Bond: A Spy-based Testing and Mocking Library

Erik Krogen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-116
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-116.html

May 24, 2016

Acknowledgement

I would like to acknowledge Professor George Necula, whose ideas created the initial incarnation of Bond and whose advising over the past two years has been invaluable in guiding me. I would also like to thank a number of the employees at Conviva, as well as the students of CS169 at UC Berkeley in the Fall of 2015, for trying Bond and providing valuable feedback. My thanks also go out to the Thomas and Stacey Siebel Foundation for their fellowship during my graduate tenure which allowed me to continue to stay focused on my academic work. Finally, I would like to thank my parents, whose ambition, intelligence and support have motivated me throughout my life to set high goals and to do everything in my power to achieve them. Nothing I have become today would be possible without you both.

Bond: A Spy-based Testing and Mocking Library

by

Erik Tao Krogen

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor George Necula, Chair
Professor Koushik Sen

Spring 2016

Bond: A Spy-based Testing and Mocking Library

Abstract

Bond: A Spy-based Testing and Mocking Library

by

Erik Tao Krogen

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor George Necula, Chair

In today's world of software projects that are constantly increasing in complexity and number of contributors, proper testing is essential. Yet, developing tests is a tedious process that is often undervalued and overlooked; most existing research into this problem has been devoted to automated test generation, but manually written tests remain a crucial part of any complete test suite. In this work we introduce spy-based testing, a novel style of unit testing in which developers specify only the variables or objects they are interested in verifying, rather than the specific values they expect. The testing framework aids the developer by collecting values from live execution and verifying them through user interaction, then saving these values to detect future regressions. We have developed Bond, an implementation of spy-based testing in Python, Ruby, and Java, in an effort to provide an easier way for developers to quickly write and maintain unit tests. In addition to the core facets of spy-based testing, Bond contains an integrated mocking library, including record-replay mocking functionality that allows a mock to be created by recording live interactions and replaying stored values back at a later time. We present Bond and discuss its usage, and finally provide case studies demonstrating the potential for Bond to save a significant amount of programming effort when developing tests.

Contents

Acknowledgments

I would like to specifically acknowledge Professor George Necula, whose ideas created the initial incarnation of spy-based testing and whose advising throughout the past two years has been invaluable in guiding me towards this achievement. I would also like to thank a number of the employees at Conviva, as well as the students of CS169 at UC Berkeley in the Fall 2015 semester, for trying an initial implementation of Bond and providing valuable feedback. My deepest thanks also go out to the Thomas and Stacey Siebel Foundation for their generous fellowship during my graduate tenure which allowed me to continue to stay focused on my academic work. Finally, I would like to thank my parents, whose incredible drive, ambition, intelligence and support have motivated me throughout my life to set high goals and to do everything in my power to achieve them. Nothing I have become today would be possible without the two of you.

1 Introduction

Testing is an essential part of any software development cycle; especially as software systems continue to grow in complexity and the teams writing them continue to grow in size, well-maintained tests are crucial to the success of any project. However, testing can be highly cumbersome and time-consuming; we refer here to the oft quoted figure that more than 50% of all development efforts are devoted to writing tests [8] and to research from Microsoft claiming that the code for unit tests is often larger than the code of the system under test [13]. Writing tests can also lack some of the appeal of developing functionality, looked down upon as uninteresting by many developers. This is clear in the reluctance of some developers to write any tests at all; one study [3] has shown that a full 12% of software developers write no tests whatsoever, a staggeringly large number implying that a great deal of application logic currently goes untested.

The software engineering community has long been aware of these issues with testing, and a great deal of research has been conducted on how to make testing software a more seamless process. A majority of this research has been devoted to automatic test generation, but much of this work focuses only on detecting fatal failures (crashes, deadlock, infinite loops, etc.) within an application rather than asserting correctness (e.g. [2, 11, 7]). A test generation tool generally cannot produce appropriate test oracles without knowledge of the expected behavior of the program, typically supplied by the user via a model or specification of the program [12, 14]. Creating such specifications can be difficult, so although it is commonplace in some industries (e.g. security critical applications), in the general case automated tests comprise only a small subset of the overall testing, meaning that the technique must still be augmented by a great deal of manual testing [9, 10, 3].

In addition to the effort needed to write tests initially, tests require maintenance as the expected behavior of a software system changes over time. This effort can be very substantial; Berner et al. have claimed that maintenance of test code, data, and tooling tends to have a bigger impact on cost than the initial implementation of tests [1]. Systems exist to help automate the process of updating tests as program behavior changes (e.g. [4]), but a study by Daka & Fraser has found that in general, it is common for failing tests to be "fixed" via deletion rather than an updating of the test's logic and/or assertions [3]. This degrades the ability of a test suite to grow over time and detect possible regressions from future changes in behavior, one of the key features of unit testing. Additionally, as the strength of assertions in a test increases, the cost of maintaining the test increases—a higher number of assertions will require more values to be updated as the expected output changes, and assertions which are more restrictive will be more likely to change as program behavior changes. This suggests a fundamental tradeoff between the strength of assertions and the maintainability of the test, a tension which has not been adequately addressed by existing testing systems.

For this reason it is imperative to make it as easy as possible for developers to both write and maintain tests. The open source community has generated an enormous wealth of testing libraries, many of them focused on unit testing and many of these in turn based off of xUnit-style frameworks, but nearly all of them rely on the traditional method of manually programmed test oracles, typically via an assertion-style mechanism. We present here a new style of unit testing framework which we refer to as *spy-based testing*, and its implementation in the Bond testing framework. Though many of its driving principles are drawn from

conventional unit testing, we replace traditional assertions with "spy" calls which can help developers to rapidly write code to verify expected values, and to more easily maintain tests as expected behavior evolves.

Spy-based testing aims to ease the process of initially writing unit tests, as well as to reduce the tension between assertion strength and test maintainability. The key insight is to separate the *code* used in the test from the *data* used in the test. Rather than specifying the variable or object upon which an assertion is to be made *as well as* the expected value, a developer simply states that he is interested in the value of a given object or variable, drawing upon the insight that the variables which are being verified change less frequently than their expected values. We allow the test framework to manage the expected value rather than leaving it to live within the test code, collecting it during live execution and storing it separately. This saves the developer from ever needing to explicitly specify the value, and allows the framework to aid the developer in easily updating expected values when they change. Since the expected value lives in a separate framework-managed area as opposed to living alongside its corresponding variable in the testing code, it is also possible to make assertions on intermediate values at arbitrarily deep locations within the call stack.

Bond is an implementation of this style of testing, combined with a mocking library designed to operate hand-in-hand with spy-based testing. Of particular note is Bond's record-replay mocking functionality, which allows the collection of live values to be replayed back as a mock during subsequent test execution. This essentially does for mocking what basic spy-based testing does for assertions: it removes the test data from the test code and enables it to be both easily generated from live values, and updated as expected behavior changes. Bond currently exists as a working open source test library in the Python[1], Ruby[2], and Java[3] programming languages; source code and documentation are available for download at `https://github.com/necula01/bond`.

The rest of this paper is formatted as follows. Section 2 provides an overview of spy-based testing in general and how it can be used to ease the burden placed on the developer of tests. Section 3 describes Bond, our implementation of spy-based testing, and some of its specific details. Sections 4 and 5 discuss how to perform mocking using Bond, including a specific type of mocking functionality known as record-replay mocking which allows for the capturing of live values to be used as mock values in subsequent test executions. Section 6 presents three case studies examining the usefulness of Bond in real-world software projects. Finally, we examine related work in Section 7, present avenues for future work in Section 8, and discuss our conclusions in Section 9.

2 Spy-Based Testing

The primary goal of spy-based testing is to make unit tests as easy to write and maintain as possible, focusing specifically on achieving this by separating testing *data* from testing *code* in such a way that the data can be managed by a testing framework rather than by the developer. In a traditional assertion-based unit test, the expected values are intertwined

[1] `https://pypi.python.org/pypi/bond/`

[2] `https://rubygems.org/gems/bond-spy`

[3] `org.necula.bond` in the Maven Central Repository (`maven.org`)

directly into the code which exercises the system under test. This means that the developer must manually specify the expected value of a variable X each time it is involved as the output of code under test. Additionally, when the system is changed in such a way that the expected value of X changes, a code change will be necessary at every assertion which involves X. By separating the data from the code, these updates can all be performed outside of the code in a managed fashion by a testing framework.

Throughout this section code examples will be presented using the Ruby language and the popular RSpec[4] testing framework. Note that RSpec uses an assertion syntax which looks like "`expect(actual_value).to eq expected_value`" as opposed to the more traditional "`assert_equals(expected_value, actual_value)`" style syntax.

2.1 Overview

To achieve this, we replace traditional assertions with calls to `spy`. Each call to `spy` records its arguments, which are key-value pairs of named values, as an "observation". The sequence of all observations in a test comprises the test data for that test. A sequence of observations is saved on disk for each test; this is known as the sequence of reference observations. When a test finishes executing, the current sequence of observations (that is, the observations from the current test execution) is compared against the reference observations; if no differences are found, the test succeeds. This implicitly creates an assertion against each value which was passed into a call to `spy`, succeeding only if they are the same across subsequent test runs. To allow us to discuss this more formally, let us define the syntax of `spy` to be a function call containing key-value pairs $(k_1 \Rightarrow v_1, k_2 \Rightarrow v_2, \ldots, k_n \Rightarrow v_n)$ in which keys are strings and values can take on any type. An observation O is the set of key-value pairs contained within one call to `spy`. During the execution of a test, any number of calls to `spy` may be made, which are combined into a sequence $S = (O_1, O_2, \ldots, O_n)$. At the time the test is first executed, some $S^{curr} = (O_1^{curr}, O_2^{curr}, \ldots, O_n^{curr})$ is generated and potentially saved as the reference observation sequence, setting $S^{ref} \coloneqq S^{curr}$. Upon subsequent executions, S^{curr} is compared against the most recent version of S^{ref}; if $S^{ref} == S^{curr}$, the test succeeds.

