Declarative Mocking

Hesam Samimi, Rebecca Hicks, Ari Fogel, Todd Millstein
Computer Science Department
University of California, Los Angeles, USA
{hesam, rdhicks, arifogel, todd}@cs.ucla.edu

ABSTRACT

Test-driven methodologies encourage testing early and often. *Mock objects* support this approach by allowing a component to be tested before all depended-upon components are available. Today mock objects typically reflect little to none of an object's intended functionality, which makes it difficult and error-prone for developers to test rich properties of their code. This paper presents *declarative mocking*, which enables the creation of expressive and reliable mock objects with relatively little effort. In our approach, developers write method specifications in a high-level logical language for the API being mocked, and a constraint solver dynamically executes these specifications when the methods are invoked. In addition to mocking functionality, this approach seamlessly allows data and other aspects of the environment to be easily mocked. We have implemented the approach as an extension to an existing tool for executable specifications in Java called PBNJ. We have performed an exploratory study of declarative mocking on several existing Java applications, in order to understand the power of the approach and to categorize its potential benefits and limitations. We also performed an experiment to port the unit tests of several open-source applications from a widely used mocking library to PBNJ. We found that more than half of these unit tests can be enhanced, in terms of the strength of properties and coverage, by exploiting executable specifications, with relatively little additional developer effort.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; K.6.3 [**Software Management**]: Software development

General Terms

Verification, Design

Keywords

mock objects, executable specifications

1. INTRODUCTION

In every stage of software development, programmers rely on testing to gain confidence in their code and guide extensions and modifications. This is particularly the case with test-driven and agile development paradigms [1, 2], where unit tests guide the design effort from the start. A dilemma that developers frequently face during the early stages of development, however, is that their code depends on other pieces of software that are not yet available. For example, another team may have not yet delivered a necessary piece of the software. Another common scenario is when code relies on large infrastructure such as databases, webservers, or data centers, but specific options have not been decided, or there is too much effort or cost to set these up properly during development and testing.

Software engineers have devised a technique, known as *mocking*, to deal with these situations. Mocking leverages the fact that although the code under test needs to interact with pieces of software that are not available, typically the interface (API) and intended behavior of that software are known. For instance, while the specific database to use may not have been determined, a Java client of this database may assume that the Java Database Connectivity (JDBC) API will be used. *Mock objects* [3, 4] serve as dummy implementations of an API, enabling programmers to write and test their code as if all the necessary dependencies are in place. Once the missing parts become available, virtually no code change is necessary to adapt to the real, instead of the mock, objects.

Mock objects in the form they exist today are undoubtedly useful as they enable unit tests to be performed despite missing dependencies, with very little effort. Yet they are severely limited in the benefits they can provide to programmers. Typically a mock object is just a stub, with little if any of the actual functionality of the code it is mocking. Therefore, mocking libraries (e.g. Mockrunner [5], Mockito [6]) require clients to explicitly indicate the results they expect from the mock object and to only indirectly test the correctness of their code through implementation-specific checks. As a result, tests are fragile, error-prone, and difficult to understand or reuse.

1.1 Motivating Example

As a small example, Fig. 1 shows how Mockito, a mocking library from Google, can be used to test an implementation

```
1  class MySet implements Set {
2    List elems;
3    void add(Object o) {
4      if (!elems.contains(o))
5        elems.add(o);
6    }
7    void testAdd() {
8      List mockList = mock(List.class);
9      MySet s = new MySet(mockList);
10     when(mockList.contains(0))
11         .thenReturn(false);
12     s.add(0);
13     verify(mockList, times(1)).add(0);
14     when(mockList.contains(0))
15         .thenReturn(true);
16     s.add(0);
17     // shouldn't add duplicates:
18     verify(mockList, times(1)).add(0);
19   }
20 }
```

Figure 1: Using the Mockito mocking library to test an implementation of sets against a mock list object

of `Set` interface that is internally built using a list object that, hypothetically, is not yet available. The programmer's goal is to test that the `add` method for `MySet` avoids adding duplicate elements. Unfortunately, this seemingly simple task is not particularly straightforward. The call to `mock` on line 8 creates a stub object that meets the `List` interface. By default this stub does not faithfully implement the intended behavior of `List`'s `contains` and `add` methods. For example, the stub has no way of knowing what boolean value to return upon a call to `contains`, so the programmer is required to explicitly provide this information. Therefore, on line 10 the programmer indicates that `contains` should return `false` when given the argument value 0. Worse, she has to update this information on line 14, since 0 has now been (conceptually at least) added to the list. Similarly, because the mock list's `add` method does not actually add the given element to the list, the programmer cannot directly check that duplicates are handled properly. Instead, she uses Mockito's `verify` method to check the number of times that the mock object's `add` method is invoked (lines 13 and 18). These checks ensure that the mock object's `add` method is not invoked on the second invocation of `s.add(0)`, which implies that the duplicate element is properly ignored.

The limitations of mock objects are clear on this simple example, and these problems are exacerbated as the objects being mocked and the code being tested become more complex. To overcome these limitations, our goal is to enable programmers to easily build mock objects that directly reflect important parts of the functionality that they mock. Another goal is to have mocks that are less coupled with any specific implementation of both the code under test and the code being mocked, which makes the tests more robust. Of course, any practical approach should require much less effort than it would take to actually implement the object being mocked, or else the benefits of mocking are lost.

1.2 Declarative Mocking

We observe that recent progress on *executable specifications* and *declarative execution* can naturally support our goals: programmers can write specifications in a high-level logical language for the methods in the API being mocked,

and a constraint solver dynamically *executes* these specifications when the methods are invoked. Specifications are often simpler than imperative code because they can directly express *what* behavior is desired without specifying *how* that behavior is achieved. Specifications also naturally support nondeterminism, which is useful both for modeling actual nondeterminism in the mocked object (e.g., the order in which messages will be received over the network) and for enabling *partial* specifications, with the nondeterminism used to represent "don't care" situations. We call the resulting approach *declarative mocking*. Note that the effort in writing specifications can be amortized over their many benefits: while stubs like those in Fig. 1 must be carefully tailored to each individual test, specifications can define the behavior of a mocked API once and for all. Furthermore, the same specifications can be employed for static verification and/or dynamic contract checking of the "real" component being mocked.

