



Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/216879/>

Version: Accepted Version

Proceedings Paper:

Maton, M., Kapfhammer, G. and McMinn, P. (2024) PSEUDOSWEEP: A pseudo-tested code identifier. In: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME). 40th International Conference on Software Maintenance and Evolution (ICSME 2024), 06-11 Oct 2024, Flagstaff, AZ, USA. Institute of Electrical and Electronics Engineers (IEEE), pp. 873-877. ISBN: 979-8-3503-9569-3. ISSN: 1063-6773. EISSN: 2576-3148.

<https://doi.org/10.1109/ICSME58944.2024.00094>

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:

<https://creativecommons.org/licenses/>

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

PSEUDOSWEEP: A Pseudo-Tested Code Identifier

Megan Maton
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

Phil McMinn
University of Sheffield, UK

Abstract—Software testing remains a crucial practice for ensuring and maintaining code quality. Yet, a critical issue remains: the existence of pseudo-tested statements. Tests cover these statements, but removing them does not trigger test failures. Since no established tools address this challenge, this paper introduces PSEUDOSWEEP, a novel tool that automatically identifies pseudo-tested methods and statements in Java projects. PSEUDOSWEEP combines method and statement deletion techniques to reveal these maintenance problems. In addition to explaining the approach used by PSEUDOSWEEP, this paper details use-cases and overviews results from experiments with PSEUDOSWEEP. The tool is available (including set-up instructions and examples) at <https://github.com/PseudoTested/PseudoSweep> and there is a video demonstration at <https://youtu.be/5QCsu7MbiXI>.

I. INTRODUCTION

If a tool can remove production code without the test suite failing, it is difficult to argue that this code is thoroughly tested. This scenario, in which a test executes some program code and yet a tool can remove it without causing a test to fail, means that the code is “pseudo-tested”, which is a quality risk [9].

For instance, Listing 1 furnishes a *formatTag* method and highlights its pseudo-tested statement on Line 3. The *testFormatTag* test executes all lines in this method but only contains assertions that check the start and end tags. This means the statement on Line 3, which places the content between the tags, has no observable impact on the method’s output. Even though the test executes this statement, a developer can delete it without causing any tests to fail. Such code is either redundant or requires further testing to detect its removal correctly.

Revealing pseudo-tested code at early testing stages will enable developers to identify weaknesses or oversights in their test suite before progressing to more expensive approaches. For example, traditional mutation testing tools evaluate a test suite’s bug-finding capability by inserting synthetic defects into the code and executing the test suite against them to see if they cause tests to fail [3], [4]. Despite extensive research and development, mutation testing often remains costly [11].

Developers can identify pseudo-tested methods using a branch of mutation testing called extreme mutation testing (XMT), which removes entire method bodies (using a dummy return value when required for compilation) and executes the tests [9]. Prior work has found pseudo-tested methods in well-tested subjects, including open-source Apache projects [9], [14]. However, Maton et al. showed that non-pseudo-tested methods still contained pseudo-tested statements [8]. Highlighting such statements will help developers address simple testing issues such as missing tests and partial assertions.

This paper introduces PSEUDOSWEEP, a tool that highlights pseudo-tested statements and methods in Java projects

```
1 public String formatTag(String tag, String content) {
2     String html = "<" + tag + ">";
3     html += content;
4     html += "</" + tag + ">";
5     return html;
6 }
7
8 @Test
9 public void testFormatTag() {
10    String str = formatTag("p", "hello world!");
11    assertThat(str, startsWith("<p"));
12    assertThat(str, endsWith(">/p"));
13 }
```

Listing 1. The source code of the *formatTag* method and its corresponding test case called *testFormatTag*. The PSEUDOSWEEP tool presented in this paper can automatically determine that the statement on line 3 is pseudo-tested.