If the values differ ($S^{ref} \neq S^{cur}$), the framework can be configured to have a number of different strategies to reconcile this differences, referred to as "reconciliation behaviors": (a) ABORT: The test can be immediately failed; this is useful when running tests in an automated fashion, e.g. in continuous integration testing; (b) ACCEPT: The current observations can be accepted as the new reference observations ($S^{ref} \coloneqq S^{cur}$), allowing the test to succeed and approving the current values as correct; this is dangerous and should be used with caution, but can be useful if the developer is confident that the changes in test data are expected; (c) INTERACTIVE: User input can be requested to determine what to do; this is the common scenario and allows the user to view the old and new values and decide which of the two is expected, or to accept some subset of the changes as expected. When no reference observations are available, the situation is the same as if reference observations were available but were blank; i.e. the same options listed above apply, and the current sequence of observations will be compared against an empty sequence ($S^{ref} = ()$).

[4] `http://rspec.info/`

2.2 Simple Usage Example

For example, a simple unit test checking the output of a function returning a scalar value might look like:

Traditional Assertions	Spy-Based
`out_var = function_under_test()` `expect(out_var).to eq 5`	`out_var = function_under_test()` `bond.spy(out_var: out_var)`

Traditionally a developer explicitly specifies the expected return value; in spy-based testing she simply specifies *which* variable she is interested in. In the situation above little is gained; however let us consider a more complex situation in which the test involves two different functions, one of which returns an object with numerous fields (using dot-notation to represent object containment):

Traditional Assertions	Spy-Based

`out_object = func_under_test1()`
`out_var = func_under_test2()`
`expect(out_object.field1).to eq 1`
`expect(out_object.field2).to eq 2`
⋮
`expect(out_object.fieldn).to eq "n"`
`expect(out_var).to eq "var"`
	`out_object = func_under_test1()`
	`out_var = func_under_test2()`
	`spy(complex object: out_object,`
	`out variable: out_var)`

In the case of spy-based testing, the entire object, as well as the secondary output variable, becomes an observation. This entire observation will be compared against the returned value in subsequent test executions, implicitly creating all of the manually-specified assertions seen in traditional unit testing. When the expected return value of `func_under_test1` changes, the developer must manually update up to n assertions in the traditional case; in spy-based testing, he will be presented with a simple choice (e.g. through a dialog window or command line prompt) as to whether or not the set of changes were expected and can complete the process in a single action. Note that in the traditional case it may also be possible to perform the object equality assertion in a single line such as `assert_equals(expected_object, ↙ ↪ out_object)`, but this still requires the developer to explicitly specify all of the fields of `expected_object` when it is created, still including expected values in the test code which must be updated as the code evolves.

The objects which comprise the set of observations are serialized into a separate file to be used for comparison in subsequent test executions; this reference observation file is distributed alongside test code, e.g. through a version control system. For example, a sample observation file serialized using JSON[5] for the test code above might look like:

[5] `http://www.json.org/`

```
[
  {
    "complex object": {
      "field1": 1,
      "field2": 2,
               ⋮
      "fieldn": "n"
    },
    "out variable": "var"
  }
]
```

Note that the entire observation is contained within a JSON array; multiple observations appear as multiple sequential entries within the array.

2.3 Inline Spying

We have seen how `spy` can be used to replace traditional assertions, but calls to `spy` can be placed within production code, and are configured in such a way that they have no effect when not running inside of a test framework. This allows spy-based testing to enable easy viewing of intermediate values and of the flow of execution in production code.

2.3.1 Intermediate Values

A call to `spy` placed within production code, when a test framework is active, will behave identically to a call to `spy` within test code (recording its arguments into the list of observations). This enables the inspection of values deep within the call stack without any outside effort or refactoring.

Consider the situation where a function is being tested which accepts some input object, inserts an entry into a database based off of that object, and returns a status code:

```
def insert_into_database(object)
  query = # processing logic here
  return submit_database_query(query)
end
```

A developer wishes to check that the query used to insert the entry is correct, but this is not returned to the caller of the function. We can use `spy` here to view this value:

```
def insert_into_database(object)
  query = # processing logic here
  bond.spy("insert into database", query: query)
  return submit_database_query(query)
end
```

Notice that we have used an extra parameter at the start of `spy`, which is a name for this observation. Naming observations, while acceptable in test code, is especially useful for inline calls to `spy`, since it is much easier to see where the observation originates from when it has an associated name.

Now, if we write a test as such:

```
object = "test string"
status_code = insert_into_database(object)
bond.spy(status_code: status_code)
```

The following JSON-serialized list of observations might be generated:

```
[
  {
    "__spy_point_name__": "insert into database",
    "query": "INSERT INTO table VALUES(\"test string\");"
  },
  {
    "status code": 1
  }
]
```

Thus, the test is able to verify that the query made to the database is correct, even without explicit test code. In this situation, we could also have refactored the method to have it return the query string and then have a separate method that calls out to the database. In general, however, this inline spy technique can be useful to avoid passing intermediate values up the call stack to be observed, especially if, for example, a method under test were to make multiple sequential calls to `insert_into_database`. Note that, if a developer does not wish to have calls to `spy` present in their production code, they could achieve the same results by injecting them through standard techniques such as dependency injection.

2.3.2 Execution Flow

Using the same mechanism as above, it can be useful to track the flow of execution through a program to ensure that it is as expected. Consider the scenario where a program makes a sequence of calls to an external web service, and that these calls must be made in a specific order. We start by placing a call to `spy` within the function which makes HTTP requests that observes, for example, the URL which is being requested:

```
def make_http_request(url, data)
  bond.spy("http request", url: url)
  # make the request
end
```

When test code exercises a portion of production code which makes HTTP requests, we might have the following JSON-serialized list of observations:

```
[
  {
    "__spy_point_name__": "http request",
    "url": "http://server.com/first_endpoint"
  },
  {
    "__spy_point_name__": "http request",
    "url": "http://server.com/second_endpoint"
  },
  {
    "__spy_point_name__": "http request",
    "url": "http://server.com/third_endpoint"
  }
]
```

By viewing the observation file, a developer can easily verify that requests are submitted in the proper order with the proper query parameters, and that no extraneous requests are made. The test will then verify this on each subsequent execution by ensuring the new list of observations is the same as the approved reference.

3 Bond Implementation

The Bond testing library is an implementation of the principles of spy-based testing discussed above, as well as a number of related features enabled through this style of testing. It is available in the Python, Ruby, and Java programming languages; see Table 1 for details on the number of lines of code used for each implementation. Only 2769 lines of code total were used for all three implementations; this is indicative of the generally simple nature of Bond, a property that we attempted to hold true to ensure that it would be easy to port Bond to as many languages as possible to increase its availability.

3.1 Spy Agents

On top of the basic principles of spy-based testing, Bond also provides the concept of an "agent". An agent can be "deployed" (i.e., registered as a handler) to a specific spy point name; whenever a call to `spy` is performed with this name, the agent is consulted to determine appropriate actions to perform. An agent may perform some side effect (e.g. modify some global test state when a spy point is called), throw an exception (e.g. to prevent some dangerous code path from being called during test execution, or to mock an error), or return some value (which can be used for mocking). Additionally, filters can be specified on an agent so that it only applies to some subset of the calls to `spy` with a matching spy point name; for example, a filter can be specified which restricts the agent to only apply to those calls which have the key "foo" whose value contains the substring "bar". If multiple agents are deployed which match the name of a spy point and satisfy all filtering criteria, the most recently deployed agent is used; this is referred to as the "active" agent.

Agents are useful primarily for inline calls to `spy`, especially when mocking functionality is desired. This will be discussed in further detail in Section 4.

	Lines of Code				
	Core Bond	Reconciliation	Spy Point Annotation	Total	Record-Replay
Python	340	437	70	847	N/A
Ruby	416	*	151	567*	136
Java	743	341	135	1219	N/A

Table 1: Non-whitespace, non-comment lines of code used in the implementation of each portion of Bond in each language. Core Bond represents the core functionality of spy-based testing. Reconciliation represents the functionality used for taking two observations, comparing them, and requesting and processing user input if necessary, as described in Section 2.1. Spy Point Annotation represents the functionality to be able to place a spy point directly on a function (see Section 4.2). Total represents the total lines of code, not including record-replay functionality. Record-Replay represents the functionality for record-replay mocking, which is only available in Ruby (see Section 5).
* The reconciliation system written in Python can be run as an executable; Ruby utilizes this, so there is no reconciliation system written in Ruby. The total figure presented here for Ruby does not include a reconciliation system.

3.2 API

The core functionality of Bond is exposed through the `spy` function, as discussed in the previous section. Bond's other main entry points, shown here in the Ruby syntax:

- `active?`: Returns true if Bond is currently in testing mode (i.e., the code is currently executing within a testing environment), otherwise false.
- `deploy_agent`: Used to deploy agents as discussed in Section 3.1.
- `spy_point`: Used to annotate a function in such a way that there is essentially an automatic call to `spy` whenever the annotated function is called; this is explained in further detail in Section 4.2.