In addition to using specifications to mock functionality, we observe that the same technology naturally supports the mocking of *data*, which provides additional value in the context of program testing. Unit tests commonly require a precondition to be established, i.e. the inputs and state of a program need to be initialized to satisfy some relevant properties, before the test can be executed. Executable specifications remove the need for imperative code to perform this initialization, instead allowing the tester to directly specify the intended precondition, reducing tester effort and increasing understandability.

Several recent works have used constraint solving to generate mock objects as part of an approach to automated test-case generation [7, 8, 9, 10, 11]. Declarative mocking uses similar technology but for a different purpose. Whereas the prior works employ a constraint solver offline in order to generate high-coverage tests, declarative mocking employs a constraint solver online to dynamically substitute for missing code or data. Therefore, declarative mocking still requires users to provide their own tests. On the other hand, declarative mocks are fully executable with arbitrary inputs, independent of any test cases. For example, these mocks can be used to perform system-level testing without having to generate explicit system-level tests, and they allow users to easily interact with a running system, where mocks fill in for any missing functionality, to manually test features of interest.

1.3 Implementation and Evaluation

We have implemented declarative mocking for Java on top of an existing Java extension supporting executable specifications, called PBnj [12]. The PBnj compiler and all mocking examples and benchmarks in this paper are available at `http://www.hesam.us/mockspecs`.

We evaluate declarative mocking with PBnj in two ways. First, we performed an exploratory study on four open-source applications that we considered good targets for declarative mocking. These applications respectively interact with a web server, a database, a server that implements a file-transfer protocol, and a data center providing computational resources in the cloud. We used this study to classify various scenarios under which declarative mocking can potentially provide value over traditional mocks, and we also identify potential limitations of the approach.

Second, we ported six existing applications from Google Code that use Mockito to instead use PBNJ and we analyze the results using the classification derived from our exploratory study. Concretely, this second study is designed to answer the following two research questions:

RQ1 What is the overhead for a developer to use declarative mocks, when used to replicate the exact behavior of traditional mocking approaches today?

RQ2 How often and under what scenarios do declarative mocks offer advantages beyond the traditional approaches, with a justifiable amount of additional effort?

We investigated RQ1 by first porting each existing Mockito-based unit test to use PBNJ in such a way that the exact behavior is preserved. This experiment is something of a "worst case" for PBNJ, since specifications are used in a very limited and unnatural manner. We observed that on average twice as much developer effort (estimated by the number of lines of code) is needed to employ specifications instead of stubs. Further, there was an average added delay of one second per test in execution times.

To investigate RQ2, we revisited the same benchmarks to see which unit tests can be enhanced, with reasonable additional effort, by taking advantage of the benefits of executable specifications that were identified during the exploratory study. The enhancement is measured in terms of increased test coverage, code reuse, and/or strength of properties tested. According to this metric, 54% of the unit tests in the applications can be enhanced by employing executable specifications. For the rest of the tests, stubs are sufficient and any additional effort to enhance the mocks with specifications was not justified. Finally, since the specifications for a mocked API are generally reusable across an entire test suite, we observed that the ported tests have on average the same number of lines of code as the original unit tests.

The paper is organized as follows. After providing background on executable specifications (Sec. 2), we introduce the idea of declarative mocking (Sec. 3). Secs. 4 and 5 respectively present our exploratory study and our experiment with Mockito-based tests from Google Code. Finally we discuss related work and conclude.

2. EXECUTABLE SPECIFICATIONS

Specifications have long been proposed as a means to express the intended semantics of a software component. Recent specification languages include JML [13] for Java and Spec# [14] for C#, which support method pre- and post-conditions as well as object invariants. These specifications are typically used during development to dynamically (by contract checking [15]) or statically (using program verification techniques [16]) check that the implementation behaves as intended.

More recently, it has been demonstrated how the latest constraint solving technologies can be utilized to declaratively *execute* specifications at run time as an alternative to imperative code. The basic idea is to execute code normally until reaching a specification that needs to be executed, which is represented as a predicate in some logic. At that point, a constraint solver is queried to find a program state that satisfies the specification, the program state is updated appropriately, and execution continues. This approach naturally allows executable specifications to be

```
class MockList implements List {
  Object[] elems, int size;
  spec int size() { return size; }
  pure boolean contains(Object o)
    ensures result <==>
      some int i : 0 .. size - 1 |
        elems[i].equals(o);
  void add(Object o)
    modifies fields
        MockList:elems, MockList:size
    ensures size == old(size) + 1
    && elems[old(size)] == o
    && all int i : 0 .. old(size) - 1 |
        elems[i] == old(elems[i]);
}
```

Figure 2: A runnable mock `List` class in PBNJ

mixed with imperative code in mainstream languages in a fine-grained manner. For example, the PBNJ [12] and SQUANDER [17] tools extend the syntax of Java with first-order relational logic specifications and use a propositional satisfiability (SAT) based relational solver called Kodkod [18] to execute specifications, while Kaplan [19] extends the Scala language with executable specifications that are represented in a subset of Scala itself and executed with a satisfiability modulo theories (SMT) solver. In this paper we illustrate declarative mocking with PBNJ, but the other systems described above could be used equally well.

As an example, the `MockList` class shown in Fig. 2 uses PBNJ specifications [12] to describe the intended semantics of its `contains` and `add` methods. As is common, postconditions are indicated with an `ensures` clause, and the keyword `result` represents the value returned by the method. The specification language supports a variety of expressive features, including integer and boolean primitives and associated operations; universal and existential quantification over arrays and other collections, via the `all` and `some` keywords; as well as set comprehensions. `spec` methods denote user-defined functions that may be invoked within specifications. Finally, the `old()` function is used to refer to the value of a variable or field on entry to the method.

