The Dissertation Committee for Yu Liu
certifies that this is the approved version of the following dissertation:

Inline Tests

Committee:

Milos Gligoric, Supervisor

Owolabi Legunsen, Co-supervisor

Christine Julien

Sarfraz Khurshid

Darko Marinov

August Shi

Inline Tests

by
Yu Liu

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin
August 2024

Acknowledgments

Five years have flown by in the blink of an eye. As I write this acknowledgment, I am grateful to people who have supported me throughout my Ph.D. journey. I have been incredibly fortunate to work with some of the most brilliant minds in software engineering. I am grateful for the mentorship, guidance, and support from my advisors, committee members, collaborators, family, and friends.

I would like to thank my advisor, Milos Gligoric for sharing his extensive knowledge and innovative ideas in research, as well as his humor and warmth in everyday life. His passion for learning and attention to detail have been exemplary. He always encourages me to explore my own research interests while providing the guidance I need to grow as a researcher. Without him, “Inline Tests” would not exist.

I would also like to thank my co-advisor, Owolabi Legunsen. His expertise in software testing has been invaluable. Whenever I have questions, he always responds with patience and wisdom. His mentorship and feedback have greatly developed my research skills. I appreciate his support and encouragement.

I also want to express my gratitude to my committee members, Christine Julien, Sarfraz Khurshid, Darko Marinov, and August Shi. I am grateful for their time and effort in reviewing my work and providing constructive suggestions.

None of the projects during my PhD would have been possible without my collaborators, Anna Guo, Alan Han, Anup K. Kalia, Junyi Jessy Li, Ali Mesbah, Pengyu Nie, August Shi, Saurabh Sinha, Aditya Thimmaiah, Zachary Thurston, Rachel Tzoref-Brill, Rahulkrishna Yandrapally, and Jiyang Zhang. I learned a great deal from each of them, especially Pengyu and August, who offered extensive advice on debugging programs, listened patiently to my project ideas, and provided invaluable feedback.

My labmates have been a constant source of support throughout my Ph.D.

journey. I would like to thank Nader Al Awar, Abdelrahman Baz, Kush Jain, Chengpeng Li, Dawei Liang, Jiefang Lin, Olivia Mitchell, Pengyu Nie, Shanto Rahman, Aditya Thimmaiah, Wenxi Wang, Yating Wu, Jiayi Yang, Zijian Yi, Ayaka Yorihiro, Hengchen Yuan, Zhiqiang Zang, and Jiyang Zhang. It has been a great pleasure to attend conferences together, share lunch, and chat in the lab.

My internship mentors and colleagues created an excellent environment for me to work on exciting projects. I enjoyed collaborating with Junyang Shen, Grace Sheng, Kaikai Sheng, Kaiyuan Wang, David You, and Jiaqi Zhang at Google, as well as Anup K. Kalia, Saurabh Sinha, and Rachel Tzoref-Brill at IBM.

I was also well supported by the excellent staff at the University of Texas at Austin. I would like to thank Thomas Atchity, Cayetana Garcia, Melanie Gulick, Alora Jones, Barry Levitch, and Melody Singleton, who helped me with paperwork so that I can focus on research.

I would like to thank other friends, including Yizhuo Du, Piao Huang, Zixin Li, Yiqing Lu, Wanying Wang, Wangyue Wang, and Zixu Wang, for their support. I am grateful for the friendship we have built over the years.

Parts of this dissertation were published at ASE 2022 [103], ICSE Demo 2023 [105] (Chapter 2), and ISSTA 2023 [104], FSE Demo 2024 [107] (Chapter 3). I would like to thank the anonymous reviewers of my papers and the audience at the conferences I attended for their comments.

My research was funded by the Google Faculty Research Award and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, CCF-2217696, and CCF-2313027.

Last but not least, I want to express my gratitude to my parents, Min Fu and Wenguo Liu, my significant other, Pengyu Nie, and my furry friend, Summer, for their unconditional love and support.

Abstract

Inline Tests

Yu Liu, PhD
The University of Texas at Austin, 2024

SUPERVISORS: Milos Gligoric, Owolabi Legunsen

Testing is essential for assuring code quality. Developers write various categories of tests, including unit tests, integration tests, and end-to-end tests to validate each program component's functionality and its interaction with other components. However, these categories of tests can be too coarse-grained or ill-suited for testing individual program statements, leading to frequent single-statement bugs. Additionally, many statements are deeply embedded within complex program logic, making it difficult to test them.

To simplify the testing of single statements and increase code coverage, we introduce *inline tests*, a novel category of tests designed to check code correctness at the statement level. We implement the first inline testing framework, I-Test, for Java and Python. Manually writing inline tests can be time-consuming and tedious, so we develop a technique to automatically extract inline tests from developer written and automatically generated unit tests.

This dissertation introduces inline tests, presents the design and implementation of an inline testing framework, I-Test, and an automatic inline-test generation technique, ExLi.

First, this dissertation motivates and introduces inline tests through several programming language features and testing scenarios in which inline tests could be

beneficial. We implement I-TEST to aid developers in writing and executing inline tests, and evaluate it on 144 statements in 31 Python projects and 37 Java projects. We also conduct a user study. All nine user study participants say that inline tests are easy to write and beneficial. The cost of running inline tests is negligible, at 0.007x–0.014x.

Second, this dissertation introduces EXLI, the first technique for automatically generating inline tests. EXLI records all variable values at a target statement (i.e., the statement to be tested) during unit test execution and uses these values as test inputs and test oracles for generating inline tests. At this point, target statements that are executed many times could have redundant inline tests. To remove redundant inline tests, EXLI uses a novel coverage-then-mutants based reduction process. Implemented for Java, EXLI generates inline tests for 718 target statements in 31 projects, reducing 17,273 initial inline tests to 905. The final set of inline tests kills up to 25.1% more mutants on target statements than developer written and automatically generated unit tests, thus improving the fault-detection capability of the test suites that they are extracted from.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Testing is essential for checking code quality during software development. Developers write various categories of tests, including unit tests, integration tests, and end-to-end tests to validate each component's functionality and its interaction with other components. However, these categories of tests can still be too coarse-grained or ill-suited for testing individual program statements. As a result, unit tests, which are commonly used for function-level testing, often fail to detect single-statement bugs [80, 81]. Existing programming languages allow developers to write complicated program logic in one statement. Some statements are hard to understand and error prone. Additionally, statements are often deeply embedded in complex program logic, posing a challenge for conventional tests to reach them.

To address these limitations of existing test categories, we introduce *inline tests* [103], a novel category of tests designed to check correctness at the statement level. They are not a replacement for unit tests but a complement to existing categories of tests. Inline tests, placed directly after the statement to be checked, i.e., the *target statement*, allow developers to specify inputs and expected outputs. For example, Figure 1.1 shows a Java code snippet that performs string manipulation. Line 5 uses a regular expression (regex) to tokenize a string. The inline test on line 6 checks if the string is correctly split into a three-element list of strings. An inline test has three components: (1) a declaration that marks its start (using `itest()` constructor), (2) assignments of inputs (e.g., assign a value to the `sql` variable with `given()`), and (3) assertions for expected outputs (e.g., check if `lines` has three elements using `checkEq()`).

To support developers in writing and executing inline tests, we define the requirements that inline testing frameworks should meet and build I-Test, the first inline testing framework, which executes inline tests and reports the results. We develop I-Test for two programming languages, Java and Python. We have also

```java
public static int executeSqlScript(Context context, Database db,
  String assetFilename, boolean transactional) throws IOException{
    byte[] bytes = readAsset(context, assetFilename);
    String sql = new String(bytes, "UTF-8");
    String[] lines = sql.split(";(\\s)*[\n\r]");
    itest().given(sql, "CREATE TABLE MINIMAL_ENTITY (_id INTEGER PRIMARY
        KEY);\nINSERT INTO MINIMAL_ENTITY VALUES (1);\nINSERT INTO
        MINIMAL_ENTITY \nVALUES (2);").checkEq(lines.length, 3);
   ... }
```

Figure 1.1: String manipulation in Java, and an inline test in blue.

integrated I-TEST with pytest, the most popular testing framework for Python, as a plugin named *pytest-inline* [105].

We evaluate I-TEST on 144 statements in 31 Python projects and 37 Java projects. The cost of running inline tests is negligible, at 0.007x for Python and 0.014x for Java on average. We conduct a user study with nine participants to evaluate the usability and effectiveness of inline tests. All study participants found inline tests easy to write and beneficial. Also, it took study participants an average of 2.5 minutes to write inline tests.

Manually writing inline tests can be time-consuming and tedious. To improve developer productivity, increase the adoption of inline tests and collect a dataset of inline tests for future research, we propose EXLI [104], a technique and tool that automatically extracts inline tests from developer written and automatically generated unit tests. EXLI automatically identifies target statements, collects runtime values of variables, and constructs inline tests from them.

Without additional processing, EXLI can generate too many inline tests, potentially exceeding the maximum allowable size of a Java method [129], and severely impacting readability and maintainability. To mitigate this excess, EXLI introduces a coverage-then-mutants based test reduction process. We consider an inline test to be redundant if it has the same fault-detection capability as other inline tests in terms of code coverage *and* mutants killed.

EXLI reduces tests in two rounds. The first round removes redundant inline tests that do not cover additional instructions of the target statement or subsequent statements in the same program scope during unit-test execution. The second round reduces the number of inline tests by performing mutation analysis, retaining a smaller set of inline tests that kill all mutants that the original inline tests kill. If no mutants are generated for a target statement, ExLi keeps all first-round coverage reduced inline tests. If no mutant for a target statement is killed by the first-round reduced inline tests, ExLi adds back all inline tests from before reduction.

For evaluation, ExLi identifies 718 target statements in 31 projects. Initially, ExLi extracts 17,273 inline tests from both developer written and automatically generated unit tests. After first-round reduction, 1,333 inline tests remain. After second-round reduction with mutant generation tools *universalmutator* [63] and *Major* [79], ExLi further reduces the inline tests to 905 with universalmutator and 930 with Major, achieving reduction rates of 94.8% and 94.6%, respectively.

We evaluate fault detection capabilities of inline tests on mutants generated by universalmutator. ExLi achieves a mutation score of 87.9% with universalmutator and 82.9% with Major. Compared to unit tests, inline tests kill 658 more mutants than unit tests with universalmutator and 645 with Major, representing 20.1% and 19.7% of all killed mutants, or 25.1% and 24.6% more mutants killed than by unit tests, respectively. The inline tests extracted by ExLi thus improve the fault detection capability of the test suites from which they are extracted. This improvement is attributed to their capacity to check the states of local and private variables, which cannot be checked by unit tests. This improvement also shows that inline tests can complement unit tests in detecting faults.

This dissertation makes the following key contributions:

⋆ **I-Test** We introduce inline tests, the benefits that they provide, and requirements for testing frameworks that support them. We implement I-Test, the first inline testing framework. We evaluate I-Test on 144 statements in 31 Python projects and 37 Java projects. Also, we conduct a user study where users find that inline tests are easy to write and beneficial. The cost of running inline tests is negligible, at 0.007x–0.014x.

⋆ **ExLi** is the first technique for automatically generating inline tests; it extracts them from unit tests. ExLi generates the largest dataset of inline tests to date. Implemented for Java, ExLi generates inline tests for 718 target statements in 31 projects, reducing 17,273 initial inline tests to 905. The final set of inline tests kills up to 25.1% more mutants on target statements than developer written and automatically generated unit tests, improving the fault-detection capability of the test suites that they are extracted from.

The code and data for I-Test [1] and ExLi [2] are open sourced to facilitate future research. We have integrated I-Test with pytest as a plugin named *pytest-inline* [3] [4] to make it easier for developers around the world to use our technique.

[1]https://github.com/EngineeringSoftware/inlinetest
[2]https://github.com/EngineeringSoftware/exli
[3]https://github.com/pytest-dev/pytest-inline
[4]https://pypi.org/project/pytest-inline/

Chapter 2: Inline Tests

Inline tests are a new category of tests that allows developers to check the correctness of single statements in their code. This chapter motivates and introduces *inline tests* in more detail. Then, we propose requirements that an inline testing framework should satisfy. Next, we describe I-Test, the first inline testing framework that allows developers to write and execute inline tests in Python and Java. We evaluate I-Test on 144 statements in 31 Python projects and 37 Java projects. The results show that I-Test has a negligible overhead, at 0.007x–0.014x. We perform a user study to understand the benefits and limitations of inline testing. All nine user study participants say that inline tests are easy to write and that inline testing is beneficial. Lastly, we discuss the limitations of I-Test and conclude the chapter.[1]

2.1 Motivating and Introducing Inline Tests

Nowadays, testing frameworks only support three levels of test granularity—unit tests, integration tests and end-to-end tests. These levels, shown in the top three layers of Figure 2.1 (known as the testing pyramid), reflect developer testing needs. Developers write unit tests to check the correctness of logical units of functionality, e.g., methods or functions [29, 149]. Integration tests are used to check that logical units interact correctly [61, 99, 130, 180]. 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 [181, 188].

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

[1]Parts of this chapter are published at ASE 2022 [103] and ICSE Demo 2023 [105]. Compared to these published papers, this chapter updates the API of the inline testing framework. For example, it changes the construct from `new Here()` to `itest()` and assertion on if conditions from `Group` to `group`.