The signatures for these functions are shown in full in Figure 1. The Python API is extremely similar, with only small differences to meet language convention (for example, `active` instead of `active?`). The Java API has significantly larger differences due to its statically typed nature; see below. Though Bond does export additional functions used, for example, to denote the start of a test and determine settings such as the directory where reference observation files should be saved, they are not relevant to this discussion; we direct the interested reader to the full Bond API[6].

Ruby and Python both allow for easy specification of key-value pair arguments, making the syntax for `spy` very simple. For Java, however, this is not possible. Instead, we use a builder pattern [6] to assemble observations, for example:

```
SpyResult<String> result = Bond.obs("key1", "value1")↙
 ↪ .obs("key2", someObject).spy("point name", String.class)
```

Calls to `obs` are chained together to build up an `Observation` object, which is then spied on using the final call to `spy`. Calls to `spy` can optionally specify an expected return type (`String`, in this case); the default is `Object` since this is the most general.

[6] `http://necula01.github.io/bond/api.html`

```
1. bond.active?
2. bond.spy(spy_point_name = nil, **observation)
3. bond.deploy_agent(spy_point_name, **options)
4. bond.spy_point(spy_point_name: nil, mock_only: false, ↙
 ↪ require_agent_result: false, spy_result: false)
```

Figure 1: The Ruby Bond API. The = and : in the parameter lists denote default values; the ** denotes that the function accepts a list of key-value pairs. Note that these calls do not require a developer-supplied object instance; the references to `bond` refer to a system-wide singleton instance. Further, note that this is not the full API, as discussed in Section 3.2.

Original Test Code

```
describe BST do
  # Automatically initializes Bond
  include_context :bond

  it 'should insert correctly' do
    tree = BST.new
    tree.insert(8)
    tree.insert(12)
    tree.insert(3)
    tree.insert(4)
    tree.insert(6)

    bond.spy(tree: tree)
  end
end
```

Updated Test Code

```
describe BST do
  # Automatically initializes Bond
  include_context :bond

  it 'should insert correctly' do
    tree = BST.new
    tree.insert(8)
    tree.insert(12)
    tree.insert(3)
    tree.insert(7)
    tree.insert(6)

    bond.spy(tree: tree)
  end
end
```

Figure 2: An example of test code and the corresponding observation file for a binary search tree test.

The same style of providing key-value pairs is used for specifying options to agents (as described in Section 3.1) in Python and Ruby; in Java we again use a builder pattern:

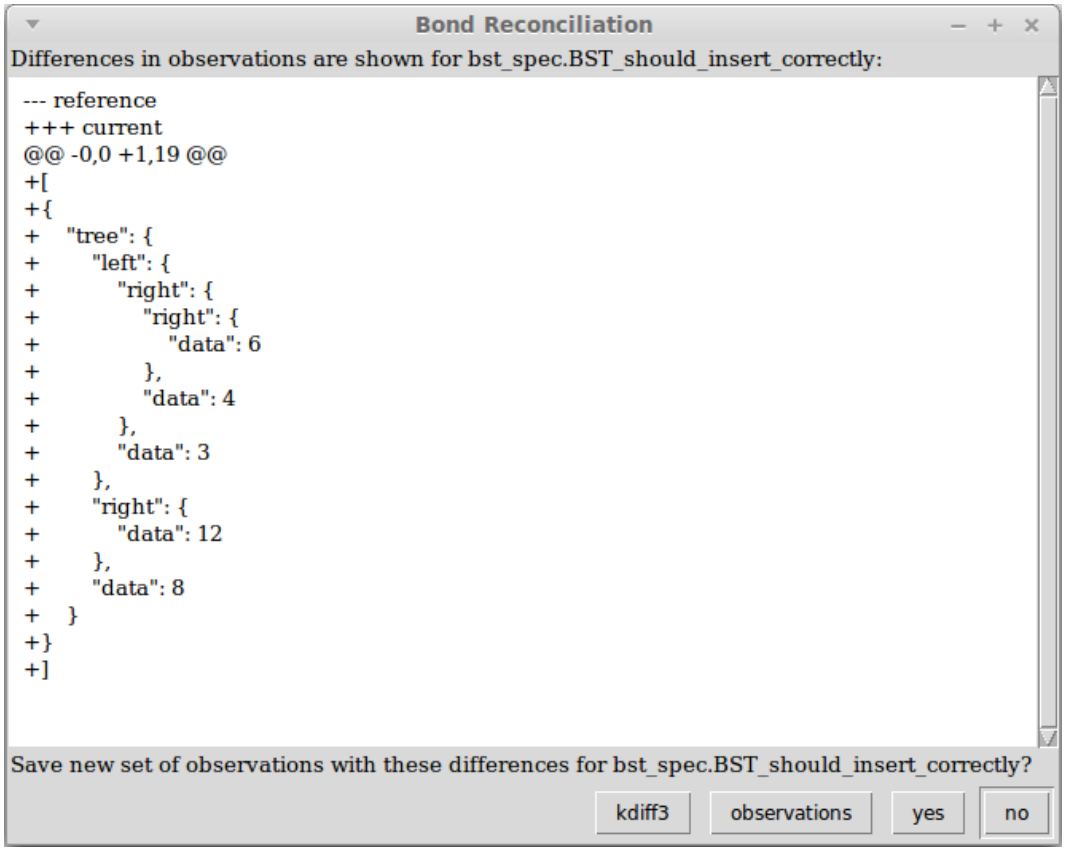
```
Bond.deployAgent("point name", ↙
 ↪new SpyAgent().withFilter(...).withResult(resultObject))
```

We first create a `SpyAgent`, then (optionally) specify various options via chained calls to `with`*`Option`*. The resulting `SpyAgent` is then passed to `deployAgent`. See Section 3.4.3 for more details and motivation behind the Java API.

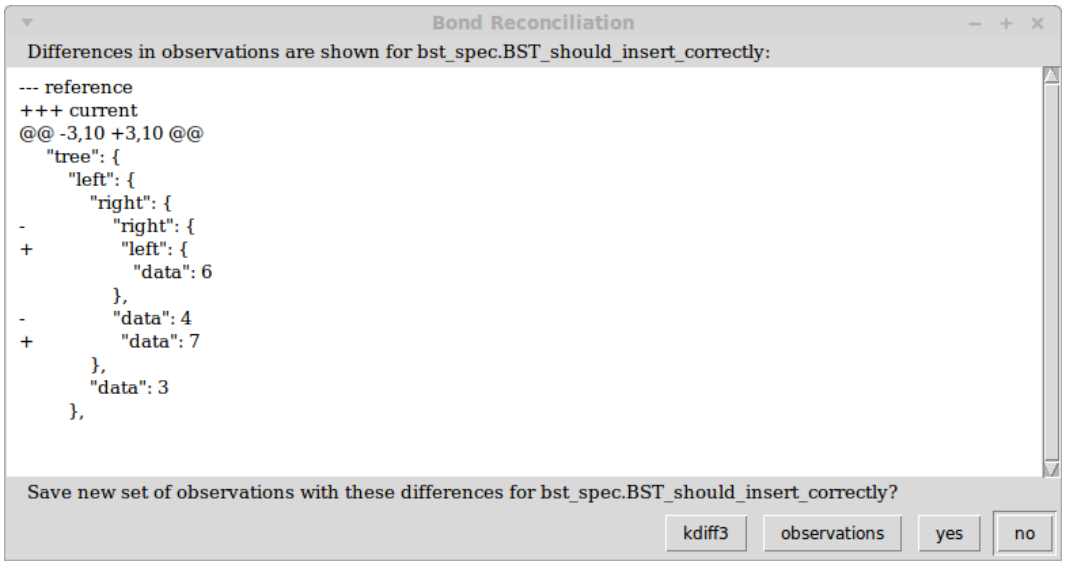
3.3 Usage Example

We present here a full usage example of how Bond would be used to write, and later update, a test for a binary search tree data structure. We again use Ruby with RSpec.

In Figure 2, we present sample testing code. The first time this code is run, the user would be presented with a dialog box ensuring that the observations generated are correct;



(a) Original observation reconciliation dialog.



(b) Observation reconciliation dialog after updating.

Figure 3: Reconciliation dialogs displayed to the user when writing a test for a binary search tree.

see Figure 3a. This assumes the user is currently in INTERACTIVE mode, which is suggested during development. Since there were no previous observations stored as a reference, the new observations are shown as a diff against an empty string, so every line is considered as added (+). The name of the test is shown, and the user is given the choice whether or not to save these observations as the new reference, view the full list of observations (rather than the diff), or to start `kdiff3`[7], a popular graphical utility for comparing and choosing between two sets of text. Let us assume that the user accepts these observations as the new reference.

Later, some change occurs. In Figure 2 we show updated test code in which we insert a 7 instead of 4 to emulate a change in behavior for the purposes of demonstration; in a real environment a change in behavior would normally come from within the production code itself. Upon rerunning the test, the observation from the call to `spy` would no longer match the reference observations, so the user would be shown the prompt in Figure 3b with all of the same options as before, but now only the specific differences displayed. Note that the beginning and end of the JSON serialization, which were shown in Figure 3a, are not displayed here because they have not changed. Upon choosing the "yes" option, Bond would automatically correct the reference observations so that in future executions the test would succeed.

3.4 Implementation Details

3.4.1 Singleton Design

All versions of Bond are designed with a single object to contain the current state; in Ruby and Python this is a singleton object, and in Java this is through the use of static members of the Bond class. This enables the developer to easily access Bond from arbitrary locations within a program, enabling the use of inline calls to `spy`. However, this has the unfortunate consequence that only one test may be running within a process at any given time, as multiple tests running concurrently would place their observations into the same sequence. We have decided that this is an acceptable tradeoff, especially since it is still possible to run multiple tests concurrently if they execute in separate processes (such that all of their program state is fully isolated).

