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, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Mock objects [?] are a common idiom in unit tests for object-oriented systems. They allow developers to test objects that have dependencies on other objects, possibly from different components, or which are simply difficult to create for test purposes (e.g. a database).

While mock objects are an invaluable tool for developers, their use complicates the static analysis of test cases. A call to a mock object resembles, by design, a call to the real object. When analyzing such code, a naive static analysis aiming to be sound would have to include all of the possible behaviours of the real object. Such an analysis, then, would not usefully capture the behaviour of the test case.

Others have defined the notion of a focal method for a test case—the method whose behaviour is being tested. The test case would set up mock objects to provide parameters to this focal method. When analyzing the test case, it would be useful to know which variables in the method contain mock objects.

We have designed a helper static analysis, MockDetector, which identifies mock objects in test cases. It starts from a list of mock object creation sites; we have provided sites for common mocking libraries. It then propagates mockiness interprocedurally through use-def chains and containers, so that an analysis can ask whether a given variable in a test case contains a mock or not. We have evaluated MockDetector on a suite of benchmarks.

Taking a broader view, we believe that helper static analyses can aid in the development of more useful static analyses. These analyses can encode useful domain properties; for instance, in our case, properties of test cases. By taking a domain-specific approach, analyses can extract useful facts about programs that would otherwise be difficult to establish.

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()*

```
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().*

```
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().*

x = Mockito.mock(X.class)

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.

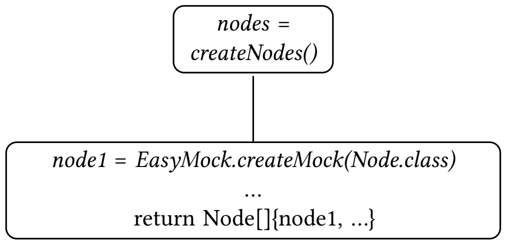
nodes = createNodes()
node1 = EasyMock.createMock(Node.class)
...
return Node[]{node1, ...}

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.

WRITE SOMETHING ABOUT DRIVER GENERATION!!!

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

Our tool also handles the case where the mock object is created via a transitive call, as seen in Listing 3. 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 uses 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 RELATED WORK

idea of custom static analysis; was part of Soot's design goals. Dynamic makes things harder and there are things like TamiFlex for dealing with reflection. Cite the soundiness paper. But tests are a really structured use of dynamic code.

Doop is also an option.

In the type system world there is the Checker framework for letting people write their own type systems. This is quite successful.

For AST-level analyses there is spoon and comby.

SpotBugs is extensible as well.

Probably also read the focal methods paper and see what it says about analyzing test cases.

6 CONCLUSION