(supporting JDK versions 8–12), thus enabling developers to address testing inadequacies. Ensuring that the program always compiles, PSEUDOSWEEP systematically removes each covered method and statement from the source code of the program under test and runs the test suite to detect elements it can remove without impacting test outcomes. It surfaces these elements to developers so that they can identify the testing deficiency that may have caused it. This tool contributes a novel implementation for identifying pseudo-tested statements, thereby helping developers to address testing inadequacies before committing resources to traditional mutation testing.

To demonstrate how developers can use our tool, we ran it on the *Caverphone2* class from *commons-codec*. We walk through the outputs for the *encode* method and contrast the findings from running PSEUDOSWEEP with what code coverage would have revealed. We also overview experimental results that explore the frequency and causes of pseudo-tested statements in 27 real-world Java projects [8].

II. RELATED MUTATION TESTING TOOLS

The DESCARTES, MUJAVA, and MAJOR tools each implement deletion operators that could be used to detect some areas of pseudo-tested code. This section investigates their approaches and operators and compares them to PSEUDOSWEEP.

Traditional Mutation Testing Tools

MUJAVA: The MUJAVA tool uses mutant schema generation to embed all mutants into a metamutant [7]. This approach manipulates the source code; however, embedding all mutants into a single metamutant only requires a single compilation. Deletion is a simple transformation, and therefore, using a metamutant enables simple control of program execution and a visual representation of the mutations for the user. MUJAVA contains several different statement deletion operators, including the capability to delete larger statement types such as *for*, *while* and *if* statements. However, although the tool’s source code is available, MUJAVA only supports up to Java 6.

MAJOR: The MAJOR tool employs a compiler-integrated approach to transform the program’s attributed abstract syntax tree (AST), enabling it to generate the mutants while compiling the source code [6]. Therefore, MAJOR avoids introducing mutants that cannot map to the original source code. MAJOR’s implementation of statement deletion currently excludes some statement types such as *if* statements. However, due to a lack of access to the source code, we could not extend it.

The MUJAVA and MAJOR tools both implement versions of the statement deletion operator. However, their divergent approaches and differing sets of operators make neither tool suitable for the focused detection of pseudo-tested statements. To implement PSEUDOSWEEP, we have taken inspiration from each of these tools and used JavaParser [12] to manipulate the AST and produce a metamutant. Section III explains these aspects of PSEUDOSWEEP’s implementation in greater detail.

PIT: PIT is a state-of-the-art tool for the mutation testing of Java programs. It uses Java bytecode manipulation to insert mutants into code, which is faster than the methods used by other tools and is suitable for the mutation operators it employs. Yet, this also means it can mutate code that does not directly map to a section of source code, and the source code descriptions of statements are not reproducible using bytecode, making it unsuitable for a statement deletion operator.

A. Extreme Mutation Testing Tools

DESCARTES: The DESCARTES Mutation Engine for PIT is the only tool focused on detecting pseudo-tested code [16]. As it depends on the PIT mutation tool for Java, the DESCARTES Engine uses Java bytecode manipulation to either remove method bodies or replace them with default return values. This implementation works successfully at the method level due to bytecode representations of methods being clear to mutate. Methods are also clearly defined code units; therefore, it does not matter that the mutation occurred at the bytecode level. Users can quickly identify which methods DESCARTES has mutated, reducing mutation analysis time. However, bytecode cannot directly map to Java statements, making it challenging to implement statement deletion in a user-friendly manner. Therefore, extending the DESCARTES engine to the statement level is unsuitable for identifying pseudo-tested statements.

RENTERI: The RENTERI plugin observes the undetected code transformations identified by DESCARTES to suggest potential causes of the pseudo-tested methods like “no-infection”, “no-propagation” and “weak-oracle” [15]. Experiments with Java programs showed that these suggestions helped developers. While useful for identifying causes of pseudo-tested methods, RENTERI does not offer suggestions at the statement level.