3.4.2 JSON Serialization & String Comparison

All versions of Bond JSON-serialize the objects passed as values in observations before comparing or saving them. Upon completion of a test, the full text of the current JSON-serialized list of observations is compared against the full text of the JSON-serialized reference list of observations, which is saved on disk from previous test executions. If any difference is found between the two strings, a unified diff[8] showing lines which were added and removed in comparison to the reference observations is displayed to the user (assuming that they have not specified to automatically accept or reject changes as discussed in Section 2.1). Developers can specify custom serialization logic for custom objects (to, for example, exclude

[7] `http://kdiff3.sourceforge.net`

[8] `https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html`

unnecessary fields) and specify serialization options such as the precision of floating point values.

Converting all objects to a JSON string before saving or comparing simplifies the comparison process greatly as opposed to attempting to perform a diff between live objects, which would eventually need to be serialized to be saved on disk in any case, and enables us to leverage existing text diff tooling to perform the comparison. JSON is a well-known and very easily human readable format, which is important when the serialized observations are displayed to the user for them to decide whether or not the values are as expected. Another serialization format fitting for this purpose is YAML[9]; we have chosen to use JSON because it is considerably more popular than YAML, but there is no reason why YAML could not be used as well.

3.4.3 Java Implementation Difficulties

Due to the statically typed nature of Java, a number of difficulties were encountered while developing the Java version that did not arise in the dynamically typed languages, primarily related to providing a clean API with as much type safety as possible.

As discussed above, in Java it is not possible to easily pass arbitrary key-value arguments (without, for example, requiring the user to create and populate a Map data structure). We considered multiple solutions, including a version lacking in any type safety in which the method accepted a variable number of arguments of type `Object` which were meant to be keys at even indices and the corresponding values at odd indices. Ultimately we decided to use `obs` in a builder pattern, which provided a clean interface relatively low in verbosity and with reasonable type safety.

In Ruby and Python it is not necessary to specify the return type of a function, so we are free to return a few special constants for use by Bond (e.g. to indicate that no agent result was provided) from `spy`, as well as arbitrary results from agents. Unfortunately this is not the case in Java, so we instead create a `SpyResult` class which is aware of a few special return types, and can otherwise contain an arbitrary return value through the use of generics. The most generic form of `spy` returns simply an `Object`, but it is desirable to be more certain of the specific return type, so we also provide versions of `spy` which specify the expected return type. Bond will internally confirm that the return type of the result provided by the active agent, if any, matches the expected return type; if it does not an exception will be thrown. This is especially useful when mocking; see Section 4.

A similar issue arises when an agent attempts to throw checked exceptions. Though unchecked (i.e. runtime) exceptions can be thrown without special handling, Java enforces that a checked exception cannot be thrown inside of a method that does not declare it may throw that exception (unless a try-catch block is used to explicitly handle the exception). Thus, to provide a way for agents to throw checked exceptions, we also provide a form of `spy` that declares it will throw an exception, `spyWithException`. Similar to the regular version of `spy`, by default it declares that it throws a generic `Exception`, but an expected exception type can be specified to narrow the scope of what may be thrown, and Bond will perform type checking as with the regular version of `spy`.

[9] `http://yaml.org/`

All of the different forms of `spy` require a great deal of method overloading, the standard way in Java to provide methods with different parameter lists and default arguments. Unfortunately these methods must be duplicated on the `Observation` object that is returned as a result of a call to `obs` and on the static Bond class for spy points with no key-value pairs; luckily all of this happens transparently to the end user.

The statically compiled nature of Java also creates difficulty when providing mocking functionality; this will be discussed further in Section 4.

4 Mocking

Though numerous mocking libraries exist for all three of the languages for which Bond is implemented, we believe Bond's style of spy-based testing places it in a unique position to provide interesting features such as inline mocking (discussed below) and record-replay mocking (discussed in Section 5).

4.1 Inline Mocking

As discussed in Section 3.1, Bond's agents can be used to provide mocking functionality by using a deployed agent to return values or throw exceptions from a call to `spy`. If no active agent is present, or the active agent does not specify a return value, a special value (varying based on language) is returned to indicate that no mocking is active. This can be used to inject mock values at arbitrary locations within production code during testing, and to make control flow decisions based on whether or not the code is currently being tested.

We present here an example of this type of mocking using the Ruby API and the RSpec testing framework. Consider the scenario where a test calls some piece of code that uses a randomly generated number; during testing it may be desirable for this number to be constant to ensure deterministic behavior across test runs. One way to achieve this would be to check if the code is currently under test by calling `bond.active?`:

```
def func_under_test(arg)
  if bond.active?
    random = 42
  else
    random = Random.new.rand(10000)
  end
  # processing logic ...
end
```

`bond.active?` will only return true during testing, thus at all other times the randomly generated number will be used as expected. During testing, however, a deterministic result will be returned.

Consider a second scenario in which a test exercises some piece of code that calls out to an external web service, for example to retrieve the temperature. During testing, the developer does not want to contact an external service; rather, she wishes to supply a mock value. Figure 4 provides an example of how to achieve this.

The default return value from `spy` (in Ruby) is `:agent_result_none`; this will be returned anytime the system is not under test or an agent has not been deployed for the given spy

Production Code

```
1  def temperature_processor
2    request_url = make_temp_url
3    temp = bond.spy('spy_temp', ↙
       ↪ request_url: request_url)
4    if temp == :agent_result_none
5      if bond.active?
6        raise RuntimeError
7      end
8      temp = get_temp(request_url)
9    end
10   # processing logic ...
11 end
```

Test Code

```
it 'should process the temp' do                1
  bond.deploy_agent('spy_temp',                2
                    result: 72)                3
  res = temperature_processor                  4
  bond.spy(result: res)                        5
end                                            6
```

Figure 4: An example of production and test code for simple inline mocking.

point. So, to perform mocking, we check for this value (line 4) and if it was returned, we continue on to call the live service (line 8). Note the additional call to `bond.active?` (line 5); this bit of logic enforces that `request_url` will never be contacted during a test (perhaps the developer must pay for each request and wants to ensure that she is never billed during testing). If any other result is returned by an agent, as in the test code above (deployed in lines 2–3), then it is used instead.

4.2 Spy Point Annotations

While the method of mocking presented above is useful, it can be rather cumbersome, as it requires the developer to check the return value of `spy`, see if it is `:agent_result_none`, and then take action appropriately. To ease this process, we also provide function annotations (see point 4 of Figure 1 for an example in Ruby) which denote a function itself as a spy point.

Annotating a function with a `spy_point` annotation essentially injects a call to `spy` before any call to the function, using the full function name as the spy point name (which can be optionally overridden using the `spy_point_name` parameter). An implicit call to `spy` can also be injected *after* the function call containing the return value by specifying the `spy_result` parameter to be true. This has some utility even beyond mocking; for example, a developer might place a `spy_point` annotation on a function which is the entry point for database calls, enabling her to view all database queries in the test observations and to verify that they are as expected. However, the feature is primarily focused on mocking. If mocking is desired without viewing the mock call as part of the observation list, for example when mocking a random number generator that is called frequently, the `mock_only` parameter can be supplied to prevent these calls from being considered as part of the observation list.

The return value of the implicit call to `spy` is used to determine the next action; if an agent returned a value, that value is returned from the function call and the original function is never entered. If, however, no return value is supplied, the original function is called as normal (unless the `require_agent_result` parameter is specified and equal to true, in which

Production Code (Under Test)

```
1 def temperature_processor
2   request_url = make_temp_url
3   temp = get_temp(request_url)
4   # processing logic ...
5 end
```

Production Code (Dependency)

```
bond.spy_point(↙                                    1
 ↪ spy_point_name: 'spy_temp', ↙
 ↪ require_agent_result: true)
def get_temp(request_url)                           2
  # unmodified code                                 3
end                                                 4
```

Figure 5: An example of production and test code for simple inline mocking.

case an exception will be thrown). This enables conditional mocking at arbitrary depth within the call stack, in a more convenient manner than inline spy points.

We revisit the example from Figure 4, but with new production code as specified in Figure 5. `temperature_processor` becomes considerably simpler; it is no longer necessary for it to be aware of any mocking that is occurring. Instead, we annotate `get_temp`, marking it as a target for mocking by Bond. Notice that we have set `require_agent_result` to true; this achieves the same behavior as lines 5-7 in Figure 4, ensuring that the live service is never called during testing. Now, *any* call to `get_temp` from within the application will refuse to be called during testing, and can easily be mocked by deploying an agent as in Figure 4.

This is very useful when dealing with potentially dangerous live services. In a normal mocking scenario, it is up to the developer of a new test to realize that their test will call some live service and create a mock for it. With spy points, the developer who wrote the dangerous code can explicitly denote it as dangerous, ensuring that no careless developer will ever allow it to be run during a test.

4.3 Implementation Details

Due to the somewhat tricky nature of intercepting and augmenting function calls, each language required somewhat different implementation approaches. We present here only an outline of the techniques employed and direct the interested user to the source code[10]. The relevant portions are the `spy_point` function in Python, the `BondTargetable` module in Ruby, and the `BondMockPolicy` and `SpyPoint` classes in Java.

The basic idea is to create a "dummy" function which replaces the original function. This dummy function makes the first call to `spy` and collects the return value, then either returns that value or calls out to the original function if no result was found; essentially the dummy function "wraps" around the original function with some extra processing logic before and after; see Figure 6 for an overview of what this wrapper function looks like. Creating the dummy function is easy in Ruby and Python, as parameters can easily be passed through to the original function. Java is slightly more complex and requires reflection to pass on all of the original parameters, but the process is still straightforward.