The `MockList` specifications naturally capture the expected behavior of the two list operations. Furthermore, with PBNJ the result is a fully executable list implementation. For example, when the `add` method is invoked, PBNJ will encode the current program state (i.e., all in-scope program variables and fields, along with the objects reachable from them) in the language of the Kodkod constraint solver. PBNJ then asks Kodkod for a *model* of the declared postcondition — values for the various program variables and fields that make the postcondition valid. This model is used to update the actual dynamic state of the Java program (see Fig. 3).

To ensure a finite search space, Kodkod requires that each variable have a *bounded* number of possible values. The search space for a variable with an object type is the set of existing instances of that type in the current program state. However, the user may override this default with an `adds` clause stating the number of additional objects of a given type that can be instantiated, in order to satisfy the postcondition of the method.

All variables and fields mentioned in a specification are considered modifiable by default. However, PBNJ syntax allows limiting this space to serve two purposes. First, it
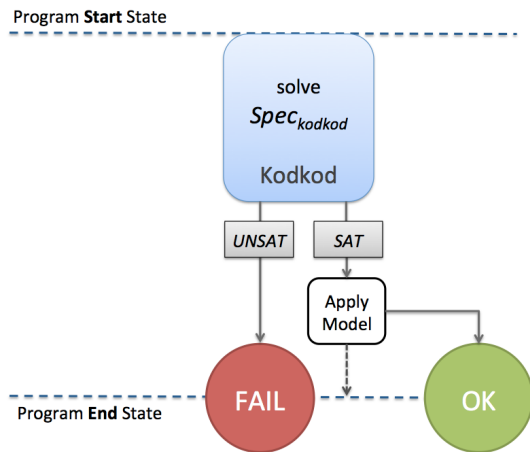
Figure 3: Declarative execution in PBNJ

is a convenient way to express *frame conditions*, which ensure that certain variables remain unchanged. For instance, the `add` method in our example may only modify the two fields declared in the `MockList` class, which is expressed using the `modifies fields` annotation. `contains`, however, may not have any side effects and gets a `pure` modifier, the equivalent of empty `modifies` clause. Second, reducing the search space can dramatically reduce the constraint solving time. Further details on PBNJ can be found in prior work [12].

3. DECLARATIVE MOCKING

Traditional mocking paradigms are inexpressive and do not allow reuse. Declarative mocking through executable specifications allows the developer to avoid having to specify the concrete outcome of every interaction with a mock object, and instead directly and declaratively express the mock object's important behaviors. The use of executable specifications additionally enables a new form of mocking, in which the data and/or environment needed for a test case is produced via specifications. In this section we describe declarative mocking of both functionality and data and also introduce extensions to PBNJ to support these tasks.

3.1 Mocking Functionality

Fig. 4 illustrates how executable specifications resolve the problems for mocking that we saw in Fig. 1. Because an instance of `MockList` from Fig. 2 already "knows" the intended behavior of a list's operations, there is no need for each individual test case to specify this information. Similarly, test cases can use ordinary `assert`s to directly ensure properties of interest, rather than the indirect and fragile approach to checking behaviors in terms of method invocation counts. We classify and quantify the benefits and limitations of declarative mocking versus stubs in our experimental studies of Secs. 4 and 5.

3.2 Mocking Data and Environment

Developers commonly want to test a feature in their application under various scenarios. For each scenario, they write initialization code to build up the state to the appropriate condition and then perform the tests. We observe that

```
class MySet implements Set {
  List elems;
  int size() { return elems.size(); }
  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }
  void testAdd() {
    List mockList = new MockList();
    MySet s = new MySet(mockList);
    s.add(0);
    s.add(0);
    // shouldn't add duplicates:
    assert (s.size() == 1);
  }
}
```

Figure 4: Testing a `Set` **implementation using the** PBNJ `MockList` **class from Fig. 2**

```
class MySet {
  List elems;
  void test1() {
    assume elems.size() > 0;
    // now run the test ...
  }
  void test2() {
    assume elems.contains(null);
    // now run the test ...
  }
}
```

Figure 5: Mocking the environment for test initialization

executable specifications can be naturally used to automatically modify program state to satisfy specified initialization conditions, relieving the tester of this burden. We call this approach *data mocking*, which is useful even for tests that do not rely on our declarative mock objects.

3.2.1 The `assume` Statement

To enable data mocking in PBNJ, we introduce a new statement of the form `assume <pred>`, where *pred* is a predicate on the current program state. When such a statement is encountered, PBNJ uses Kodkod to identify a program state satisfying *pred*, updates the program state, and continues execution. In the context of testing, the `assume` statement is useful both to synthesize the inputs to use in the test as well as to properly set the state of the mocked objects. We refer to the latter capability as *environment mocking*.

Consider again the `MySet` class with a mocked `List` object. In Fig. 5 we use environment mocking to easily set up two different test scenarios of interest: when the mocked list is non-empty and when it contains the `null` value as an element. When each test is run, PBNJ will nondeterministically find a state of the mocked object satisfying the given initialization condition.

3.2.2 The `unique` Modifier

In order to allow the developer to take full advantage of nondeterministic specifications and achieve higher coverage, we introduce an annotation for specifications called *unique*. This annotation can appear as a modifier on a method as well as on an `assume` statement. Consequently, every invocation of the associated specification on the same inputs will

choose a result not previously chosen, unless all possible solutions have already been produced. For example, when the postcondition `result * result == 9` is invoked for the first time, either solution `result = 3` or `result = -3` may be produced. Now, should the `unique` modifier be present, a second invocation (within the same process) will only return the solution not previously chosen. In this way, a tester can cycle through all possible scenarios satisfying a given initialization condition. We implemented this feature by leveraging the Kodkod solver's ability to (surprisingly efficiently) solve for all possible solutions within a given set of bounds.

4. EXPLORATORY STUDY

In this section we report on an exploratory study we performed in order to gain insights into the benefits and limitations of declarative mocking. We used PBnJ specifications to perform mocking on four open-source applications. This process allowed us to freely experiment without being bound to how mock objects are used today. Below we present the experiments and classify the advantages and disadvantages of declarative mocking that we discovered.

