MockDetector: Detecting and tracking mock objects in unit tests

by

Qian Liang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis consists of all chapters written for conference research paper submission (later rejected), with minor word changes and styling updates.

Qian Liang was the sole author for Chapters 2, 3, 5, which were written under the supervision of Dr. Patrick Lam. Qian was responsible for developing the imperative Soot implementation for mock analysis, carrying out data collection and analysis from both imperative Soot and declarative Doop's implementations.

Qian Liang and Dr. Patrick Lam were the co-authors for Chapters 1, 4, 6 and 7.

Abstract

Unit testing is a widely used tool in modern software development processes. A well-known issue in writing tests is handling dependencies: creating usable objects for dependencies is often complicated. Developers must therefore often introduce mock objects to stand in for dependencies during testing.

Test suites are an increasingly important component of the source code of a software system. We believe that the static analysis of test suites, alongside the systems under test, can enable developers to better characterize the behaviours of existing test suites, thus guiding further test suite analysis and manipulation. However, because mock objects are created using reflection, they confound existing static analysis techniques. At present, it is impossible to statically distinguish methods invoked on mock objects from methods invoked on real objects. Static analysis tools therefore currently cannot determine which dependencies' methods are actually tested, versus mock methods being called.

In this thesis, we introduce MockDetector, a technique to identify mock objects and track method invocations on mock objects. We first built a Soot-based imperative dataflow analysis implementation of MockDetector. Then, to quickly prototype new analysis features and to explore declarative program analysis, we created a Doop-based declarative analysis, added features to it, and ported them back to the Soot-based analysis. Both analyses handle common Java mock libraries' APIs for creating mock objects and propagate the mock objects information through test cases. Following our observations of tests in the wild, we have added special-case support for arrays and collections holding mock objects. On our suite of 8 open-source benchmarks, our imperative dataflow analysis approach reported 2,095 invocations on mock objects intraprocedurally, whereas our declarative dataflow approach reported 2,130 invocations on mock objects (under context-insensitve base analyses in intraprocedural mode), out of a total number of 63,017 method invocations in test suites; across benchmarks, mock invocations accounted for a range from 0.086% to 16.4% of the total invocations. Removing confounding mock invocations from consideration as focal methods can improve the precision of focal method analysis, a key prerequisite to further analysis of test cases.

Acknowledgements

I would like to thank my advisor, Professor Patrick Lam, for his guidance in Soot implementation, and data analysis suggestions for Doop's base analyses. I sincerely appreciate the time, effort and patience he has put in throughout my master's program, and specifically for the thesis.

I would also like to thank Professor Gregor Richards for reading my thesis and providing invaluable suggestions and feedback on background chapter, high level description section and many other areas, and Professor Derek Rayside for providing his great insight regarding the content goes into background work.

I would also like to thank the Doop authors for responding to our questions surrounding Doop implementations and the performance of basic-only base analysis in a timely and helpful fashion.

Dedication

I dedicate the thesis to my family, who have always supported me during my study and life at University of Waterloo.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mock objects [1] are a common idiom in unit tests for object-oriented systems. They allow developers to test objects that rely on other objects, likely from different components, or that are simply complicated to build for testing purposes (e.g. a database).

While mock objects are an invaluable tool for developers, their use complicates the static analysis and manipulation of test case source code, one of our planned future research directions. Such static analyses can help IDEs provide better support to test case writers; enable better static estimation of test coverage (avoiding mocks); and detect focal methods in test cases.

Ghafari et al discussed the notion of a focal method [11] for a test case—the method whose behaviour is being tested—and presented a heuristic for determining focal methods. By definition, the focal method's receiver object cannot be a mock object. Ruling out mock invocations can thus improve the accuracy of focal method detection and enable better understanding of a test case's behaviour.

Mock objects are difficult to analyze statically because, at the bytecode level, a call to

a mock object statically resembles a call to the real object (as intended by the designers of mock libraries). A naive static analysis attempting to be sound would have to include all of the possible behaviours of the actual object (rather than the mock) when analyzing such code. Such potential but unrealizable behaviours obscure the true behaviour of the test case.

We have designed a static analysis, MockDetector, which identifies mock objects in test cases. It starts from a list of mock object creation sites (our analyses include hardcoded APIs for common mocking libraries EasyMock[1], Mockito[2], and PowerMock[3]). It then propagates mockness through the test and identifies invocation sites as (possibly) mock. Given this analysis result, a subsequent analysis can ask whether a given variable in a test case contains a mock or not, and whether a given invocation site is a call to a mock object or not. We have evaluated MockDetector on a suite of 8 benchmarks plus a microbenchmark. We have cross-checked results across the two implementations and manually inspected the results on our microbenchmark, to ensure that the results are as expected.

Taking a broader view, we believe that helper static analyses like MockDetector 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.

[1] https://easymock.org/
[2] https://site.mockito.org/
[3] https://github.com/powermock/powermock

We make the following contributions in this thesis:

- We designed and implemented two variants of a static mock detection algorithm, one as a dataflow analysis implemented imperatively (using Soot) and the other declaratively (using Doop).
- We evaluate both the relative ease of implementation and precision of the imperative and declarative approaches, both intraprocedurally and interprocedurally (for Doop).
- We characterize our benchmark suite (8 open-source benchmarks, 184 kLOC) with respect to their use of mock objects, finding that 1084 out of 6310 unit tests use intraprocedurally detectable mock objects, and that there are a total of 2095 method invocations on mock objects.

At a higher level, we see the thesis as making both a contribution and a meta-contribution to problems in source code analysis. The contribution, mock detection, enables more accurate analyses of test cases, which account for a significant fraction of modern codebases. The meta-contribution, comparing analysis approaches, will help future researchers decide how to best solve their source code analysis problems. In brief, the declarative approach allows users to quickly prototype, stating their properties concisely, while the imperative approach is more amenable to use in program transformation; we return to this question in Chapter 7.

Chapter 2

Background

Static analysis is an important tool for code analysis, especially when analyzing benchmarks with large code bases. Running an application does not always expose bugs (for instance, due to missing branch coverage), where static analysis tools are more complete (at the price of false positives). A good static analysis tool can help developers to find bugs that are hard to spot manually (e.g., array out of bounds exception), which reduces time and costs involved in fixing bugs.

2.1 Dataflow Analysis

Dataflow analysis [13] is a static analysis which is performed on a program's control flow graph (CFG). A CFG is a directed graph in which each node stands for a statement in the program (the statement could be an assignment statement, an if statement, etc.), and the set of directed edges represents the overall control flow of the program.

Dataflow analysis, meanwhile, is typically implemented using a fixed point algorithm that computes and gathers all the facts at each program point (i.e., each node in the CFG). The facts would usually be a mapping between program variables and the abstractions specifically defined to solve the problem at hand. The most important decision for a dataflow analysis is the abstraction and associated flow functions to update the abstraction at each program point.

Forward vs. Backward Researchers or developers need to make two additional decisions before implementation. First, they must decide if the problem should be categorized as a forward or backward dataflow problem. For a forward dataflow analysis, the facts propagate along the direction of the control flow. Determining whether expressions are available (i.e. have already been computed) at each program point is a type of forward dataflow problem. On the other hand, the facts propagate in the opposite direction from the control flow in a backward dataflow analysis, and most importantly, we need to access the future usage of the variables. Analyzing the liveness of variables is a known backward dataflow problem. Few dataflow problems require bidirectional flows.

May vs. Must Researchers or developers also need to decide if it is a may or must dataflow problem. The core difference between the two types of problems is how they handle the facts at all the join points in the program, where multiple branches meet. A may dataflow analysis keeps facts that hold true on any joined path. "Reaching Definitions" is a may analysis problem. It checks if a definition (or assignment) of a variable reaches a specific program point. On the other hand, must dataflow analysis only keeps facts

that hold true from all the branches. Determining available expressions is a must analysis problem.

Gen and Kill Set All dataflow analyses requires a transfer function of some kind for each program point. Generally, the actions on the dataflow analysis domain are often called "gen" and "kill". When we have set of facts at each statement, the kill action removes the facts that are no longer true from the set, while the gen action adds new facts that are now true. The flow function describing such process are of the form

$$out(s) = gen(s) \bigcup (in(s) \setminus kill(s)) \tag{2.1}$$

Dataflow analyses are usually implemented imperatively, which focuses on the "how" part of the solution, providing a set of commands or operations to tackle the problem.

2.1.1 Soot

Among all the dataflow analysis tools, Soot [20] is a representative Java optimization framework, which provides a total of four types of intermediate representations to analyze Java bytecode.

An intermediate representation (IR) is an abstract language designed for ease of analysis and transformation. A good IR is independent of the source and target languages, and thus convenient to translate into code for retargetable architectures.

For our project, we use Jimple (simple Java), which is a stackless, typed 3-address IR in

the CFG. To familiarize the reader with Jimple IR's syntax, the following is an example of high-level Java code and its corresponding Jimple IR for code that defines a mock object.

Java source code:

```
MavenSession session = mock(MavenSession.class);
```

Jimple IR:

```
org.apache.maven.execution.MavenSession r1;
java.lang.Object $r2;
$r2 = staticinvoke <org.mockito.Mockito:
       java.lang.Object mock(java.lang.Class)>
       (class "Lorg/apache/maven/execution/MavenSession;");
r1 = (org.apache.maven.execution.MavenSession) $r2;
```

For each method invocation, the Jimple IR contains the type of invocation (i.e. staticinvoke, specialinvoke, dynamicinvoke, interfaceinvoke, or virtualinvoke) along with the method signature. The method's signature consists of the class where the method is declared and the method's subsignature (method name, parameter types, and return type).

The actual return object (r1) is of a different type (ORG.APACHE.MAVEN.EXECUTION.MAVENSESSION) than the return type (JAVA.LANG.OBJECT) from Mockito's mock source creation method *mock()*'s return type. Java parametrized types exist only at source code level but not at bytecode level. They are erased from the bytecode[1]. The Java compiler then produces a cast to bridge from JAVA.LANG.OBJECT (in the bytecode) to the mock return type in the parametrized source code.

Heros [4] is an inter-procedural, finite, distributive subset problems (IFDS) / interprocedural distributive environment (IDE) solver that provides interprocedural support to the

[1]They are preserved in attributes but not the bytecode itself.

Soot framework.

2.2 Declarative Analysis

While an imperative approach focuses on the "how" component of a solution, declarative analysis focuses on the "what" part during the implementation. It gives a set of constraints which must be solved.

A declarative approach normally comes with an underlying imperative layer, where some type of tool handles the imperative processes for the developer.