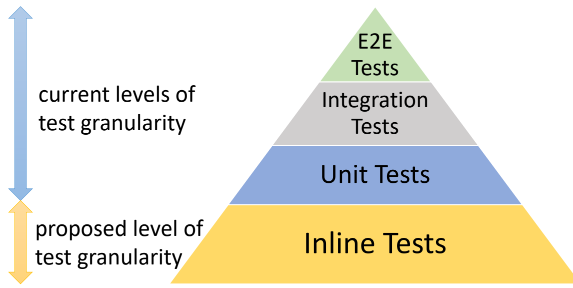


Figure 2.1: The classic testing pyramid and how inline tests extend it.

four reasons:

1. Single-statement bugs occur frequently [80, 81], but unit tests rarely fail on commits that introduce single-statement bugs [91].
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 hard-to-understand traditional programming language features like regular expressions (regexes) [30, 31, 85, 118, 193], bit manipulation [8, 95], and string manipulation [37, 90, 140].
4. Recent programming language features, e.g., Java Stream API [34], 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 2.2. First, in the commonly-practiced "`printf` debugging" [9, 14, 60, 75, 100, 137], 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` [59] developers use the "@VisibleForTesting" annotation to expose non-public variables or methods for unit testing [134, 135]. Lastly, developers lose productivity when they repeatedly use any of the many third-party websites [16, 35, 179] or in-IDE pop-ups like the one in IntelliJ [78] 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 to 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.

We introduce *inline tests*, a new category of tests that makes it easier to check individual program statements. An inline test is a statement that allows developers 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 2.1.

Inline tests could provide software development benefits beyond testing. For example, prior work showed that tests and code do not usually co-evolve gracefully [13]. 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 [113]. 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., Java [128] and Python [174]. `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's value is in range, or that a method does not return `null`. Differently, inline tests require developer-provided inputs and oracles, and they only enable *test-time checking* of individual statements.

We define language-agnostic requirements for inline testing frameworks (Section 2.3.1). For example, it should *not* be possible to use inline tests in place of unit tests or debuggers.

We implement I-Test, the first inline testing framework. The requirements that we define provide a basis for I-Test and they can provide guidance for the development of other 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 projects and 37 Java projects. 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 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 chapter 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 projects.

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

2.2 Examples

We show 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.

2.2.1 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 2.3.4). Consider this inline test that we write for a target statement in `apprenticeharper/DeDRM_tools` [168]; its target statement is shown and described in Figure 2.5:

$$\underbrace{\text{itest()}}_{\text{Declare}}\ \underbrace{\text{.given(dt, (1980, 1, 25, 17, 13, 14))}}_{\text{Assign}}\ \underbrace{\text{.check\_eq(dosdate, 57)}}_{\text{Assert}}$$

The “Declare” portion tells the inline testing framework to process the statement as an inline test. The “Assign” portion allows a 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 a 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.

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

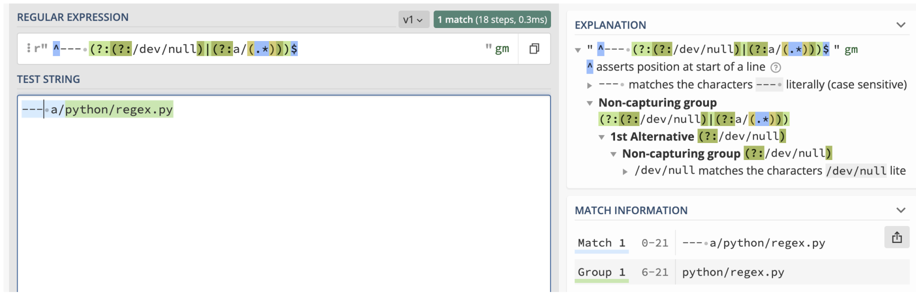
Figure 2.2: Regex example in Python code, and an inline test in blue.

2.2.2 Some Programming Language Features that Inline Tests Can Help Check

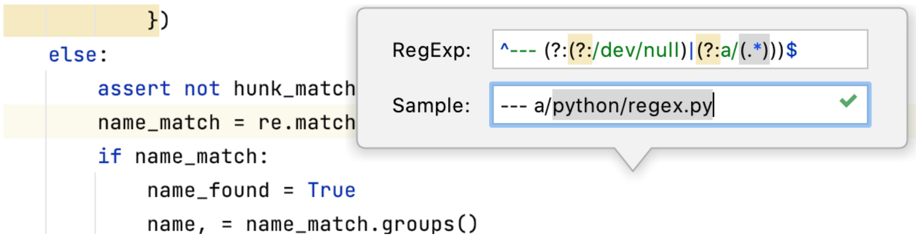
Regular expressions (regexes). Prior work showed that regexes are widely used, but they are difficult for developers to understand and to use correctly [22, 30, 31, 118]. So, inline tests allow developers to check what regexes do, and to test them in place. Consider the Python code fragment in Figure 2.2, which is simplified from `pytorch/pytorch` [86]. 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.

Checking what the regex on line 3 matches, or testing if it is correct, is difficult without direct support for statement-level testing. Three unit tests check `parse_diff` (these unit tests are written in a different file and executed using `pytest` [141]), but they mock [176] the `parse_diff` inputs and do not directly test the regex. In fact, we are unaware 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 [16, 35, 179]. Figure 2.3a shows one such website. One could also use in-IDE pop-ups like the one in Figure 2.3b for IntelliJ [78]. 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 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



(a) A regex-checking website [35]



(b) IntelliJ pop-up for checking regexes [78]

Figure 2.3: Screenshots of a website and an in-IDE pop-up for checking regexes.

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 [92, 93]. 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 2.2 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 2.2 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 2.4 shows a fix that we report to developers of a project

```
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   itest().given(orig,
        '0123456789ABCDEF0123456789ABCDEF0123').check_true(group(1))
8   clean_title = cleanup_name(book.getBookTitle())
```

Figure 2.4: Fix for faulty regex that an inline test helped find.

in our evaluation, who accepted our pull request[2]. 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 2.4, 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 2.5 shows a simplified code fragment from `apprentice-harper/DeDRM_tools` [168]. 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 2.5 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

[2]`https://github.com/noDRM/DeDRM_tools/commit/012ff53`

```
1 def FileHeader(self):
2  dt = self.date_time
3  dosdate = (dt[0] - 1980) << 9 | dt[1] << 5 | dt[2]
4  itest().given(dt, (1980, 1, 25, 17, 13, 14)).check_eq(dosdate, 57)
5  dostime = dt[3] << 11 | dt[4] << 5 | (dt[5] // 2)
6  itest().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 2.5: Bit manipulation example in Python code, 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      itest().given(sql, "CREATE TABLE MINIMAL_ENTITY (_id INTEGER PRIMARY
           KEY);\nINSERT INTO MINIMAL_ENTITY VALUES (1);\nINSERT INTO
           MINIMAL_ENTITY \nVALUES (2);").checkEq(lines.length, 3);
7      int count;
8      if (transactional) {
9          count = executeSqlStatementsInTx(db, lines);
10     }
11     ...
12     return count;
13 }
```

Figure 2.6: Another string manipulation example in Java code, and an inline test in blue.

be hard to understand. With the inline tests on lines 4 and 6, we are able to directly check these statements individually. Also the inputs and expected outputs in those inline tests document what the target statements compute.

String manipulation. Figure 2.6 shows simplified code in a method from `greenrobot/GreenDAO` [62]. Line 5 uses a regex to tokenize a string. The result of line 5 is subsequently used to query a database on line 9, 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 7 to 11. The inline test on line 6 directly tests line 5.

```
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-    itest().given(ch, 0x7F).check_eq((ord(ch)>>4)&0x000F, 0x07)
7+    itest().given(ch, chr(0x7F)).check_eq((ord(ch)>>4)&0x000F, 0x07)
8     out.write(hexdigits[ord(ch) & 0x000F])
```

Figure 2.7: Inline test helped find this string manipulation fault.

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 2.7. 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[3]. Line 7 is our updated inline test after our fix. No unit test covers this function, but there are other functions that can call it in production.

Streams. The target statement on lines 3 to 8 in Figure 2.8 uses Java Stream API; it is from `apache/flink` [43] 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 [114] (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 new `apache/flink` developer is likely to be better able to understand the code with the inline test than they would do without it.

[3] `https://github.com/python/cpython/commit/5535f3f`

```java
private CalculatedQueryOperation unwrapFromAlias(CallExpression call) {
 List<Expression> children = call.getChildren();
 List<String> aliases =
  children.subList(1, children.size())
  .stream()
  .map(alias -> ExpressionUtils.extractValue(alias, String.class)
   .orElseThrow(() -> new ValidationException("Unexpected: " + alias)))
  .collect(toList());
  itest().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;"));
 CallExpression tc = (CallExpression) children.get(0);
 return createFunctionCall(tc, aliases, tc.getResolvedChildren());
}
```

Figure 2.8: Java code using Streams, and an inline test in blue.

2.2.3 A Common Scenario: "`printf` debugging"

Developers often perform "`printf` debugging" by temporarily adding 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 link [51]. (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 [19] and others [27, 84, 111] clean up after "`printf` debugging". Figure 2.9 shows a GitHub commit[4] 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 [14], lack of platform-specific debuggers [9, 75], perceived speed [137] and simplicity [100] of "`printf` debugging", the inability of debuggers to handle parallel programming language constructs [60], etc.

[4] `https://github.com/redis/redis-om-spring/commit/f808c9b`

```
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  itest().given(subField, new Object() {@Indexed CharSequence
       f;}.getClass().getDeclaredField("f")).checkEq(sfIsTagField, true);
14   ...
15 }}}
```

Figure 2.9: Inline tests can nicely replace 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 in Figure 2.9 shows how one could manually migrate the print statements from "`printf` debugging" into inline tests.

2.3 The I-Test Framework

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

2.3.1 Inline Testing Framework Requirements

Section 2.2 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 granularity levels 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. ✓
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 [54]. ◆
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. ✗

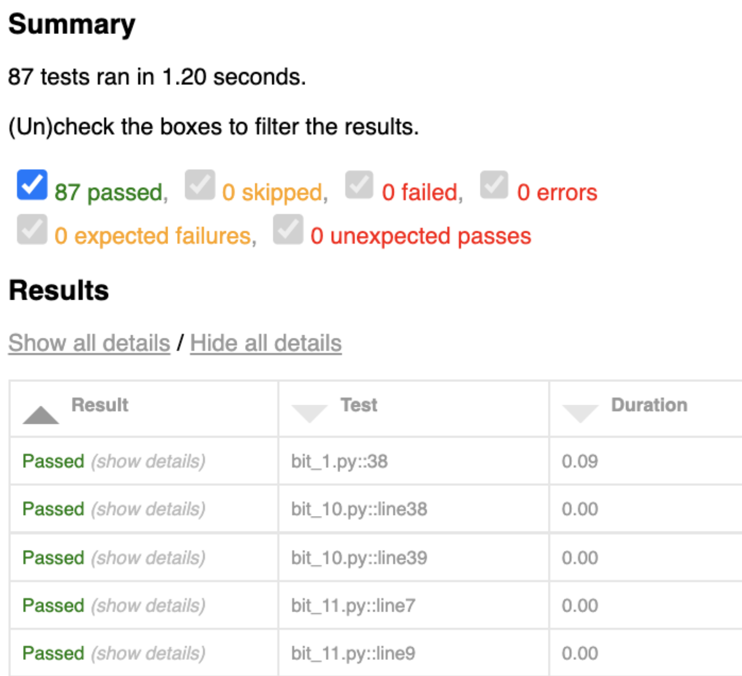


Figure 2.10: A test report in the HTML format generated by I-Test.

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 with ✓, partially supports those in Python and Java marked with ✓*and does not support those marked with ✗. The ◆ in requirements 11 and 12 means that our current Python implementation satisfies the requirements but our current Java implementation does not.

These are *initial* requirements 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.

2.3.2 Overview of the I-Test Framework

I-Test is our inline-testing framework that provides 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 integrate I-Test with two popular unit testing frameworks—`JUnit` and `pytest`. Figure 2.10 is an example test report generated by I-Test, based on the `pytest-html` plugin [142] that it uses. Inline tests must be the next statements after a target statement. 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.

2.3.3 I-Test Development Process

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 2.2.2, 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 for 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 projects, manually inspect them, and iteratively refine our I-Test API. Each example is a file from an open-source project that contains at least one target statement that we aim to test. 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 use them to guide our API design. Inline tests are not limited to these kinds of statements, but we focus on them to bootstrap. We next describe our example collection process, and provide more details on the current API.

2.3.3.1 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 2.1 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 our repo[5]. 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 the search sometimes matches the closing tag of a parameterized generic 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,

[5]https://github.com/EngineeringSoftware/inlinetest/blob/main/appendix.pdf

Table 2.1: Number of examples and the inline tests that we write to guide API design. PL= programming language, #Projs= number of projects, #Examples= number of examples, #Target stmts= number of target statements, and #Inline tests= number of inline tests.

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

based on randomly extracted 50 examples of Python and 50 examples of Java, we design the I-Test API.

2.3.3.2 Corpus

Data about the selected examples that we base our design of I-Test API on are shown in Table 2.1. 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 2.2 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 results report this variation in preferences (Section 2.6).

2.3.4 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-

Table 2.2: Breakdown of the inline tests that we wrote.

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
Streams	Java	2	2	2	2

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 `itest()` and `itest(test_name = "")` functions in Python and the `itest()` and `itest(testName)` methods in Java. With these `itest()` 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 in case it is flaky [12, 88]; (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 an inline test; (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` [169]); (5) set the timeout for an inline test, so that an inline test fails if its execution time exceeds the given duration; (6) set the assumption under which an inline test should run; and (7) set the order in which inline tests should run first.

(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 `itest(parameterized = True)` is used. This feature is similar to parameterized unit tests [177, 178].

(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. These three suffice to check the target statements in our corpus (Section 2.3.3.2). 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 support more kinds of assertions, for example, we support eight kinds of assertions in total in pytest-inline plugin [105].

Even though our initial design is based on selected examples from open-source projects, we are encouraged that our API design has produced components that should be familiar to developers who already know how to write unit tests. The API is consistent across both Java and Python. Even if small tweaks may be needed to support other programming languages, current evidence suggests that the same inline testing API components could be broadly applicable.