III. PSEUDOSWEEP: DETECTING PSEUDO-TESTED ELEMENTS IN PROJECT SOURCE CODE

We designed PSEUDOSWEEP to identify and report pseudo-tested statements and methods within the source code of a Java project. By revealing these testing deficiencies before developers commit resources to traditional mutation testing, PSEUDOSWEEP provides helpful insights engendering test

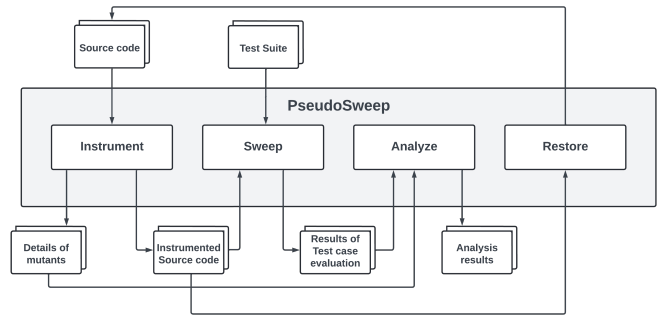


Fig. 1. Diagram of PSEUDOSWEEP’s main components. Instrument, Sweep, Analyze and Restore are entry points, run from the command line.

suite maturity. The tool focuses on detecting pseudo-tested elements through deletion mutations. To achieve this, we designed and implemented an appropriate set of operators. Figure 1 overviews the core components of PSEUDOSWEEP.

A. Instrument

To keep the program’s source code, PSEUDOSWEEP duplicates each Java Class file with a temporary one that has the *.java.orig* file extension, enabling the reverting of the instrumented source code to the original with the Restore function.

1) *Operators*: Detecting pseudo-tested elements requires operators that can delete elements of the code. The current version of the tool operates at the statement and method levels; therefore, the operator-set design targets these component types. We look to extend this in future tool iterations to give users control over the granularity of the results they receive.

a) *Method Operators*: We have developed our method-level operators based on this set and described them in Table I. If a Java method declares a non-void return type, then the method requires at least one *return* statement at the end of the method to compile. To check if a method is pseudo-tested, PSEUDOSWEEP must delete the whole method. Therefore, we must add a placeholder return statement to ensure compilation. We use two default return values for each return statement to mitigate the risk of returning the expected value. Table II presents the default method return values.

b) *Statement Operators*: We have assembled six operators to detect pseudo-tested statements, as described in Table I. We looked towards the statement deletion operator (SDL) defined by Deng et al. [5]; however, their operator set required additional operators such as a *return value mutation operator* to make it suitable for identifying pseudo-tested statements.

For example, SDL only provides a single default return value for all types other than boolean. These single defaults leave room for equivalent mutants such as returning ‘0’ when an assertion is expecting a ‘0’. To overcome this problem, the DESCARTES Mutation Engine for PIT applies two default value mutants and adds defaults for further types such as Collections and Maps. We have extended PSEUDOSWEEP’s implementation of SDL to include this as presented in Table II.

2) *Metamutant*: PSEUDOSWEEP implements these operators by creating a metamutant [13] that adds *if* statements to

TABLE I

Descriptions of the deletion mutation operators employed by PSEUDOSWEEP. In the “Example Transformation” column, the method calls to `exec`, `fix` and `eval` are simplified. In the full instrumentation inserted by the tool, this source code would also include class information and the element type.