2.2.1 Doop

We chose Doop [6] as the declarative framework for our project. It comes with a number of implementations solving the constraints stated in the declarative form. The collection of analyses are expressed using Datalog rules. The current version of Doop uses Soufflé[2] as the Datalog engine for these analysis rules.

Doop takes as input facts generated by Soot (imported into a database) as well as declarative Datalog rules. It then asks Soufflé to solve the rules on the inputs. Doop also computes fixed points by embedding them in the constraints it gives to the constraint solver.

[2] https://souffle-lang.github.io/docs.html

2.3 Mock Objects

Mock objects are commonly used in unit test cases. They substitute for the real objects that are normally hard to create (e.g., a database), or slow to process (e.g., a connection). It is noteworthy that mock objects are never being tested in test cases. Their purpose is to stand in for the dependencies, assisting for the behavioural test of the real object. For this to happen, the mock objects must at least mimic the behaviour at interfaces connecting to the real object under test.

Imagine you want to test the behaviour of a detonator. It is infeasible to always test the detonator's behaviour with a real bomb. So in this scenario, you build a mock bomb. The mock bomb does not do anything other than check whether it has received instructions to explode. Note that in the whole process, you are testing the detonator's behaviour, not the mock bomb's behaviour.[3]

The current static analysis tools, however, could not differentiate a mock object and a real object, because the containing variables have the same type in Java and in the IR. Section 3.3 runs a toy example and explains why the current tools are insufficient in locating mock objects.

For our project, we consider for Java mock source methods from three mocking frameworks: EasyMock[4], Mockito[5], and PowerMock[6]. According to a prior study [14], EasyMock

[3]The idea of this example is sparked from https://stackoverflow.com/questions/28783722/when-using-mokito-what-is-the-difference-between-the-actual-object-and-the-mock?noredirect=1&lq=1.

[4]https://easymock.org/

[5]https://site.mockito.org/

[6]https://github.com/powermock/powermock

and Mockito are used in about 90% of the 5,000 randomly sampled projects. We then added a third mocking framework by our own choice. Thus, we believe our analysis results should be applicable to most Java benchmarks.

Chapter 3

Motivation

In this chapter, we run through a real-world example selected from benchmark `maven`. The unit test case contains both a real object and a mock object, and there is a method invocation on each of them. We then present a simplified toy example with corresponding IR processed and used by static analysis tools. We demonstrate why the current tools are incompetent to differentiate mock objects from real objects. However, before diving into the current project, let us take a step back and talk about what led us to the research on detecting mock objects and tracking mock invocations.

3.1 Preliminary Research

[2]The content of this figure have been incorporated within a NIER paper published by IEEE in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), available online: https://ieeexplore.ieee.org/xpl/conhome/9240597/proceeding[doi: 10.1109/ICSME46990.2020.00075]. Qian Liang and Patrick Lam, "SiblingClassTestDetector: Finding Untested Sibling Functions"

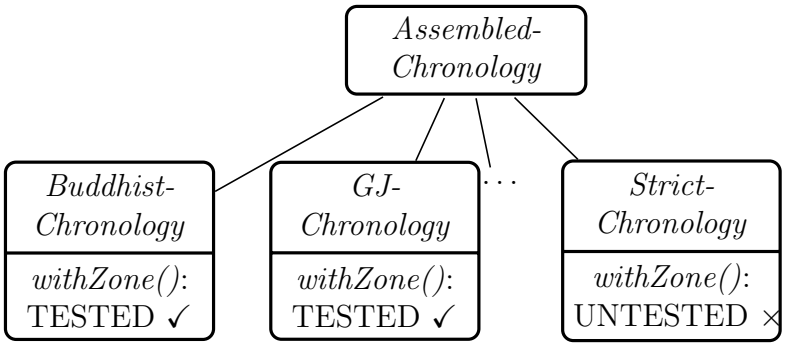

Figure 3.1: joda-time contains superclass *AssembledChronology* and subclasses *BuddhistChronology*, *GJChronology*, *StrictChronology*, and others. Method *withZone()* is tested in *Buddhist-* and *GJChronology* but not *StrictChronology*.[2]

Prior to the development for MockDetector, we were working on a project to detect untested functions that have analogous implementations in sibling classes (they share a common superclass), where at least one of the related implementations are tested. The overall goal of that project is to reduce untested code. Though testing could not guarantee desired program behaviour, developers certainly know nothing about any untested code. Since the sibling methods share the same specification, it is likely that a unit test case covering for one sibling class's implementation may also work for the untested after small modifications, potentially increasing the statement coverage and consequently having a better chance to gain behavioural insight of the benchmark.

Figure 3.1 illustrates such an example from an open-source benchmark, joda-time (version 2.10.5). The abstract class *AssembledChronology* inherits the specification of method *withZone()* from its parent class, which is not shown in the Figure. *Assembled-Chronology*'s subclasses *BuddhistChronology*, *GJChronology*, *StrictChronology*, and many others are at the same hierarchy level, which are defined as sibling classes. These sibling

classes all have an implementation of *withZone()*; however, the *withZone()* implementations in *BuddhistChronology* and *GJChronology* are tested, whereas the implementation in *StrictChronology* is not.

As the research progressed, we encountered the test case in Listing 3.1, and realized that it would be a necessary step to remove method invocations on mock objects from the call graph generated by existing static analysis frameworks, as otherwise we may mistakenly treat such test case as the one covering for the tested sibling method, since the existing static analyses tools could not distinguish method invocations on mock objects from method invocations on real objects.

```
1
2 @Test
3 public void addAllForIterable() {
4 final Collection<Integer> inputCollection = createMock(Collection.class);
5 ...
6 final Collection<Number> c = createMock(Collection.class);
7 ...
8 expect(c.addAll(inputCollection)).andReturn(false);
9 }
```

Listing 3.1: This code snippet illustrates an example from commons-collections4, where the method *addAll()* invoked on the mock object `c` could be mislabelled as a focal method.

3.2 Running Example: Detecting Mock Objects and Mock Invocations

To motivate our work, consider Listing 3.2, which presents a unit test case from the Maven project. Line 7 calls *getRequest()*, invoking it on the mock object `session`. Line 11 then calls *getToolchainsForType()*, which happens to be the focal method whose behaviour is being tested in this test case. At the bytecode level, the two method invocations are indistinguishable with respect to mockness; to our knowledge, current static analysis tools cannot easily tell the difference between the method invocation on a mock object on line 7 and the method invocation on a real object on line 11. Given mockness information, an IDE could provide better suggestions. The uncertainty about mockness would confound a naive static analysis that attempts to identify focal methods. For instance, Ghafari et al's heuristic [11] would fail on this test, as it returns the last mutator method in the object under test, and the focal method here is an accessor.

```
1 @Test
2 public void testMisconfiguredToolchain() throws Exception {
3     MavenSession session = mock( MavenSession.class );
4     MavenExecutionRequest req = new DefaultMavenExecutionRequest();
5     when( session.getRequest() ).thenReturn( req );
6
7     ToolchainPrivate[] basics =
8             toolchainManager.getToolchainsForType("basic", session);
```

```
9
10      assertEquals( 0, basics.length );
11  }
```

Listing 3.2: This code snippet illustrates an example from maven-core, where calls to both the focal method `getToolchainsForType()` and to mock `session`'s `getRequest()` method occur in the test *testMisconfiguredToolchain()*.

3.3 A Toy Example

In this section, we show how hard it is to differentiate mocks from actuals on IR. The toy example serves the same purpose as the one in Listing 3.2 but comes with a simpler Jimple IR for easier explanation.

Listing 3.3 shows the creation of two `java.lang.Object` objects: `object1` and `object2`. `object1` is created via normal initialization by calling to a constructor, thus it is a real object. Meanwhile, `object2` is created by invoking Mockito's mock API returning a mock object, thus `object2` is a mock object.

Listing 3.4 displays the Jimple IR for Listing 3.3, where `$r1` and `r2` are the Jimple IRs of `object1` and `object2`, respectively. Though the initialization of `$r1` and `r2` are from different types of invocations, since the static analysis tool does not recognize the mock source method from Mockito, the tool views both `$r1` and `r2` as real objects (they are both of type `java.lang.Object` after all). Therefore, the tool is unable to recognize the invocation of *foo()* on line 10 is definitely *not* a focal method.

```
1   Object object1 = new Object();
2   object1.foo();
3
4   Object object2 = mock(Object.class);
5   object2.foo();
```

Listing 3.3: A toy example illustrates a real object `object1` and a mock object `object2`, and their corresponding method invocations of *foo()*.

```
1   java.lang.Object $r1, r2;
2
3   $r1 = new java.lang.Object;
4   specialinvoke $r1.<java.lang.Object: void <init>()>();
5   virtualinvoke $r1.<java.lang.Object: void foo()>();
6
7   r2 = staticinvoke <org.mockito.Mockito:
8                 java.lang.Object mock(java.lang.Class)>
9                 (class "Ljava/lang/Object;");
10  virtualinvoke r2.<java.lang.Object: void foo()>();
```

Listing 3.4: Jimple Intermediate Representation for the code in Listing 3.3.

Chapter 4

Technique

We present two complementary ways of statically computing mock information: an imperative implementation of a dataflow analysis (using the Soot program analysis framework), and a declarative implementation (using the Doop framework). We started this project with the usual imperative approach to implementing a static analysis—in our context, that meant using Soot. Then, when we wanted to experiment with adding more features to the analysis, we decided that this was a good opportunity to learn about Doop's declarative approach as well. We added new features to the Doop implementation and backported them to the Soot implementation. While the core analysis is similar, the different implementation technologies have different affordances. For instance, it is easier for the Doop version to mark a field as mock-containing (we added 3 rules) than for the Soot version to do so. We start by describing each implementation in turn, and conclude this section with the commonalities between the two implementations. Chapter 5 then presents the results obtained using each technology and compares them.

4.1 High Level Definitions

A mock object may be created and carried in several ways: it could be the return value from a mock creation site; it could be a casted version of the return value from a mock creation site in the intermediate level if the object is not of type `java.lang.Object`; it could be a copy of an already-flagged mock; it could be a field mock object defined via annotations, in constructor `<init>`, or in `@Before`/*setUp()* methods; or it could be hold by an array or container before re-accessing it.

Therefore, the following is our definition for mockness:

Definition 1. *Object x is a mock object if it is either the return value from a mock creation site (including field mocks), or if the initial mock definition reaches a use point through casting, transitive definition, or passed along through mock-containing arrays or containers. We consider paths over the intraprocedural control-flow graph and interprocedural call graph edges.*

In this section, we present the high level definitions for different categories of mock using Jimple IR. The Soot implementation looks for these Jimple IRs during the "gen" action for each statement. The declarative Doop analysis is quite close to the high level definitions.