2.3.5 I-Test Implementation

Figure 2.11 shows the workflow of I-TEST for Python; it is similar for Java. Given a source file, `Finder` searches for statements that start with `itest` calls. `Parser`

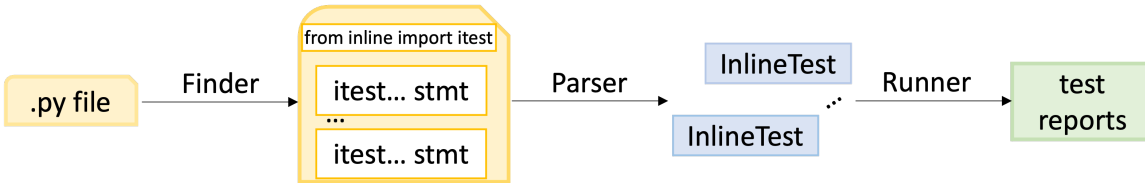


Figure 2.11: Workflow of I-Test for Python.

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 inline tests of a target statement 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 2.10.

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 [165] to parse the source code, extract the target 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 target statement exists. More precisely, I-Test uses the visitor design pattern [49] 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 [76] 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)`, I-Test looks up the declared type of `a` 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. I-Test also supports 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 that can be executed using a JUnit runner.

2.4 Usage

In this section, we describe how to use `pytest-inline` to execute tests.

2.4.1 Installation

We recommend conda [26] for installing `pytest` and `pytest-inline`. A conda environment with Python 3.9 can be created like so (`pytest` requires Python 3.7 or higher):

```
$ conda create --name inlinetest python=3.9 pip -y
$ conda activate inlinetest
```

Next, install `pytest` and `pytest-inline` in the conda environment:

```
$ pip install pytest-inline
```

2.4.2 Command-Line Interface

By default, `pytest` recursively discovers and runs all "test_*.py" or "*_test.py" files in the current directory. `pytest-inline` also recursively processes all ".py" files in the current directory. Users can specify what files to process, e.g., to run inline tests in ".py" files that start with "a":

```
$ pytest  a*.py
```

Use `inlinetest-group` to run tagged inline tests:

```
# run only the tests with tags "str" and "bit"
$ pytest --inlinetest-group="str" --inlinetest-group="bit"
```

The `-k` option allows specifying inline tests to run by name:

```
$ pytest -k "add" # run the inline tests whose names match the
    given string expression
```

Inline tests can be run in three modes: default, inlinetest-only, and inlinetest-disable. The default mode runs inline tests and unit tests; inlinetest-only mode runs only inline tests; and inlinetest-disable mode skips inline tests but runs unit tests:

```
$ pytest # run all tests
$ pytest --inlinetest-only  # run only inline tests
$ pytest --inlinetest-disable  # skip inline tests
```

When collecting inline tests, `pytest-inline` imports dependencies and throws an error if those dependencies are not installed. Users can use `inlinetest-ignore-import-errors` to ignore such errors and skip the collection of the affected files (doing so also skips the inline tests in those files):

```
$ pytest --inlinetest-ignore-import-errors
```

The default line-number order of running inline tests can be overridden using tags and `inlinetest-order`:

```
# run test tagged "str", then "bit", and then the rest
$ pytest --inlinetest-order="str" --inlinetest-order="bit"
```

Inline tests can be run in parallel after installing *pytest-xdist* [143] by using `-n` to specify the number of processes.

```
$ pip install pytest-xdist
$ pytest -n 4 # run tests in parallel with 4 processes
$ pytest -n auto # run tests in parallel with all CPU cores
```

Lastly, to generate HTML test reports, users can use the pytest-html plugin and the `html` option:

```
$ pip install pytest-html
$ pytest --html=report.html
```

2.5 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 2.3.3.2). 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.

2.5.1 Experimental Setup

Standalone experiments. To run the inline tests available 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 its inline tests. For example, the code fragment in Figure 2.6 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 method `itest` from I-TEST. On the contrary, running a unit test for the same example requires loading all these 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 `pytest-inline` 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 both Python and Java projects, 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 [12, 87, 89, 109, 157] 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 2.4a).

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 projects. 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 2.4c.

Duplicating inline tests. Since the work in this chapter is 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, 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 that we use in this chapter, 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 [129]. 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 versions required by each project in the integrated experiments.

2.5.2 Results

Table 2.3: Results of standalone experiments. Dup = duplication count, #IT= total number of inline tests, T_{IT}[s]= total inline-tests running time, t_{IT}[s]= inline-test running time per test.

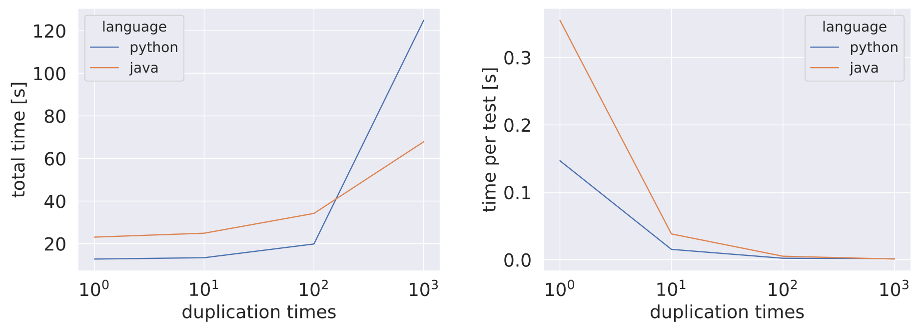
(a) Python

Dup	**#IT**	$T_{\mathbf{IT}}$**[s]**	$t_{\mathbf{IT}}$**[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**	$T_{\mathbf{IT}}$**[s]**	$t_{\mathbf{IT}}$**[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

RQ1: cost of running only inline tests. Table 2.3 shows the results of running Python and Java inline tests in 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



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

Figure 2.12: Line plots of duplication times vs. total/per-test time when running inline tests in standalone mode.

for running each inline test reduces to 0.001s for Python and 0.001s for Java. There could be two reasons for this reduction in average time. First, the cost of reading a file and extracting inline tests is amortized. Second, repeatedly executing the same inline test is faster than different inline tests.

Figure 2.12 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), but grows dramatically when duplicating 1000 times. I-Test for Java scales better than I-Test for Python, 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 2.4 shows the results of running Python and Java inline tests after integrating with the open-source projects and their unit tests. There, the $O_{\mathbf{ITE}}$ columns show the overhead when inline tests are enabled and executed during the execution of existing unit tests. Overall, without duplicating inline tests (Tables 2.4a and 2.4c), 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 2.4b and 2.4d); for example, when duplicating inline tests 1000 times, which

Table 2.4: Results of integrated experiments. Project= project name, Dup = duplication times, #UT= total number of unit tests, #IT= total number 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 all 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

Project	#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

Project	#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

brings the number of inline tests closer the number of unit tests, the overhead is 0.088x for Python and 0.008x for Java.

RQ3: overhead of running unit tests with inline tests disabled. The $O_{\textbf{ITD}}$ columns in Table 2.4 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 codebase 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 nondeterministic execution.

2.6 User Study

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

2.6.1 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. . A sample user study (without responses) is in our GitHub repository [6]. We briefly describe the activities that participants undertake.

(1) Tutorial. We provide an overview of I-Test's API (Section 2.3.4), then ask each

[6]https://github.com/EngineeringSoftware/inlinetest/tree/main/userstudy/content

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 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 with I-Test), (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 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 I-Test framework, and we confirm that none of them contributes to the open-source projects being tested. We got nine valid responses. We exclude one response who did not meet the requirement and did not provide a rationale for why inline tests are beneficial. We exclude another response that directly copied the target statement into the assignment function call (`given(var, value)`), which makes the test always pass and is not the intended use of the I-TEST API. 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.

2.6.2 User Study Results

Quantitative analysis. Our user study results are shown in Table 2.5, 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 2.8 (median: 2.8) 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, 2, and 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 (*"I would like shortcut for checking for multiple inputs"*), which we have now implemented.

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*

Table 2.5: User study results. T_{u}[min]= time to understand each task, T_{w}[min]= time to write all inline tests per task, #IT= number of inline tests, T_{w}/#IT [min]= average 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

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 2.5.2. 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"*.

2.7 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 2.3.1 shows the list of language agnostic requirements that I-Test does not yet support (✗) and those that it only partially supports (✓*). This chapter 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 [129]).

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 3.7 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 2.3.1). We already observed a consequence of this limitation in our attempt to write inline tests for statements that use Java 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 like 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) [7, 11, 150]. (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 [47] 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.

2.8 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 categories of tests are ill-suited. We introduced a new category 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.

Chapter 3: Extracting Inline Tests from Unit Tests

Despite the progress we made in chapter 2, developers still have to write inline tests manually for each target statement they want to test. But, existing code can have many target statements. So, automatic generation of inline tests is an important next step towards increasing their adoption. In this chapter, we propose ExLi, the first technique for automatically generating inline tests. [1]

3.1 Motivating and Introducing ExLi

Automatic generation of inline tests is an important next step towards increasing their adoption for two reasons. First, automatic generation can reduce manual developer effort for retrofitting inline tests into existing codebases that have many target statements. Second, automatic generation can enable future inline testing research by providing more inline tests for evaluation than exist today. For example, we previously simulated runtime costs by repeatedly executing 152 manually-written inline tests thousands of times (in chapter 2).

We propose ExLi, *the first technique for automatically generating inline tests.* ExLi extracts inline tests from unit tests. Unit tests are an attractive source of inline tests: they are abundant in practice and they can be automatically generated [44, 131, 145]. In turn, the extracted inline tests can help find single-statement bugs that unit tests miss [91]. Extracted inline tests can also help find bugs in executed statements that are deeply-nested in conditional expressions, which can be missed by automatically generated unit tests [3].

Given the code under test (CUT), a target statement, and unit tests that cover the target statement, ExLi generates a set of inline tests for the target statement.

[1] This chapter is published at ISSTA 2023 [104].

ExLi can automatically discover four kinds of target statements that we identified in previous chapter as being able to benefit from inline testing, and extract inline tests from the unit tests that cover them.

ExLi is agnostic to the source of unit tests; they can be manually written by developers or automatically generated by tools like Randoop [131, 145] or EvoSuite [44]. ExLi outputs a new version of the CUT in which the target statement is immediately followed by the generated inline tests. Since ExLi is a first step towards inline test generation, we assume that unit tests correctly exercise the CUT. That is, the inline tests generated by ExLi on one code version can detect regression bugs in future versions of the code.

ExLi first instruments the CUT to record all observed variable values in the target statement during unit testing. Then, the recorded values are used to automatically generate inline tests. For example, consider assignment statements. The recorded values of right-hand side variables are used as input values, and the recorded values of the left-hand side variable are used as expected values in the generated inline test. ExLi can also generate inline tests for declarations and expressions in `if` conditions. We plan to support more locations of target statements in the future.

Inline tests are co-located with target statements, so an important concern is that readability could be degraded if too many inline tests are generated per target statement. Compilation could also fail if adding the generated inline tests causes a method's body to exceed the maximum allowable size [129]. Too many inline tests can be generated for target statements in which many sets of values are observed during unit testing. Such many-valued target statements could be covered by many unit tests, or they may be in loops. In an extreme case, 14,928 sets of values were recorded for a target statement during our experiments.

To address the concern of generating too many initial inline tests per target statement, ExLi introduces a *coverage-then-mutants based* test reduction process. We consider an inline test to be redundant if it has the same fault-detection capability as other inline tests with respect to code covered and mutants killed. Code coverage [20, 53] and mutation score [77, 148] are established metrics for measuring the quality and fault-detection capability of unit tests. ExLi adapts these two metrics to guide inline-test reduction.

ExLi uses both *target coverage*—code covered while executing the target statement—and *context coverage*—code covered while executing the enclosing program scope of the target statement. ExLi also builds on existing mutation analysis tools [63, 79] but it only mutates the target statements.

The coverage-then-mutants based test reduction process in ExLi works as follows. ExLi tracks the code covered in the target statement and its context during unit testing, and only records sets of values that cover code that was not covered by previously extracted sets of values. ExLi also mutates the target statement and ensures that each generated inline test kills at least one unique mutant. If no mutant is generated for a target statement, ExLi's reduction is based on coverage. If coverage and mutation scores are computed, reduction prioritizes coverage, followed by mutation score. Coverage is prioritized because it is collected on the fly during unit test execution and considers the context of the target statement, while mutation score considers the target statement itself. But mutation score is also important because some previous studies have shown that mutation score is a more accurate metric of the fault-detection capability than coverage [155].

We implement ExLi for Java and apply it to 718 target statements in 31 projects. ExLi generates an initial set of 17,273 inline tests. ExLi-UM, which uses universalmutator [63] for mutation analysis, generates a final set of 905 inline tests (reduction rate: 94.8%). ExLi-Major, which uses Major [79] for mutation analysis, generates a final set of 930 inline tests (reduction rate: 94.6%).

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

This chapter makes the following contributions:

⋆ **Technique.** ExLi is the first technique for automatically generating inline tests; it extracts inline tests from unit tests.

⋆ **Reduction approach.** ExLi uses a novel inline test reduction approach that is based on both code coverage and mutation score.

⋆ **Evaluation.** ExLi's reduction strategy is effective, yielding inline tests that improve the fault-detection capability of unit test suites.

⋆ **Dataset.** ExLi generates the largest dataset of inline tests to date. ExLi and our dataset can enable future work on inline tests.

