that there is a learning curve: *"I experienced a learning curve to using the framework. I was able to understand the structure of how to make ... tests much better after doing the first task"*. It will be important in the future to investigate ways to lower the learning curve. A participant was curious to know what the overhead is when inline tests are disabled: *"Does inline testing add overhead during production runs (i.e. no testing is needed)?"*. We answer this question in Section 4.2. Also, a participant thinks inline tests may be better than `assert` statements (*"Inline tests can be good replacement for assertions"*). Lastly, a participant made the connection to "`printf` debugging": *"I would legitimately want to use a framework like this next time I felt the need to do printf debugging"*.

Table 5: User study results. T_u[min]= time to understand each task, T_w[min]= time to write all inline tests per task, #IT= no. of inline tests, T_w/#IT [min]= avg. time to write each inline test, Corr= ratio of participants who write passing inline tests, Adv= ratio of participants who find inline tests beneficial.

Task	T_u[min]		T_w[min]		#IT		T_w/#IT [min]		Corr	Adv
	avg	med	avg	med	avg	med	avg	med		
1	4.0	4.0	3.7	3.0	1.7	1.0	2.8	2.0	9/9	8/9
2	1.6	1.0	3.4	3.0	1.6	1.0	2.5	2.0	9/9	8/9
3	2.2	2.0	4.1	4.0	1.7	2.0	3.0	2.0	9/9	5/9
4	3.3	3.0	2.8	2.0	1.8	2.0	1.9	1.0	9/9	9/9
avg	2.8	2.8	3.5	3.6	1.7	1.7	2.5	2.6	N/A	N/A

6 LIMITATIONS

We design the I-Test API based on 100 examples that we select from open-source projects. Also, the inline test inputs and expected outputs that we use in those tests were neither chosen by the open-source project developers nor confirmed by them. So, it is not yet clear if those developers will find our inline tests acceptable.

Our own programming experience tells us that more kinds of oracles will likely need to be supported in I-Test. For example, we do not yet support expected exceptions or allow checking near equality between floating point values. The current limited set of oracles in I-Test results from using 100 examples to guide our design. In the future, by collecting more examples and requirements, I-Test can possibly be extended to support more kinds of oracles.

In terms of implementation, Section 3.1 shows the list of language agnostic requirements that I-Test does not yet support (✗) and those that it only partially supports (✓*). This paper motivates, defines, and evaluates inline tests as a way to prove the concept. The engineering effort to fully support all the requirements is a matter of time and resources that we will invest into seeing that inline tests become more mature.

An inline test is inserted as code directly following the code under test. In the unlikely case when the code under test is in a large method or file, inserting inline tests may cause code-too-large errors due to limitations of compilation tool chains (for example, a Java method can only have a maximum of 65535 bytes of bytecode [67]).

Our current Java I-Test implementation is designed to support language features of Java 8, and it may not work for newer language features in more recent Java versions. In the opposite direction, our current Python I-Test implementation is designed to support language features of Python 3.6 and above, so it may not work for older Python versions.

If a target statement invokes a method with arguments that need to be assigned in an inline test, then the current I-Test implementation cannot be used to check that target statement (Hence, the ✗ on

Requirement 13 in Section 3.1). We already observed a consequence of this limitation in our attempt to write inline tests for statements that use Java's stream API. Most stream operations invoke the kind of method-with-arguments that we do not yet support. Also, stream operations typically invoke several methods, so testing them with inline tests can seem like writing unit tests. Finding smart ways to support the testing of stream operations will be a priority—the complexity and popularity of stream operations make them attractive candidates for inline testing.

Inline testing may not generalize well to programming languages that do not use the imperative style of Java and Python. In particular, more thoughts need to be given in the future on whether and how inline testing can be realized effectively for functional languages like Haskell, logic programming languages like Prolog, or domain-specific languages like SQL.

We have not investigated how well inline testing can fit into different software and test design processes. So, it is not yet clear what impact, if any, inline tests will have in the presence of different testing methodologies. For example, since inline tests check *existing* target statements, its role may be limited in organizations that follow test-driven development (TDD) [3, 7, 78]. (In TDD, tests are written prior to writing code.) As another example, what role should inline tests play during regression testing and how often should they be re-run during software evolution? Similarly, it may be that inline tests are more useful in systems where testability [24] was not a first-class concern during programming. That is, inline tests may be more helpful in legacy systems or systems with large monolithic components than in newer systems that are designed to be unit-testable from the ground up. We leave the investigation of how to fit inline tests into different software- and test-design processes as future work.