4.1 Applications

4.1.1 JStock—Mocking Webserver Data

JStock [20] is an open-source Java stock watchlist GUI application, which frequently queries a web page to display live quotes in a table. We augmented JStock's source code with executable specifications to mock data that is received from the webserver, allowing us to test the application without accessing the network or requiring any web service libraries.

4.1.2 JDBC—Mocking Database Behavior and Data

The Java Database Connectivity (JDBC) API [21] allows Java applications to interact with database management systems using simple method calls with strings of SQL statements as their parameters. We used PBnJ to create a functional, in-memory mock database meeting this API.

4.1.3 TFTP—Mocking Errors and Network Nondeterminism in a Client-Server Protocol

TFTP [22] is a simple protocol for transferring files between a client and a server. We created a mock server object in order to test an implementation of the client. Writing an imperative mock server that responds appropriately to client messages is relatively straightforward. However, by running the mock server locally, we lose the nondeterministic behavior that may occur due to dropped or misordered packets over the network. Our goal was to explore the use of specifications to express this network nondeterminism in a declarative manner.

4.1.4 Hadoop—Mocking Cloud Behavior and Environment

Hadoop [23] is an open-source framework for processing MapReduce jobs in the cloud. Testing a MapReduce application is a challenging task. Using cloud resources is often not a practical option for development and testing. Moreover, the performance of a MapReduce job is greatly influenced by numerous execution factors. These include the resources dedicated to the job, the workload, as well as scheduling policies.

```
class Table ensures uniqueRows() {
  String primaryKey;
  List<String> columns, List<Tuple> rows;
  spec boolean uniqueRows() {
    int primaryIdx =
      columns.indexOf(primaryKey);
    return
      all int i : 0 .. rows.size() - 1 |
        all int j : 0 .. rows.size() - 1 |
          (i != j ==>
            rows.get(i).get(primaryIdx) !=
              rows.get(j).get(primaryIdx));
  }
}
class Tuple extends ArrayList<Literal> { }
```

Figure 6: Partial invariants of a JDBC `Table` object

MapReduce simulators have been built (e.g. [24]) to help developers simulate their applications locally. To utilize the simulators, the users are required to provide cluster and workload trace information from real previous executions on the cloud. This data is not always available and tedious to produce synthetically [25]. Our first goal was to use data mocking to produce realistic trace information for input to such simulators. Secondly, we employed specifications to mock Hadoop's standard FIFO and fair (HFS) schedulers (see [26]), which these simulators rely upon. Our goal here was to experiment with the usage of specifications for rapid prototyping and design experimentation.

4.2 Advantages and Disadvantages

We now present the results of our exploratory study in terms of a classification of the advantages and disadvantages of declarative mocking. On each point, we compare our proposed approach with both traditional stub-based mocking and mocking by simply writing imperative code.

4.2.1 Advantages

Data Integrity: Objects often come with implicit integrity constraints that should be always satisfied. One of the benefits of declarative mocking is that these integrity constraints can be stated once and for all (as object invariants), and the runtime guarantees that any mocked behavior or data always conforms to these constraints. On the other hand, if done manually, it is easy for the tester to accidentally set up a program state that does not in fact conform to the necessary integrity constraints.

Example1. The in-memory JDBC-style database mock leverages the ability to express integrity constraints. Fig. 6 shows the representation of a database `Table` in the mock `Jdbc` objects, with `Literal` representing literal values storable in a cell. Each `Table` object must satisfy the `uniqueRows()` specification to enforce the exclusion of rows with duplicate primary keys. Consequently, any mocked `Jdbc` database or operation automatically ensures this property is preserved.

Declarative Expression: The expressiveness of specifications depends upon the flexibility of the employed solver. In performing these experiments, we found PBnJ's specification syntax to be adequate for concise and declarative expression of user intentions.

```
class DatabaseGUI {
  Jdbc jdbc, ResultSet results;
  void buttonTest1() {
    assume databaseInit()
      && results.size() == 0;
    // now test button behavior
  }
  void buttonTest2() {
    assume databaseInit()
      && results.size() == 2;
    // now test button behavior
  }
  spec boolean databaseInit() {
    String dbID = "shop";
    String tableID = "inventory";
    BExpr where =
      new CmpExpr(EQ, "price", 0);
    Database db = jdbc.databases.get(dbId);
    Table table = db.tables.get(tableId);
    return jdbc != null
      && jdbc.databases.containsKey(dbId)
      && db.tables.containsKey(tableId)
      && table.columns.contains("price")
      && table.select(where, results);
  }
}
```

Figure 7: Declaratively initializing tests

Example2. Testing various functions of a JDBC GUI client requires a database initialized in a particular way. A tester would be interested in the behavior of a GUI under various database conditions, e.g.:

> "Does button1 work properly assuming we are connected to a database named `shop`, which has a table named `inventory`, for which when I run the query `select * from inventory where price = 0`, I get no results? What about when I get two results?"

In Fig. 7 we show a PBNJ translation of the above scenario. The `select` method is a *spec* method that selects database rows based on a given query. Declarative specifications naturally and directly capture the programmer's intent.

Underspecification: Declarative mocking is useful when the programmer does not want to implement all aspects of the functionality being mocked. Specifications naturally support this through *underspecification*. For example, if the only relevant fact about a method is that its return value is always a positive integer, this can be stated directly as a specification, and PBNJ will nondeterministically choose a value to return at run time.

Example3. To achieve the sample database initialization requirements given above without declarative mocks, the tester must manually bring the state of the mock database to the desired condition. This process involves choosing concrete values for all aspects of the database state, for example selecting exactly which two results should be returned to the given test query. In contrast, with specifications the tester does not have to specify any details about the database state other than the high-level requirements described earlier.