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

3.2 Example

Figure 3.1 shows an example code with a target statement and inline tests that ExLi generates for that target statement after reduction. The example is simplified from `mp911de/logstash − gelf` [108]. Method `setAdditionalFields` splits the value stored in `spec` using `MULTI_VALUE_DELIMITTER` (",") as the delimiter, stores the results in `properties`, and adds each `field` in `properties` that contains `EQ` ("=") to `gelfMsg`. Line 7 is the target statement; it finds the index of the first occurrence of `EQ` in `field`. All variables in this example have primitive or String types, but ExLi

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

Figure 3.1: A target statement with ExLi-generated inline tests.

supports complex non-primitive types as well (see example in Figure 3.6, Section 3.4). A developer could use ExLi to generate inline tests for this target statement; it is in a loop and it is reached by lots of other methods.

Line 8 is one of the two inline tests that ExLi generates. All inline tests have three parts. First, the "Declare" part—`itest()`—marks the current statement as an inline test. Second, the "Assign" part—`given(field, "profile.requestStart.ms")` and `given(EQ, '=')`—provides inputs to the inline test. Third, the "Assert" part—`checkEq(index,` $-$`1)`—specifies a test oracle, including an expected output. In Figure 3.1, given the inputs for `field` and `EQ`, the `index` variable computed by the target statement should be $-$`1` for the inline test on line 8 to pass.

The example target statement is executed 2,413 times with 215 unique sets of values during unit testing. But, directly generating 215 inline tests to check one statement could be overkill for two reasons. First, many of the 215 sets of values are redundant because they exercise the target statement in the same way. So, using them all is wasteful. Second, adding 215 inline tests for this target statement will likely make the code harder to read and maintain. So, ExLi must reduce the number of generated inline tests by eliminating redundancy. ExLi's coverage-then-mutants based reduction process reduces those 215 inline tests to the two shown in Figure 3.1, without loss in fault-detection capability.

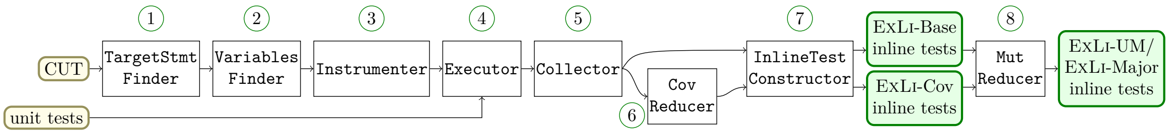


Figure 3.2: The steps in ExLi's workflow.

3.3 Technique

Figure 3.2 shows ExLi's procedure for generating inline tests. The inputs are the CUT (required), the unit tests (required), and line numbers of target statements (optional, not shown). ExLi outputs the generated inline tests after the coverage-then-mutants based reduction. ExLi also produces two intermediate outputs for evaluation and debugging purposes: ExLi-Base inline tests without any reduction; and ExLi-Cov inline tests with reduction based only on code coverage but not mutation score.

3.3.1 Finding and Analyzing Target Statements

The first two steps of ExLi's workflow are for finding and analyzing the target statements. In step 1, `TargetStmtFinder` parses the abstract syntax tree (AST) of the CUT and extracts the target statements. If developers provide the optional input of line numbers of target statements, ExLi will skip this step and directly use the developer-specified target statements. Then, in step 2, `VariablesFinder` identifies the variables used in each target statement, which will be used as the input or output variables in the generated inline tests. For example, `VariablesFinder` should identify three variables for the target statement in Figure 3.1: two input variables, `field` and `EQ`, and one output variable `index`.

3.3.2 Generating Inline Tests

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

The `Instrumenter` (step 3) adds code *before* each target statement to collect the values of input variables and *after* each target statement to collect the values of output variables. Figure 3.3 shows how we instrument the code in Figure 3.1: `collectInputs` (line 7) is added before the target statement to collect the values of `field` and `EQ`, and `collectOutputs` (line 9) is added after the target statement to collect the value of `index`. Other code added by `Instrumenter` for inline-test reduction is described in Section 3.3.3.

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

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

The `InlineTestConstructor` (step 7) also edits the CUT to insert constructed inline tests right after the target statement. After that, ExLi uses I-Test (our inline testing tool for Java in chapter 2) to run each generated inline test. If any inline test fails, ExLi filters it out: the failing inline test is removed from the CUT. Such failing inline tests are due to the target statement using inputs other than the input variables (e.g., a static variable used in a method invoked from the target statement). ExLi does not collect such inputs; future work can explore storing such inputs from the global program state.

```
public static void setAdditionalFields(String spec,GelfMsg gelfMsg){
  if (null != spec) {
    String[] properties = spec.split(MULTI_VALUE_DELIMITTER);
    for (String field : properties) {
      try {
        collectCov(); // cov1
        collectInputs(field, EQ);
        final int index = field.indexOf(EQ); // target statement
        collectOutputs(index);
        collectCov(); // cov2
        if (-1 == index) { continue; }
        ... // add field to gelfMsg
      } finally { collectCov(); } // cov3
}}}
```

Figure 3.3: Example showing how ExLi instruments a target statement.

3.3.3 Coverage-then-Mutants Based Reduction

ExLi-Base generates an inline test for each unique set of values collected while executing unit tests. But, too many sets of values could be collected for some target statements even if we only keep unique sets of values (Section 3.1). We observe in our experiments that many sets of values are redundant with respect to one another: they have similar fault-detection capability and exercise the target statement in the same way. (Recall that, from a unit testing point of view, the sets of values that ExLi collects are intermediate values.)

To avoid generating redundant inline tests, ExLi uses a novel *coverage-then-mutants based* test reduction process: reducing the inline tests (or sets of values, if reducing before constructing inline tests) that have redundant fault-detection capability, using both code coverage [20, 53] and mutation score [77, 148] as metrics for fault-detection capability.

3.3.3.1 Reduction by Code Coverage

ExLi collects code coverage using JaCoCo [121], a widely-used code coverage tool for Java. To fit the inline testing scenario, ExLi considers two kinds of code coverage: *target coverage*, the coverage collected while executing the target statement;

and *context coverage*, the coverage after executing the target statement while executing the *context* of the target statement. The context of a target statement is defined as code between the target statement and the end of its enclosing program scope. For example, for the target statement in Figure 3.1 (line 7), its enclosing program scope is the `for` loop from lines 6 to 12, and its context is the code from lines 10 to 12.

Using context coverage in addition to target coverage makes reduction more accurate. The target coverage alone may not provide enough information to distinguish non-redundant inline tests. For example, the inline tests at line 8 and line 9 in Figure 3.1, which have different fault-detection capability, have the same target coverage, but they have different context coverage because only the first inline test covers the `then` branch of the `if` statement in the context at line 10.

To collect target coverage and context coverage, `Instrumenter` (step ③) adds code to collect code coverage at three points, see the `collectCov` calls in Figure 3.3: (1) the instruction-level coverage just before the target statement (line 6, `cov1`), (2) the instruction-level coverage right after the target statement (line 10, `cov2`) and (3) the instruction-level coverage at the end of the target statement's enclosing program scope (line 13, `cov3`). Then, `CovReducer` (step ⑥) processes each collected set of values and instruction-level coverage information. Only sets of values that increase either target coverage or context coverage of the corresponding target statement are kept and sent to `InlineTestConstructor`.

The SHOULDKEEPVALUES procedure in Algorithm 1 describes how `CovReducer` computes the target coverage and context coverage and decides when to keep a set of values. The inputs are code coverage information `cov1`, `cov2`, `cov3`, and target statement ℓ0. `CovReducer` uses a global map, `tgtStmtToCovered`, to store the code coverage metric: the lines of code covered by the collected sets of values (which is initialized to empty) of each target statement. SHOULDKEEPVALUES checks if the target coverage changed (line 2) and if the context coverage changed (line 3) and returns `true` if either changed. COVCHANGED compares the code coverage at two

Algorithm 1 `CovReducer`

Global var: `tgtStmtToCovered`: mapping from target statement to the set of lines covered by the target statement's collected values

Inputs: `cov1`, `cov2`, `cov3`: code coverage information for the current set of covered instructions; `ℓ0`: target statement's line number

Output: `true` if the set of values should be kept, `false` otherwise

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

points, and checks if the later one has covered any line not covered by the former one (line 8) and that line was not covered by previously collected values (line 9). If so, covChanged updates `tgtStmtToCovered` and returns `true`. The instruction-level coverage reported by JaCoCo is a mapping from line number to the count of instructions on that line being covered. So, line 8 considers a line's coverage as changed if its instruction counts changed (from zero to non-zero; or, from non-zero to a larger value for ternary operators or Boolean expressions).

3.3.3.2 Reduction by Mutation Score

Mutation score is an established measure of the fault-detection capability of tests [77, 148]; it is the ratio of mutants killed by tests (i.e., that cause the tests to fail) to the total number of mutants. Mutants are typically small syntactic modifications to the CUT that simulate seeded faults. ExLi uses two popular mutation generators for Java: universalmutator [63] and Major [79]. ExLi uses all default mutation operators in the two generators, but it only mutates target statements. To do so, we specify line numbers to mutate (for universalmutator) or filter out mutants that are not for the target statements (for Major).

`MutReducer` (step 8) performs reduction by mutation score, given the ExLi-Base inline tests without reduction and ExLi-Cov inline tests after reduction by code coverage. Note that the mutant generator may fail to generate mutants for some target statements (9.6% for universalmutator, 8.9% for Major), in which case mutation score cannot be computed, and `MutReducer` will directly output the ExLi-Cov inline tests for those target statements. For all other target statements, `MutReducer` further reduces the coverage-reduced inline tests by mutation score, which prior work suggests measures fault-detection capability more accurately than coverage [155].

`MutReducer` first executes the ExLi-Base and ExLi-Cov inline tests on the mutants and maps each inline test to mutants that it kills. Then, `MutReducer` uses the Greedy test-suite reduction algorithm [189] (used in prior work [155, 156, 158]), based on the mapping of ExLi-Cov inline tests to killed mutants, to minimize the set of ExLi-Cov inline tests that kill the same mutants. Each inline test in the reduced set kills at least one unique mutant. Finally, if ExLi-Base inline tests kill any mutant that is not killed by the reduced ExLi-Cov inline tests, then reduction by coverage would result in a loss in mutation score. So, `MutReducer` adds one ExLi-Base inline test that killed that mutant to the reduced inline tests to remedy this loss.

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

Remark 1. Conceptually, ExLi could directly use test-suite reduction with respect to mutants on the target statement to reduce the collected sets of values. Instead, we make the design choice to first use reduction by code coverage for three reasons. First, using mutants for minimization requires to first generate inline tests for all the collected sets of values. It is not always possible to do so due to limits on method sizes [129]. Second, using reduction by code coverage has the benefit that we can use mutation testing as a sanity check of the fault-detection capability of the reduced set of inline tests. There would be no automated sanity check if mutation testing is used initially. Lastly, ExLi will need to preserve all inline tests for target statements in which no mutant is created. So, if ExLi only uses reduction by mutation score and if a frequently covered target statement has no mutants, then readability may degrade because too many inline tests are generated.

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

3.4 Implementation

We describe our ExLi implementation, using the same step numbers as in Section 3.3 to make our descriptions easier to follow.

1 **Find target statements**. ExLi currently supports finding the same four kinds of Java target statements mentioned in chapter 2.2.2: regular expressions, string manipulation, bit manipulation, and stream processing. Given a kind of target statement,

Table 3.1: Search terms used to filter statements.

Type	**API**
Regex	Matcher.matches(), Matcher.find(), Matcher.group()
String	String.split(), String.substring(), String.indexOf(), String.format(), String.replace()
Bit	», «, &, —, ˆ, ˜, &=, —=, ˆ=, »=, «=
Streams	Stream.of(), *.stream()

`TargetStmtFinder` searches for target statements that use APIs that are commonly used in the kinds of statements of interest. Table 3.1 lists the terms that ExLi searches for. Unlike our earlier I-Test prototype that searches program text, ExLi improves accuracy by parsing the AST (using JavaParser [76]) to find target statements.

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

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

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

③ **Instrument CUT**. `Instrumenter` is implemented using JavaParser. ExLi currently supports instrumenting target statements at three syntactic locations:

- *Condition of an* `if` *statement.* Figure 3.4 shows an example from `json-schema-validator` [123]. Line 7 is the target statement; it checks if `value` matches a pattern. `Instrumenter` adds code before the `if` statement (line 6) to collect input variables, at the beginning of the `then` branch (line 8) to collect `true` as the value of the output variable—the result of evaluating a conditional expression, and at the start of the `else` branch (line 15) to collect false as the value of the output variable.

```java
public String[] match(String value) { ...
  for (int i = 0; i < patterns.length; i++) {
    try {
      Matcher matcher = patterns[i].matcher(value);
      collectCov(); // cov1
      collectInputs(matcher);
      if (matcher.matches()) { // target statement
        collectOutputCond(true);
        collectCov(); // cov2
        int count = matcher.groupCount();
        String[] groups = new String[count];
        for (int j = 0; j < count; j++)
          groups[j] = matcher.group(j + 1);
        return groups;
      } else { collectOutputCond(false); }
    } finally { collectCov(); } // cov3
  }
  return null; }
```

Figure 3.4: Example of ExLi instrumenting a target statement at a condition of an `if` statement.

- *Declaration statement.* `Instrumenter` adds code before the target statement to collect right-hand side variable values and after the target statement to collect left-hand side variable values.
- *Assignment statement.* `Instrumenter` adds code to collect left- and right-hand side variable values before the target statement and to collect left-hand side variable values after the target statement. Left-hand side variables are collected both before and after the target statement, because they may be both input and output variables in compound assignment statements like `a += 1`.

Moreover, `Instrumenter` handles the following special cases:

- If there is an increment/decrement expression in an array index, `Instrumenter` rewrites the array-indexing expression such that the correct element is collected. For example, in Figure 3.5, the output variable on line 6 is $\texttt{mOutBuffer}[\texttt{ptr}++]$, but its value is collected on line 7 as $\texttt{mOutBuffer}[\texttt{ptr}-1]$ because `ptr` would be incremented after executing the target statement.