The difficult part of this process is in replacing the original function with the dummy function. Python has an easy built-in way to do this using function decorators; the decorator need simply be placed immediately above the function definition and the process is taken

[10] `https://github.com/necula01/bond/`

```
# Developer-supplied code
class MyClass
  extend BondTargetable # enable mocking for this class

  bond.spy_point
  def method_to_mock(arg1); end
end

# Dynamically generated dummy function
def method_to_mock_wrapped(arg1_value)
  ret = bond.spy("MyClass#method_to_mock", arg1: arg1_value)
  if ret == :agent_result_none
    if require_agent_result
      raise "No result found" # required but not found; raise error
    else
      ret = method_to_mock(arg1_value)
    end
  end
  if spy_result
    bond.spy("MyClass#method_to_mock.result", result: ret)
  end
  return ret
end
```

Figure 6: A sketch of the wrapper function that is created when mocking, shown here in Ruby. This is not the actual full implementation, but gives a sense of the necessary logic. Note that this is generated by Bond, not the test developer.

care of internally. Ruby is slightly more complex, as it does not have a built-in way to achieve this functionality. We first include a class, `BondTargetable`, into whichever class or module defines the method we wish to spy on. Then, we hook into method addition for the class (through well-defined Ruby APIs). The call to `bond.spy_point` is actually only a method call which sets internal state within the class; this state is then consumed when the next method—the one immediately following the `bond.spy_point` call within the source code—is added. Within the method addition hook, Ruby's monkey patching functionality allows us to replace the new method with our own dummy method.

Java is by far the most difficult language to implement spy points within, as it is not a dynamically interpreted language and thus it is not natively possible to programatically modify classes. A language extension to Java, AspectJ[11], enables such features through what is referred to as "aspect-oriented programming", but we did not wish to restrict users of Bond to use such a system (which requires significant specialized functionality). Instead, we make use of the popular PowerMock mocking library[12], which integrates into standard Java and uses bytecode manipulation to modify classes only when in a testing environment—perfect for our needs. We place an annotation on the method to be spied, which in Java does nothing except to mark the method for other systems to be able to differentiate it, for example

[11] `http://www.eclipse.org/aspectj/`

[12] `https://github.com/jayway/powermock`

through reflection. PowerMock operates by intercepting requests to the Java classloader, so we instruct the PowerMock classloader to intercept any requests to retrieve classes for which there are annotated methods. We then specify that as those classes are loaded, the annotated methods should be replaced with our own dummy method. This approach is somewhat fragile, and likely would fail in the context of some other complex frameworks that make use of reflection; however, it has worked well so far in the authors' experience. Unfortunately the setup code required in each test class is somewhat cumbersome; we are continuing to investigate ways to increase the robustness of our approach and the ease of use.

5 Record-Replay

Bond offers a specialized form of mocking referred to as "record-replay" mocking. In record-replay mocking, instead of directly specifying return values for mock calls, the developer runs the system in RECORD mode and calls live code, saving the outputs that are generated. Then, in REPLAY mode, the saved outputs are used as mock values. While this idea is not novel, we believe we have made improvements over the state of the art because Bond's `spy` mechanism places it in a unique position to apply record-replay to arbitrary mocking; see Section 7.3 for further discussions.

In general, record-replay mocking is useful because it reduces the need for developers to directly specify mock results. If mocking is utilized heavily, the necessary code to set up the mocks can often comprise the bulk of the code for a test; record-replay alleviates this by automating the mock setup process and moving mock data out of the testing code. This is especially useful for functions which have complicated input parameters or return values. For example, when mocking HTTP requests with a standard mocking library it is necessary to place a large block of text (the HTTP response text) into the testing code itself, but this is highly undesirable.

5.1 Usage

Record-replay functionality adds only one additional function to Bond's public API (shown here in Ruby):

```
bond.deploy_record_replay_agent(spy_point_name,↙
 ↪ order_dependent: false, record_mode: false, **filters)
```

Similar to `deploy_agent`, this accepts the name of a spy point to which the agent should apply, as well as optional filters (as described in Section 3.1) and a toggle to enable or disable order dependence (discussed below). Switching between RECORD and REPLAY mode can be done on a test-wide level, but also can be overridden on a per-agent basis using the `record_mode` parameter. Note that `deploy_record_replay_agent` can only be applied to spy points corresponding to function annotations.

When a record-replay spy agent is encountered during test execution, the behavior taken depends on the mode of the test/agent and the reconciliation behavior (as discussed in Section 2.1). The cases are as follows:

- RECORD mode: Record the values generated by the live code
- REPLAY mode with a saved value: Play back the saved value
- REPLAY mode with no saved value: Varies based on reconciliation behavior
  - ACCEPT: Record values generated (emulate RECORD mode)
  - ABORT: Throw an exception
  - INTERACTIVE: Ask the user which of the two above options to perform

This allows the developer to never have to explicitly turn on RECORD mode, assuming they are in the INTERACTIVE reconciliation mode (which should generally be true during test development). The first time the test is run, by default the state will be REPLAY mode with no saved value, so in INTERACTIVE mode the developer can specify to continue with recording generated values. Subsequent executions will play back the saved value automatically, so the developer need only enable RECORD mode if she needs to re-record the values.

When in RECORD mode, after the live code has been called and the values have been recorded, they are displayed to the user for verification. At this point the user is free to edit the mock response before it is saved, or to reject it as incorrect. The ability to edit the response can be useful to, for example, simulate extraordinary conditions such as error states.

To determine which saved value applies to which spy point call, the arguments to the annotated function are saved along with the return value. On subsequent test executions, the current arguments are compared against the arguments for all saved values. When `order_dependent` is false, a saved value with matching arguments will be used multiple times if a function is called multiple times with the same arguments. When `order_dependent` is true, values will be saved alongside their call order, and will be played back in the same order. This can be useful to, for example, simulate interacting with a stateful web service in which multiple calls to the same endpoint will result in different return values.

5.2 Example

We consider a situation in which some application makes a call out to an external sensor to check its temperature, and if it is above some threshold, sends an alert message. Code for this application is shown in Figure 7a. We wish to mock out the HTTP requests so that the test will run entirely locally without any reliance on external services, so we mock out the `make_request` method using record-replay mocking.

In a traditional mocking framework, we would have to explicitly specify exactly what HTTP response we expect, which may include headers or large responses. However, by using record-replay agents, we can simply allow the test to run once and contact a real sensor, record the interaction, and save this for the future.

We first run the test code for the "should not send an alert" test as shown in Figure 7b when no values have yet been recorded. We are in INTERACTIVE mode, so although we have not explicitly specified `record_mode = true`, we will be shown the dialog in Figure 8a asking if we want to continue recording; we accept. Then, a request is made, and we are

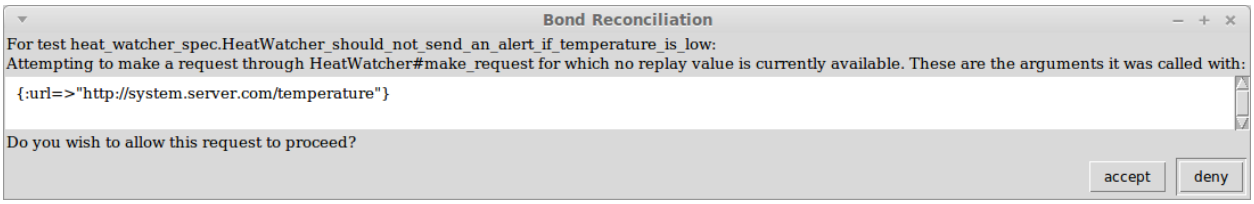
```
class HeatWatcher
  extend BondTargetable

  def send_alert_if_high_temp
    temp = get_temperature
    if temp > 100
      send_alert("Warning! High Temperature: #{temp} C")
    end
  end

  # Read the temperature from a sensor
  def get_temperature
    resp_code, temp_data =
        make_request('http://system.server.com/temperature')
    raise 'Error while retrieving temperature!' unless resp_code == 200
    match = /<temperature>([0-9.]+)<\/temperature>/.match(temp_data)
    raise "Error while parsing temperature from: #{temp_data}" if match.nil?
    match[1].to_f
  end

  # Send an alert
  def send_alert(message)
    make_request('http://backend.server.com/messages', {message: message})
  end

  bond.spy_point
  # HTTP request (GET, or POST if the data is provided)
  def make_request(url, data = nil)
    full_url = "#{url}?#{URI.encode_www_form(data)}"
    resp = Net::HTTP.get_response(URI(full_url))
    [resp.code.to_i, resp.body]
  end
end
```

(a) Sample code for a heat watching alert system.