7 RELATED WORK

Testing and debugging. Karampatsis and Sutton [39], and Kamienski et al. [38] curated datasets of single-statement bugs (SStuBs) in Java and Python, respectively. Also, Latendresse et al. [47] find that continuous integration (CI) rarely detects SStuBs. These works on SStuBs further motivate the need for direct support for checking individual statements, which inline tests provide.

Michael et al. [62] found that regexes are hard to read, find, validate, and document. Eghbali and Pradel [20] also found that string-related bugs are common in JavaScript programs. Section 2 discussed how inline tests can mitigate these problems and how I-Test helped find regex-related and string-manipulation bugs.

Doctest [84] in Python allows writing tests in function docstrings. Inline tests are similar to doctests in that both can help with code comprehension. But, doctest only supports function-level testing, while inline tests only support statement-level testing.

Regression test selection (RTS) [21, 27–29, 52, 81, 102] speeds up regression testing by only re-running tests that are affected by code changes. Section 4 showed that each inline test runs very fast compared to unit tests, but RTS for inline tests may become important as inline tests usage increases.

In-vivo testing [64] executes tests in the deployment environment, to find defects that are hidden by the clean test environment. In-vivo tests are method-level tests, while inline tests statement-level tests, and I-Test currently targets the test environment.

Fault localization [1, 2, 56, 72, 98, 99] helps finds faulty statements that cause a test failure. Inaccurate fault localization can occur for unit tests that cover many statements [53, 82]. We expect fault localization for inline tests to be more accurate since they check the immediately preceding statement that is not an inline test.

Assertions and design by contract. The `assert` construct in many programming languages, e.g., [35, 68, 83, 91], allows checking that a condition holds on the current program state. An inline tests, like an `assert` [97], can be written after any statement in the production (not test) code. Inline tests allow developer-provided input and are only to be used for test-time checking, but `asserts` do not allow developers to provide arbitrary inputs and can be used for production-time checking [76].

There is a lot of work on design-by-contract (DBC) [6, 50, 61, 63, 65, 76, 79, 89, 90] for specifying preconditions, postconditions, and invariants. DBC tools include PyContracts [90], Crosshair [79], Icontract [89] for Python, and JML [50], Jass [6], Squander [63], Deuterium [65] for Java. DBC helps check and comprehend hard-to-understand programs—goals that inline tests also target. DBC typically requires developers to use a different programming language/paradigm developers, so there may be a higher learning curve. In contrast, inline tests are written in the same language/paradigm as the code. Also, DBC enables method-level checks (except for loop invariants [22, 25, 34]), but inline tests check statements.

Domain specific languages. We provide I-Test as an API in both Python and Java. However, the design of our API was inspired by prior work on domain specific languages for writing executable comments [66] and contracts [65].

8 CONCLUSION

If developers could write tests for individual program statements, then they would be able to meet testing needs for which they currently have little to no support. Such needs are at a lower granularity level than what today's testing frameworks support, or for which currently supported levels of test granularity are ill-suited. We introduced a new kind of tests, called inline tests to help test individual statements. We implemented the first inline testing framework, I-Test, to meet language-agnostic requirements that we define. Our assessment of I-Test via a user study and via performance measurements showed that inline testing is promising—participants find it easy to learn and use inline testing and the additional cost of running inline tests is tiny. We outline several directions in which I-Test can be extended to make it more mature and to meet developer needs across programming languages.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Darko Marinov, August Shi, Aditya Thimmaiah, Zhiqiang Zang, Jiyang Zhang and the anonymous reviewers for their feedback on this work. We also thank all the user study participants. This work was partially supported by a Google Faculty Research Award and the US National Science Foundation under Grant Nos. 1652517, 2019277, 2045596, 2107291, 2217696.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *SIGPLAN Notices* 25, 6 (1990), 246–256.

[2] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *ISSRE*. 143–151.

[3] Dave Astels. 2003. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference.

[4] Sammie Bae. 2019. Bit manipulation. In *JavaScript Data Structures and Algorithms*. 339–349.

[5] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *ICSE*. 752–763.

[6] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. 2001. Jass—Java with assertions. In *RV*. 103–117.