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

Figure 3.5: Example of ExLi instrumenting a target statement with an increment expression in an array index.

- Some target statements are in `if` blocks that have jump (`return`, `break`, `continue`, `throw`, etc.) instructions in the `then` and `else` branches. To avoid compilation error (unreachable code) that would occur if `Instrumenter` adds code to the end of blocks in such branches, `Instrumenter` always wraps the parent node of the target statement in the AST in a `try` block. If the target statement's parent node is a constructor body whose first statement is a constructor call (e.g., `super`() or `this`()), ExLi excludes such constructor calls from the `try` block to avoid compilation error (super/this has to be the first statement).

4 Execute unit tests and 5 collect values. `Executor` runs unit tests on the instrumented CUT and the `Collector` stores the values of input and output variables that are observed during execution. ExLi is agnostic to the source of unit tests; they can be manually written or automatically generated. We currently use Randoop [131, 145] and EvoSuite [44] for automatic unit test generation; future work can investigate other test generators.

When the variable whose value is to be collected is of a primitive type, a wrapper type for a primitive type, a `String`, or an array of these types, `Collector` directly stores the collected values (which will be used on the constructed code for the inline test). Otherwise, `Collector` uses XStream [187] to serialize the values, which

```
1 public CompiledTemplate compile(IdentifiableStringTemplateSource
2     templateSource) throws TemplateException {
3   // target statement
4   String id = templateSource.getId().replace('/', ';');
5   .itest().given(templateSource, "25.xml")
6           .checkEq(id, ";root;body@;folder;descriptor.txt");
7   String source = templateSource.getSource();
8   StringTemplateSource currentTemplateSource =
9     (StringTemplateSource) templateLoader.findTemplateSource(id)
        ;
10  ... }
```

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

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

(b) The contents that are serialized to "25.xml".

Figure 3.6: An inline test that saves an object to an XML file.

will be deserialized in future executions of the generated inline test. This support for complex non-primitive types was introduced in ExLi, not I-Test.

Figure 3.6 shows an example inline test using XStream to support complex non-primitive types, from craftercms/core [167]. Line 4 is the target statement; it replaces "/" in `templateSource`'s `id` with ";". Line 5 is an inline test that ExLi generates. The variable being assigned, `templateSource`, is of a complex non-primitive type `IdentifiableStringTemplateSource`, whose value is serialized into "`25.xml`" (Figure 3.6b).

6 **Reduction by code coverage**. `CovReducer` reduces redundancy by removing sets of variable values that do not increase the coverage rate of the target statement, which means that they have the same effect on the statement.

We set JaCoCo [121], the code coverage tool used by ExLi, to instrument and collect all classes in the current project and dependency libraries, including the Java standard library. However, some classes in the Java standard library (e.g., `java.lang.String`) are loaded during JaCoCo initialization and are thus not instrumented. To avoid missing coverage information in such classes, especially for string-related and regex-related target statements, our implementation uses wrapper classes that we write for `java.lang.String` and `java.util.Matcher` so that the method calls of these classes can be instrumented. It is necessary to wrap `java.util.Matcher` because some `java.lang.String` methods that are used by our evaluation subjects depend on it.

7 **Construct inline tests**. `InlineTestConstructor` creates the inline tests at the AST level with the help of JavaParser [76].

8 **Reduce by mutation score**. `MutReducer` performs mutation analysis, using universalmutator [63] and Major [79], and test-suite reduction, using an existing implementation [154], to further reduce the generated inline tests. The test-suite reduction implementation [154] supports four algorithms: Greedy [189], GE, GRE [23], as well as HGS [71]. We found that the four algorithms always result in the same number of inline tests in the reduced set (but different inline tests are selected) in our experiments, so we set Greedy as the default algorithm.

3.5 Evaluation

We answer the following research questions:

RQ1: How many inline tests does ExLi generate *before* reduction?

RQ2: How many inline tests does ExLi generate *after* reduction?

RQ3: How effective are the generated inline tests in terms of fault-detection capability, compared with unit tests?

RQ4: What is the runtime cost of ExLi?

Experimental environment. We run all experiments on a machine with Intel Core i7-11700K @ 3.60GHz (8 cores, 16 threads) CPU, 64 GB RAM, Ubuntu 20.04, Java 8, and Maven 3.8.6.

Table 3.2: Projects used in our evaluation.

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

Table 3.3: Statistics about unit tests used in this chapter.

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

3.5.1 Curating an Evaluation Dataset

We start with a large set of projects from a work on learning to complete unit tests [126]. That prior work used different experimental requirements than this work to filter projects. So, we start from the original unfiltered set containing 1,535 Java projects that use Maven, have no compilation error, and have appropriate licenses. To simplify our experiments, we select the subset of 1,209 single-module projects. From these, we select the 128 actively-maintained projects that have commits after January 1, 2022, to facilitate future work on integrating the generated inline tests into these

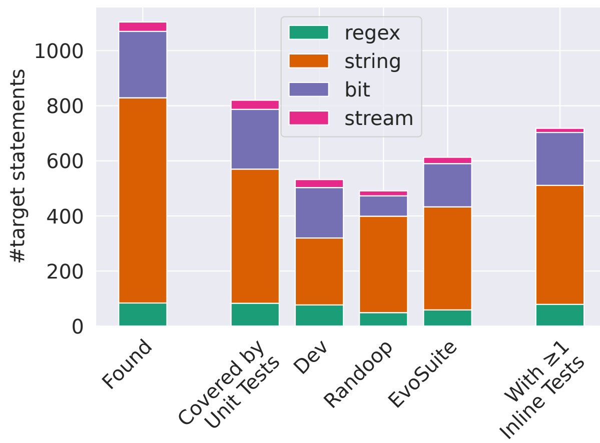


Figure 3.7: Number of target statements that we find for four kinds of APIs, covered by (all, developer written, Randoop, and EvoSuite) unit tests, and for which ExLi generates inline tests.

projects. Next, we filter out projects in which developer written unit tests fail (84 remain), in which JaCoCo fails (73 remain), and in which Randoop or EvoSuite fails (48 remain).

On these remaining 48 projects, we use ExLi to find target statements and generate inline tests. We filter out 6 projects that do not have the kinds of target statement that we look for (section 2.2.2); one project where all target statements are not covered by any unit test; and one project for which ExLi does not generate any passing inline test because XStream could not serialize an object. We also filter out 8 projects where ExLi's instrumentation clashes with the projects' instrumentation for other purposes, and one project where developer written tests take more than one hour.

We use the remaining 31 projects as our evaluation subjects. Table 3.2 shows the PIDs and names of these projects, the commit SHA that we use, and total lines of Java code (LOC). Figure 3.7 shows statistics about the number of target statements in the 31 projects. ExLi initially finds 1,104 target statements (84 for regular expression, 745 for string manipulation, 241 for bit manipulation, and 34 for stream operations). Of these, 820 target statements are covered by at least one unit test (532 are covered by at least one developer written unit test, 491 are covered by at least one Randoop-

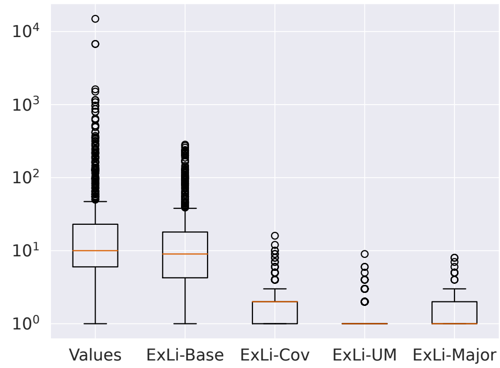


Figure 3.8: Distribution of inline tests per target statement.

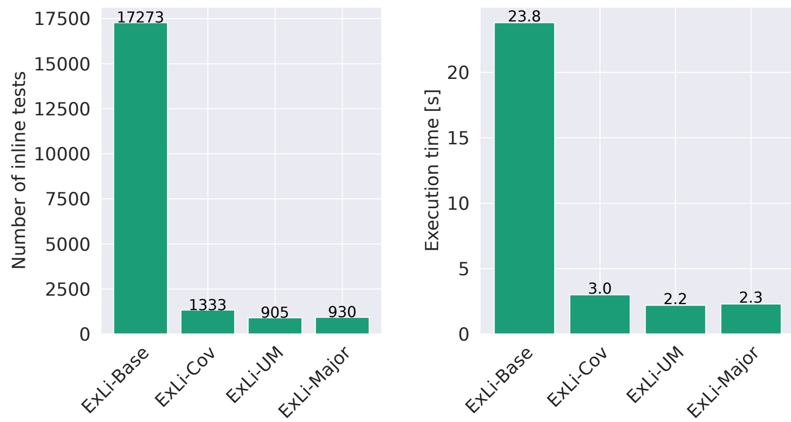
generated unit test, and 613 are covered by at least one EvoSuite-generated unit test). After removing failing inline tests and corresponding target statements, ExLi generates inline tests for 718 target statements (79 for regular expression, 432 for string manipulation, 192 for bit manipulation, and 15 for stream operations); we use them in the rest of our evaluation.

3.5.2 Extracting Inline Tests

First, we run Randoop and EvoSuite to obtain automatically generated unit tests for each project in our dataset. We run Randoop with a time limit of 10 minutes to generate unit tests for each project (as suggested by the Randoop user manual [175]); we set other options to default values. We run EvoSuite with a time limit of 120 seconds (as suggested by the configuration in the recent SBST competition [152]) for each class with at least one target statement.

Table 3.3 shows the statistics about the unit tests: number of test methods (**#tests**), test-running time (**T[s]**), line coverage (**L[%]**), and branch coverage (**B[%]**). Note that EvoSuite's line and branch coverage for some projects are low. Because it is setup to only generate unit tests for classes with target statements, which may be a small proportion of the CUT.

Next, we run ExLi to extract inline tests from unit tests. We compile and run developer written, Randoop-generated, and EvoSuite-generated tests separately



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

Figure 3.9: Number and execution time of inline tests extracted by ExLi with different levels of reduction.

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

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

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

We compare the four sets of inline tests generated by ExLi as intermediate or final results (also see workflow in Figure 3.2): *ExLi-Base* without reduction, *ExLi-Cov* with only reduction by code coverage, *ExLi-UM* with coverage-then-mutants based reduction using universalmutator, and *ExLi-Major* with coverage-then-mutants based reduction using Major.

Figure 3.8 shows the distribution of generated inline tests per target statement. We also include the number of unique sets of variable values collected during execution of unit tests (denoted as *Values*), to show the number of inline tests that ExLi would generate without setting the 300 upper limit. The average number of inline tests per target statement for Values, ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major are 88.9, 24.1, 1.9, 1.3, and 1.3, respectively. The medians for Values, ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major are 10.0, 9.0, 2.0, 1.0, and 1.0, respectively.

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

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

Figure 3.9 shows the number of inline tests and their execution time (note that we did not include compilation time here). To evaluate the effectiveness of ExLi's reduction, we consider ExLi-Base as the baseline before reduction; it generates 17,273 inline tests that take 23.8 seconds to execute.

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

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

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

3.5.3 Performing Mutation Analysis

In this section, we perform mutation analysis using the mutants [25, 138] for the target statements generated by universalmutator. We reuse the same mutants that universalmutator generated during step 9 in Section 3.4 for reducing inline tests. We report results based on the 649 target statements that have non-stillborn mutants [4], and compare the mutation scores of inline tests generated by ExLi against unit tests. Note that universalmutator did not generate any mutant for any target statement in `liquibase/liquibase − oracle` (P15), so we excluded it from the mutation analysis evaluation.

Table 3.4: Mutation analysis evaluation results. P15 is excluded because no mutant was generated for it.

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

Table 3.4 shows the number of tests and mutation scores of developer written, Randoop-generated, and EvoSuite-generated unit tests, and ExLi-Base, ExLi-Cov, ExLi-UM, and ExLi-Major inline tests. Note that the mutation scores of ExLi-UM and ExLi-Base are always the same by design, because during the mutation-based reduction, ExLi adds any inline test from ExLi-Base that kills a mutant that survives ExLi-Cov inline tests. The average mutation score of ExLi-Base is 87.9%, which is much higher than the mutation score of developer written (67.9%), Randoop-generated (31.4%), and EvoSuite-generated (43.8%) unit tests. These scores are computed only on the target statement. ExLi-Cov achieves 80.4%, slightly lower than ExLi-Base, but higher than the mutation score of unit tests. By performing additional mutation-based reduction, ExLi-UM fully recovers the mutation score to 87.9%, and ExLi-Major improves the mutation score to 82.9%. The difference between ExLi-UM and ExLi-Major is small, and suggests that the two mutation generation tools are quite similar (see also reports in prior work [63]).

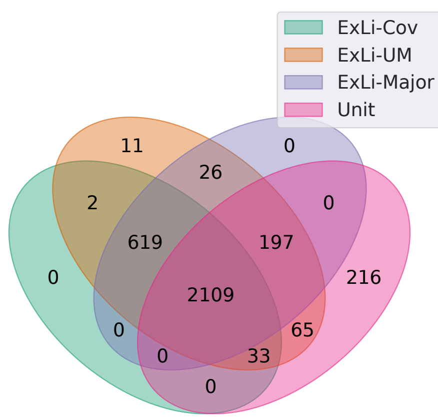


Figure 3.10: Sets of mutants killed by inline tests and unit tests.