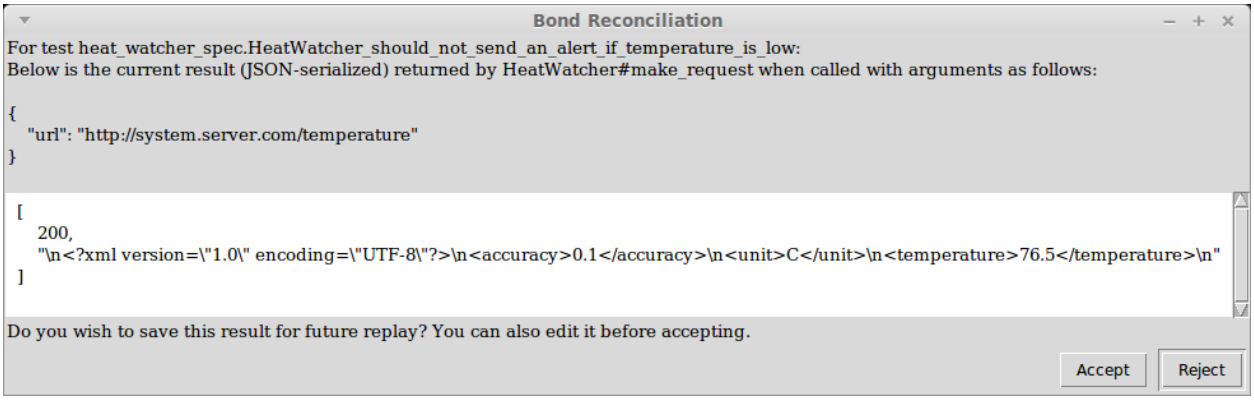
```
describe HeatWatcher do
  include_context :bond

  let(:heat_watcher) { HeatWatcher.new }

  it 'should sent an alert if temperature is high' do
    bond.deploy_record_replay_agent('HeatWatcher#make_request')
    heat_watcher.send_alert_if_high_temp
  end

  it 'should not send an alert if temperature is low' do
    bond.deploy_record_replay_agent('HeatWatcher#make_request')
    heat_watcher.send_alert_if_high_temp
  end

end
```

(b) Test code using record-replay mocking with Bond

Figure 7: A demonstration of using record-replay mocking in Bond.



(a) Checking whether the user wants to allow recording.



(b) Confirming the recorded response as correct.

Figure 8: Dialog windows displayed to the user when recording an interaction for the low temperature test.

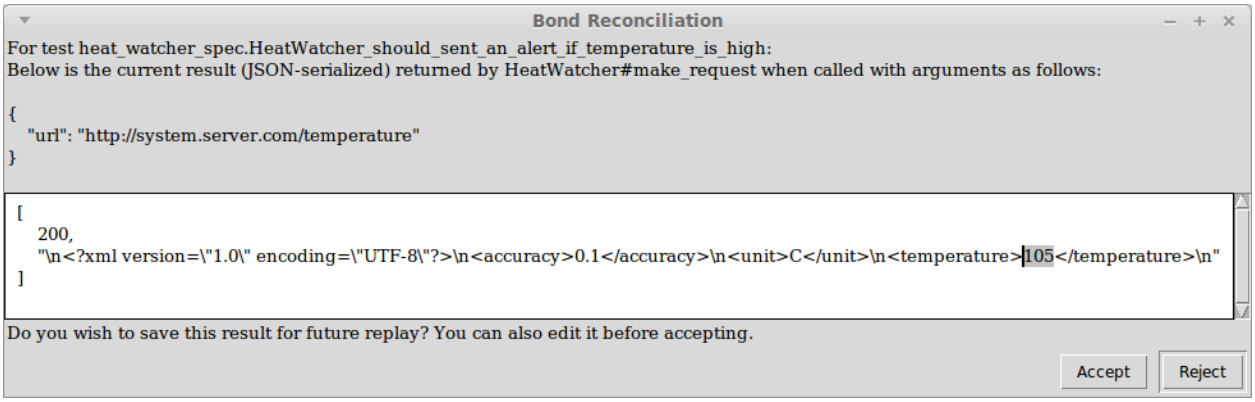
shown the value to accept or reject, as shown in Figure 8b. We accept, and the value is saved to be replayed in future test runs.

Next, we run the "should send an alert" test. Again, we have no saved value, so after confirming that we want to record we are shown the dialog in Figure 9a. Let us assume that the sensor is still in a low temperature state, so the returned response still contains a temperature which would not trigger an alert. This is where it is useful to be able to edit the response; we simply replace 76.5 with 105 and accept the value, triggering an alert. The alert is another HTTP request, so we are shown the dialog in Figure 9b to confirm the interaction, and we save this for replay as well.

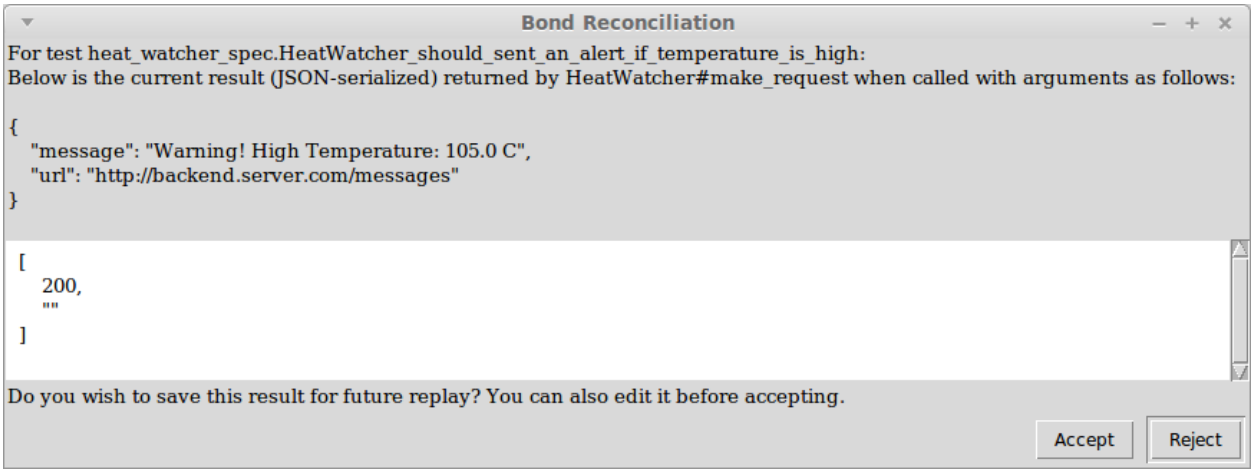
Now, for future test executions, all HTTP interactions will be mocked out by replaying these saved values, and at no point did the user have to manually write or copy-paste any HTTP responses.

5.3 Implementation Details

Currently, record-replay is only available in the Ruby implementation of Bond; porting to Python and Java is planned for future work. The implementation of record-replay agents relies heavily on the previously described mechanisms for saving observations. In RECORD mode, we essentially save the spy point observation as normal, using `spy_result = true` to save the return value as well. The only additional modifications are to request input from the user, and to add a special value to the observation denoting it as a recorded value. At the start of a test, all of these special values are loaded from the observation file and stored to be queried whenever a REPLAY mode spy point is encountered.



(a) Confirming that the temperature response is correct *after* editing it to contain a temperature value which will trigger an alert.



(b) Confirming the message response as correct.

Figure 9: Dialog windows displayed to the user when recording an interaction for the high temperature test.

This has the benefit of making the implementation quite simple, and also of placing these mock calls into the observation file so that they can be viewed as part of the developer's test verification in the same manner as any other observation.

6 Case Studies

We now present three small case studies into various aspects of Bond, all within the Ruby language. First, we examine spy-based testing in general by applying it to the tests for the Ruby library for interacting with the Twitter API[13]. Next, we examine record-replay functionality in a general context by applying it to mocking within the Flapjack monitoring and notification processing system[14]. Finally, we examine record-replay functionality in an

[13] `https://github.com/sferik/twitter`
[14] `http://flapjack.io/`

HTTP context by applying it to Flapjack::Diner, a library for consuming the Flapjack API[15]. Code for all of these case studies is available at `https://github.com/xkrogen/{twitter,flapjack,flapjack-diner}/tree/bond_case_study`, with the last commit on the branch demonstrating the changes applied as part of the case study.

6.1 Spy-Based Testing

The Ruby library for interacting with the Twitter API is a very active project, with over 150 contributors, more than 2,500 commits on GitHub, and nearly 3 million downloads from RubyGems[16]. We applied spy-based testing to all of the "streaming" tests, located within the `spec/twitter/streaming` directory. We converted 24 test cases within 6 test files, although due to some consolidation between test cases, after conversion only 22 test cases remained (though no validations were lost; three of the original test cases were simply more natural to write as a single test case using Bond). We added a total of 45 lines of code to various objects involved in these tests to implement `to_json` methods for those objects to allow their values to be easily serialized during calls to `spy`; note that these additions would continue to be useful throughout any tests which involve the use of these objects, not only the tests studied here. In total 71 lines of assertion code were removed, and not including 2 lines of setup code per file, only 24 lines of Bond code were added back. Using only approximately $\frac{1}{3}$ of the number of lines of code is indicative of lower development effort, as well as less code to maintain as expected behavior changes, without losing strength of assertion. In fact, as discussed below, the strength of assertions was increased despite this decrease in the amount of test code.

Figure 10 shows an example of how much tedious testing code can be saved by using spy-based testing. Rather than explicitly specifying the type and contents of each individual element of the returned array, we can simply observe the array as a whole; the JSON serialization process will automatically inspect each element and display its contents. We omit many values from the observation file here for brevity, but the type of each object in the array and all of the contents which are being asserted about, so no verification was lost as a result, and in fact many new fields are also being asserted on. It is worth pointing out again that this file is maintained by Bond based on occasional user interaction; it is not manually created by the developer.

Figure 11 shows a situation in which the amount of code required in the test does not change significantly, but the strength of the verification is greatly increased. The original code in Figure 11a only ensures that the tweet which returned has the correct `id`. What if the process somehow disturbed the tweet itself, or returned a new tweet object with only an `id` field? The test would not catch such behavior. Using Bond, much more information is captured about the returned object (see Figure 11c), at no extra effort to the developer. The developer is able to see that the tweet has all of the expected fields without needing to explicitly state them all in test code, and is able to save these to ensure that a regression never occurs in the future.