| Operator: Description | Applicable Element Types | Example Source Code | Example Transformation |
|---|--|--|--|
| MDR : Delete method body and return default value | Non-void Method | <pre> 1 public int method () { 2 methodBody; 3 return 5; 4 }</pre> | <pre> 1 public int method () { 2 if (exec(id)) { 3 methodBody; 4 return 5; 5 } 6 return (execDefault(id)) ? 0 : 1; 7 }</pre> |
| MDV : Delete method body | Void Method | <pre> 1 public void method () { 2 methodBody; 3 }</pre> | <pre> 1 public void method () { 2 if (exec(id)) { 3 methodBody; 4 } 5 }</pre> |
| SDSS : Delete Statement | Break, Continue, Do-while, Expression, For, For Each, If, Contains Inner Class, Lambda, Switch, Try, While | <pre> 1 statement;</pre> | <pre> 1 if (exec(id)) { 2 statement; 3 }</pre> |
| SDSL : Delete statement with label | Break, Continue | <pre> 1 statement label;</pre> | <pre> 1 if (exec(id)) { 2 statement label; 3 }</pre> |
| SDSR : Delete the statement and return default value | Return | <pre> 1 return "string";</pre> | <pre> 1 if (exec(id)) { 2 return "string"; 3 } 4 return (execDefault(id)) ? "" : "A";</pre> |
| SDVD : Delete declaration initialisation | Variable declarations | <pre> 1 int i=3;</pre> | <pre> 1 int i=0; 2 if (exec(id)) { 3 i=3; 4 }</pre> |
| SFCT : Fix conditional expression to True | Do-while, For, If, While | <pre> 1 if (a < b) { 2 }</pre> | <pre> 1 if (fix(eval() && a < b, id)){ 2 }</pre> |
| SDLL : Delete loop statement with label | For, While | <pre> 1 label: 2 for(){ 3 }</pre> | <pre> 1 if (exec(id)) { 2 label: 3 for(){ 4 } 5 }</pre> |

TABLE II
Default return values for return statements.

| Type | Default Values | |
|----------------|-----------------------|----------------|
| Boolean | false | true |
| Byte | 0 | 1 |
| Double | 0 | 1 |
| Float | 0 | 1 |
| Char | \40 (space character) | 'A' |
| Short | 0 | 1 |
| Int | 0 | 1 |
| String | "" | "A" |
| Long | 0 | 1 |
| Collection | null | New ArrayList |
| Iterable | null | New ArrayList |
| List | null | New ArrayList |
| Queue | null | New LinkedList |
| Set | null | New HashSet |
| Map | null | New HashMap |
| Reference Type | null | null |

the source code of the project that PSEUDOSWEEP manipulates to “remove” individual code elements during a “Sweep” (Section III-B). The metamutant approach ensures that the mutants always directly map to the developer’s source code.

a) Statement-level metamutant: As shown by Table I, our tool instruments statements by inserting an *if* statement around them. The *if* statement’s conditional makes a call to PSEUDOSWEEP to check whether it should execute the contained statement. The expression includes the class name, element

type and element number to enable PSEUDOSWEEP to identify it. Where moving a statement into an *if* block causes scoping issues and thus compilation concerns, the instrumentation uses scenario-specific tactics. For instance, variable declarations and return statements cannot change scope, so PSEUDOSWEEP uses default values so that the program still compiles.

b) Method-level metamutant: We implement method-level instrumentation by inserting a conditional *if* statement around the method logic as demonstrated in Table I. We also addressed scoping issues at the method level by adding default values to relevant places to ensure the code could still compile.

The details of the mutants are recorded in class information files, enabling developers to identify where PSEUDOSWEEP placed the mutants in the original source code without requiring them to inspect the metamutant. The Analyze command also uses these files to identify the pseudo-tested code.

Methods such as simple getter/setter methods may be unimportant to test, therefore PSEUDOSWEEP is configurable to skip their instrumentation, with a view to extend this to other less-relevant statements, such as logging calls, in future work.

Due to the use of JavaParser [12], the projects that PSEUDOSWEEP can evaluate are limited to Java versions 8 to 12. The tool cannot evaluate code containing *var* type definitions, as type information is required to set default values. Currently, the tool supports JUnit 4 and 5 test suites, with the scope

of adding compatibility with other frameworks. Deng et al.'s definitions of SDL [5] and the terms used in JavaParser guide PSEUDOSWEEP's definitions of a *statement*; therefore, our tool's use may differ from other tools' use of this word.

After instrumenting the source code, the user must compile it with their configured build tool, including all their dependencies and test classes, before proceeding to the next phase.