Figure 3.10 shows a Venn diagram illustrating the overlap among the sets of mutants killed by all unit tests (named `Unit` in the figure) and inline tests from ExLi-Cov, ExLi-UM (which is the same as ExLi-Base), and ExLi-Major. All inline tests and unit tests kill 3,278 mutants in total. 2,404 mutants are killed by both inline tests and unit tests. The set of mutants killed by ExLi-Major inline tests is a subset of the set of mutants killed by ExLi-UM inline tests, but the difference is small: ExLi-UM inline tests kills 111 or 3.8% more mutants than ExLi-Major inline tests. Compared with ExLi-UM inline tests, ExLi-Cov inline tests miss 299 mutants (9.1% of all killed mutants). Compared with unit tests, ExLi-UM inline tests miss 216 mutants (6.6% of all killed mutants). This is because unit tests can check global program state (e.g., fields) that is modified by the target statement, but inline tests currently cannot; future extensions of inline tests can address this limitation. But, ExLi-UM kills 658 more mutants than unit tests (20.1% of all killed mutants or 25.1% of mutants killed by unit tests).

We manually inspect surviving mutants that lead to loss of mutation scores when ExLi-Cov is compared with ExLi-Base. So far, we found two limitations of ExLi that lead to such intermediate losses. (1) There are multiple clauses in an `if` condition, but the mutation operator only modifies one of them. This limitation occurs because, unlike pytest-inline [105], I-Test does not yet support testing individual clauses in a condition. This limitation will go away as I-Test matures. (2) Multiple sets of values can kill a mutant but they all cover the target statement and its context in the same way as a chosen set of values that cannot kill the mutant. This is a limitation of reduction by coverage as we discussed in Section 3.3.

Observe from Figure 3.10 that inline tests and unit tests are complementary in terms of their fault-detection capability on the target statements. So, inline tests can enhance the fault-detection capability of the unit test suites from which they are extracted. To understand why some mutants on target statements can be killed by inline tests but not by the unit tests, we manually inspected 63 randomly sampled mutants from the 658. We found two reasons: (1) unit tests lack good assertions to kill the mutants, i.e., the mutant could be killed if we add assertions to the unit tests (77.8% of cases); (2) the mutant does not change program state that propagates to unit tests, i.e., it only changes local variables or control flow but not the return value or global variables, but inline tests' "local" assertions kill such mutants (22.2% of the cases).

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

3.5.4 Measuring ExLi's Runtime Cost

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

We are very encouraged by these early results on runtime costs, especially when compared with our estimated amount of time that it would take developers to write all 905–930 inline tests that ExLi generates. Our prior user study (section 2.6) showed that participants spent around 6.3 minutes (378s) to understand and write inline tests for each target statement in Python. Assume that the times to understand target statements and write inline tests is uniformly distributed and are the same for Java and Python. Then, on average, participants would have needed 271,404s (∼75 hours) to write inline tests for all 718 target statements that we use.

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

3.6 Discussion

Limitations. (1) ExLi uses coverage of the target statement and its context for initially reducing the set of inline tests. Flaky tests [12, 65, 88, 109, 133, 157] can cause coverage to fluctuate. We do not control for flaky tests in the unit tests that ExLi uses. (2) Extracted inline tests may be flaky and fail if the expected output in the oracles that are generated depend on data that may change, e.g., current date or device configuration. (3) When potential inputs cause the target statement or its context to throw an exception, ExLi does not use such values to construct inline tests because I-Test [103] does not yet support using expected exceptions as test oracles. (4) We do not evaluate the extracted inline tests with developers of the open-source projects that we evaluate. But, we have initial confidence from our prior user study, which showed that participants find inline tests useful. We plan to communicate more with open-source developers in the future, especially as I-Test matures.

Threats to validity. Our code to instrument target statements, collect coverage rates, and perform reduction could contain bugs. To mitigate this threat, we reviewed the code and inspected the results. Our findings could be limited to projects that we evaluate and their unit tests. To mitigate this threat, we used open-source projects with various characteristics and used automatically generated unit tests. The ideas in ExLi are general but our results may not generalize to other programming languages. We plan to use our *pytest-inline* tool [105] as a basis for a tool that extracts inline tests from Python unit tests.

3.7 Conclusion

In this chapter, we presented ExLi, a technique for automatically generating inline tests with coverage-then-mutants based test reduction. The coverage-based reduction is based on context-aware coverage feedback, and the mutation-based reduction is based on killed mutants. We evaluate ExLi on 31 Java projects and find that ExLi generates between 905 (when using universalmutator to reduce tests) and

930 (when using Major to reduce tests) inline tests for 718 target statements. ExLi reduces initially generated inline tests by more than 94%. ExLi enables developers to enhance the fault-detection capability of their test suites by easily obtaining and adding inline tests.

Chapter 4: Related Work

This chapter presents prior work in the area of testing that are most related to inline tests that are presented in this dissertation.

Testing and debugging. Karampatsis and Sutton [81], Kamienski et al. [80], and Richter and Wehrheim [144] curated datasets of single-statement bugs (SStuBs) in Java and Python. Also, Latendresse et al. [91] find that continuous integration rarely detects SStuBs. These works show that many bugs are caused by faults in single statements, and that unit tests miss such bugs. They further motivate the need for direct support for checking individual statements, which inline tests provide.

The ManySStuBs4J [81] dataset contains single-statement bugs that are curated by statically analyzing open-source Java projects and their version histories. As the ManySStuBs4J dataset evolves to capture more recent versions of those projects, it can be a benchmark for evaluating the bug-detection capability of inline tests. We do not use ManySStuBs4J because (1) the filtering process that was followed to curate the dataset resulted in many false positives during our initial search for target statements; (2) the commits used in the dataset are from before 2019, so we had trouble running the unit tests in some projects.

Michael et al. [118] found that regexes are hard to read, find, validate, and document. Eghbali and Pradel [37] also found that string-related bugs are common in JavaScript programs. Section 2.2 discussed how inline tests can mitigate these problems and how I-Test helped find regex-related and string-manipulation bugs.

Doctest [166] in Python allows writing tests in function docstrings. Inline tests are similar to doctests in helping with code comprehension. But, doctest only supports function-level testing, while inline tests only support statement-level testing.

In-vivo testing [122] executes tests in the deployment environment to find defects hidden by the clean test environment. In-vivo tests are method-level tests, while inline tests are statement-level tests, and I-Test targets the test environment.

"ppx inline tests" [162] and the inline tests in our paper [103] share a name and the characteristic that they are co-located with code. But, "ppx inline tests" check the correctness of functions instead of single statements. Xiong et al. [186] propose inner oracles: assertions declared in unit tests to check internal states. Inline tests allow specifying both oracles and test inputs to check single statements.

Fault localization [1, 2, 101, 136, 184, 185] helps find faulty statements that cause a test failure. Inaccurate fault localization can occur for unit tests that cover many statements [98, 160]. We expect fault localization for inline tests to be more accurate since they check the immediately preceding statement.

Regression test selection (RTS) [39, 54–56, 66, 96, 97, 106, 159, 191, 194] speeds up regression testing by only re-running tests that are affected by code changes. Section 2.5 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.

There have been many techniques for automatically generating assertions and invariants, including those that (1) infer invariants from runtime information [18, 28, 40]; (2) generate assertions from comments and documentation [15, 57, 120]; and (3) learn assertions from code [36, 67, 126, 183, 190]. ExLi is most similar to approaches in the first category, as it extracts inline tests from runtime information. But, ExLi additionally (1) uses the collected information to construct inputs and expected outputs for the generated inline tests; and (2) reduces the set of generated inline tests.

Assertions, invariants and design by contract. The `assert` construct [5, 17, 58, 74, 147, 182] in many programming languages, e.g., [10, 128, 161, 164, 174], allows checking that a condition holds on the current program state. Inline tests are similar to assert statements [182]: both are co-located with program statements and they can be turned off in production. Inline tests differ in at least three ways from `asserts`. First, `asserts` do not allow providing arbitrary inputs and oracles for a statement. Second, `asserts` only run if they are in code covered by unit tests or in production [147], but inline tests run in a different context even if the target statement is not covered by unit tests in the testing environment. `asserts` can check global program state at a code location, but inline tests are more local and test the input-output behavior of one statement. Lastly, existing inline testing frameworks provide features that are typically not supported in assert statements: parameterized tests, repeating test runs (helpful to see if inline tests are flaky), grouping tests, and running tests in parallel.

There is a lot of work on design-by-contract (DBC) [10, 94, 117, 119, 125, 147, 151, 171, 173] for specifying preconditions, postconditions, and invariants. DBC tools include PyContracts [173], Crosshair [151], Icontract [171] for Python, and JML [94], Jass [10], Squander [119], Deuterium [125] 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, 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 [42, 48, 73]), 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 [124] and contracts [125].

Automatic test generation. Automatic generation of tests is a popular research topic and many test generation techniques have been proposed for Java [3, 6, 21, 32, 44, 50, 52, 125, 131, 153]. But, ExLi is the first automatic generation technique for inline tests. Elbaum et al.'s technique [33, 38] extracts unit tests from system tests. ExLi is similar in spirit—it also extracts lower granularity tests from higher granularity tests—but differs in the granularity levels that it targets. Also, unlike Elbaum et al.'s technique, ExLi further reduces generated inline tests.

Random testing [68], a black-box testing technique, generates unit tests by randomly selecting inputs from the input domain of the program under test. Randoop [131] is a popular tool that uses a feedback-directed random approach to generate unit tests in the form of method-call sequences. Search-based techniques, e.g., [6, 115], are alternatives to random approaches; they are white-box techniques that generate unit tests by searching for tests that satisfy a criterion. One notable search-based tool is EvoSuite [44], which focuses on optimizing coverage [146] or mutation scores [46]. ExLi uses Randoop and EvoSuite as generators to obtain unit tests from which inline tests are extracted. Beyond that usage, our work is orthogonal to all prior unit-test generation approaches: we focus on inline-test generation.

Test suite reduction/minimization. Test-suite reduction techniques [25, 45, 77, 83, 112, 127, 148, 155, 156, 158, 189, 192] find a minimal subset of a test suite that preserves some measure of test effectiveness, e.g., fault-detection capability or coverage. Some of those test-suite reduction techniques use (1) greedy algorithms [24, 70, 163], (2) heuristics [23, 71], or (3) integer programming [72, 102]. ExLi supports four reduction algorithms [154] (by default is Greedy algorithm) to reduce generated inline tests, and aims to preserve mutation scores on the target statement.

Shi et al. [155] found that techniques based on statement coverage reduce test-suite sizes by 62.9% but lose 20.5% in killed mutants. Conversely, techniques based on killed mutants have no loss in killed mutants but have test-suites that are 10.9 percentage points larger than those produced by coverage-based minimization, on average. Shi et al.'s study gives more confidence in preservation of fault-detection capability in ExLi reduction based on killed mutants.

Noemmer and Haas [127] recently compare test suite minimization techniques on open-source projects and find that, on average, test suites reduce by 70% while losing 12.5% of the fault-detection capability. In ExLi, we use a combined change of coverage rate of target statements and their enclosing program scope. Our results show that traditional test suite minimization reduces generated inline tests by 32.1% and ExLi preserves fault-detection capability.

Mutation testing. Mutation testing is a technique for evaluating the effectiveness of test suites [69, 132, 139]. Popular mutant generators for Java include universalmutator [63], Major [170], PIT [172], and MuJava [110]. ExLi uses the first two tools which perform mutation on the source code level, thus allowing filtering mutants for the target statements. We evaluated universalmutator and Major, and found that there is a small advantage of using the latter instead of the former during inline-test generation. But, future work can explore integrating other mutation tools with ExLi.

Program synthesis. Program synthesis [64, 116] generates programs from specifications or input/output examples. LooPy [41, 82] allows developers to interactively synthesize program blocks. Future work could develop IDE plugins to enable interactive synthesis of inline tests, e.g., based on recent work on automatic test completion [126]. Doing so could be a valuable way to bring developers into the inline-test generation loop.

Chapter 5: Future Work

We now present our plans for future work that can build upon our current contributions and results as described in chapters 2 and 3.

Finding more target statements. Existing target statements found by ExLi only include four types regular expressions, bit manipulation, string manipulation and stream operations. However, there are many other types of statements that may be worth testing. We could define a metric to rank statements based on their complexity and testability. Also, we could develop an interactive tool like a VSCode plugin for developers to select target statements.

Improving the readability of generated tests. Current inline tests use XML files to store the serialized objects that are not primitive types or `String`. However, these XML files are not readable. In the future, we could develop a tool to convert the XML files into code that directly constructs the objects. This would involve analyzing the constructors of the objects represented in the XML files and generating code to construct these objects. For constructors that require non-primitive object arguments, we could use mocking frameworks.

Improving the user study. We conducted an initial user study of I-Test with nine participants, who completed four tasks in Python. In the future, we plan to conduct a larger, more comprehensive user study of I-Test for Java and other programming languages. This user study will involve a diverse group of participants, including both novice and experienced developers, to gather insights into the usability and overall user experience of I-Test across different programming environments. Additionally, we will consider more factors that may affect the learning of I-Test API, such as the order of tasks and the difficulty of tasks.

Exploring usage modes. The inline tests that ExLi generates can help find regressions in future versions of the code. There is need for future work on co-evolving inline tests with code. Such future work could involve developing a technique to automatically update the inline tests when the code changes. That technique could be done by analyzing the diff between the two versions of the code and updating the inline tests accordingly.

Chapter 6: Conclusion

We proposed a new category of tests, named inline tests, to test individual statements, meeting the need for statement-level testing. We implemented I-Test, the first inline testing framework to help developers write and execute inline tests and meet language-agnostic requirements that we define. Our evaluation of I-Test via a user study and 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 negligible.

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

References

[1] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990. https://doi.org/https://doi.org/10.1145/93542.93576.

[2] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering*, pages 143–151, 1995. https://doi.org/https://doi.org/10.1109/ISSRE.1995.497652.