In the following Jimple IR definitions, the ones highlighted in dashed boxes are considered as mock, and the ones highlighted in frame boxes are considered as mock-containing arrays or containers.

The first definition is for mock source creation methods:

```
java.lang.Object r1;
r1 = staticinvoke <org.mockito.Mockito:
          java.lang.Object mock(java.lang.Class)>
          (class "Ljava/lang/Object;");
```

In the 3 mocking libraries, we consider a total of 9 mock source creation methods. For the Jimple IR definition , we pick Mockito's `<org.mockito.Mockito: java.lang.Object mock(java.lang.Class)>` as a representative.

The mock source methods return `java.lang.Object`. For objects of other types, Jimple IR casts the return object from mock source methods. The Jimple IR below uses a generalized type X (other than `java.lang.Object`):

```
X r1;
java.lang.Object $r2;
$r2 = staticinvoke <org.mockito.Mockito:
          java.lang.Object mock(java.lang.Class)>
          (class "X;");
r1 = (X) $r2;
```

Field Mocks are commonly defined and used in several of the benchmarks we select. There are usually three ways to define field mocks.

The first way is to define fields as referencing mock objects via Mockito or EasyMock's @Mock annotations. This is unfortunately not reflected in the Jimple IR. Instead, we implemented the following annotation type check to locate the field mock defined via annotation.

```
for (AnnotationTag annotation : tag.getAnnotations()) {
    if ( annotation.getType().equals("Lorg/mockito/Mock;")
         annotation.getType().equals("Lorg/easymock/Mock;") ) {
    }
}
```

The second approach defines field mocks in the constructor `<init>` method.

```
public void <init>() {
    java.lang.Object $r1;
    X $r2;
    $r1 = staticinvoke <org.mockito.Mockito:
                java.lang.Object mock(java.lang.Class)>
                (class "X;");
    $r2 = (X) $r1;
}
```

The last approach defines field mocks in the `@Before`/*setUp()* method:

```
private X x;
public void init() {
    TESTCLASS r0;
    java.lang.Object $r1;
    X $r2;

    r0 := @this: TESTCLASS;
    $r1 = staticinvoke <org.mockito.Mockito:
                java.lang.Object mock(java.lang.Class)>
                (class "X;");
    $r2 = (X) $r1;
    r0.<TESTCLASS: X x> = $r2;
}
```

For mock-containing array (write method), the following is a sample Jimple IR definition:

```
java.lang.Object r1, $r2;
java.lang.Object[] $r3;

$r2 = staticinvoke <org.easymock.EasyMock:
            java.lang.Object createMock(java.lang.Class)>
            (class "java.lang.Object;");
r1 = (java.lang.Object) $r2;
$r3 = newarray (java.lang.Object)[1];
$r3[0] = r1;
```

Similarly, for mock-containing container (write method), the Jimple IR definition is:

```
java.util.ArrayList $r1;
java.lang.Object $r2;
X r3;

$r1 = new java.util.ArrayList;
specialinvoke $r1.<java.util.ArrayList: void <init>()>();
$r2 = staticinvoke <org.mockito.Mockito:
          java.lang.Object mock(java.lang.Class)>
          (class "X;");
r3 = (X) $r2;
virtualinvoke $r1.<java.util.ArrayList: boolean add(java.lang.Object)>(r3);
```

The container APIs for read and write methods are hard-coded in both the Soot and Doop implementations.

4.2 Imperative Soot Implementation

We first describe the Soot-based imperative dataflow analysis to find mock invocation sites. Our tool tracks information from the creation sites through the control-flow graph using a forward dataflow may-analysis—an object is declared a mock if there exists some execution path where it may receive a mock value. Our implementation also understands containers like arrays and collections, and tracks whether containers hold any mock objects. The abstraction marks all contents of collections as potential mocks if it observes any mock object being put into the array or container.

4.2.1 Forward Dataflow May Analysis

Our forward dataflow analysis maps values (locals and field references) in the Jimple intermediate representation to our abstraction:

$$\texttt{Value} \mapsto \texttt{MockStatus}.$$

`MockStatus` records three bits: one for the value being a mock, one for it being an array containing a mock, and one for it being a collection containing a mock. At most one of the three bits may be true for any given value. Not having a mapping in the abstraction is equivalent to mapping to a MockStatus having all three bits false.

We chose to implement a may-analysis rather than a must-analysis for two reasons: 1) we did not observe any cases where a value was assigned a mock on one branch and a real object on the other branch of an if statement; 2) implementing a must-analysis would not help heuristics to find focal methods, as a must-analysis would rule out fewer mock invocations. Our merge operation is therefore a fairly standard pointwise disjunction of the two incoming values in terms of values and in terms of the 3 bits of `MockStatus`.

Our dataflow analysis uses fairly standard gen and kill sets in the flow function. We set bits in `MockStatus` as follows:

First, the gen set includes pre-analyzed fields containing mock objects defined via annotation (e.g. `@Mock`), inside a constructor `<init>`, or in JUnit's `@Before`/`setUp()` methods. We discuss the pre-analysis below in Section 4.2.4.

Second, it includes local variables assigned from mock-creation source methods, which

consist of Mockito's *java.lang.Object mock(java.lang.Class)*, EasyMock's *java.lang.Object createMock(java.lang.Class)*, and PowerMock's *java.lang.Object mock(java.lang.Class)*:

```
X x = mock(X);
```

Third, it includes values assigned from return values of read methods from mock-containing collections or arrays:

```
// array read;
// r1 is in the in-set as an array mock
X x = r1[0];
// collection read;
// r2 is in the in-set as a collection mock
X x = r2.get(0);
```

Fourth, if `x` is a mock and casted and assigned to `x_cast`, then the gen set includes `x_cast` (e.g. `r1` in Listing 4.2):

```
// x is a mock in the in-set
X x_cast = (X) x;
```

Finally, the gen set includes copies of already-flagged mocks:

```
// x is a mock in the in-set
X y = x;
```

The copy-related rules also apply to mock-containing arrays and collections. We add some additional rules for generating mocks that the program reads from collections and arrays, as well as rules for marking arrays and collections as mock-containing. For instance, in the

below array write, if the in set has `r2` as a mock, then the destination `r1` will be generated as a mock-containing array. Similarly, if `r3` is a known mock, then the collection `$r4` to which it is added (the list of collection add methods is hardcoded) will be generated as a mock-containing collection.

```
// r2 is in the in set as a mock
r1[0] = r2;
// r3 is in the in set as a mock
$r4.<java.util.ArrayList: boolean add(java.lang.Object)>(r3);
```

4.2.2 Toy Example Revisited — Soot Implementation

Let us now revisit the toy example first introduced in Section 3.3.

Figure 4.1 shows how our dataflow analysis works. At the top of the Jimple IR in Listing 4.1, we begin with an empty abstraction (no mapping for any values, equivalent to all bits false for each value) before line 3. For the creation of `$r1` on line 3 and 4, since the call to the no-arg `<init>` constructor is not one of our hardcoded mock APIs, our analysis does not declare `$r1` to be a mock object. In practice, our abstraction simply does not create an explicit binding for `$r1`, instead leaving the mapping empty as it was prior to line 3; but it would be equivalent to create a new `MockStatus` with all bits false and bind it to `$r1`. Thus, we may conclude that the invocation `object1.foo()` on line 5 in Figure 4.1 is not known to be a mock invocation. Tying back to our focal methods application, we would not exclude the call to `foo()` from being a possible focal method.

On the other hand, our imperative analysis sees the mock-creation source methods

```
1   //       mock: ✗    mockAPI: ✗
2   Object object1 = new Object();
3
4   // mock: ✗
5   object1.foo();
6
7   //       mock: ✓    mockAPI: ✓
8   Object object2 = mock(Object.class);
9
10  // mock: ✓
11  object2.foo();
```

Figure 4.1: Our static analysis propagates mockness from sources (e.g. `mock(Object.class)`) to invocations.

`<org.mockito.Mockito: java.lang.Object mock(java.lang.Class)>` on line 7 and 8 in the Jimple IR. It thus adds a mapping from local variable `r2` to a new `MockStatus` with the mock bit set to true. When the analysis reaches line 10, because `r2` has a mapping in the abstraction with the mock bit being set, MockDetector will deduce that the call on line 10 is a mock invocation. This implies that the call to method *foo()* on line 11 in Figure 4.1 cannot be a focal method.

```
1   java.lang.Object $r1, r2;
2
3   $r1 = new java.lang.Object;
4   specialinvoke $r1.<java.lang.Object: void <init>()>();
5   virtualinvoke $r1.<java.lang.Object: void foo()>();
6
7   r2 = staticinvoke <org.mockito.Mockito:
```

```
8                   java.lang.Object mock(java.lang.Class)>
9                   (class "Ljava/lang/Object;");
10    virtualinvoke r2.<java.lang.Object: void foo()>();
```

Listing 4.1: Jimple Intermediate Representation for the code in Figure 4.1.

4.2.3 Arrays and Containers

To be explicit about our treatment of arrays and containers: at a read from an array into a local variable where the source array is mock-containing, we declare that the local destination is a mock. At a write of a local variable into an array where the local variable is mock-containing, we declare that the array is mock-containing.

We treat collections analogously. However, while there is one API for arrays—the Java bytecode array load and array store instructions—Java's Collections APIs include, by our count, 60 relevant methods, which we discuss further in Section 4.4. For our purposes here, we use our classification of collection methods to identify collection reads and writes and handle them as we do array reads and writes, except that we say that it is a mock-containing collection, not a mock-containing array.

Mock-containing Array Toy Example

Figure 4.2 illustrates the process of identifying a mock-containing array, and Listing 4.2 displays the Jimple IR of the code in Figure 4.2. Our analysis reaches the mock API call on line 4–6, where it records that `$r2` is a mock object—it creates a MockStatus

```
1   //       mock: ✓    mockAPI: ✓
2   Object object1 = createMock(Object.class);
3
4   //         arrayMock: ✓   ⇐ array-write     mock: ✓
5   Object[]   objects         = new Object[]   { object1 };
```

Figure 4.2: Our static analysis also finds array mocks.

abstraction object with mock bit set to 1 and associates that object with `$r2`. The tool then handles the cast expression assigning to `r1` on line 7, giving it the same MockStatus as `$r2`. When the analysis reaches line 9, it finds an array reference on the left hand side, along with `r1` stored in the array on the right-hand side of the assignment statement. At that point, it has a MockStatus associated with `r1`, with the mock bit turned on. It can now deduce that `$r3` on the left-hand side is an array container which may hold a mock object. Therefore, MockDetector's imperative static analysis associates `$r3` with a MockStatus with mock-containing array bit ("arrayMock") set to 1.

```
1   java.lang.Object r1, $r2;
2   java.lang.Object[] $r3;
3
4   $r2 = staticinvoke <org.easymock.EasyMock:
5           java.lang.Object createMock(java.lang.Class)>
6           (class "java.lang.Object;");
7   r1 = (java.lang.Object) $r2;
8   $r3 = newarray (java.lang.Object)[1];
```

```
9     $r3[0] = r1;
```

Listing 4.2: Jimple Intermediate Representation for the array in Figure 4.2.