B. Sweep

The main algorithm for evaluating the instrumented class files is similar to regular mutation analysis. Given a test method, PSEUDOSWEEP executes it three times, without any active mutations, to record the test results, times taken (to calculate timeout values), and the methods and statements executed. The tool execute the tests three times because a flaky test result [10] could lead to missed pseudo-tested elements if a test fails for reasons not related to the deleted element. If a test fails at this stage, the tool discards it as it cannot be relied upon for testing the code. Secondly, PSEUDOSWEEP iterates through the list of executed elements, activating the relevant mutations individually and checking for behaviour that differs from the recorded outcomes, specifically failing, throwing an exception or timing out. The test runs three times, attempting to induce one of these outcomes. If one of these occurs, PSEUDOSWEEP stops and moves on to the next element. It then records the test outcomes in the result files for analysis purposes. Due to internal thread tracking, projects containing threaded code may cause errors and unreliable results.

C. Analyze

After evaluating all the mutants, PSEUDOSWEEP uses the Analyze functionality to identify the not-covered, covered, and pseudo-tested elements. The tool presents the lists of elements and relevant statistics (e.g., % pseudo-tested) in a *.json* file per Java class, and an overall project summary file. At this stage, the developer can look directly at the metamutant to identify the pseudo-tested elements, or invoke the Restore functionality to return to their original code and use the provided location information to begin addressing pseudo-tested code.

D. Restore

The Restore functionality deletes the instrumented file and renames the original file back to *filename.java*. Although this is a manual step in the current prototype, future versions of the tool will internalise code restoration while also providing visual coverage reports that display pseudo-tested elements.

IV. APPLYING THE PSEUDOSWEEP TOOL

We summarize how PSEUDOSWEEP has been evaluated and how it can be used with a case study and a demonstration. Full instructions on setting up the tool are available on GitHub [2].

A. Evaluation

This section summarises the results of an empirical evaluation that applied PSEUDOSWEEP to 27 real-world projects, including Apache projects *commons-codec* and *commons-cli* [8]. The study explored pseudo-tested statements' frequency, weaknesses, and causes within these Java projects. The study

found that identifying only pseudo-tested methods would miss half of the pseudo-tested statements. PSEUDOSWEEP identified 350 pseudo-tested statements within methods required for the test suite to pass. The study also revealed that these statements obtained lower mutation scores, suggesting they are less well-tested, with the leading causes of pseudo-tested statements being missing/partial tests and assertions, which accounted for 86 of the 119 pseudo-tested statements sampled.

B. Case Study

Our study of the causes of pseudo-tested statements in 27 real-world Java projects surfaced many examples [8], including the *commons-codec.language.Caverphone2.encode* method in the *commons-codec* project that this section uses as a case study. The *encode* method encodes a source string into a CaverPhone 2.0 value. The method uses a sequence of *java.lang.String.replace(t, r)* and *java.lang.String.replaceAll(r;r)* method calls to substitute characters and regular expression patterns within the source string to produce CaverPhone 2.0 value. Twelve tests cover the method, achieving 100% statement coverage. This method is also "required" for the test suite to pass (i.e., it cannot be removed without causing tests to fail). However, PSEUDOSWEEP identified that of the 63 statements in the method, 23 were pseudo-tested. The first author manually confirmed this by deleting each statement and running the test suite. We detail full replication instructions within the replication GitHub Repository [1] with a link to the video presentation of the tool evaluating this Java class in the README. We will now look at how the tool identifies these statements.