[3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 263–272, 2017. https://doi.org/https://doi.org/10.1109/icse-seip.2017.27.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[5] Mauricio Finavaro Aniche, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. What do the asserts in a unit test tell us about code quality? A study on open source and industrial projects. In *European Conference on Software Maintenance and Reengineering*, pages 111–120, 2013. https://doi.org/https://doi.org/10.1109/csmr.2013.21.

[6] Andrea Arcuri and Gordon Fraser. Java enterprise edition support in search-based JUnit test generation. In *Symposium on Search-Based Software Engineering*, pages 3–17, 2016. https://doi.org/https://doi.org/10.1007/978-3-319-47106-8_1.

[7] Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. https://doi.org/https://doi.org/10.5555/864016.

[8] Sammie Bae. Bit manipulation. In *JavaScript Data Structures and Algorithms*, pages 339–349, 2019. https://doi.org/https://doi.org/10.1007/978-1-4842-3988-9_20.

[9] Herman Banken, Erik Meijer, and Georgios Gousios. Debugging data flows in reactive programs. In *International Conference on Software Engineering*, pages 752–763, 2018. https://doi.org/https://doi.org/10.1145/3180155.3180156.

[10] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass—Java with assertions. In *Runtime Verification*, pages 103–117, 2001. https://doi.org/https://doi.org/10.1016/S1571-0661(04)00247-6.

[11] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[12] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*, pages 433–444, 2018. https://doi.org/https://doi.org/10.1145/3180155.3180164.

[13] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *International Symposium on the Foundations of Software Engineering*, pages 179–190, 2015. https://doi.org/https://doi.org/10.1145/2786805.2786843.

[14] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *International Confer-*

ence on Software Engineering, pages 572–583, 2018. https://doi.org/https://doi.org/10.1145/3180155.3180175.

[15] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis*, pages 242–253, 2018. https://doi.org/https://doi.org/10.1145/3213846.3213872.

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

[17] Kevin Boos, Chien-Liang Fok, Christine Julien, and Miryung Kim. Brace: An assertion framework for debugging cyber-physical systems. In *International Conference on Software Engineering*, pages 1341–1344, 2012. https://doi.org/https://doi.org/10.1109/icse.2012.6227084.

[18] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *International Symposium on Software Testing and Analysis*, pages 169–180, 2006. https://doi.org/https://doi.org/10.1145/1146238.1146258.

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

[20] Xia Cai and Michael R Lyu. The effect of code coverage on fault detection under different testing profiles. In *A-MOST*, pages 1–7, 2005. https://doi.org/https://doi.org/10.1145/1082983.1083288.

[21] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. Bounded exhaustive test-input generation on GPUs. *Proceedings of the ACM on Programming Languages*, 1(International Conference on Object-Oriented Programming,

Systems, Languages, and Applications):1–25, 2017. https://doi.org/https://doi.org/10.1145/3133918.

[22] Carl Chapman and Kathryn T Stolee. Exploring regular expression usage and context in Python. In *International Symposium on Software Testing and Analysis*, pages 282–293, 2016. https://doi.org/https://doi.org/10.1145/2931037.2931073.

[23] Tsong Yueh Chen and Man Fai Lau. Heuristics towards the optimization of the size of a test suite. *WIT Transactions on Information and Communication Technologies*, 14, 1970. https://doi.org/https://doi.org/10.2495/SQM950372.

[24] Tsong Yueh Chen and Man Fai Lau. A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, 40(13):777–787, 1998. https://doi.org/https://doi.org/10.1016/s0950-5849(98)00094-9.

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

[26] Conda. Conda. https://docs.conda.io/projects/conda/en/stable.

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

[28] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering*, pages 281–290, 2008. https://doi.org/https://doi.org/10.1145/1368088.1368127.

[29] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *International Symposium on Software Reliability Engineering*, pages 201–211, 2014. https://doi.org/https://doi.org/10.1109/issre.2014.11.

[30] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *International Symposium on the Foundations of Software Engineering*, pages 246–256, 2018. https://doi.org/https://doi.org/10.1145/3236024.3236027.

[31] James C Davis, Daniel Moyer, Ayaan M Kazerouni, and Dongyoon Lee. Testing regex generalizability and its implications: A large-scale many-language measurement study. In *Automated Software Engineering*, pages 427–439, 2019. https://doi.org/https://doi.org/10.1109/ASE.2019.00048.

[32] Matthew Davis, Sangheon Choi, Sam Estep, Brad Myers, and Joshua Sunshine. NaNofuzz: A usable tool for automatic test generation. In *International Symposium on the Foundations of Software Engineering*, pages 1114–1126, 2023. https://doi.org/https://doi.org/10.1145/3611643.3616327.

[33] Amirhossein Deljouyi and Andy Zaidman. Generating understandable unit tests through end-to-end test scenario carving. In *International Working Conference on Source Code Analysis and Manipulation*, pages 107–118, 2023. https://doi.org/https://doi.org/10.1109/scam59687.2023.00021.

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

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

[36] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. TOGA: A neural method for test oracle generation. In *International Conference*

on Software Engineering, pages 2130–2141, 2022. https://doi.org/https://doi.org/10.1145/3510003.3510141.

[37] Aryaz Eghbali and Michael Pradel. No strings attached: An empirical study of string-related software bugs. In *Automated Software Engineering*, pages 956–967, 2020. https://doi.org/https://doi.org/10.1145/3324884.3416576.

[38] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *International Symposium on the Foundations of Software Engineering*, pages 253–264, 2006. https://doi.org/https://doi.org/10.1145/1181775.1181806.

[39] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010. https://doi.org/https://doi.org/10.1016/j.infsof.2009.07.001.

[40] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. https://doi.org/https://doi.org/10.1016/j.scico.2007.01.015.

[41] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. LooPy: Interactive program synthesis with control structures. *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 5:1–29, 2021. https://doi.org/https://doi.org/10.1145/3485530.

[42] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81, 1993. https://doi.org/https://doi.org/10.1007/978-94-011-1793-7_4.

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

[44] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *International Symposium on the Foundations of Software Engineering*, pages 416–419, 2011. https://doi.org/https://doi.org/10.1145/2025113.2025179.

[45] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Transactions on Software Engineering*, 39(2):276–291, 2012. https://doi.org/https://doi.org/10.1109/tse.2012.14.

[46] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20:783–812, 2015. https://doi.org/https://doi.org/10.1007/s10664-013-9299-z.

[47] Roy S Freedman. Testability of software components. *Transactions on Software Engineering*, 17(6):553–564, 1991. https://doi.org/https://doi.org/10.1109/32.87281.

[48] Carlo A Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys*, 46(3):1–51, 2014. https://doi.org/https://doi.org/10.1145/2506375.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1995. https://doi.org/https://doi.org/10.5555/180415.

[50] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: An automatic test generation tool for industrial Java applications with strings. In *International Conference on Software Engineering*, pages 992–1001, 2013. https://doi.org/https://doi.org/10.1109/icse.2013.6606649.

[51] GitHub. Github commits containing 'remove debug'. `https://github.com/search?q=remove+debug&type=commits`, 2022.

[52] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234, 2010. `https://doi.org/https://doi.org/10.1145/1806799.1806835`.

[53] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis*, pages 302–313, 2013. `https://doi.org/https://doi.org/10.1145/2483760.2483769`.

[54] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Automated Software Engineering*, pages 361–372, 2014. `https://doi.org/https://doi.org/10.1145/2642937.2643019`.

[55] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. In *International Conference on Software Engineering, Demonstrations*, pages 713–716, 2015. `https://doi.org/https://doi.org/10.1109/ICSE.2015.230`.

[56] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015. `https://doi.org/https://doi.org/10.1145/2771783.2771784`.

[57] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis*, pages 213–224, 2016. `https://doi.org/https://doi.org/10.1145/2931037.2931061`.

[58] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588, 2003. https://doi.org/https://doi.org/10.1109/DFTVS.2003.1250158.

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

[60] Siegfried Grabner, Dieter Kranzlmüller, and Jens Volkert. Debugging parallel programs using ATEMPT. In *International Conference on High-Performance Computing and Networking*, pages 235–240, 1995. https://doi.org/https://doi.org/10.1007/BFb0046633.

[61] Mark Grechanik and Gurudev Devanla. Generating integration tests automatically using frequent patterns of method execution sequences. In *International Conference on Software Engineering and Knowledge Engineering*, pages 209–280, 2019. https://doi.org/https://doi.org/10.18293/seke2019-001.

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

[63] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering, Demonstrations*, pages 25–28, 2018. https://doi.org/https://doi.org/10.1145/3183440.3183485.

[64] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[65] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *International Conference on Software Testing, Verification,*

and Validation, pages 993–997, 2016. https://doi.org/https://doi.org/10.1145/2950290.2983932.

[66] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *International Symposium on Software Reliability Engineering*, pages 112–122, 2018. https://doi.org/https://doi.org/10.1109/ISSRE.2018.00022.

[67] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. Assertion generation through active learning. In *International Conference on Formal Engineering Methods*, pages 174–191, 2017. https://doi.org/https://doi.org/10.1007/978-3-319-68690-5_11.

[68] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994. https://doi.org/https://doi.org/10.1017/9781108974073.015.

[69] Farah Hariri, August Shi, Owolabi Legunsen, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. Approximate transformations as mutation operators. In *International Conference on Software Testing, Verification, and Validation*, pages 285–296, 2018. https://doi.org/https://doi.org/10.1109/icst.2018.00036.

[70] Preethi Harris and Raju Nedunchezhian. A greedy approach for coverage-based test suite reduction. *International Arab Journal of Information Technology*, 12:17–23, 2015. https://doi.org/https://www.ccis2k.org/iajit/PDF/vol.12,no.1/5246.pdf.

[71] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993. https://doi.org/https://doi.org/10.1109/icsm.1990.131378.

[72] Joshua Hartmann and Dave J Robson. Revalidation during the software maintenance phase. In *International Conference on Software Maintenance*, pages 70–80, 1989. `https://doi.org/https://doi.org/10.1109/icsm.1989.65195`.

[73] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. `https://doi.org/https://doi.org/10.1145/363235.363259`.

[74] Charles Antony Richard Hoare. Assertions: A personal perspective. *Annals of the History of Computing*, 25(2):14–25, 2003. `https://doi.org/https://doi.org/10.1142/9781848162914_0005`.

[75] Keiichi Ida, Yasuyuki Ohno, Shunsuke Inoue, and Kazuo Minami. Performance profiling and debugging on the K computer. *Fujitsu Scientific and Technical Journal*, 48(3):331–339, 2012.

[76] JavaParser Team. JavaParser. `https://github.com/javaparser/javaparser`, 2023.

[77] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *Transactions on Software Engineering*, 33(2):108–123, 2007. `https://doi.org/https://doi.org/10.1109/tse.2007.18`.

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

[79] René Just. The major mutation framework: Efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis, Tool Demonstrations*, pages 433–436, 2014.

[80] Arthur V Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. PySStuBs: Characterizing single-statement bugs in popular open-source Python projects. In *International Working Conference on Mining Software Repositories*, pages 520–524, 2021. `https://doi.org/https://doi.org/10.1109/msr52588.2021.00066`.

[81] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? The ManySStuBs4J dataset. In *International Working Conference on Mining Software Repositories*, pages 573–577, 2020. `https://doi.org/https://doi.org/10.1145/3379597.3387491`.

[82] Tomer Katz and Hila Peleg. Programming-by-example with nested examples. In *Symposium on Visual Languages and Human-Centric Computing*, pages 280–282, 2023. `https://doi.org/https://doi.org/10.1109/vl-hcc57772.2023.00053`.

[83] Saif Ur Rehman Khan, Sai Peck Lee, Reza Meimandi Parizi, and Manzoor Elahi. A code coverage-based test suite reduction and prioritization framework. In *World Congress on Information and Communication Technologies*, pages 229–234, 2014. `https://doi.org/https://doi.org/10.1109/wict.2014.7076910`.

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

[85] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836, 2013. `https://doi.org/http://hdl.handle.net/1721.1/79645`.

[86] Facebook AI Research lab. PyTorch. `https://github.com/pytorch/pytorch`, 2022.

[87] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *International Symposium on Software Testing and Analysis*, pages 101–111, 2019. https://doi.org/https://doi.org/10.1145/3293882.3330570.

[88] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*, pages 312–322, 2019. https://doi.org/https://doi.org/10.1109/ICST.2019.00038.

[89] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *International Conference on Software Engineering*, pages 1471–1482, 2020. https://doi.org/https://doi.org/10.1145/3377811.3381749.

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

[91] Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. How effective is continuous integration in indicating single-statement bugs? In *International Working Conference on Mining Software Repositories*, pages 500–504, 2021. https://doi.org/https://doi.org/10.1109/msr52588.2021.00062.

[92] Thomas D LaToza and Brad A Myers. Designing useful tools for developers. In *Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 45–50, 2011. https://doi.org/https://doi.org/10.1145/2089155.2089166.

[93] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *International Conference on Software Engineering*, pages 492–501, 2006. https://doi.org/https://doi.org/10.1145/1134285.1134355.

[94] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999. https://doi.org/https://doi.org/10.1007/978-1-4615-5229-1_12.

[95] Wonyeol Lee, Rahul Sharma, and Alex Aiken. Verifying bit-manipulations of floating-point. In *Conference on Programming Language Design and Implementation*, pages 70–84, 2016. https://doi.org/https://doi.org/10.1145/2908080.2908107.

[96] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *International Symposium on the Foundations of Software Engineering*, pages 583–594, 2016. https://doi.org/https://doi.org/10.1145/2950290.295036.

[97] Owolabi Legunsen, August Shi, and Darko Marinov. STARTS: STAtic Regression Test Selection. In *Automated Software Engineering, Tool Demonstrations*, pages 949–954, 2017. https://doi.org/https://doi.org/10.1109/ASE.2017.8115710.