4.2.4 Pre-Analyses for Field Mocks Defined in Constructors and Before Methods

A number of our benchmarks define fields as referencing mock objects via EasyMock or Mockito's `@Mock` annotations, or initialize these fields in the `<init>` constructor or `@Before` methods (*setUp()* in JUnit 3), which test runners will execute before any test methods from those classes. These mock field or mock-containing container fields are then used in tests. In the Soot implementation, we use two pre-analyses before running the main analysis, under the assumption that fields are rarely mutated in the test cases (and that it is incorrect to do so). We have validated our assumption on benchmarks. An empirical analysis of our benchmarks shows that fewer than 0.3% of all fields (29/9352) are mutated in tests.

The first pre-analysis handles annotated field mocks and field mocks defined in the constructors (`<init>` methods), while the second pre-analysis handles `@Before` and `setUp()` methods.

MockDetector retrieves all fields in all test classes, and marks fields annotated `@org.mockito.Mock` or `@org.easymock.Mock` as mocks.

Listing 4.3 depicts an example where instance fields are initialized using field initializers. Java copies such initializers into all class constructors (`<init>`). To detect such mock-

containing fields, we thus simply apply the forward dataflow analysis on all constructors in the test classes prior to running the main analysis, using the same logic that we use to detect mock objects or mock-containing containers in the main analysis. The second pre-analysis handles field mocks defined in `@Before` methods just like the first pre-analysis handled constructors.

```
1    private GenerationConfig config = mock(GenerationConfig.class);
2    private RuleFactory ruleFactory = mock(RuleFactory.class);
```

Listing 4.3: Example for field mocks defined by field initializations from `TypeRuleTest.java` in jsonschema2pojo.

4.2.5 Interprocedural Support

The Heros framework implements IFDS/IDE for program analysis frameworks including Soot. With some effort, it would be possible to rewrite our mock analysis with Heros; however, this would be a more involved process than in the declarative case, where we simply added two rules. In particular, Heros uses a different API in its implementation than Soot. Conceptually, though, it should be no harder to implement an interprocedural Heros analysis than an intraprocedural Soot dataflow analysis.

4.3 Declarative Doop Implementation

We next describe the declarative Doop-based technique that MockDetector uses. We implemented this technique by writing Datalog rules. Similarly to the dataflow analysis,

the declarative approach propagates mockness from known mock sources, through the statements in the intermediate representation, to potential mock invocation sites.

The core of the implementation starts by declaring facts for 9 mock source methods manually gleaned from the mock libraries' documentation, as specified through method signatures (e.g. `<org.mockito.Mockito: java.lang.Object mock(java.lang.Class)>`.) It then declares that a variable `v` satisfies `isMockVar(v)` if it is assigned from the return value of a mock source, or otherwise traverses the program's interprocedural control-flow graph, through assignments, which may possibly flow through fields, collections, or arrays. Finally, an invocation site is a mock invocation if the receiver object `v` satisfies `isMockVar(v)`.

```
// v = mock()
isMockVar(v) :-
AssignReturnValue(mi, v),
callsMockSource(mi).
// v = (type) from
isMockVar(v) :-
isMockVar(from),
AssignCast(_ /* type */, from, v, _ /* inmethod */).
// v = v1
isMockVar(v) :-
isMockVar(v1),
AssignLocal(v1, v, _).
```

The predicates `AssignReturnValue`, `AssignCast`, and `AssignLocal` are provided by

Doop, and resemble Java bytecode instructions. Unlike Java bytecode, however, their arguments are explicit. For instance, `AssignLocal(?from:Var, ?to:Var, ?inmethod:Method)` denotes an assignment statement copying from `?from` to `?to` in method `?inmethod`. (It is Datalog convention to prefix parameters with `?`s).

We designed the analysis in a modular fashion, such that the interprocedural, collections, arrays, and fields support can all be disabled through the use of `#ifdef`s, which can be specified on the Doop command-line.

4.3.1 Toy Example Revisited — Doop Implementation

Again referring to Jimple Listing 4.1, this time we ask whether the invocation on Jimple line 10 satisfies predicate `isMockInvocation` (facts Listing 4.4, line 2), which we define to hold the analysis result—namely, all mock invocation sites in the program. It does, because of facts lines 3–7: Jimple line 10 contains a virtual method invocation, and the receiver object `r2` for the invocation on that line satisfies our predicate `isMockVar`, which holds all mock-containing variables in the program (Section 4.3 provides more details). Predicate `isMockVar` holds because of lines 8–10: `r2` satisfies `isMockVar` because Jimple line 7 assigns `r2` the return value from mock source method `createMock` (facts line 8), and the call to `createMock` satisfies predicate `callsMockSource` (facts line 10), which requires that the call destination `createMock` be enumerated as a constant in our 1-ary relation `MockSourceMethod` (facts line 11), and that there be a call graph edge between the method invocation at line 7 and the mock source method (facts line 12).

```
1   isMockInvocation(<Object: void foo()>/test/0,
```

```
2   <Object: void foo()>, test, _. r2).
3   |VirtualMethodInvocation(<Object: void foo()>/test/0,
4   |                       <Object: void foo()>, test).
5   |VirtualMethodInvocation_Base(<Object: void foo()>/test/0,
6   |                                r2).
7   |isMockVar(r2).
8   |-AssignReturnValue(<Mockito: Object mock(Class)>/test/0,
9   |                       r2).
10  |-callsMockSource(<Mockito: Object mock(Class)>/test/0).
11  |MockSourceMethod(<Mockito: Object mock(Class)>).
12  |CallGraphEdge(_, <Mockito: Object mock(Class)>/test/0, _,
13  |             <Mockito: Object mock(Class)>).
```

Listing 4.4: Facts about invocation `r2.foo()` in method `test`.

4.3.2 Interprocedural Support

From our perspective, including (context-insensitive) interprocedural support is almost trivial; we only need to add two rules

```
// v = callee(), where callee's return
// var is mock
isInterprocMockVar(v) :-
AssignReturnValue(mi, v),
mainAnalysis.CallGraphEdge(_, mi, _, callee),
```

```
ReturnVar(v_callee, callee),
isMockVar(v_callee).

// callee(v) results in formal param
// of callee being mock
isInterprocMockVar(v_callee) :-
isMockVar(v),
ActualParam(n, mi, v),
FormalParam(n, callee, v_callee),
mainAnalysis.CallGraphEdge(_, mi, _, callee),
Method_DeclaringType(callee, callee_class),
ApplicationClass(callee_class).
```

using Doop-provided call graph edges (relation `mainAnalysis.CallGraphEdge`) between the method invocation `mi` and its callee `callee`. The first rule propagates information from callees back to their callers, while the second rule propagates information from callers to callees through parameters. Note that we restrict our analysis to so-called "application classes", excluding in particular the Java standard library. We chose to run our conext-insensitive analysis on top of Doop's context-insensitive call graph, but have also reported results with Doop's `basic-only` analysis, which implements Class Hierarchy Analysis. Mirroring Doop, it would also be possible to add context sensitivity to our analysis, but our results suggest that this would not help much; we'll return to that point in Chapter 5.

4.3.3 Arrays and Containers

Consistent with our analysis being a may-analysis, we define a predicate `isArrayLocalThatContainsMocks` to record local variables pointing to arrays that may contain mock objects. This predicate is true whenever the program under analysis stores a mock variable into an array; we also transfer array-mockness through assignments and casts. When a local variable `v` is read from a mock-containing array `c`, then `v` is marked as a mock variable, as seen in the first rule below. An store of a mock variable `mv` into an array `c` causes that array to be marked as `isArrayLocalThatContainsMocks`. Note that these predicates are mutually recursive. A similar predicate is also applied to containers. We also handle `Collection.toArray` by propagating mockness from the collection to the array.

```
// v = c[idx]
isMockVar(v) :-
isArrayLocalThatContainsMocks(c),
LoadArrayIndex(c, v, _ /* idx */).

// c[idx] = mv
isArrayLocalThatContainsMocks(c) :-
StoreArrayIndex(mv, c, _ /* idx */),
isMockVar(mv).
```

4.3.4 Fields

Apart from the obvious rule stating that a field which is assigned from a mock satisfies `fieldContainsMock`, we also label fields that have the `org.mockito.Mock` or `org.easyMock.Mock` annotation as mock-containing. We declare that a given field *signature* may contain a mock, i.e. the field with a given signature belonging to all objects of a given type. We also support containers stored in fields.

4.3.5 Arrays and Fields

We also support not just array locals but also array fields. That is, when an array-typed field is assigned from a mock-containing array local, then it is also a mock. And when an array-typed local variable is assigned from a mock-containing array field, then that array local is a mock-containing array.

4.4 Common Infrastructure

We have parameterized our technique with respect to mocking libraries and have instantiated it with respect to the popular Java libraries Mockito, EasyMock, and PowerMock. We also support different versions of JUnit[1]: 3, and 4+, and we support the Java Collections API. We discuss this parameterization in this subsection.

Both JUnit and mocking libraries rely heavily on reflection, and would normally pose problems for static analyses. In particular, the set of reachable test methods is enumerated

[1] https://junit.org

at run-time, and the mock libraries create mock objects reflectively. Fortunately, their use of reflection is limited and stylized, and we have designed our analyses to soundly handle these libraries.

4.4.1 JUnit and Driver Generation

JUnit tests are simply methods that developers write in test classes, appropriately annotated (in JUnit 3 by method name starting with "test", in 4+ by a `@Test` annotation). A JUnit test runner uses reflection to find tests. Out of the box, static analysis engines do not see tests as reachable code.

Thus, to enable static analysis over a benchmark's test suite, our tool uses Soot to generate a driver class for each Java sub-package of the suite (e.g. `org.apache.ibatis.executor.statement`). In each of these sub-package driver classes, our tool creates a *runall()* method, which invokes all methods within the sub-package that JUnit (either 3 or 4) considers to be non-constructor test cases, all surrounded by calls to class-level init/setup and teardown methods. Test methods from concrete test classes are particularly easy to call from a driver, as they are specified to have no parameters and are not supposed to rely on any particular ordering. Our tool then creates a RootDriver class at the root package level, which invokes the *runall()* method in each sub-package driver class, along with the `@Test`/`@Before`/`@After` methods found in classes located at the root. The drivers that we generate also contain code to catch all checked exceptions declared to be thrown by the unit tests. Both our Soot and Doop implementations use the generated driver classes.

