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

Qian Liang, Patrick Lam
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
{q8liang, patrick.lam}@uwaterloo.ca

***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. Static analysis is usually a helpful tool to analyze a program's test suite to help identify uncovered method pre-compilation. However, the methods invoked on mock objects are wrongly included in the call graph analysis, which would provide incorrect testing information for the dependency objects during the static analysis stage. 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, followed by a forward flow must analysis, which aims to include all locals that are indeed mock objects on all possible paths, as well as the containers such as array or collection that have must mock objects stored. 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.

***Index Terms*—Forward Must Analysis, Mock Objects, Unit Tests**

I. 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.

II. 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);
  // ...
  return new Node[]{node1, ...};
}

@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()*.

III. Technique and Implementation

IV. Evaluation

V. Related Work

VI. Conclusions