MockDetector: A technique to identify mock objects created in unit tests

Qian Liang
Patrick Lam
q8liang@uwaterloo.ca
patrick.lam@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Software dependencies are ubiquitous and may pose problems during testing, because creating usable objects from dependencies is often complicated. Developers, therefore, often introduce mock objects to stand in for dependencies during testing. However, to our knowledge, no static analysis framework provides a tool to automatically identify mock objects created in the unit test cases. The lack of mock object detection can decrease the precision of static analyses, as they are unable to separate methods invoked on mock objects from methods invoked on actual objects.

In this paper, we introduce MockDetector, a technique to identify mock objects. It is able to detect common Java mock libraries' APIs that create mock objects, checking whether there is a call to a mock creation site and then a def-use chain reaching the point of use. Implications of understanding which objects are mock objects include helping static analysis tools identify which dependencies' methods are actually tested, versus mock methods being called.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

static analysis, mock objects, unit tests

ACM Reference Format:

Qian Liang and Patrick Lam. 2018. MockDetector: A technique to identify mock objects created in unit tests. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

2 MOTIVATING EXAMPLE

In this section, we illustrate how our MockDetector tool finds a mock object created within a unit test case. Our tool identifies variables which have been assigned an object flowing from a mock creation site through a def-use chain (possibly of length 0).

First, we would like to discuss an example to illustrate our motivation for this project. Listing 1 illustrates a method *addAll()* that is invoked on a mocked object of Type Collection<Number>. Current static analysis tools, to our knowledge, cannot easily distinguish this method invocation on a mocked object from the method invocation on an actual object. Therefore, a naive static analysis would perceive method invocations on mocked objects as the behaviour getting tested, whereas the purpose of the method invocations on mocked objects are intended model behaviours, so that the actual object's behaviour can be properly tested.

Listing 2 shows the unit test case *testSimpleResolution()* in the benchmark byte-buddy-dep (version 1.7.10) where the mock object TypeDescription is created via a direct call to Java mocking library Mockito's *mock(java.lang.class)*. In this example, our MockDetector tool would utilizes Soot [**?**] to locate the statements that are instances of Assignment Statement with an invoke expression at the right operand, i.e. def-use chain of length 0. It then checks if the method invoked matches with any Java mocking libraries' APIs creating a mock object, by matching the method name, parameter types, and return type (i.e. the method subsignature).

Meanwhile, Listing 3 illustrates the unit test case *testGetIterator()* in the benchmark commons-collections4 (version 4.3), where the array of Node, consists of mock objects created in the helper function *createNodes()*, under this transitive call scenario. In this example with a def-use chain, our tool would first detect the Java mocking library that is in use within the benchmark, and retrieve the corresponding API creating a mock object from the detected Java mocking library. It then utilizes Soot's ReachableMethods with the input of a constructed call graph and the iterator consists of the specific, and checks if any of the statements in the unit test case's body, contains a method invocation that could eventually reach the API.

```
@Test
public void addAllForIterable() {
        // ...
        final Collection<Number> c = createMock(Collection.class);
        // ...
        expect(c.addAll(inputCollection)).andReturn(true);
        // ...
}
```

Listing 1: This code snippet illustrates an example where a method is invoked on a mocked object in unit test case *addAllForIterable()*

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

```java
import static org.mockito.Mockito.mock;

// ...

@Test
public void testSimpleResolution() throws Exception {
        TypeDescription typeDescription =
                        mock(TypeDescription.class);
// ...
}
```

Listing 2: This example illustrates a direct call to Mockito's *mock(java.lang.class)* function from test case *testSimpleResolution().*

```java
private Node[] createNodes() {
        final Node node1 = createMock(Node.class);
// ...
}

@Test
public void testGetIterator() {
// ...
        final Node[] nodes = createNodes();
// ...
}
```

Listing 3: This example illustrates a transitive call to EasyMock's *CreateMock(java.lang.class)* function from test case *testGetIterator().*

Figure 1: Illustration of the immediate mock (def-use chain of length 0) created in Listing 2. The actual benchmark from which this was drawn contains TypeDescription instead of X.

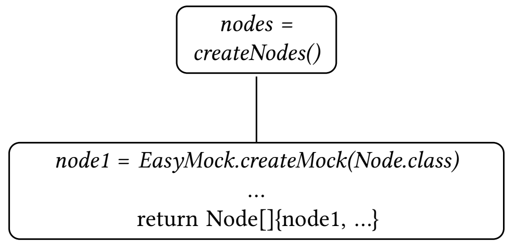

Figure 2: Illustration of the wrapper mock (def-use chain of length 1) created in Listing 3.