All static frameworks must somehow approximate the set of entry points as appropriate for their targets. Additionally, the Wala framework [21] creates synthetic entry points, but it does this to perform pointer analysis on a program's main code rather than to enumerate the program's test cases.

4.4.2 Intraprocedural Analysis

The Soot analysis is intraprocedural and the Doop analysis has an intraprocedural version. In both of these cases, we make the unsound (but reasonable for our anticipated use case) assumption that mockness can be dropped between callers and callees: at method entry points, no parameters are assumed to be mocks, and at method returns, the returned object is never a mock. Doop's interprocedural version drops this assumption, and instead context-insensitively propagates information from callers to callees and back; we discuss the results of doing so in Chapter 5.

Call graphs are useful to our intraprocedural analysis in two ways: first, because they help identify calls to mock source methods (that we identify explicitly); and second, because they come with entry points (which we effectively supply to the call graph using our generated driver, as explained above). We assume that developers do not call inherited versions of mock creation sites (for example, it could be a wrapper of the mock source method). However, if a call graph of any inherited versions of mock create sites is available in Doop, our Doop implementation will use it.

4.4.3 Mock Libraries

Our supported mock libraries take different approaches to instantiating mocks. All of the libraries have methods that generate mock objects; for instance, EasyMock contains the `createMock()` method. We consider return values from these methods to be mock objects. Additionally, Mockito contains a fluent `verify()` method which returns a mock object. Finally, Mockito and EasyMock also allow developers to mark fields as `@Mock`; we treat reads from such fields as mock objects. Both implementations start the analysis with these hard coded facts on mock source methods, as described in the mock libraries' documentation.

4.4.4 Containers

As stated above, we hardcode all relevant methods from the Java Collections API. There are 60 such methods in total, which together account for about 1/6th of the total lines in our Doop analysis. In addition to straightforward get and put methods, we also support iterators, collection copies via constructors, and add-all methods. An iterator can be treated as a copy of the container, with the request of an object from the iterator being tantamount to a container get. An add-all method copies the mock-containing collection bit.

Chapter 5

Evaluation

In this chapter, we perform a qualitative evaluation of our MockDetector tool and its potential contribution to finding focal methods, as well as a quantitative evaluation of the tool's effectiveness in identifying mock objects and finding invocations on them. We compare running times and efficacy between our Soot and Doop implementations. In addition, we also investigate four Doop base analyses and the effect of the base analysis on running times and efficacy.

5.1 Qualitative Evaluation

We plan to reproduce Ghafari's algorithm [11] to automatically identify focal methods, and see how much it would benefit from mock removal. However, first, before implementing it, we use manual inspection to evaluate how necessary it is to remove mocks from consider-

ation as focal methods. We present two examples in this section to showcase how our tool eliminates method invocations that are definitely not focal methods.

We begin by revisiting the example first discussed in Section 3.2. Figure 5.1 shows the process of locating the mock object `session` and thus the mock invocation `getRequest()` in the example. Both the Soot and Doop implementations report one mock invocation from the test case. Soot outputs the Invoke Expression for `getRequest()`, whereas Doop outputs the corresponding call-graph edge in the isMockInvocation relation. With the assistance of MockDetector, we could remove `getRequest()` from consideration as a focal method. We judge `getToolchainsForType()` to be the focal method because it is the only method invocation remaining after the elimination process. In addition, as the `length` attribute of the return object `basics` from the invocation `getToolchainsForType()` is checked in the assertion statement on Line 13, this test case indeed tests the behaviour of `toolchainManager.getToolchainsForType()`.

For this example, since Ghafari's algorithm does not consider accessors (or so-called inspector methods in the paper) as focal methods, they will presumably report no focal method for this unit test case. This clearly is an incorrect result from Ghafari's algorithm.

Figure 5.2 displays the second example, a test from vraptor-core with multiple mock invocations. The field mock objects are defined via mock annotations.

Both the Soot and Doop implementations identify all the field mocks, and consequently report three mock invocations from the test case. They also successfully output the Invoke Expressions or the corresponding call-graph edges for the three mock invocations: `getContentType()`, `deserializerFor()`, and `unsupportedMediaType()`. By a process of

elimination, we could deduce that `intercept()` on Line 32 is the focal method.

The heuristic of Ghafari's algorithm requires the unit test case to have at least one assertion statement. Since this unit test case does not have any assertion statement, Ghafari's algorithm will presumably have undefined behaviour analyzing this unit test.

In summary, Ghafari's algorithm does not seem to have a provision for methods with no mutators (presumably returns empty set), nor does it seem to be able to handle test cases with no assertions (presumably undefined behaviour).

Both examples qualitatively demonstrate that our MockDetector tool can help remove invocations on mocks from consideration as focal methods, thus making it easier to identify focal methods. We believe our tool could augment the precision of Ghafari's algorithm. We could also potentially construct a novel elimination-based algorithm. Such an algorithm might be better since it would consider methods that are actually focal methods, like sophisticated getters.

5.2 Description of Benchmark Suite

We have quantitatively evaluated MockDetector on 8 open-source benchmarks, along with a micro-benchmark that we developed to test our tool. We ran all of our experiments on a 32-core Intel(R) Xeon(R) CPU E5-4620 v2 at 2.60GHz with 128GB of RAM running Ubuntu 16.04.7 LTS.

Table 5.1 presents summary information about our benchmarks and run-times, namely the LOC and Soot and Doop analysis run-times for each benchmark. The 9 benchmarks

```
1   @Test
2   public void testMisconfiguredToolchain() throws Exception {
3       //             mock:✓   mockAPI:✓
4       MavenSession session = mock ( MavenSession.class );
5       MavenExecutionRequest req = new DefaultMavenExecutionRequest();
6       //     mock invocation:✓ ⇒ focal method:✗
7       when( session.getRequest() ).thenReturn( req );
8
9       ToolchainPrivate[] basics =
10          //                         focal method:✓
11          toolchainManager.getToolchainsForType("basic", session);
12
13      assertEquals( 0, basics.length );
14  }
```

Figure 5.1: Qualitative evaluation of removing one mock invocation from focal method consideration.

include over 383 kLOC, with 184 kLOC in the test suites, as measured by SLOCCount[1]. The Soot total time is the amount of time that it takes for Soot to analyze the benchmark and test suite in whole-program mode, including our analyses. The Soot intraprocedural analysis time is the sum of run-times for the main analysis plus two pre-analyses, as described in Section 4.2. Meanwhile, the reported Doop run-time is from the context-insensitive analysis, while the Doop analysis time for intraprocedural mock invocation analysis is for running the analysis alone based on recorded facts from the benchmark. The total Doop run-time is much slower than the total Soot run-time because Doop always computes a call-graph, which is an expensive operation. We believe that the Doop analysis-only time is also slower because it computes a solution over the entire program, as opposed

[1] https://dwheeler.com/sloccount/

```
1   // Fields used in the unit test case
2   private DeserializingInterceptor interceptor;
3   // mock annotation:✓
4   @Mock private HttpServletRequest request;
5   // mock annotation:✓
6   @Mock private InterceptorStack stack;
7   // mock annotation:✓
8   @Mock Deserializers deserializers;
9   private MethodInfo methodInfo;
10  // mock annotation:✓
11  @Mock Container container;
12  // mock annotation:✓
13  @Mock private Status status;
14
15  @Before
16  public void setUp() throws Exception {
17  MockitoAnnotations.initMocks(this);
18
19  methodInfo = new DefaultMethodInfo();
20  interceptor = new DeserializingInterceptor(request, deserializers, methodInfo,
        container, status);
21  }
22
23
24  @Test
25  public void willSetHttpStatusCode415IfThereIsNoDeserializerButIsAccepted() throws
        Exception {
26      //    mock invocation:✓ ⇒ focal method:✗
27      when( request.getContentType() ).thenReturn("application/xml");
28      //        mock invocation:✓ ⇒ focal method:✗
29      when( deserializers.deserializerFor("application/xml", container) ).thenReturn(
            null);
30
31      //      focal method:✓
32      interceptor.intercept(stack, consumeXml, null);
33      //        mock invocation:✓ ⇒ focal method:✗
34      verify(status).unsupportedMediaType("Unable to handle media type [application/
            xml]: no deserializer found.");
35      verifyZeroInteractions(stack);
36  }
```

Figure 5.2: Qualitative evaluation of removing multiple mock invocations from focal method consideration.

to Soot, which works one method at a time.

5.3 Field Mocks

We perform an evaluation on the necessity of our pre-analyses finding field mocks. Table 5.2 displays the number of field mock objects that are defined via `@Mock` annotations, in the constructors, and in the `@Before`/`setUp()` methods, respectively.

We focus on the 5 open-source benchmarks that have defined field mock objects. They are BOOTIQUE, MAVEN-CORE, JSONSCHEMA2POJO-CORE, MYBATIS, and VRAPTOR-CORE. Among these 5 benchmarks, JSONSCHEMA2POJO-CORE, MYBATIS, and VRAPTOR-CORE have a high number (565) or a high percentage (over 50%) of of test-related methods containing mock objects, and have many intraprocedural mock invokes. From the results collected in Table 5.2, we can tell these benchmarks also prefer to define field mock objects. Instead of repetitively creating the same mock objects in each test case within the same test class, these benchmarks create the field mock objects once and and consequently use them in all the test cases. This pattern reduces the need for code maintenance. In addition, although BOOTIQUE and MAVEN-CORE have lower number of tests using mock objects, these 2 benchmarks also prefer to define field mock objects. Therefore, 5 out of the 8 open-source benchmarks prefer to define field mock objects for the ease of testing. This suggests that our pre-analysis for field mocks described in Section 4.2.4 is indeed a necessary and an effective step for analyzing mock objects and mock invocations in the test suites.

5.4 Prevalence of Mocks

We next investigated the prevalence of mocks. Table 5.3 presents the number of test-related (Test/Before/After) methods which contain local variables or which access fields that are mocks, mock-containing arrays, or mock-containing collections, as reported by our Soot-based intraprocedural analysis. Across the 8 benchmarks, test-related methods containing local/field mocks or mock-containing containers accounted for 0.35% to 51.8% of the total number of test-related methods found in public concrete test classes. Our benchmarks are from different domains and created by different groups of developers. The difference in mock usage reflects their different philosophies and constraints regarding the creation and usage of mock objects in tests. Benchmarks like VRAPTOR-CORE and JSONSCHEMA2POJO-CORE have more than half of their test-related methods containing mock objects (and mock-containing arrays); in both of these, most field mocks are created via annotations and reused in multiple test cases in the same class.