Nondeterminism: Similarly, nondeterministic behavior can be expressed more naturally in specifications than in manual code or stub.

```
spec boolean
  isWellFormedErrorInducingMsg(Msg m) {
  return !isNonErrorInducingMsg(m)
    && isWellFormedMsg(m);
}
spec boolean
  isNonErrorInducingMsg(Msg m) { ... }
spec boolean isWellFormedMsg(Msg m) { ... }
```

Figure 8: Composing specs in TFTP

Example4. In TFTP, to test that the client side properly handles all possible scenarios of received messages, we wrote a declarative mock server and specified conditions for various types of messages it may send out to the client. The nondeterminism in the messages sent by the mock server, due to network conditions and/or server errors, was naturally captured as a logical disjunction of conditions, each specifying one legal type of message to send.

Compositionality: Logical specifications *compose* effortlessly, which allows relatively complex requirements to be expressed by composition of several simpler conditions. On the other hand, mocks implemented as imperative code do not easily compose. This was demonstrated by the TFTP application.

Example5. One class of messages that the TFTP server can send are well-formed-error-inducing (WFEI) messages. These messages could be sent due to a buggy server implementation and are used to test the error-handling behavior in the client. As seen in Fig. 8, we were able to generate messages that are WFEI simply by composing predicates for well-formed and non-error-inducing messages, whose specifications are relatively straightforward and were gleaned from the TFTP specification document [22].

We cannot in general compose two different stubs in order to produce a stub that has the desired characteristics. Nor does there appear to be a natural way to employ composition in this way using imperative code. For example, both the well-formed and non-error-inducing conditions impose constraints on the block number of a given message. Therefore, the programmer must manually deduce the range of block numbers that satisfies all constraints and then implement a method that produces block numbers in that range.

Reconfigurability of Data: Software testing often involves testing under numerous representative scenarios. One scenario may be different from another in a conceptually simple way, easily expressed by tweaking or composing logical conditions. Yet there may not be a simple way to tweak a stub representing one condition to obtain a stub representing the other. Similarly, the imperative code to produce each one may be very different.

Example6. To enable mocking of job trace and cluster data for the Hadoop application, we specified the general hierarchy of a data center cluster as well as the structure of a MapReduce job. Once the class hierarchies, integrity constraints, and specification methods were declared, we could readily produce any number of very different workload scenarios for units test with just a tweak of a few lines of spec-

ifications. When testing our Hadoop mock schedulers this proved very useful, as it enabled us to quickly produce a range of scenarios to run on.

Extensibility and Reusability: Declarative mocking is useful when the mock object is itself subject to frequent modifications or experimentation. For example, with specifications the programmer can naturally start with a bare-bones mock whose specifications are very weak and then incrementally strengthen the specifications based on the requirements of the client code under test. This kind of evolutionary process is much less natural with stubs or imperative code, since conceptually small updates to the mock's intended behavior can easily translate into tedious and sizable modifications to examples or an implementation.

Example7. We used the extensibility of specifications to our advantage in implementing the Hadoop schedulers declaratively. We first wrote the general task assignment policies (e.g. no reduce jobs can be assigned unless all map jobs are complete) in the superclass's method called `MockScheduler.assignTasksSpec()`. Moving to a stronger policy of FIFO, we added FIFO-specific policies for the subclass `MockScheduler_FIFO` and used the conjunction `super.assignTasksSpec() && assignTasksSpec_FIFO()` as the functional mock of the FIFO scheduler.

4.2.2 Disadvantages of Specifications in General

Effort vs. Stubs: Software engineers find stub-based mock objects appealing precisely because of the very low overhead to employ them. Clearly, using logic to describe the general functionality of mocks can require substantially more developer effort.

Effort vs. Imperative Code: For small and/or simple mocks, many of the benefits of using specifications can be achieved using ordinary imperative Java code instead. For example, after writing the in-memory JDBC mock both entirely in specs and entirely in code, we realized that the SQL operations are not complex enough to justify the use of executable specifications for the purpose of mocking the functionality of a database (while they remain very useful for mocking the *data* in a database).

On the flip side, some simple algorithms like sorting on arrays and insertion into red-black trees can be succinctly specified but can be much more onerous to implement due to low-level details and subtle corner cases [12]. Even our simple `MockList` example has a subtle issue when implemented imperatively: when adding an element to the list, regular Java code must check that the `elems` array has space for the new element and must allocate a bigger array if not. In contrast, the specification handles this issue implicitly.

Specifications Are Error-Prone Too: Just as in imperative code, logical specifications can be error-prone and hard to debug. Stubs are typically simple input-output pairs and so more straightforward to implement.

Efficiency and Scalability: In general constraint solving is severely limited in its efficiency and scalability versus imperative code. However, state-of-art solvers are constantly improving, and PBNJ's frame annotations help a lot in making the approach practical. In our experiments, we observed a slowdown of seconds and in a few times minutes per test. Nevertheless, during development and testing, developers may well be willing to trade off some performance for the software engineering benefits of declarative mocking illustrated here.

4.2.3 Limitations Specific to PBNJ

We encountered a few problems while performing these experiments that are not inherent to executable specifications, but are rather a result of limitations in the current implementation of the PBNJ tool.

We employ Kodkod, a bounded solver that directly encodes constraints into SAT, and which supports integers as the only primitive type. Constraints involving other kinds of primitive data such as reals, floats, and strings are therefore not directly supported. In our current implementation of PBNJ we only support equality constraints on primitives other than integers. Moreover, SAT-based constraint solving can become prohibitively slow when too many integer variables are involved due to the large search space. Therefore we ran the constraints assuming only 8-bit integers. Both of the above limitations can be partly avoided by employing a satisfiability modulo theories (SMT) solver such as Z3 [27], which employs specialized constraint solvers for arithmetic and other logical theories. Finally, PBNJ currently does not support constraints over data of a nested generic type (e.g. `List<List<Integer>>`), and some refactoring of code was necessary to work around this limitation.

5. EVALUATION

Based on our exploratory study discussed in the previous section, it is clear that there are real software engineering benefits to using declarative specifications for mocking, but there are also important limitations and costs to consider. To investigate the research questions posed in Sec. 1, we ported the unit tests of six existing applications from Mockito to PBNJ.