3 TECHNIQUE

In this section, we describe the technique that MockDetector applies to find unit test cases with mock objects created in the test body. Our tool tracks the sites and occurrences of the mock object We separate the tracking and counting of the special case where mock objects created with def-use chain of length 0, from the general case where the def-use chain has length more than 0. We believe the study of immediate mock creation (i.e, def-use length of 0), as well as wrapper mock creation (i.e, def-use chain length more than 0), would provide additional insight of the benchmark.

3.1 Define Common Mocking Library APIs

Our tool stores a pool of common APIs, provided by the analysis designer, which are used to create mock objects when using popular Java mocking libraries, including Mockito, EasyMock, and PowerMock. These APIs are the possible mock creation sites—the candidates for def nodes in the def-use chain.

3.2 Load all classes and Determine the Mock Library

Given a pool of possible APIs to search for, our tool can analyze tests for their uses of these APIs.

JUnit tests are simply methods that developers write in test classes, suitably annotated (in JUnit 3 by method name, in 4+ by a @Test annotation). A JUnit test runner uses reflection to find tests. This is a problem for static analyses.

Thus, to enable static analysis over the test suite classes, our tool first generates a driver class which invokes all public, non-constructor test cases. Then it uses Soot to analyze the benchmark's test suite, treating the driver class as the main class, so that all test classes and their public, non-constructor test cases are analyzed. Our tool loads all benchmark classes and queries Soot to find out which mocking library the benchmark uses, by checking for references to each of the common APIs.

3.3 Find Immediate Mocks

In this stage, our tool detects and counts the unit test cases which create at least one mock object directly (i.e. not transitively through method calls). With all the unit test cases processed in the previous step, our tool next retrieves the body for each unit test case, and looks for statements similar to the one illustrated in Listing 2, i.e. instances of assignment statements containing a invoke expression on the right hand side. Our tool would then determine if it matches with the determined mocking library's API. In the example, the right operand (source) of the assignment statement is a method invocation of *Mockito.mock()*. After our tool examines the library and method signature, it concludes that this unit test case contains a mock object created with def-use length of 0, as depicted in Figure 1.

3.4 Find Wrapper Mocks

Refer to Listing 3, our tool also considers the case where the mock object is created via a transitive call. Figure 2 presents the example's def-use path. It illustrates an array of mocked Node objects created in the helper function *createNodes()*, which is an approximation of mock object creation that is considered within our tool's scope. On top of the scheme finding immediate mocks, MockDetector implements Soot's ReachableMethods to find these wrapper mocks. With the determined mocking library's API taken as the end point, our tool could now tell if this end point is reachable by any statements in the test case, excluding the ones creating immediate mocks. The reason to differentiate the counting of wrapper mocks from the immediate mocks is that, we believe the analysis of these two separately could provide more insight of the benchmark, such as to obtain the percentage of mocks, or the mocks within certain modules created via a wrapper.

4 EVALUATION

5 CONCLUSION

6 MATH EQUATIONS

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

6.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the **math** environment, which can be invoked with the usual `\begin`…`\end` construction or with the short form `$`…`$`. You can use any of the symbols and structures, from α to ω, available in LaTeX [?]; this section will simply show a few examples of in-text equations in context. Notice how this equation: $\lim_{n\rightarrow \infty}x=0$, set here in in-line math style, looks slightly different when set in display style. (See next section).

6.2 Display Equations

A numbered display equation—one set off by vertical space from the text and centered horizontally—is produced by the **equation** environment. An unnumbered display equation is produced by the **displaymath** environment.

Again, in either environment, you can use any of the symbols and structures available in LaTeX; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n\rightarrow \infty}x=0 \tag{1}$$

Notice how it is formatted somewhat differently in the **displaymath** environment. Now, we'll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty}x_i=\int_{0}^{\pi+2} f \tag{2}$$

just to demonstrate LaTeX's able handling of numbering.

7 FIGURES

Your figures should contain a caption which describes the figure to the reader.

Figure captions are placed *below* the figure.

Every figure should also have a figure description unless it is purely decorative. These descriptions convey what's in the image to someone who cannot see it. They are also used by search engine crawlers for indexing images, and when images cannot be loaded.