The core result of this thesis is in Table 5.4, which presents the number of method invocations on mocks detected by our implementations. We present numbers from the imperative intraprocedural Soot implementation, as well as a total of eight versions of the declarative Doop implementation[2]: { "basic-only" (class hierarchy analysis), "context-insensitive" (CI), "context-insensitive-plusplus" (CIPP), "1-object-sensitive" } Doop base analysis × { intraprocedural, interprocedural }.

Note that our declarative and imperative implementations find exactly the same number

[2]bootique, mybatis and vraptor timed out for Doop's 1-object-sensitive analysis without our mock analysis, and we report "–" for their times.

of intraprocedural mock invocations for 4 of our benchmarks. On the others, the main source of missed mock calls in the Soot implementation is missing support for array- or collection-related functions. Our intraprocedural analysis finds that method invocations on mock objects account for a range from 0.086% to 16.4% of the total number of invocations. Our counts of intraprocedural mock invocations show that the benchmarks with more than half of their test-related methods containing mock objects also have a considerable percentage of invocations on mocks. Knowing that existing static analysis tools can not pinpoint mock invocations, this result further suggests that it is necessary to have a tool to exclude mock invocations from call graph edges for subsequent analyses.

The two implementations of mock analysis serve to cross validate each other. Often there is only one implementation for a static analysis project, thus it is difficult to judge on the implementation's soundness and correctness. It is possible to formally prove analysis properties, but even then, nothing guarantees conformance of the implementation to the formal description. For this project, I can cross check the results from two implementations, investigate the discrepancies and decide which implementation to further improve on. The improvements on finding intraprocedural mock invocations will be in future work.

Combining the mock counts result from Table 5.3 and intraprocedural mock invocations result from Table 5.4, we can see that benchmarks such as JSONSCHEMA2POJO-CORE, MYBATIS and VRAPTOR-CORE rely quite heavily on mock objects in their tests, which supports our motivation that it is quite necessary to track mock objects and invocations on mocks, to refine existing algorithms to find focal methods and remove candidates that are definitely not focal methods.

5.4.1 Intraprocedural vs. Interprocedural

In Section 4.4 we discussed the implementation of our intraprocedural and interprocedural analyses. We can now discuss the effects of these implementation choices on the experimental results. Recall that we chose, unsoundly, to not propagate any information across method calls in the intraprocedural analysis. Thus, the intraproc columns in Table 5.4 show smaller numbers than the interproc columns, as expected. Note also that there is a sometimes drastic increase from the intraprocedural to the interprocedural result, e.g. from 40 to 1300 for FLINK-CORE. In 5 out of the 9 benchmarks analyzed, the difference is less than 30, which is relatively minor. Meanwhile, for the remaining 4 benchmarks, the interprocedural analysis reports up to 1200 more mock invocations than the intraprocedural analysis. Other than the source of mock objects propagated from helper methods to unit test cases within the test suite, the difference also includes the mock invocations on the mock objects unavoidably propagated to the main program.

This is because mocks will (especially context-insensitively) propagate from tests to the methods that they call and throughout the main program code. It would be desirable to be able to differentiate test helper methods, which we do want to propagate mocks to, from methods in the main program, which we generally do not want to propagate mocks to. (Although the main program may make mock invocations on objects it is given, we do not want to report these mock invocations from main code.) However, our current analysis infrastructure treats test and main code identically.

Therefore, in the future, we will need a filtering step after the current Doop implementation, which will eliminate mock invocations that occur outside test code. With a

working filtering step, we can get closer to seeing the real difference between intraprocedural and interprocedural results on mock invocations. We will still need to check the validity and correctness of each interprocedural mock invocation before giving definite conclusions about the difference (which again suggests the importance of having two implementations for cross checking). After verifying validity and correctness, we can deduce the set of mock objects that are propagated from helper methods to test cases.

Because of our unsound assumption for the intraprocedural analysis that mocks don't traverse method boundaries, a Doop interprocedural implementation will presumably always report more mock invocations than our intraprocedural. Assuming our current Doop interprocedural implementation correctly finds all mock invocations within the test suite (which can be verified by more manual inspection, or by cross checking after we have a Soot interprocedural implementation), then it is ready to apply for the focal method candidate elimination process (remove the mock invocations from consideration).

5.5 Doop Analysis Results

We explored the run-time performance of our 8 declarative analysis variants based on recorded program facts. We used hyperfine[3] to perform 10 benchmarking runs for the command that performs mock analysis, and present the means and standard deviations from the 10 runs in Table 5.5 and 5.6.

The running times show that the Doop runs with basic-only base analysis spend more time on mock analysis for most benchmarks than the run-times from the more advanced

[3] https://github.com/sharkdp/hyperfine

base analyses. Basic-only is a naive base analysis that generates a much larger call graph than more advanced base analyses (though it still misses some edges, as discussed in Section 5.5.1), and as it would need to check through more call graph edges, we expect that basic-only would generally spend more time on performing mock analysis. Table 5.7 highlights the number of source classes and target classes presented in the call graph generated by the four base analyses, which supports the idea that basic-only generates a bigger call graph, and spends most of the extra time checking over unnecessary classes generated in the call graph instead of performing actual mock analysis.

To further investigate the interprocedural running times on benchmarks including JSONSCHEMA2POJO-CORE, MYBATIS, and VRAPTOR-CORE, we remove call graph edges that have library classes or dependency classes as source classes for each base analysis, and then count the total number of source classes and target classes in the filtered call graph. As Table 5.8 shows, if we take the difference of the number of target classes from the number of source classes for each base analysis, the results suggest that the more advanced base analyses (CI, CIPP) reach more target classes from a better defined subset of source classes that are application classes.

5.5.1 Investigating Basic-Only CHA

We believe that the basic-only base analysis has some undesired behaviour in the process of generating the CHA. We have discussed with the Doop authors. They consider that calls to concrete implementations of abstract methods are not included in CHA. We believe this is not correct from our data collection but it is their assumption. Fig-

ure 5.3 shows the call graph edges from the method *isApplicableType* on the class PatternRule to `java.lang.String fullName()`, which is a sibling method (i.e. a method that has implementations in multiple classes under the same superclass) that is first declared in the abstract base class `com.sun.codemodel.JType`. From Figure 5.3, we can tell that the basic-only base analysis only reports edges to the (abstract) method `com.sun.codemodel.JType.fullName`, whereas the call graph generated by CI base analysis reports edges to implementation sites of the target method. We believe that this shows that there are missing edges in basic-only from its choice to stop at the abstract level. Combining this information with the total number of interprocedural mock invocations reported for the three benchmarks, we know that CI and CIPP have found notably more mock invocations in the process of searching through the edges to the implementation sites of the sibling methods, and the time spent on this process possibly accounts for the higher mock-analysis running times of CI and CIPP on benchmarks JSONSCHEMA2POJO-CORE, MYBATIS, and VRAPTOR-CORE.

5.5.2 Mock Invocations Results by Different Base Analyses

The four base analyses report the same number of intraprocedural mock invocations in 8 benchmarks. The minor difference in VRAPTOR-CORE is due to one method (which ought to be present) not showing up in Doop's context-insensitive call-graph.

From our investigation of basic-only's CHA, it is probably safe to disregard interprocedural basic-only mock invocation results.

```
Call-graph Edge in Basic-only:
0   <org.jsonschema2pojo.rules.PatternRule: boolean
isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
0   <com.sun.codemodel.JType: java.lang.String fullName()>

Call-graph Edges in CI:
<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JTypeVar: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JTypeWildcard: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JNarrowedClass: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JArrayClass: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JDefinedClass: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JDirectClass: java.lang.String fullName()>

<<immutable-context>> <org.jsonschema2pojo.rules.PatternRule:
boolean isApplicableType(com.sun.codemodel.JFieldVar)>/com.sun.codemodel.JClass.fullName/0
<<immutable-context>> <com.sun.codemodel.JCodeModel\$JReferencedClass: java.lang.String fullName()>
```

Figure 5.3: The call graph edges to the method *fullName()* from both basic-only and context-insensitive base analyses. The method is declared in abstract class `com.sun.codemodel.JType` and implemented in its children classes.

More Sophisticated Analyses Analyzing the Doop interprocedural results among the three more advanced base analyses (CI, CIPP and 1-object-sens), we see that they report the same number of interprocedural mock invocations in 5 out of 9 benchmarks. CI and CIPP report the same number of interprocedural mock invocations in 2 more benchmarks that 1-object-sensitive base analysis timed out in generating the call graph.

We can observe that CI and CIPP have comparable run-times and mock invocation counts, with CI having slightly higher run-times, both intraprocedurally and interprocedurally. This makes sense, as CIPP is "context-insensitive with an enhancement for low-hanging fruit: methods that have their params flow to their return value get a methods that have their params flow to their return value get a 1-obj treatment" [18]. Data presented in Table 5.7 demonstrates that the call graph sizes are comparable for CI and CIPP, with CI's call graphs generally slightly bigger as expected.

On the other hand, since 1-object-sensitive base analysis is a more sophisticated analysis, it spends more time on building the call graph (3 benchmarks ending up with timed-out runs on building the call graph). We feed the same input to all four base analyses, but a more sophisticated analysis, like 1-object-sensitive, returns a much smaller call-graph. We believe it produces a better defined call-graph and thus executes faster on the actual mock analysis. However, we need to communicate to Doop developers more and have a better understanding of the logic involved in these sophisticated analyses before making a conclusion.

Table 5.1: Our suite of 8 open-source benchmarks (8000–117000 LOC) plus our microbenchmark. Soot and Doop analysis run-times.

Benchmark	Total LOC	Test LOC	Soot intraproc total time (s)	Doop intraproc total time (s)	Soot intraproc mock analysis (s)	Doop intraproc mock analysis (s)
bootique-2.0.B1-bootique	15530	8595	58	2810	0.276	19.93
commons-collections4-4.4	65273	36318	114	694	0.386	14.20
flink-core-1.13.0-rc1	117310	49730	341	1847	0.415	27.21
jsonschema2pojo-core-1.1.1	8233	2885	313	1005	0.282	29.33
maven-core-3.8.1	38866	11104	183	588	0.276	19.49
micro-benchmark	954	883	47	387	0.130	11.73
mybatis-3.5.6	68268	46334	500	4477	0.662	59.83
quartz-core-2.3.1	35355	8423	155	736	0.231	21.06
vraptor-core-3.5.5	34244	20133	371	1469	0.455	34.95
Total	384033	184405	2082	14013	3.123	237.73

Table 5.2: Counts of Field Mock Objects defined via `@Mock` annotation, in the constructors, and in `@Before` methods, in each benchmark's test suite.