5.1 Selection Criteria

We searched Google's open-source code repository `code.google.com` for Java applications that employ Google's Mockito library as part of their unit tests. We selected the first 6 applications in the search results whose purpose was fairly clear to us based on the descriptions, and for which we were able to gain a fair amount of understanding about the tests and the mocked components in a relatively short amount of time. We only examined unit tests that employed Mockito. We excluded tests that rely on the mock object throwing an exception, since our tool currently lacks support for specifications about exceptions. This limitation excluded 5% of the tests that use Mockito.

5.2 Strategy

We applied a two-phase evaluation strategy on each benchmark, which respectively address our two research questions RQ1 and RQ2 posed in the introduction:

Phase A: To learn about the overhead of using specifications and constraint solving, we first replicated each unit test by replacing existing mocks with declarative mocks that behave exactly as the developer's Mockito stubs. We wrote a separate mock class with associated PBNJ specifications mimicking the stubs for each individual test.

For example, if the original stub appeared as

```
when(mockList.contains(0)).thenReturn(false)
```

then we would create a mock `List` class with the method

```
boolean contains(int x) ensures x == 0 ==> !result;
```

We compared the two versions in terms of programmer effort, expressiveness, and running time for each test.

Phase B: In the second phase, we evaluate whether access to declarative mocking could have enhanced the unit tests in terms of strength of properties tested, test coverage, and reusability, with a justifiable amount of additional effort. We consider both mocked functionality and mocked data for test initialization. For instance, in our `List` example above we would generalize the specification as follows:

```
      contains(int x) ensures result <==>
   some int i :  0 ..  size − 1 | elems[i] == x;
```

Unlike the first phase, here we generalized and reused a single mock class and its associated PBNJ specifications across multiple tests, as this is the natural style to use with declarative mocking.

5.3 Benchmarks

We examined a total of 114 unit tests among 6 benchmarks, briefly described below.

j2bugzilla is an API for interacting with a Bugzilla bug repository within Java. Unit tests mock the `BugzillaConnector` class, which uses an Apache XML RPC library to access a given Bugzilla repository.

jscep is the implementation of the Simple Certificate Enrollment Protocol (SCEP) in Java. To avoid having to test with the real objects, the unit tests mock both the certificate (an X.509 certificate in the `java.security` package), as well as the `CertificateCertifier`, the interface for verifying the identity of a given certificate.

tjays1-project1 is a personal code repository with a collection of small applications, which all employ Mockito stubs in their unit tests.

gcm-server is the server side implementation of cloud messaging service for Android. Google developers use a mock of the `Sender` class, responsible for sending messages over an HTTP connection, to test any number of possible scenarios.

shivaminesweeper is a servlet-based Minesweeper game running in the browser. Unit tests mock the HTTP connection and stub the requested parameters to verify the functionality of the game implementation based on user events.

birthdefectstracker is a web-based application to query and manipulate a database of medical records. The most notable use of mock objects is that of various Data Access Objects (DAOs). Unit tests are generally used to test proper behavior of DAO controllers, for example that an error is signaled when a username is entered that already exists in the database.

5.4 Phase A Results

In phase A all unit tests were successfully refactored to replace stubs with specs that retain their exact behavior. Table 1 summarizes the results for both phases of the evaluation[1]. As a rough measure of developer "effort" required to replace stubs with specifications, we compared the number

---

[1]Solving on a Core i7-3930K, 3.20GHz, with 8-bit integers.

of lines between the stub and spec versions. The third column of Table 1 shows that on average there was a 2:1 lines-of-code ratio between specs and stubs, respectively. The average slowdown due to constraint solving was one second per test. Based on this data, there is considerable effort overhead for the tester to employ specifications merely as stubs, which is to be expected since input-output stubs can make very little use of the benefits of specifications.

The functionality of Mockito's `verify()` (which tracks invocation counts for a stubbed method) cannot be directly replicated using specifications. We replicated this task indirectly by declaring auxiliary counter fields and adding additional assertions in the postconditions to increment these counters on each invocation. In many cases there was a more direct property to be checked which would obviate the need for simulating `verify()`; phase B below explores that possibility.

5.5 Phase B Results

In phase B we revisited each test to examine whether the specifications from the previous phase could be generalized to take advantage of the benefits of declarative mocking illustrated in Sec. 4. Below we sample some of the positive and negative scenarios that we encountered.

5.5.1 When Specifications Were Useful

(D) Data Mocking, Data Integrity: In *shivaminesweeper* there are many implicit relationships among objects representing various aspects of the game, such as the dimensions of the board, the mine count for each cell on the board, etc. Many tests set up the game board manually by constructing a specific game state. We instead specified the relationships between various objects using object invariants. This simplified the task of test initialization. For example, once the dimensions for the board are provided, the invariants automatically determine the appropriate number of mines to include and place them on the board nondeterministically. This use of invariants prevents the creation of inconsistent states, which are easy to accidentally introduce when initializing state manually.

(R) Reusability: In *birthdefectstracker* we removed the existing stubs and reused the database specifications from our JDBC mock from the exploratory study to generalize each test. We used data mocking to initialize various snapshots of each database declaratively, and recycled the initialization conditions from one test to another to reduce effort.

(U) Underspecification: One of the applications in *tjays1-project1* is an implementation of an elevator unit, where tests verify that the implementation chooses the right floor to stop at next. An object keeping a priority set of floors that have requested service is mocked. Instead of hardcoding a set of floors as done in the original stubs, we request an underspecified set of floors, and employ the `unique` modifier to test a variety of context within a single test, with only minimal modifications to the original unit tests. Here, specifications increase the coverage of each test while requiring the same amount of developer effort.