[98] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. How test suites impact fault localisation starting from the size. *IET Software*, 12(3):190–205, 2018. https://doi.org/https://doi.org/10.1049/iet-sen.2017.0026.

[99] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *International Conference on Software Maintenance*, pages 290–301, 1990. https://doi.org/https://doi.org/10.1109/icsm.1990.131377.

[100] Xiangqi Li and Matthew Flatt. Medic: Metaprogramming and trace-oriented debugging. In *Proceedings of the Workshop on Future Programming*, pages 7–14, 2015. https://doi.org/https://doi.org/10.1145/2846656.2846658.

[101] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005. `https://doi.org/https://doi.org/10.1145/1064978.1065014`.

[102] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *International Conference on Software Engineering*, pages 1039–1049, 2018. `https://doi.org/https://doi.org/10.1145/3180155.3180174`.

[103] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. Inline tests. In *Automated Software Engineering*, pages 1–13, 2022. `https://doi.org/https://doi.org/10.1145/3551349.3556952`.

[104] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. Extracting inline tests from unit tests. In *International Symposium on Software Testing and Analysis*, pages 1458–1470, 2023. `https://doi.org/https://doi.org/10.1145/3597926.3598149`.

[105] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. pytest-inline: An inline testing tool for Python. In *International Conference on Software Engineering, Demonstrations*, pages 1–4, 2023. `https://doi.org/https://doi.org/10.1109/icse-companion58688.2023.00046`.

[106] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. More precise regression test selection via reasoning about semantics-modifying changes. In *International Symposium on Software Testing and Analysis*, pages 664–676, 2023. `https://doi.org/https://doi.org/10.1145/3597926.3598086`.

[107] Yu Liu, Aditya Thimmaiah, Owolabi Legunsen, and Milos Gligoric. ExLi: An inline-test generation tool for Java. In *International Symposium on the Foundations of Software Engineering, Demonstrations*, pages 1–5, 2024. `https://doi.org/https://doi.org/10.1145/3663529.3663817`.

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

[109] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *International Symposium on the Foundations of Software Engineering*, pages 643–653, 2014. https://doi.org/https://doi.org/10.1145/2635868.2635920.

[110] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, 2005. https://doi.org/https://doi.org/10.1002/stvr.308.

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

[112] Alessandro Marchetto, Giuseppe Scanniello, and Angelo Susi. Combining code and requirements coverage with execution cost for test suite reduction. *Transactions on Software Engineering*, 45:363–390, 2019. https://doi.org/https://doi.org/10.1109/tse.2017.2777831.

[113] Paul Marinescu, Petr Hosek, and Cristian Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *International Symposium on Software Testing and Analysis*, pages 93–104, 2014.

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

[115] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004. https://doi.org/https://doi.org/10.1002/stvr.294.

[116] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013. https://doi.org/https://doi.org/10.5555/3042817.3042840.

[117] Bertrand Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992. https://doi.org/https://doi.org/10.1109/2.161279.

[118] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *Automated Software Engineering*, pages 415–426, 2019. https://doi.org/https://doi.org/10.1109/ASE.2019.00047.

[119] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *International Conference on Software Engineering*, pages 511–520, 2011. https://doi.org/https://doi.org/10.1145/1985793.1985863.

[120] Manish Motwani and Yuriy Brun. Automatically generating precise oracles from structured natural language specifications. In *International Conference on Software Engineering*, pages 188–199, 2019. https://doi.org/https://doi.org/10.1109/ICSE.2019.00035.

[121] Mountainminds GmbH & Co. KG and Contributors. JaCoCo - Java code coverage library. https://www.jacoco.org/jacoco, 2023.

[122] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *International Conference on Software Testing, Verification, and Validation*, pages 111–120, 2009. https://doi.org/https://doi.org/10.1109/ICST.2009.18.

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

[124] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. A framework for writing trigger-action todo

comments in executable format. In *International Symposium on the Foundations of Software Engineering*, pages 385–396, 2019. https://doi.org/https://doi.org/10.1145/3338906.3338965.

[125] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. Unifying execution of imperative generators and declarative specifications. *Proceedings of the ACM on Programming Languages*, 4(International Conference on Object-Oriented Programming, Systems, Languages, and Applications), 2020. https://doi.org/https://doi.org/10.1145/3428285.

[126] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *International Conference on Software Engineering*, pages 2111–2123, 2023. https://doi.org/https://doi.org/10.1109/icse48619.2023.00178.

[127] Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In *Software Quality: Quality Intelligence in Software and Systems Engineering*, pages 51–66, 2019. https://doi.org/https://doi.org/10.1007/978-3-030-35510-4_4.

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

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

[130] Alessandro Orso. Integration testing of object-oriented software. *Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano*, pages 1–119, 1998. https://doi.org/https://sites.cc.gatech.edu/people/home/orso/papers/orso.thesis.pdf.

[131] Carlos Pacheco and Michael D Ernst. Randoop: Feedback-directed random testing for Java. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 815–816, 2007. https://doi.org/https://doi.org/10.1145/1297846.1297902.

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

[133] Owain Parry, Gregory M. Kapfhammer, Michael C Hilton, and Phil McMinn. A survey of flaky tests. *Transactions on Software Engineering and Methodology*, 31:17:1–17:74, 2022. https://doi.org/https://doi.org/10.1145/3476105.

[134] Jaroslaw Pawlak. 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, 2022.

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

[136] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *International Conference on Software Engineering*, pages 609–620, 2017. https://doi.org/https://doi.org/10.1109/ICSE.2017.62.

[137] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110, 2017. https://doi.org/https://doi.org/10.1109/ISSREW.2014.36.

[138] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from Google. *Transactions on Software Engineering*, 48(10):3900–3912, 2021. `https://doi.org/https://doi.org/10.1109/tse.2021.3107634`.

[139] Goran R. Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *International Conference on Software Engineering*, pages 910–921, 2021. `https://doi.org/https://doi.org/10.1109/icse43902.2021.00087`.

[140] Jan Ploski, Matthias Rohr, Peter Schwenkenberg, and Wilhelm Hasselbring. Research issues in software fault categorization. *Software Engineering Notes*, 32(6):6–es, 2007. `https://doi.org/https://doi.org/10.1145/1317471.1317478`.

[141] pytest-dev team. pytest. `https://docs.pytest.org`, 2022.

[142] pytest-html Team. pytest-html plugin. `https://github.com/pytest-dev/pytest-html`, 2022.

[143] Pytest-xdist. Pytest-xdist. `https://github.com/pytest-dev/pytest-xdist`, 2022.

[144] Cedric Richter and Heike Wehrheim. TSSB-3M: Mining single statement bugs at massive scale. In *International Working Conference on Mining Software Repositories*, pages 418–422, 2022. `https://doi.org/https://doi.org/10.1145/3524842.3528505`.

[145] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Automated Software Engineering*, pages 23–32, 2011. `https://doi.org/https://doi.org/10.1109/ase.2011.6100059`.

[146] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Symposium on Search-Based Software Engineering*, pages 93–108, 2015. https://doi.org/https://doi.org/10.1007/978-3-319-22183-0_7.

[147] David S. Rosenblum. A practical approach to programming with assertions. *Transactions on Software Engineering*, 21(1):19–31, 1995. https://doi.org/https://doi.org/10.1109/32.341844.

[148] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *International Conference on Software Maintenance*, pages 34–43, 1998. https://doi.org/https://doi.org/10.1109/icsm.1998.738487.

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

[150] Adrian Santos, Sira Vegas, Oscar Dieste, Fernando Uyaguari, Ayşe Tosun, Davide Fucci, Burak Turhan, Giuseppe Scanniello, Simone Romano, Itir Karac, et al. A family of experiments on test-driven development. *Empirical Software Engineering*, 26(3):1–53, 2021. https://doi.org/https://doi.org/10.1007/s10664-020-09895-8.

[151] Phillip Schanely. Crosshair. https://github.com/pschanely/CrossHair, 2022.

[152] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. EvoSuite at the SBST 2022 tool competition. In *International Workshop on Search-Based Software Testing*, pages 33–34, 2022. https://doi.org/https://doi.org/10.1145/3526072.3527526.

[153] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults?

An empirical study of effectiveness and challenges (t). In *Automated Software Engineering*, pages 201–211, 2015. https://doi.org/https://doi.org/10.1109/ase.2015.86.

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

[155] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *International Symposium on the Foundations of Software Engineering*, pages 246–256, 2014. https://doi.org/https://doi.org/10.1145/2635868.2635921.

[156] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *International Symposium on the Foundations of Software Engineering*, pages 237–247, 2015. https://doi.org/https://doi.org/10.1145/2786805.2786878.

[157] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *International Conference on Software Testing, Verification, and Validation*, pages 80–90, 2016. https://doi.org/https://doi.org/10.1109/ICST.2016.40.

[158] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. Evaluating test-suite reduction in real software evolution. In *International Symposium on Software Testing and Analysis*, pages 84–94, 2018. https://doi.org/https://doi.org/10.1145/3213846.3213875.

[159] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. Reflection-aware static regression test selection. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187:1–187:29, 2019. https://doi.org/https://doi.org/10.1145/3360589.

[160] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *International Symposium on Software Testing and Analysis*, pages 314–324, 2013. https://doi.org/https://doi.org/10.1145/2483760.2483767.

[161] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006. https://doi.org/https://doi.org/10.1016/j.entcs.2006.02.007.

[162] Jane Street. Inline tests. https://github.com/janestreet/ppx_inline_test, 2023.

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

[164] Richard N Taylor. Assertions in programming languages. *ACM SIGPLAN Notices*, 15(1):105–114, 1980. https://doi.org/https://doi.org/10.1145/954127.954139.

[165] CPython Team. Python AST library. https://github.com/python/cpython/blob/main/Lib/ast.py, 2022.

[166] CPython Team. Doctest. https://docs.python.org/3/library/doctest.html, 2022.

[167] Crafter Core Team. Crafter CMS Core. https://github.com/craftercms/core, 2023.

[168] DeDRM Team. DeDRM tools. https://github.com/apprenticeharper/, 2024.

[169] JUnit Team. JUnit. https://junit.org, 2022.

[170] Major Team. Major mutation framework. https://mutation-testing.org/, 2023.

[171] Parquery Team. icontract. https://github.com/Parquery/icontract, 2022.

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

[173] PyContracts Team. PyContracts. https://github.com/AndreaCensi/contracts, 2022.

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

[175] Randoop Team. Randoop manual. https://randoop.github.io/randoop/manual/, 2023.

[176] Dave Thomas and Andy Hunt. Mock objects. *IEEE Software*, 19(3):22–24, 2002. https://doi.org/https://doi.org/10.1109/MS.2002.1003449.

[177] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *Software Engineering Notes*, 30(5):253–262, 2005. https://doi.org/https://doi.org/10.1145/1095430.1081749.

[178] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006. https://doi.org/https://doi.org/10.1109/MS.2006.117.

[179] Serge Toarca. debuggex. https://www.debuggex.com, 2022.

[180] Fabian Trautsch. *An Analysis of the Differences Between Unit and Integration Tests*. PhD thesis, Niedersächsische Staats-und Universitätsbibliothek Göttingen, 2019.

[181] W.T. Tsai, Xiaoying Bai, R. Paul, Weiguang Shao, and V. Agarwal. End-to-end integration testing design. In *IEEE Annual International Computer Software and Applications Conference*, pages 166–171, 2001. https://doi.org/https://doi.org/10.1109/cmpsac.2001.960613.

[182] Jeffrey M Voas and Keith W Miller. Putting assertions in their place. In *International Symposium on Software Reliability Engineering*, pages 152–157, 1994. https://doi.org/https://doi.org/10.1109/issre.1994.341367.

[183] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *International Conference on Software Engineering*, pages 1398–1409, 2020. https://doi.org/https://doi.org/10.1145/3377811.3380429.

[184] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. University of Michigan, 1979.

[185] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *Transactions on Software Engineering*, 42 (8):707–740, 2016. https://doi.org/https://doi.org/10.1109/TSE.2016.2521368.

[186] Yingfei Xiong, Dan Hao, Lu Zhang, Tao Zhu, Muyao Zhu, and Tian Lan. Inner oracles: Input-specific assertions on internal states. In *International Symposium on the Foundations of Software Engineering*, pages 902–905, 2015. https://doi.org/https://doi.org/10.1145/2786805.2803204.

[187] XStream. XStream Team, 2022. https://x-stream.github.io/index.html.

[188] Rahulkrishna Yandrapally and Ali Mesbah. Mutation analysis for assessing end-to-end web tests. In *International Conference on Software Maintenance*

and Evolution, pages 183–194, 2021. https://doi.org/https://doi.org/10.26226/morressier.613b5417842293c031b5b5c3.

[189] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification & Reliability*, 22(2): 67–120, 2012. https://doi.org/https://doi.org/10.1002/stv.430.

[190] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. Automated assertion generation via information retrieval and its integration with deep learning. In *International Conference on Software Engineering*, pages 163–174, 2022. https://doi.org/https://doi.org/10.1145/3510003.3510149.

[191] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. Comparing and combining analysis-based and learning-based regression test selection. In *ICSE Workshop on Automation of Software Test*, pages 17–28, 2022.

[192] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An empirical study of JUnit test-suite reduction. In *International Symposium on Software Reliability Engineering*, pages 170–179, 2011. https://doi.org/https://doi.org/10.1109/issre.2011.26.

[193] Zexuan Zhong, Jiaqi Guo, Wei Yang, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Generating regular expressions from natural language specifications: Are we there yet? In *AAAI Conference on Artificial Intelligence*, pages 791–794, 2018. https://doi.org/https://aaai.org/papers/791-ws0512-aaaiw-18-17262/.

[194] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. A framework for checking regression test selection tools. In *International Conference on Software Engineering*, pages 430–441, 2019. https://doi.org/https://doi.org/10.1109/ICSE.2019.00052.