1) *Generating mutants*: Given the source code of *commons-codec.language.Caverphone2.encode*, as shown in Listing 2, PSEUDOSWEEP applies the statement operators described in Table I to each statement. The statement-level operators also use *if* statements to execute the statement logic conditionally. For example, given Line 14 in Listing 2, PSEUDOSWEEP applies the **SDSS** operator as follows:

```
if (org.pseudosweep.I.exec("EXPRESSION", 9, "org.apache.
commons.codec.language.Caverphone2", "sdl")) {
    txt = txt.replaceAll("^trough", "trou2f");
}
```

2) *Executing mutants*: During the Sweep command, PSEUDOSWEEP executes the tests against the mutants by recording the elements covered by each test and then running each test against each of the mutated versions of the elements it covers.

3) *Identifying pseudo-tested code*: The Analyze command enables the user to sweep test results further to reveal the pseudo-tested components. The developer has two choices at this stage: to Restore the code to the source code or to analyze the metamutant directly. The metamutant contains the element information displayed in the analysis file, enabling the user to use file searches to quickly reach the location of the pseudo-tested code. Otherwise, they can restore the system to its source code and use the class information files to identify the locations of pseudo-tested code elements. Once the user has found the pseudo-tested element in the source code, they can begin to analyse it. Using file search, the user can discover which test cases covered the element and manually analyse the

```

1  @Override
2  public String encode(final String source) {
3      String txt = source;
4      if (SoundexUtils.isEmpty(txt)) {
5          return TEN_1;
6      }
7      txt = txt.toLowerCase(java.util.Locale.ENGLISH);
8      txt = txt.replaceAll("[a-z]", "");
9      txt = txt.replaceAll("e$", "");
10     txt = txt.replaceAll("^cough", "cou2f");
11     txt = txt.replaceAll("^rough", "rou2f");
12     txt = txt.replaceAll("^tough", "tou2f");
13     txt = txt.replaceAll("^enough", "enou2f");
14     txt = txt.replaceAll("^trough", "trou2f");
15     txt = txt.replaceAll("^gn", "2n");
16     txt = txt.replaceAll("mb$", "m2");
17     txt = txt.replaceAll("cq", "2q");
18     txt = txt.replace("ci", "si");
19     txt = txt.replace("ce", "se");
20     txt = txt.replace("cy", "sy");
21     txt = txt.replace("tch", "2ch");
22     txt = txt.replace("c", "k");
23     txt = txt.replace("q", "k");
24     txt = txt.replace("x", "k");
25     txt = txt.replace("v", "f");
26     txt = txt.replace("dg", "2g");
27     txt = txt.replace("tio", "sio");
28     txt = txt.replace("tia", "sia");
29     txt = txt.replace("d", "t");
30     txt = txt.replace("ph", "fh");
31     txt = txt.replace("b", "p");
32     txt = txt.replace("sh", "s2");
33     txt = txt.replace("z", "s");
34     txt = txt.replaceAll("[aeiou]", "A");
35     txt = txt.replaceAll("[aeiou]", "3");
36     txt = txt.replace("j", "y");
37     txt = txt.replaceAll("^y3", "Y3");
38     txt = txt.replaceAll("^y", "A");
39     txt = txt.replace("y", "3");
40     txt = txt.replace("3gh3", "3kh3");
41     txt = txt.replace("gh", "22");
42     txt = txt.replace("n", "k");
43     txt = txt.replaceAll("s+", "S");
44     txt = txt.replaceAll("t+", "T");
45     txt = txt.replaceAll("p+", "P");
46     txt = txt.replaceAll("k+", "K");
47     txt = txt.replaceAll("f+", "F");
48     txt = txt.replaceAll("m+", "M");
49     txt = txt.replaceAll("n+", "N");
50     txt = txt.replace("w3", "W3");
51     txt = txt.replace("wh3", "Wh3");
52     txt = txt.replaceAll("w3", "3");
53     txt = txt.replace("w", "2");
54     txt = txt.replaceAll("h", "A");
55     txt = txt.replace("h", "2");
56     txt = txt.replace("r3", "R3");
57     txt = txt.replaceAll("r3", "3");
58     txt = txt.replace("x", "2");
59     txt = txt.replace("l3", "L3");
60     txt = txt.replaceAll("l3", "3");
61     txt = txt.replace("l", "2");
62     txt = txt.replace("2", "");
63     txt = txt.replaceAll("33", "A");
64     txt = txt.replace("3", "");
65     txt = txt + TEN_1;
66     return txt.substring(0, TEN_1.length());
67 }