(N) Nondeterminism: Several tests in *gcm-server* check that the `send` method properly retries message sends when-

```
Sender sender = Mock(Sender.class);
Result message = new Result();
doReturn(null) // fails 1st time
  .doReturn(null) // fails 2nd time
    .doReturn(result) // succeeds 3rd time
      .when(sender).send(message, "1");
```

Figure 9: Use of stubs in *gcm* for simulating a scenario that includes failures and success

```
class MockSender extends Sender {
  spec int sendCount;
  Result send(Message msg, String id)
    ensures sendCount == old(sendCount) + 1
      // up to 4 times
      // ok/fail nondeterministic:
      && (result != null
         || old(sendCount) < 5);
}
```

Figure 10: Specifications generalize Fig. 9 scenario.

ever they get dropped. Fig. 9 shows how Google developers use Mockito's cascaded stub feature to test particular scenarios involving dropped messages. Specifications express this nondeterminism naturally, as a disjunction of possible outcomes, as illustrated in Fig. 10. We added the `unique` modifier to generalize tests such as this to cover any number of possible outcomes, making several other existing tests redundant. Writing these specs does not require much more effort than the stubs in Fig. 9.

5.5.2 When Specifications Were Not Useful

When Stub Is Irrelevant to the Test: In the *jscep* benchmark, as shown by the example in Fig. 11, unit tests use stubs to check that the certificate certifier is properly invoked by the client code under various circumstances. This represents a case where specifications do not enhance this test in any way, as there is no relation between the property being tested (the certifier has been properly invoked) and the logic of the stubbed components (the certifier's behavior).

When Mocking Functionality Is Simple with Code: As we discovered in the exploratory study (in mocking SQL operations) sometimes mocking an object's behavior is straightforward using imperative code and the overheads of using specifications and constraint solving are not justifiable.

```
void testHandlerForCertificate() {
  certifier = mock(Certifier.class);
  cert = mock(X509Certificate.class);
  when(certifier.certify(cert))
    .thenReturn(true);
  // perform handler test here...
  // verify certifier's certify method
  // was invoked:
  verify(certifier).certify(cert);
}
```

Figure 11: *jscep* example where specs not useful

5.5.3 Results

The last 7 columns in Table 1 report the result of the second phase of the evaluation. We state the percentage of examined unit tests for each benchmark that were enhanced by declarative data mocking and data integrity (D), reuse and reconfiguration (R), and nondeterminism and underspecification (NU). Clearly, many tests belong to multiple categories, and some of properties we mentioned in Sec. 4 were left out due to being difficult to accurately quantify. We dub tests that exhibit at least one of these properties as "enhanced." The next column indicates that 54% of all unit tests were able to be enhanced in this way.

In performing this experiment and studying the results, we observed a general pattern. Among unit tests where there was a strong relationship between the logic of the unit test and the stubbed component, declarative mocking of functionality was typically beneficial. On the other hand, when mocks were simply there to enable running of the tests, with no direct relation to the properties being tested, stubs were sufficient and specifications were not worth the effort. Declarative data mocking, on the other hand, was typically beneficial any time there existed complex test initialization data.

The third-to-last column compares the lines of code as a rough measure of developer "effort," among those tests that benefited from declarative mocking. Because the specifications were generally reusable across the tests for a given application, this ratio dropped to 1:1 on average. Thus employing specifications can produce their many benefits for the purpose of mocking, while requiring comparable amount of developer efforts over a test suite when compared to traditional approaches.

The last two columns report on constraint solving times by Kodkod. The average solving time in Phase B was 34 seconds per test. As we mentioned, this solver works by direct translation to SAT and constraints involving a lot of integer arithmetic can take a long time to solve. This was the case in both *shivaminesweeper* and *birthdefectstracker*, where specifications involved arithmetic constraints over the elements of a multidimensional array. The current PBNJ tool is not optimized for these situations.

6. RELATED WORK

Declarative mocking is related to several strands of prior research.

6.1 Mock Objects

Several libraries are designed to allow testers to produce simple mock objects, including Mockito [6], Mockrunner [5], and Microsoft Moles [28]. These frameworks make traditional stub-based mock objects easier to create, while our work focuses on making mock objects more expressive and declarative. Ostermann incorporates nondeterministic choice to make mock objects more expressive [29]; declarative mocking naturally supports nondeterminism as well as additional expressiveness.

Saff *et al.* [30] propose an approach to automatically create mock objects for the purpose of test factoring by capturing the interactions between a component and its environment on a set of system-wide tests. This approach requires that the full system be available initially. Similarly, Qi *et al.*'s method [31] creates environment models based on execution traces, so it also requires a fully executable version of

Table 1: Benchmark data

		Phase A						Phase B			
Application	#Tests with stubs	Spec/Stub LoC ratio	Avg time	Worst time	% (D)	% (R)	% (NU)	%Tests enhanced	Spec/Stub LoC ratio	Avg time	Worst time
j2bugzilla	13	1.4	4 sec.	10 sec.	77	85	69	85	0.4	12 sec.	95 sec.
jscep	4	2.6	0	0	0	0	0	0	–	–	–
tjays1-project1	18	1.8	1	2	44	44	39	44	0.8	1	2
gcm-server	23	0.9	1	2	22	30	30	30	1.0	2	3
shivaminesweeper	15	1.8	1	2	93	93	93	93	0.7	52	64
birthdefectstracker	41	2.8	0	1	73	73	66	73	1.9	104	335

the program including of the mocked environment. Our approach does not have this limitation and allows more control over what properties of the environment to mock, but it requires explicit specifications.

In prorogued programming [32], the system interactively asks the user to supply an appropriate return value upon a call to the method. The supplied values are recorded for later use, which has the effect of incrementally building up an appropriate mock for the method.

Henkel *et al.*'s approach [33] is conceptually similar to ours but uses term rewriting on specifications rather than constraint solving for producing mocks. Their approach relies on heuristics to guide the rewriting, which can miss solutions and/or lead to infinite search, while our specifications are more general, and soundness and completeness are guaranteed, up to the search bounds. Wilmore [34] proposes an automatic database state preparation approach for test initialization of database applications via intensional specifications as constrained queries. This work can be thought of as an instance of declarative mocking of data, and our approach can handle it, as evidenced by our JDBC and MapReduce examples.