[7] Kent Beck. 2003. *Test-driven development: By example*. Addison-Wesley Professional.

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

[9] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *ESEC/FSE*. 179–190.

[10] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *ICSE*. 572–583.

[11] Nathan Broadbent. 2022. git-remove-debug. https://github.com/ndbroadbent/git-remove-debug.

[12] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *ISSTA*. 282–293.

[13] Luis Couto. 2022. grunt-groundskeeper. https://github.com/Couto/grunt-groundskeeper.

[14] Cyril. 2022. Regex Tester. https://extendsclass.com/regex-tester.html.

[15] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *ISSRE*. 201–211.

[16] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *ESEC/FSE*. 246–256.

[17] James C Davis, Daniel Moyer, Ayaan M Kazerouni, and Dongyoon Lee. 2019. Testing regex generalizability and its implications: A large-scale many-language measurement study. In *ASE*. 427–439.

[18] Java developers. 2022. Java stream API. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[19] Firas Dib. 2022. RegEx101. https://regex101.com.

[20] Aryaz Eghbali and Michael Pradel. 2020. No strings attached: An empirical study of string-related software bugs. In *ASE*. 956–967.

[21] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *IST* 52, 1 (2010), 14–30.

[22] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification*. 65–81.

[23] Apache Software Foundation. 2022. Flink. https://github.com/apache/flink.

[24] Roy S Freedman. 1991. Testability of software components. *TSE* 17, 6 (1991), 553–564.

[25] Carlo A Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: Analysis, classification, and examples. *CSUR* 46, 3 (2014), 1–51.

[26] GitHub. 2022. Github commits containing 'Remove Debug'. https://github.com/search?q=remove+debug&type=commits.

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

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

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

[30] Google. 2022. Guava. https://github.com/google/guava.

[31] Siegfried Grabner, Dieter Kranzlmüller, and Jens Volkert. 1995. Debugging parallel programs using ATEMPT. In *SC*. 235–240.

[32] Mark Grechanik and Gurudev Devanla. 2019. Generating integration tests automatically using frequent patterns of method execution sequences. In *SEKE*. 209–280.

[33] greenDAO Team. 2022. greenDAO. https://github.com/greenrobot/greenDAO.

[34] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Communications* 12, 10 (1969), 576–580.

[35] Charles Antony Richard Hoare. 2003. Assertions: A personal perspective. *IEEE Annals of the History of Computing* 25, 2 (2003), 14–25.

[36] Keiichi Ida, Yasuyuki Ohno, Shunsuke Inoue, and Kazuo Minami. 2012. Performance profiling and debugging on the k computer. *FSTJ* 48, 3 (2012), 331–339.

[37] JetBrains. 2022. IntelliJ IDEA regular expression syntax reference. https://www.jetbrains.com/help/idea/regular-expression-syntax-reference.html#tips-tricks.

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

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

[40] Andrew Kovalyov. 2022. Debug Statements Fixers. https://github.com/akovalyov/DebugStatementsFixers.

[41] Holger Krekel and pytest-dev team. 2022. pytest. https://docs.pytest.org.

[42] Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *NAACL*. 826–836.

[43] Facebook AI Research lab. 2022. PyTorch. https://github.com/pytorch/pytorch.

[44] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*. 101–111.

[45] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE*. 1471–1482.

[46] Eric Larson and Todd Austin. 2003. High coverage detection of input-related security faults. In *USENIX Security*.

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

[48] Thomas D LaToza and Brad A Myers. 2011. Designing useful tools for developers. In *PLATEAU*. 45–50.

[49] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: A study of developer work habits. In *ICSE*. 492–501.

[50] Gary T Leavens, Albert L Baker, and Clyde Ruby. 1999. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. 175–188.

[51] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *PLDI*. 70–84.

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

[53] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How test suites impact fault localisation starting from the size. *IET Software* 12, 3 (2018), 190–205.

[54] Hareton KN Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *ICSM*. 290–301.

[55] Xiangqi Li and Matthew Flatt. 2015. Medic: Metaprogramming and trace-oriented debugging. In *FPW*. 7–14.

[56] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *SIGPLAN Notices* 40, 6 (2005), 15–26.

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

[58] Eswar Malla. 2022. DebugPurge. https://github.com/eswarm/DebugPurge.

[59] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *ISSTA*. 93–104.

[60] Glen McCluskey. 2022. Using Java reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[61] Bertrand Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51.