Benchmark	# of Annotated Field Mock Objects	# of Field Mock Objects defined in the `<init>` constructor	# of Field Mock Objects defined in @Before methods
bootique-2.0.B1-bootique	0	0	8
commons-collections4-4.4	0	0	0
flink-core-1.13.0-rc1	0	0	0
jsonschema2pojo-core-1.1.1	26	126	0
maven-core-3.8.1	7	0	1
micro-benchmark	2	0	29
mybatis-3.5.6	41	0	0
quartz-core-2.3.1	0	0	0
vraptor-core-3.5.5	263	128	83

Table 5.3: Counts of Test-Related (Test/Before/After) methods in public concrete test classes, along with counts of mocks, mock-containing arrays, and mock-containing collections, reported by Soot intraprocedural analysis.

Benchmark	# of Test-Related Methods	# of Test-Related Methods with mocks (intra)	# of Test-Related Methods with mock-containing arrays (intra)	# of Test-Related Methods with mock-containing collections (intra)
bootique-2.0.B1-bootique	420	32	7	0
commons-collections4-4.4	1152	3	1	1
flink-core-1.13.0-rc1	1091	4	0	0
jsonschema2pojo-core-1.1.1	145	76	1	0
maven-core-3.8.1	337	24	0	0
micro-benchmark	59	43	7	25
mybatis-3.5.6	1769	330	3	0
quartz-core-2.3.1	218	7	0	0
vraptor-core-3.5.5	1119	565	15	0
Total	6310	1084	34	26

Table 5.4: Number of InstanceInvokeExprs on Mock objects analyzed by Soot and Doop, and Total Number of InstanceInvokeExprs, in each benchmark's test suite. "–" = timed out after 90 minutes. Runs [mybatis, basic-only] and [flink-core, 1-object-sensitive] take close to 90 minutes and sometimes time out.

Benchmark	Total Number of Invocations	Mock Invokes intraproc (Soot)	Mock Invokes intraproc (Doop)				Mock Invokes interproc (Doop)			
			basic -only	CI	CIPP	1-obj -sens	basic -only	CI	CIPP	1-obj -sens
bootique	3366	99	99	99	99	–	120	122	122	–
commons-collection4	12753	11	3	3	3	3	23	23	23	23
flink-core	11923	40	40	40	40	40	1262	1389	1374	1229
jsonschema2pojo-core	1896	276	282	282	282	282	462	604	604	604
maven-core	4072	23	23	23	23	23	31	39	39	39
microbenchmark	471	108	123	123	123	123	132	132	132	132
mybatis	19232	575	577	577	577	–	644	1345	1345	–
quartz-core	3436	21	21	21	21	21	23	31	31	31
vraptor-core	5868	942	963	962	962	–	1301	1630	1627	–

Table 5.5: Intraprocedural Doop analysis-only run-time (in seconds) after basic-only, context-insensitive, context-insensitive-plusplus and 1-object-sensitive base analyses. "–" = timed out after 90 minutes. Runs [mybatis, basic-only] and [flink-core, 1-object-sensitive] take close to 90 minutes and sometimes time out.

Benchmark	intraproc							
	basic -only	σ	CI	σ	CIPP	σ	1-obj -sens	σ
bootique	21.26	1.73	19.93	1.85	19.51	1.74	–	–
commons-collections4	15.39	1.39	14.20	1.36	14.48	0.97	13.39	1.08
flink-core	29.87	2.29	27.21	30.34	27.78	4.54	21.17	0.49
jsonschema2pojo	31.72	1.44	29.33	2.33	30.72	3.52	29.35	3.69
maven-core	20.40	0.97	19.49	1.80	16.09	1.61	15.72	1.09
microbenchmark	13.20	0.76	11.73	0.73	11.61	0.59	12.23	1.15
mybatis	83.29	5.57	59.83	4.07	59.76	0.59	–	–
quartz-core	21.95	2.28	21.06	2.08	18.88	2.84	17.22	1.91
vraptor	49.27	2.59	34.96	0.79	36.49	1.82	–	–

Table 5.6: Interprocedural Doop analysis-only run-time (in seconds) after basic-only, context-insensitive, context-insensitive-plusplus and 1-object-sensitive base analyses. "–" = timed out after 90 minutes. Runs [mybatis, basic-only] and [flink-core, 1-object-sensitive] take close to 90 minutes and sometimes time out.

Benchmark	interproc							
	basic -only	σ	CI	σ	CIPP	σ	1-obj -sens	σ
bootique	26.44	1.73	24.90	0.69	25.50	1.69	–	–
commons-collections4	17.80	1.42	16.64	1.06	16.82	0.79	15.61	1.60
flink-core	68.26	1.98	62.12	3.99	63.94	5.64	54.11	0.32
jsonschema2pojo	39.76	2.33	41.05	2.90	35.37	1.96	37.29	4.55
maven-core	26.41	1.60	23.42	1.98	22.16	1.82	19.62	1.75
microbenchmark	13.76	1.03	12.92	0.95	13.02	1.18	12.30	1.53
mybatis	113.38	4.57	192.16	23.36	181.92	4.01	–	–
quartz-core	23.49	2.42	21.92	1.84	21.97	1.78	19.80	2.75
vraptor	70.92	2.85	149.38	5.73	148.35	2.62	–	–

Table 5.7: Call graph statistics: total number of source classes and target classes from inter-procedural Doop analysis with basic-only, context-insensitive, context-insensitive-plusplus, and 1-object-sensitive base analyses. "–" = timed out after 90 minutes. Runs [mybatis, basic-only] and [flink-core, 1-object-sensitive] take close to 90 minutes and sometimes time out.

Benchmark	Source Classes				Target Classes			
	basic -only	CI	CIPP	1-obj -sens	basic -only	CI	CIPP	1-obj -sens
bootique	12187	3683	3678	–	12910	3422	3417	–
commons-collections4	8576	2762	2757	2726	8916	2657	2652	2622
flink-core	17381	3962	3946	3928	18062	3764	3748	3731
jsonschema2pojo	16876	2976	2967	2962	17832	2817	2808	2804
maven-core	13228	2858	2849	2845	14102	2726	2717	2713
microbenchmark	7227	1603	1603	1591	7569	1527	1527	1515
mybatis	31778	4774	4771	–	32043	4542	4539	–
quartz-core	11036	2919	2918	2906	11424	2786	2785	2774
vraptor	18294	4352	4339	–	19603	4240	4227	–

Table 5.8: Call graph statistics: total number of source classes that are application classes (i.e., excluding classes from dependencies or libraries) and total number of target classes reached from application classes by interprocedural Doop analysis with basic-only, context-insensitive, context-insensitive-plusplus, and 1-object-sensitive base analyses. "–" = timed out after 90 minutes. Runs [mybatis, basic-only] and [flink-core, 1-object-sensitive] take close to 90 minutes and sometimes time out.

Benchmark	Source Classes				Target Classes			
	basic -only	CI	CIPP	1-obj -sens	basic -only	CI	CIPP	1-obj -sens
bootique	514	358	358	–	631	663	627	–
commons-collections4	896	799	798	778	1042	1101	1093	1044
flink-core	4730	1372	1365	1362	4951	1855	1812	1777
jsonschema2pojo	123	119	119	119	256	301	301	299
maven-core	798	291	291	291	1244	554	531	515
microbenchmark	13	12	12	12	36	46	46	44
mybatis	1981	1271	1271	–	2385	1872	1842	–
quartz-core	386	288	288	288	559	546	520	501
vraptor	904	649	649	–	1259	1376	1353	–

Chapter 6

Related Work

We discuss related work in the areas of declarative versus imperative static analysis, treatment of containers, and taint analysis.

6.1 Imperative vs Declarative

Kildall contributed perhaps the first dataflow analysis [13] as the concept is understood today, describing an algorithm for intraprocedural constant propagation and common subexpression elimination. His algorithm, operating on the program graph, is described in quite imperative pseudocode (and proven to terminate). In some sense, implementing algorithms imperatively is the default, and doesn't need further discussion, except to point out that program analysis frameworks such as Soot [20] provide libraries that can ease the implementation burden.

To our knowledge, Corsini et al did some of the first work in declarative program analysis [8]; however, that work performed abstract interpretation on (tiny) logic programs rather than imperative programs. Dawson et al [9] did similar work. Around the same time, Reps proposed [16] a declarative analysis to perform on-demand versions of interprocedural program analyses, which is similar to what we have here; however, we compute all of the analysis results rather than performing an on-demand analysis. CodeQuest by Hajijev et al [12] also allows developers to perform AST-level code queries using a declarative query language. DIMPLE$^+$ [3][2, Chapter 3] by Benton and Fischer may be closest to what we are advocating as the declarative analysis approach. While Benton's dissertation presents a simple DIMPLE$^+$ implementation of Andersen's points-to analysis, the DIMPLE$^+$ work does not have Doop's sophisticated pointer analysis available to it. Soufflé, by Scholz et al [17], advocates for declarative static analysis (but without comparing it directly to an imperative approach as we do here), and presents performance optimizations needed to achieve this goal. Finally, Doop [6], which is now primarily implemented with a Soufflé backend, is perhaps the most powerful extant declarative program analysis, and focuses on expressing sophisticated pointer analyses in Datalog.

In terms of comparing implementations, Prakash et al [15] compare pointer analyses as provided by Doop and Wala; in some sense, the present work is similar to that work in that both works compare two frameworks. However, that work compares empirical results from two families of pointer analysis implementations (and finds that the specific intermediate representation used doesn't change the results much), while we discuss the process of implementing a static analysis declaratively versus imperatively. Like us, they note that Doop is difficult to incorporate into a program transformation framework (it works better

in standalone mode) while Wala's results are readily available; a similar result applies to any result that a Soot-based data flow analysis produces as compared to a Doop-based declarative analysis.

6.2 Treatment of Containers

In this work, we use coarse-grained abstractions for containers, consistent with the approach from Chu et al [7]. In our experience, test cases do not perform sophisticated container manipulations where it would be necessary to track exactly which elements of a container are mocks. Were such an analysis necessary, we could use the fine-grained container client analysis by Dillig et at [10].

6.3 Taint Analysis

Like many other static analyses, our mock analysis can be seen as a variant of a static taint analysis: sources are mock creation methods, while sinks are method invocations. There are no sanitizers in our case. However, for a taint analysis, there is usually a small set of sink methods, while in our case, every method invocation in a test method is a potential sink. Additionally, the goal of our analysis (detecting possible mocks) is different in that it is not security-sensitive, so the balance between false positives and false negatives is different—it is less critical to not miss any potential mock invocations, whereas missing a whole class of tainted methods would often be unacceptable.