A figure description must be unformatted plain text less than 2000 characters long (including spaces). **Figure descriptions should not repeat the figure caption – their purpose is to capture important information that is not already provided in the caption or the main text of the paper.** For figures that convey important and complex new information, a short text description may not be adequate. More complex alternative descriptions can be placed in an appendix and referenced in a short figure description. For example, provide a data table capturing the information in a bar chart, or a structured list representing a graph. For additional information regarding how best to write figure descriptions and why doing this is so important, please see https://www.acm.org/publications/taps/describing-figures/.

7.1 The "Teaser Figure"

A "teaser figure" is an image, or set of images in one figure, that are placed after all author and affiliation information, and before the body of the article, spanning the page. If you wish to have such a figure in your article, place the command immediately before the `\maketitle` command:

```
\begin{teaserfigure}
  \includegraphics[width=\textwidth]{sampleteaser}
  \caption{figure caption}
  \Description{figure description}
\end{teaserfigure}
```

8 CITATIONS AND BIBLIOGRAPHIES

The use of BibTeX for the preparation and formatting of one's references is strongly recommended. Authors' names should be complete — use full first names ("Donald E. Knuth") not initials ("D. E. Knuth") — and the salient identifying features of a reference should be included: title, year, volume, number, pages, article DOI, etc.

The bibliogrphy is included in your source document with these two commands, placed just before the `\end{document}` command:

```
\bibliographystyle{ACM-Reference-Format}
\bibliography{bibfile}
```

where "`bibfile`" is the name, without the "`.bib`" suffix, of the BibTeX file.

Citations and references are numbered by default. A small number of ACM publications have citations and references formatted in the "author year" style; for these exceptions, please include this command in the **preamble** (before the command "`\begin{document}`") of your LaTeX source:

```
\citestyle{acmauthoryear}
```

Some examples. A paginated journal article [?], an enumerated journal article [?], a reference to an entire issue [?], a monograph (whole book) [?], a monograph/whole book in a series (see 2a in spec. document) [?], a divisible-book such as an anthology or compilation [?] followed by the same example, however we only output the series if the volume number is given [?] (so Editor00a's series should NOT be present since it has no vol. no.), a chapter in a divisible book [?], a chapter in a divisible book in a series [?], a multi-volume work as book [?], a couple of articles in a proceedings (of a conference, symposium, workshop for example) (paginated proceedings article) [? ?], a proceedings article with all possible elements [?], an example of an enumerated proceedings article [?], an informally published work [?], a couple of preprints [? ?], a doctoral dissertation [?], a master's thesis: [?], an online document / world wide web resource [? ? ?], a video game (Case 1) [?] and (Case 2) [?] and [?] and (Case 3) a patent [?], work accepted for publication [?], 'YYYYb'-test for prolific author [?] and [?]. Other cites might contain 'duplicate' DOI and URLs (some

SIAM articles) [?]. Boris / Barbara Beeton: multi-volume works as books [?] and [?]. A couple of citations with DOIs: [? ?]. Online citations: [? ? ?]. Artifacts: [?] and [?].

9 ACKNOWLEDGMENTS

Identification of funding sources and other support, and thanks to individuals and groups that assisted in the research and the preparation of the work should be included in an acknowledgment section, which is placed just before the reference section in your document.

This section has a special environment:

```
\begin{acks}
...
\end{acks}
```

so that the information contained therein can be more easily collected during the article metadata extraction phase, and to ensure consistency in the spelling of the section heading.

Authors should not prepare this section as a numbered or unnumbered `\section`; please use the “`acks`” environment.

10 APPENDICES

If your work needs an appendix, add it before the “`\end{document}`” command at the conclusion of your source document.

Start the appendix with the “`appendix`” command:

```
\appendix
```

and note that in the appendix, sections are lettered, not numbered. This document has two appendices, demonstrating the section and subsection identification method.

11 SIGCHI EXTENDED ABSTRACTS

The “`sigchi-a`” template style (available only in LaTeX and not in Word) produces a landscape-orientation formatted article, with a wide left margin. Three environments are available for use with the “`sigchi-a`” template style, and produce formatted output in the margin:

- `sidebar`: Place formatted text in the margin.
- `marginfigure`: Place a figure in the margin.
- `margintable`: Place a table in the margin.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

A RESEARCH METHODS

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

B ONLINE RESOURCES

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.