```

Listing 2. The *commons-codec.language.Caverphone2.encode* method (with blank space and comments removed for presentation purposes). Pseudo-tested statements identified by PSEUDOSWEEP are highlighted with grey boxes.

deletion to decide their action plan for addressing the issue. According to the code comments, the statement on Line 14 in Listing 2 is exclusive to the CaverPhone 2.0 specification. However, PSEUDOSWEEP can remove this statement without impacting test outcomes. Adding a test assertion such as:

```
assertEquals(encode("trough"), "TRF11111111")
```

to check the encoding of a string beginning with “trough” means this statement is required for the test suite to pass.

V. CONCLUSIONS AND FUTURE WORK

This paper introduces PSEUDOSWEEP, a tool that detects pseudo-tested statements so that developers can identify areas of insufficient testing before committing resources to the costly process of mutation testing. This paper explains PSEUDOSWEEP’s design and its approach to identifying pseudo-tested methods and statements in Java programs. Finally, we demonstrate how a developer can use the output from our tool in their workflow, using an example from *commons-codec*. Building on the experimental results demonstrating the effectiveness of PSEUDOSWEEP [8], future work will 1) Visualise pseudo-tested components into a coverage report; 2) Use pseudo-tested methods to remove redundant execution of the statement-level mutants that they contain; 3) Reduce the pseudo-tested code reported to a user at a given time by identifying the most relevant elements, specifically high-coverage, yet pseudo-tested, methods and pseudo-tested statements found in required methods; and 4) Investigate pseudo-tested components at different granularities and in other programming languages. This work will further PSEUDOSWEEP’s capability to expose the most critical pseudo-tested elements, enabling developers to methodically improve their test suites.

Acknowledgements. Megan Maton is funded by the EPSRC Doctoral Training Partnership with the University of Sheffield, grant EP/W524360/1. Phil McMinn is supported, in part, by the EPSRC grant “Test FLARE” (EP/X024539/1).

REFERENCES

- [1] Demo replication: <https://github.com/PseudoTested/pseudosweep-demo>.
- [2] PseudoSweep Tool: <https://github.com/PseudoTested/PseudoSweep>.
- [3] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, 2005.
- [4] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 1978.
- [5] L. Deng, J. Offutt, and N Li. Empirical Evaluation of the Statement Deletion Mutation Operator. In *Proc. ICST*, 2013.
- [6] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proc. ASE*, 2011.
- [7] Y-S. Ma, J. Offutt, and Y-R. Kwon. MuJava: A mutation system for Java. In *Proc. ICSE*, 2006.
- [8] M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring pseudo-testedness: Empirically evaluating extreme mutation testing at the statement level. In *Proc. ICSME*, 2024.
- [9] R. Niedermayr, E. Jürgen, and S. Wagner. Will my tests tell me if I break this code? In *Proc. Int. Workshop on Continuous Software Evolution and Delivery*, 2016.
- [10] O. Parry, M. Hilton, G. M. Kapfhammer, and P. McMinn. A survey of flaky tests. *ACM TOSEM*, 2022.
- [11] A. V Pizzoleto, F. C Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *JSS*, 2019.
- [12] N. Smith, D. van Bruggen, and F. Tomassetti. *JavaParser: Visited*. LeanPub, 2023.
- [13] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *Proc. ISSA*, 1993.
- [14] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *EmSE*, 2019.
- [15] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. In *arXiv:1909.04770*, 2019.
- [16] O. L. Vera-Pérez, M. Monperrus, and B. Baudry. Descartes: A PITest engine to detect pseudo-tested methods — tool demonstration. In *Proc. ASE*, 2018.