Chapter 7

Discussion

Having described our imperative and declarative approaches to implementing mock analysis, we now comment on the strengths and weaknesses of these two approaches. We hope that our discussion will help future designers of source code analyses and frameworks.

7.1 Subsequent Use of Results

Doop is a standalone tool. It depends on other tools to provide input, but provides output in the form of `csv` files, whose content can be matched to the program source, if a subsequent analysis has the appropriate internal representation. On the other hand, Soot is a compiler framework. Thus, using the Soot analysis results in a subsequent compilation phase is quite easy. Doop works quite well for producing analysis results, and not quite as well for using these results in a compilation process. Our Soot analysis also doesn't need to process the whole program for itself to produce the analysis results that we're interested

in here—our intraprocedural analysis can use the existing in-memory representation and pass it on to the next phase, while Doop reads the whole program, throws it away, and leaves nothing for the next compilation phase.

7.2 Expressiveness vs Concision

In [6], Bravenboer and Smaragdakis point out that:

> Even conceptually clean program analysis algorithms that rely on mutually recursive definitions often get transformed into complex imperative code for implementation purposes.

The presentation of the declarative approach in Chapter 4 could meaningfully include direct excerpts from the Datalog; including Java code is rarely meaningful, as there is too much boilerplate in that language.

The declarative approach takes 237 non-comment lines, compared to about 533 non-comment lines for the main part of the imperative approach, which is a significant point in favour of Doop. A head-to-head comparison is tricky, as the imperative approach also uses pre-analyses which are not present in the declarative approach.

We comment on the reasons for using helper analyses in the imperative version and not the declarative version. Recall that the helper analyses pre-computed information about 1) mock annotations and 2) constructors and setup methods. The mock annotations are an inessential difference; they could be computed on the fly in the imperative version, as

they are in the declarative version. As for the constructors: when thinking imperatively, it is more intuitive to explicitly order the computations for constructors before regular test methods. On the other hand, thinking declaratively, it is more natural to use mutual recursion to declare a dependency on the results of previous computations for fields (our relation `isCollectionFieldThatContainsMocks` in particular) than to declare an explicit ordering. There is a small semantic difference in the two implementations, as the declarative implementation does not require field writes to be confined to constructors and setup methods; in this particular case, we empirically verified that the imperative assumption was almost always satisfied.

We also contrast how we store the abstraction in the two versions. The imperative version uses a standard dataflow analysis abstraction (three bits per local variable/field reference), along with an explicitly specified merge operator, while the declarative version uses one relation for each of the three bits. Propagating and merging data happens automatically in Doop.

Another difference between the declarative and imperative versions is in the support for interprocedural analysis. As stated earlier, in Chapter 4, the declarative version implements a context-insensitive interprocedural analysis while the imperative version is intraprocedural. The choice of intraprocedural versus interprocedural depends strongly on the particular analysis being implemented. Implementing the interprocedural analysis declaratively was impressively easy, while it is significantly more challenging to implement an interprocedural analysis in Soot, requiring the use of Heros [5], an additional framework. On the other hand, the Heros implementation would be IFDS-based and be context-sensitive; it would be somewhat harder to upgrade our context-insensitive implementation to a context-sensitive

Doop implementation.

It is easier to add instrumentation, e.g. timers, to the imperative version than the declarative version. Doop contains some built-in timers, but it is unclear how to add new ones.

7.3 Development Velocity

To help the reader calibrate our descriptions, we describe our experience levels with Soot and Doop. I initially had little Soot experience, developed the Soot implementation through countless hours of testing and debugging, and took guidance and suggestions from my supervisor, who is an early code contributor to the Soot framework. My supervisor, the co-author of the submitted conference research paper, developed the Doop implementation.

Soot is a mature program analysis framework and many of the common sticking points have, over the years, been addressed by the developers. Nevertheless, it can be intimidating to start working with Soot. Our experience with Doop is that it is overall robust, yet still being actively developed (i.e. occasionally, at the start, some daily snapshots didn't work with some versions of the underlying Soufflé engine). There is more documentation for Soot than for Doop, although even for Doop, it is often possible to scrape together answers to one's questions from the source code and the online documentation. Finding the right API (or relation, in Doop) to use can be challenging for both Soot and Doop; it's impossible for us to fairly compare them, due to our different experiences with Soot and Doop.

We thank the Doop developers for their timely and helpful answers to our questions;

developer or community support is necessary to successfully use Doop (or, for that matter, most research-grade program analysis frameworks, including Soot).

Most of the time, adding a feature to the declarative version (e.g. field support) required an evening of work. This typically happened first; the declarative version is better for cleanly describing some approximation of the desired behaviour. Somewhat to our surprise, it was then possible to fast-follow with the imperative version, which ended up not taking much more than an evening to implement either. We believe that the existence of the declarative specification helped with designing the imperative version.

The declarative version was still subject to the combinatorial feature interaction problem; for instance, when we added support for fields and containers, we also specifically needed to add support for containers stored in fields.

Debugging is an inevitable part of any development process, including this one; declarative languages are no proof against debugging. Some Doop errors were just frustrating, e.g. hardcoding a syntactically incorrect method signature for a collection method. Other times, better type system enforcement in Datalog, and in particular, identifying relations that are unsatisfiable due to type conflicts, would help. Soot errors are typical programming errors.

Iteration speed can help with more effective debugging. On some benchmarks, Soot iterations could finish in under a minute, while Doop analysis-only iterations could finish in 10 seconds (but we didn't know that at the time). To expand on that: while developing our analysis, we ran our analysis together with the main analysis, and recomputed the main analysis every time we iterated. Yet, Doop supports running add-on analyses like

ours, in isolation, after the main analysis terminates. If we were doing it again, we would develop our analysis as a run-after analysis. Running with the main analysis requires at least a 2.5-minute iteration time due to the necessity of re-compiling and re-running the entire analysis every time the analysis changes, while running an analysis after the main analysis can take 10 seconds, as mentioned above. Setting up the analysis to run after the main analysis is trickier and requires understanding of Doop which we did not have until late in the process.

As stated above, instrumentation is easier in Soot than in Doop, and that extends to printing debug information and using traditional debugging tools, which works as well for Soot as traditional debugging does in general. To debug the Doop analysis, we resorted to outputting relevant relations after a Doop run and manually pinpointing which facts were missing or extraneous. Because Doop uses Soot to generate program facts, understanding Soot in particular and compilers in general was invaluable while developing the Doop implementation—we also looked at the Soot intermediate representation to understand what analysis information was flowing to which intermediate variables.

7.4 Future Work

In this project, we presented two implementations for detecting mock objects and tracking mock invocations in test suites. There are multiple areas require further investigation to improve MockDetector's quality:

- We will work on the Soot interprocedural implementation in our tool for a more

complete mock analysis. It will be useful to compare and further understand the two approaches analyzing mock objects.

- The current interprocedural Doop implementation could not differentiate method invocations in the test suites from the ones in the source code. We will work on modifying Doop's implementation to only report results from the test suites for move accurate results.

- We plan to build construct an automated focal method analysis tool. It aims to help quantitatively evaluate our MockDetector tool in assisting for the focal method findings.

7.5 Summary

We would conclude that, especially with the knowledge we have now gained about Doop, prototyping in Doop is easier than in Soot, but that it is no panacea; it remains subject to the feature interaction problem as well as debugging. Additionally, trying to add certain functional behaviours to the Doop implementation, such as timers, can be challenging.

Overall, in this thesis, we have described a case study of the MockDetector static analysis, which we intend to use for further static analyses of test cases. We have implemented MockDetector twice—once imperatively in Soot, and once declaratively in Doop—and characterized its performance on a test suite. Finally, we have discussed our experience implementing this analysis twice, and pointed out the benefits and disadvantages of the imperative and declarative approaches for writing static analyses.

References

[1] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2002.

[2] William C. Benton. *Fast, Effective Program Analysis for Object-Level Parallelism.* PhD thesis, University of Wisconsin–Madison, December 2008.

[3] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: From prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, page 13–24, New York, NY, USA, 2007. Association for Computing Machinery.

[4] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 3–8, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2012)*, pages 3–8, July 2012.

[6] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

[7] Hang Chu and Patrick Lam. Collection disjointness analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 45–50, Beijing, China, 2012.

[8] Marc-Michel Corsini, Kaninda Musumbu, Antoine Rauzy, and Baudouin Le Charlier. Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains (extended abstract). In Maurice Bruynooghe and Jaan Penjam, editors, *Progamming Language Implementation and Logic Programming*, pages 75–91, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[9] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 117–126, New York, NY, USA, 1996. Association for Computing Machinery.

[10] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 187–200, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70, 2015.

[12] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[14] Shaikh Mostafa and Xiaoyin Wang. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th International Conference on Quality Software*, pages 127–132, October 2014.

[15] Jyoti Prakash, Abhishek Tiwari, and Christian Hammer. Effects of program representation on pointer analyses — an empirical study. In Esther Guerra and Mariëlle

Stoelinga, editors, *Fundamental Approaches to Software Engineering*, pages 240–261, Cham, 2021. Springer International Publishing.

[16] Thomas W. Reps. *Demand Interprocedural Program Analysis Using Logic Databases*, pages 163–196. Springer US, Boston, MA, 1995.

[17] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Yannis Smaragdakis. context insensitive plusplus source code. https://bitbucket.org/yanniss/doop/src/master/souffle-logic/analyses/context-insensitive-plusplus/analysis.dl, 2021.

[19] Yannis Smaragdakis. Doop bitbucket repository. https://https://bitbucket.org/yanniss/doop/src/master/, 2021.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot—a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 214–224. IBM Press, 1999.

[21] WALA. T.J. Watson Libraries for Analysis. https://github.com/wala/WALA, Jan 2019.

APPENDICES

Appendix A

Field Mutation Analysis

Table A.1: Counts of fields mutated in test cases, and counts of all fields found in test cases.

Benchmark	Total # of Fields Mutated in Test Cases / Total # of Fields
bootique-2.0.B1-bootique	0 / 271
commons-collections4-4.4	3 / 697
flink-core-1.13.0-rc1	8 / 2675
jsonschema2pojo-core-1.1.1	0 / 228
maven-core-3.8.1	0 / 765
microbenchmark	5 / 32
mybatis-3.5.6	0 / 2618
quartz-core-2.3.1	2 / 878
vraptor-core-3.5.5	10 / 1193
Total	29 / 9352