[15] `https://github.com/flapjack/flapjack-diner`

[16] RubyGems is the standard "gem" distribution system for Ruby, where a "gem" is a package that can be used in Ruby programs

```
describe '#user' do
  it 'returns an arary of Tweets' do
    # initial setup code
    expect(objects.size).to eq(6)
    expect(objects[0]).to be_a Twitter::Streaming::FriendList
    expect(objects[0]).to eq([488_736_931, 311_444_249])
    expect(objects[1]).to be_a Twitter::Tweet
    expect(objects[1].text).to eq("The problem with your code is that it's ↙
     ↪ doing exactly what you told it to do.")
    expect(objects[2]).to be_a Twitter::DirectMessage
    expect(objects[2].text).to eq('hello bot')
    expect(objects[3]).to be_a Twitter::Streaming::Event
    expect(objects[3].name).to eq(:follow)
    expect(objects[4]).to be_a Twitter::Streaming::DeletedTweet
    expect(objects[4].id).to eq(272_691_609_211_117_568)
    expect(objects[5]).to be_a Twitter::Streaming::StallWarning
    expect(objects[5].code).to eq('FALLING_BEHIND')
  end
end
```

(a) Original test code.

```
describe '#user' do
  it 'returns an arary of Tweets' do
    # initial setup code
    bond.spy(objects: objects)
  end
end
```

(b) Bond testing code.

```
[
{
  "objects": [
    [ 488736931, 311444249 ],
    {"_type":"Tweet","id":...,"text":"...","uri":"...","user":{"_type":"User","id":7505382,↙
     ↪"uri":"...","screen_name":"sferik","name":"Erik Michaels-Ober"}},
    {"_type":"DirectMessage","id":...,"text":"hello bot","recipient":{"_type":"User",↙
     ↪"id":...,"uri":"...","screen_name":"onediarybot","name":"One Diary Bot"},"sender":↙
     ↪{"_type":"User","id":...,"uri":"...","screen_name":"adambird","name":"Adam Bird"}},
    {"_type":"Event","name":"follow", "source":{"_type":"User","id":...,"uri":"...","↙
     ↪ screen_name":"adambird","name":"Adam Bird"},"target":{"_type":"User","id":...,↙
     ↪"uri":"...","screen_name":"onediarybot","name":"One Diary Bot"}},
    {"_type":"DeletedTweet","id":...,"user_id":...},
    {"_type":"StallWarning","code":"FALLING_BEHIND","message":"...","percent_full":60}
  ]
}
]
```

(c) Observation file. Many fields have been omitted and replaced with ... for the sake of brevity.

Figure 10: Sample of testing code within `client_spec.rb` before and after converting to Bond demonstrating that some assertions can be much more compactly expressed.

```
it 'should have the tweet as the target object' do
  expect(subject.target_object).to be_a(Twitter::Tweet)
  expect(subject.target_object.id).to eq(394_454_214_132_256_768)
end
```

(a) Original test code.

```
it 'should have the tweet as the target object' do
  bond.spy(target_object: subject.target_object)
end
```

(b) Bond testing code.

```
[
{
  "target_object": {
    "_type":"Tweet",
    "id":394454214132256768,
    "text":"@darrenliddell my programmers thought that they had that ↙
     ↪ one covered. I have admonished them.",
    "uri":"https://twitter.com/onediarybot/status/ 394454214132256768",
    "user":{
      "_type":"User",
      "id":1292911088,
      "uri":"https://twitter.com/onediarybot",
      "screen_name":"onediarybot",
      "name":"One Diary Bot"
    }
  }
}
]
```

(c) Bond observation file

Figure 11: Sample of testing code within `event_spec.rb` before and after converting to Bond demonstrating that some assertions can be much more powerful with a similar amount of effort.

These two examples are particularly notable for exemplifying the qualities discussed, but the same benefits were seen in smaller amounts all throughout the conversion process. We believe that this validates spy-based testing as a testing methodology which is very helpful in reducing the amount of testing code written / development effort expended, as well as increasing the strength of validations.

6.2 General Record-Replay

Flapjack, though not as prolific as the Ruby Twitter library, is still an active project with over 130,000 downloads from RubyGems. For this study, we used Bond's record-replay mocking functionality in a generic manner to mock out objects of type `Flapjack::Data::Alert` (a frequently used class which represents an object to be sent as a notification) in the "gateway" tests residing within `spec/lib/flapjack/gateways`. To perform this study required adding 11 lines of code to the `Flapjack::Data::Alert` object to enable spy points on its methods. Across 8 test files and 18 test cases, 256 lines of code to set up mock `Flapjack::Data::Alert` objects were removed, and only 117 lines of code were added back to deploy record-replay agents and 1–2 regular agents per test to return results that couldn't be returned via record-replay (both calls returned other mock objects produced in the test). This is a reduction by approximately $\frac{1}{2}$ in terms of lines of code needed to be produced by the developer. However, we note that there is some additional effort required to place the `Flapjack::Data::Alert` object into the correct state to be recorded.

Figure 12 shows the result of converting RSpec-style mocking of the `Flapjack::Data::Alert` object into a record-replay style using Bond. We see that the amount of setup code has been reduced, though unfortunately we still need to individually specify each method which we would like to record. This is not a fundamental limitation to record-replay mocking and in the future we would like to implement the ability to mark an entire object as record-replay, tracking and recording all method calls to the object, in which case the Bond mocking code would become even shorter. One other nice feature of this style of testing is that the setup code is generic beyond specifying which methods are of interest; this means that all of the Bond record-replay code could easily be placed in a shared setup block while still allowing the returned values to vary based on the individual test, something not possible with traditional mocking.

6.3 HTTP Record-Replay

Flapjack::Diner, a subproject of Flapjack, is a wrapper around the API for interacting with the Flapjack notification and monitoring service. As such, it has numerous HTTP interactions (with Flapjack's external API), so we use this project as a subject for a case study of using Bond's record-replay functionality for HTTP requests. As discussed in Section 5, record-replay type functionality often shines when used in HTTP request contexts due to the frequently large size of parameter lists and of results for HTTP requests. Flapjack::Diner currently makes use of the Pact testing library, discussed further in Section 7.3, to mock out its HTTP requests. In this case study we convert the `contacts_spec` test, located under `spec/resources`, which tests interactions with a specific "contact" Flapjack resource. Though Pact is powerful, it has very verbose setup code; in this case study we will demon-

```
expect(alert).to receive(:address).and_return('pdservicekey')
expect(alert).to receive(:check).twice.and_return(check)
expect(alert).to receive(:state).and_return('critical')
expect(alert).to receive(:state_title_case).and_return('Critical')
expect(alert).to receive(:summary).twice.and_return('')
expect(alert).to receive(:type).twice.and_return('problem')
expect(alert).to receive(:type_sentence_case).and_return('Problem')

fpn.send(:handle_alert, alert)
```

(a) Original RSpec mock code.

```
bond.deploy_agent('Flapjack::Data::Alert#check', result: check)
meths_to_record = %w'method_missing state state_title_case summary ↙
 ↪ type type_sentence_case'
meths_to_record.each do |meth|
  bond.deploy_record_replay_agent("Flapjack::Data::Alert##{meth}")
end

fpn.send(:handle_alert, Flapjack::Data::Alert.new)
```

(b) Bond record-replay mocking code. The `%w'...'` notation denotes an array of space-separated strings, each of which corresponds to a method to which record-replay mocking will be applied. The `#{...}` syntax is used to interpolate a variable into a string.

Figure 12: Sample of testing code within `pager_duty_spec.rb` before and after converting to Bond record-replay mocking, demonstrating that mocking code has the potential to be much shorter when using record-replay.

strate how concisely Bond's record-replay functionality can be used to mock out HTTP interactions. A total of 201 lines of Pact code were removed, and only 37 lines of Bond code needed to be added back to replace Pact. A total of 421 lines of observation files representing these interactions were produced, roughly equivalent to the 201 lines of Pact code due to the more verbose nature of JSON-serialization. Additionally, 12 lines of code had to be modified and 33 additional lines of code added to the base functions used to send HTTP requests to enable Bond to hook into this process; these changes would be applicable to any HTTP record-replay mocking throughout the entire test suite.

Figure 13 shows an example of Pact mocking code as compared to Bond's record-replay mocking code. Clearly, a great deal of coding effort is saved by using record-replay mocking, allowing the developer to simply observe the interaction and verify that it is correct as opposed to explicitly specifying every portion of the interaction. However, Pact also has the advantage that it verifies not only the client side of the interaction (Flapjack::Diner), but also the server side (Flapjack) (see Section 7.3 for more detail). By using Bond this two-sided verification capability is lost; however, again we do not see this as a fundamental limitation. In situations such as this, currently record-replay mocking could be employed simultaneously on both sides of the interaction; however, this would result in two distinct

```
it "has some data" do
  contact_data = {:id         => 'abc',
                  :first_name => 'Jim',
                  :last_name  => 'Smith',
                  :email      => 'jims@example.com',
                  :timezone   => 'UTC',
                  :tags       => ['admin', 'night_shift']}

  flapjack.given("a contact with id 'abc' exists").
    upon_receiving("a GET request for all contacts").
    with(:method => :get, :path => '/contacts').
    will_respond_with(
      :status => 200,
      :headers => {'Content-Type' => 'application/vnd.api+json; ↙
       ↪ charset=utf-8'},
        :body => {:contacts => [contact_data]} )

  result = Flapjack::Diner.contacts
  expect(result).not_to be_nil
  expect(result).to eq([contact_data])
end
```

(a) Original Pact mock code.

```
it "has some data" do
  bond.deploy_record_replay_agent('GET_REQ')
  bond.spy(result: Flapjack::Diner.contacts)
end
```

(b) Bond record-replay mocking code.