As mentioned in Sec. 1, declarative mocking uses similar technology to prior work on automated test generation, but with distinct goals. Closest to our work is prior research that automatically produces mock objects for use with generated tests. For instance, Galler *et al.* [7] generate test inputs by automatically extracting mock object stubs that satisfy user-specified preconditions. However, these mock objects are limited in expressiveness; for example, the values returned from mocked methods are determined statically and may not depend on the inputs or state of the object under test. Our approach solves constraints dynamically and so does not suffer from these limitations.

6.2 Declarative Execution

The idea of employing a *mixed interpreter* for mock objects was mentioned by Rayside *et al.* [35], yet the idea was not investigated concretely.

We use PBnJ [12] to enable declarative execution. Other recent declarative execution systems include Squander [17] for Java and Kaplan [19] for Scala. Squander also uses Kodkod for finding models. In PBnJ the specifications are expressed over concrete Java variables in the program. Squander, on the other hand supports abstract, logical variables that are used for specification purposes only. Abstraction and concretization functions can be then provided to relate the concrete and logical states of a given program. Kaplan utilizes the state-of-art SMT solver Z3 [27] for constraint solving. As we discussed, Kodkod has a clear efficiency disadvantage compared to SMT solvers for problems involving primitive values such as integers.

7. CONCLUSIONS

We have presented a new approach to creating mock objects. Programmers write high-level specifications for the methods in an API being mocked, and a constraint solver dynamically executes these specifications. As a result, code that depends on the API can be tested exactly as if it is invoking a "real" implementation of the API. Further, we show that executable specifications naturally support other testing tasks, in particular the initialization of state for both the object under test as well as the mocked objects.

Our implementation of declarative mocking for Java extends the PBnJ executable specifications tool, and we have used the implementation both to explore the potential capabilities of the approach as well as to directly compare with the usage of traditional mock objects on existing applications. Declarative mocking of behavior is most beneficial for unit tests where there is a strong relation between the logic of the unit test and the stubbed component, and declarative data mocking can often simplify initialization code and increase test coverage.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] Beck, *Test Driven Development: By Example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[3] T. Mackinnon, S. Freeman, and P. Craig, "Extreme programming examined," in *Extreme programming examined*, G. Succi and M. Marchesi, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ch. Endo-testing: unit testing with mock objects, pp. 287–301.

[4] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," in *OOPSLA '04*. New York, NY, USA: ACM, 2004, pp. 236–246.

[5] G. L. Alwin Ibba, Jeremy Whitlock, "Mockrunner," http://mockrunner.sourceforge.net.

[6] S. Faber, "Mockito: Simpler and better mocking," http://code.google.com/p/mockito.

[7] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from design by contract specification for test data generation," in *AST '10*. New York, NY, USA: ACM, 2010, pp. 43–50.

[8] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 365–368.

[9] S. Kong, N. Tillmann, and J. d. Halleux, "Automated testing of environment-dependent programs - a case study of modeling the file system for pex," in *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, ser. ITNG '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 758–762.

[10] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *AST'09*, 2009, pp. 149–153.

[11] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. de Halleux, "Environmental modeling for automated cloud application testing," *Software, IEEE*, vol. 29, no. 2, pp. 30 –35, march-april 2012.

[12] H. Samimi, E. D. Aung, and T. Millstein, "Falling back on executable specifications," in *Proceedings of the 24th European conference on Object-oriented programming*, ser. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 552–576.

[13] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: a behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[14] M. Barnett, R. M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *CASSIS '05*, ser. LNCS, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362. Springer-Verlag, 2005, pp. 49–69.

[15] B. Meyer, "Design by contract: Making object-oriented programs that work," in *TOOLS (25)*, 1997, p. 360.

[16] R. M. Leino, "Specifying and verifying software," in *ASE '07*. New York, NY, USA: ACM, 2007, pp. 2–2.

[17] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson, "Unifying execution of imperative and declarative code," in *ICSE '11*. New York, NY, USA: ACM, 2011, pp. 511–520.

[18] E. Torlak, "A constraint solver for software engineering: Finding models and cores of large relational specifications," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[19] A. S. Köksal, V. Kuncak, and P. Suter, "Constraints as control," in *POPL '12*. New York, NY, USA: ACM, 2012, pp. 151–164.

[20] JStock, "Jstock," http://jstock.sourceforge.net.

[21] JDBC, "Jdbc," oracle Corporation. http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc.

[22] K. R. Sollins, "The TFTP protocol (revision 2)," *Internet RFC 1350*, July 1992.

[23] Hadoop, "Apache hadoop," http://hadoop.apache.org.

[24] H. Tang, "Mumak: Map-reduce simulator," https://issues.apache.org/jira/browse/MAPREDUCE-728.

[25] G. Wang, A. R. Butt, H. Monti, and K. Gupta, "Towards synthesizing realistic workload traces for studying the hadoop ecosystem," in *MASCOTS '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 400–408.

[26] M. T. Jones, "Scheduling in hadoop," http://www.ibm.com/developerworks/linux/library/os-hadoop-scheduling/index.html.

[27] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *TACAS'08/ETAPS'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[28] J. de Halleux and N. Tillmann, "Moles: tool-assisted environment isolation with closures," ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 253–270.

[29] M. Achenbach and K. Ostermann, "Testing object-oriented programs using dynamic aspects and non-determinism," in *ETOOS '10*. New York, NY, USA: ACM, 2010, pp. 3:1–3:6.

[30] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *ASE '05*. New York, NY, USA: ACM, 2005, pp. 114–123.

[31] D. Qi, W. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury, "Modeling software execution environment," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 415–424.

[32] M. Afshari, E. T. Barr, and Z. Su, "Liberating the programmer with prorogued programming," in *Onward! '12*. New York, NY, USA: ACM, 2012, pp. 11–26.

[33] J. Henkel, C. Reichenbach, and A. Diwan, "Developing and debugging algebraic specifications for java classes," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 3, pp. 14:1–14:37, Jun. 2008.

[34] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications," in *ICSE '06*. New York, NY, USA: ACM, 2006, pp. 102–111.

[35] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson, "Agile specifications," in *OOPSLA '09*. New York, NY, USA: ACM, 2009, pp. 999–1006.