[62] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *ASE*. 415–426.

[63] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying execution of imperative and declarative code. In *ICSE*. 511–520.

[64] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. 2009. Quality assurance of software applications using the In Vivo testing approach. In *ICST*. 111–120.

[65] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying execution of imperative generators and declarative specifications. In *OOPSLA*. 217:1–217:26.

[66] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. 2019. A framework for writing trigger-action todo comments in executable format. In *ESEC/FSE*. 385–396.

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

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

[69] Alessandro Orso. 1998. Integration testing of object-oriented software. (1998), 119.

[70] Jaroslaw Pawlak. 2022. Bad practices of testing. https://github.com/Jarcionek/Bad-Practices-of-Testing/blob/master/src/java/presentation/_09_test_verifying_implementation_rather_than_behaviour/description.md.

[71] Michael Peacock. 2022. A case for using the @VisibleForTesting annotation. https://michael-peacock.com/a-case-for-using-the-visiblefortesting-annotation.

[72] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault

localization. In *ICSE*. 609–620.
[73] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *SQJ* 25, 1 (2017), 83–110.
[74] Jan Ploski, Matthias Rohr, Peter Schwenkenberg, and Wilhelm Hasselbring. 2007. Research issues in software fault categorization. *Software Engineering Notes* 32, 6 (2007), 6–es.
[75] pytest-html team. 2022. pytest-html plugin. https://github.com/pytest-dev/pytest-html.
[76] David S. Rosenblum. 1995. A practical approach to programming with assertions. *TSE* 21, 1 (1995), 19–31.
[77] Per Runeson. 2006. A survey of unit testing practices. *IEEE Software* 23, 4 (2006), 22–29.
[78] Adrian Santos, Sira Vegas, Oscar Dieste, Fernando Uyaguari, Ayşe Tosun, Davide Fucci, Burak Turhan, Giuseppe Scanniello, Simone Romano, Itir Karac, et al. 2021. A family of experiments on test-driven development. *ESE* 26, 3 (2021), 1–53.
[79] Phillip Schanely. 2022. Crosshair. https://github.com/pschanely/CrossHair.
[80] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. 80–90.
[81] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. In *OOPSLA*. 187:1–187:29.
[82] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*. 314–324.
[83] Richard N Taylor. 1980. Assertions in programming languages. *SIGPLAN Notices* 15, 1 (1980), 105–114.
[84] CPython team. 2022. Doctest. https://docs.python.org/3/library/doctest.html.
[85] CPython team. 2022. Python AST library. https://github.com/python/cpython/blob/main/Lib/ast.py.
[86] DeDRM Team. 2022. DeDRM tools. https://github.com/apprenticeharper/DeDRM_tools.
[87] Javaparser team. 2022. Javaparser library. https://github.com/javaparser/javaparser.
[88] JUnit team. 2022. JUnit. https://junit.org.
[89] Parquery Team. 2022. icontract. https://github.com/Parquery/icontract.
[90] PyContracts Team. 2022. PyContracts. https://github.com/AndreaCensi/contracts.
[91] Python 3.10.5 Documentation Team. 2022. Simple statements. https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement.
[92] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. *Software Engineering Notes* 30, 5 (2005), 253–262.
[93] Nikolai Tillmann and Wolfram Schulte. 2006. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software* 23, 4 (2006), 38–47.
[94] Serge Toarca. 2022. debuggex. https://www.debuggex.com.
[95] Fabian Trautsch. 2019. *An analysis of the differences between unit and integration tests.* Ph. D. Dissertation.
[96] W.T. Tsai, Xiaoying Bai, R. Paul, Weiguang Shao, and V. Agarwal. 2001. End-to-end integration testing design. In *COMPSAC*. 166–171.
[97] Jeffrey M Voas and Keith W Miller. 1994. Putting assertions in their place. In *ISSRE*. 152–157.
[98] Mark David Weiser. 1979. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* University of Michigan.
[99] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* 42, 8 (2016), 707–740.
[100] Rahulkrishna Yandrapally and Ali Mesbah. 2021. Mutation analysis for assessing end-to-end web tests. In *ICSME*. 183–194.
[101] Zexuan Zhong, Jiaqi Guo, Wei Yang, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. Generating regular expressions from natural language specifications: Are we there yet?. In *AAAI*. 791–794.
[102] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*. 430–441.