Figure 13: Sample of testing code within `contacts_spec.rb` before and after converting to Bond record-replay mocking.

observation files which represent the same interaction. It may be interesting to pursue as future work the possibility of understanding this type of interaction and saving it as a single unified observation file which can be produced through recording rather than manual specification as in Pact.

Regardless of this behavior difference between Pact and Bond, it is clear to see that use of Bond's record-replay functionality for HTTP mocking can be very concise and powerful.

7 Related Work

7.1 Regression Testing

Orstra is a tool that automatically performs regression testing on Java programs [15]. It exercises the system using already existing automatically generated tests, which typically do

not have any test oracles, but force the execution of numerous distinct code paths. During execution, Orstra collects the state of objects used by the test and saves these. Then, after a code change, Orstra repeats the same process and checks where these states differ, warning the user of possible regressions.

Diffy, a tool developed and used at Twitter, also performs automatic regression testing, though in a very different manner [5]. It is tailored specifically to web servers which produce a response to inbound requests. Diffy starts two copies of known-good code, and as well as one instance of the new code which is being checked for regressions. Diffy executes the same series of requests against all three servers, and checks the two copies of known-good code against each other to deduce where non-determinism is to be expected in the responses. Then, the responses produced by the known-good code are compared against those produced by the new code, and differences modulo the expected non-determinism are reported to the user as possible regressions.

Both of these systems compare an old version of known-good code against the new version which is to be regression tested, though they approach the problem in very different ways. These are both similar to Bond's approach to testing: using known-good results to verify subsequently generated outputs. However, Bond differs in that the known-good results are verified by the user rather than trusted from a previous version, enabling them to be used for initial testing as well as regression testing.

7.2 Test Maintenance

ReAssert is a tool that attempts to aid the user in automatically updating Java tests as expected program behavior changes [4]. ReAssert analyzes standard Java tests written in JUnit[17] and examines assertions which are causing tests to fail after a change is made to the system's code. It employs a variety of strategies to attempt to fix the failing assertions, including changing the expected values. The user is then prompted to determine whether or not the changes are correct; if they are, ReAssert will automatically apply them.

This is very similar to how Bond handles changes in tests; newly generated outputs are compared against old, trusted outputs and the user is prompted to determine whether or not the changes are expected, updating the expected values as necessary. Bond takes a more holistic approach to this problem by moving the potentially changing test outputs into a separate file and managing them independently of the test code. This provides a much more simplistic way to update the expected output as behavior changes. However, ReAssert applies to traditional assertion-style unit tests, which is desirable as it applies to an enormous number of existing tests.

7.3 Record-Replay Mocking

A number of systems exist for providing record-replay functionality, mostly focused on HTTP traffic. Most notable among these are VCR[18] for Ruby and Betamax[19] for Java (based off of VCR), which do an excellent job of recording outbound requests and inbound responses

[17] `http://junit.org/`

[18] `https://github.com/vcr/vcr`

[19] `https://github.com/betamaxteam/betamax`

and playing these back at a later time. Similarly to Bond, these values are saved in a separate file to keep test code and data separate; these systems are very useful when the only record-replay functionality necessary is for HTTP traffic.

Pact[20] provides functionality similar to HTTP record-replay, though not identical. The developer must specify the expected requests and responses, but Pact will then verify the expected behavior on *both* sides of the interface. It is used for testing communication within a system rather than to external systems, and has the nice property that by specifying the communication a single time, both participants in the interaction can be tested. However, it requires rather verbose setup code within tests, and you must manually specify the expected interactions as opposed to recording them.

Python's aspectlib[21] does provide full record-replay functionality over arbitrary functions; however, it requires the developer to copy the recorded sequence of expected value-result pairs and place them into the test code. This step places extra burden on the developer and colocates the test data with the test code, which Bond strives to avoid.

8 Future Work

While we are pleased with the functionality that Bond is able to provide in its current state, we see a number of ways to continue to expand the framework. As discussed previously, we plan to implement record-replay functionality in Python and Java; though Python will be a straightforward implementation similar to Ruby, Java will likely be significantly more complex due to issues with the type system, similar to what is discussed in Sections 3.4.3 and 4.3. Additionally, the complexities and fragility of spy point mocking in Java as discussed in Section 4.3 is undesirable and we are continuing to investigate ways to improve this, perhaps through the creation of a custom version of PowerMock or by reimplementing some of its functionality within Bond.

We would also like to continue to implement Bond in more programming languages; due to its relative simplicity, it is not an unrealistic task to be able to reimplement it on top of many languages. One language we are particularly interested in is Scala; it is intriguing because it is statically typed like Java, but has a type system which is significantly more advanced and is amenable to complex typing situations not possible in Java, which can possibly simplify the Bond API.

One more complicated area we are interested in exploring is the notion of observation diff consolidation. When a component which is depended on by many areas of a system is changed, it is likely that many tests will need to be updated in a very similar manner. Though Bond already makes this easier than in a traditional unit testing framework by prompting the user for changes rather than requiring the user to manually specify them, we see an opportunity for Bond to understand that all of these test changes originate from the same root cause, and prompt the user to update all relevant tests simultaneously.

[20] `https://github.com/realestate-com-au/pact`

[21] `https://github.com/ionelmc/python-aspectlib`

9 Conclusions

We have presented spy-based testing, a new testing methodology which aims to make it easier for developers to write unit tests. Spy-based testing eases the pains of initially writing unit tests by allowing developers to use calls to `spy` to replace traditional assertions, which can easily view entire objects with a single call and does not require the developer to explicitly hard-code expected value(s). Spy-based testing is particularly useful as a project evolves over time, since expected test behavior will change, and some of these changes can become streamlined because the test data lives separately from the test code, enabling expected results to be updated with low developer effort.

We have also presented Bond, our implementation of spy-based testing in the Ruby, Python, and Java languages. We demonstrated how Bond can be used both in testing code and in production code to not only replace assertions, but also to provide an easy way to verify that the overall flow of execution is as expected, and to verify intermediate values. We demonstrated how Bond's spy agents can be used for mocking, and how spy point annotations can be employed to utilize this functionality in a concise and intuitive way which can also provide protection against calling dangerous functions during testing. We presented record-replay mocking, a feature of Bond which, although not entirely novel in its concept, can be applied more generally than most existing record-replay libraries. We argued that Bond's spy-based nature makes it especially suitable for employing record-replay mocking, particularly since it separates mock data from test code in a similar manner to the way spy-based testing separates test data from test code. Finally, we presented multiple small case studies to demonstrate the usefulness of spy-based testing in general and of Bond's specific features, and how these can help to enable developers to write tests with less code and higher productivity.

Bibliography

[1] Stefan Berner, Roland Weber, and Rudolf K. Keller. "Observations and Lessons Learned from Automated Testing". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 571–579. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062556. URL: http://doi.acm.org/10.1145/1062455.1062556.

[2] Christoph Csallner and Yannis Smaragdakis. "JCrasher: An Automatic Robustness Tester for Java". In: *Softw. Pract. Exper.* 34.11 (Sept. 2004), pp. 1025–1050. ISSN: 0038-0644. DOI: 10.1002/spe.602. URL: http://dx.doi.org/10.1002/spe.602.

[3] Ermira Daka and Gordon Fraser. "A Survey on Unit Testing Practices and Problems". In: *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*. ISSRE '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 201–211. ISBN: 978-1-4799-6033-0. DOI: 10.1109/ISSRE.2014.11. URL: http://dx.doi.org/10.1109/ISSRE.2014.11.

[4] Brett Daniel et al. "ReAssert: Suggesting Repairs for Broken Unit Tests". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 433–444. ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.17. URL: http://dx.doi.org/10.1109/ASE.2009.17.

[5] *Diffy: Testing services without writing tests*. https://blog.twitter.com/2015/diffy-testing-services-without-writing-tests. Accessed: 2016-04-08.

[6] Debasish Ghosh. *DSLs in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 9781935182450.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065036. URL: http://doi.acm.org/10.1145/1065010.1065036.

[8] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN: 0471469122.

[9] R. Ramler, D. Winkler, and M. Schmidt. "Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?" In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. 2012, pp. 286–293. DOI: 10.1109/SEAA.2012.42.

[10] R. Ramler, K. Wolfmaier, and T. Kopetzky. "A Replicated Study on Random Test Case Generation and Manual Unit Testing: How Many Bugs Do Professional Developers Find?" In: *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. 2013, pp. 484–491. DOI: 10.1109/COMPSAC.2013.82.

[11] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". in: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750. URL: http://doi.acm.org/10.1145/1081706.1081750.

[12] Haruto Tanno et al. "TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 717–720. URL: http://dl.acm.org/citation.cfm?id=2819009.2819147.

[13] Nikolai Tillmann and Wolfram Schulte. "Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution". In: *IEEE Software* 23.4 (2006), pp. 38–47. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=77414.

[14] Chunhui Wang et al. "Automatic Generation of System Test Cases from Use Case Specifications". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 385–396. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771812. URL: http://doi.acm.org/10.1145/2771783.2771812.

[15] Tao Xie. "Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking". In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP'06. Nantes, France: Springer-Verlag, 2006, pp. 380–403. ISBN: 3-540-35726-2, 978-3-540-35726-1. DOI: 10.1007/11785477_23. URL: http://dx.doi.org/10.1007/11785477_23.