Java Reflection API: Revealing the Dark Side of the Mirror

Felipe Pontes
Rohit Gheyi
Federal University of Campina Grande
Campina Grande, Brazil
felipepontes@copin.ufcg.edu.br,rohit@dsc.ufcg.edu.br

Sabrina Souto
State University of Paraiba
Campina Grande, Brazil
sabrinadfs@gmail.com

Alessandro Garcia
PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Brazil
marcio@ic.ufal.br

ABSTRACT

Developers of widely used Java Virtual Machines (JVMs) implement and test the Java Reflection API based on a Javadoc, which is specified using a natural language. However, there is limited knowledge on whether Java Reflection API developers are able to systematically reveal i) underdetermined specifications; and ii) non-conformances between their implementation and the Javadoc. Moreover, current automatic test suite generators cannot be used to detect them. To better understand the problem, we analyze test suites of two widely used JVMs, and we conduct a survey with 130 developers who use the Java Reflection API to see whether the Javadoc impacts on their understanding. We also propose a technique to detect underdetermined specifications and non-conformances between the Javadoc and the implementations of the Java Reflection API. It automatically creates test cases, and executes them using different JVMs. Then, we manually execute some steps to identify underdetermined specifications and to confirm whether a non-conformance candidate is indeed a bug. We evaluate our technique in 439 input programs. Our technique identifies underdetermined specification and non-conformance candidates in 32 Java Reflection API public methods of 7 classes. We report underdetermined specification candidates in 12 Java Reflection API methods. Java Reflection API specifiers accept 3 underdetermined specification candidates (25%). We also report 24 non-conformance candidates to Eclipse OpenJ9 JVM, and 7 to Oracle JVM. Eclipse OpenJ9 JVM developers accept and fix 21 candidates (87.5%), and Oracle JVM developers accept 5 and fix 4 non-conformance candidates.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Reflection API, Non-conformance, Underdetermined Specification

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338946

ACM Reference Format:
Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java Reflection API: Revealing the Dark Side of the Mirror. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338946

1 INTRODUCTION

Reflection is the ability to examine a program and to change its structure and behavior at run time [16]. It is a feature present in Python, C#, and Java. In Java, 78% of open source Java projects use reflection [10], such as JBoss, JUnit5, Maven, and Spring Boot. These projects often depend on the Java Reflection API to implement critical tasks in a program, such as handling dependencies dynamically, inspecting program components, manipulating fields, and invoking methods at run time.

The reliable use of a reflection API largely depends on the systematic conformance testing of the API implementation. Otherwise, two basic problems may occur. First, the implementation of each API method might not be in conformance with its specification. Second, API developers may not be able to reveal issues in the API specification. These two problems may induce misunderstandings of API methods even by experienced Java programmers [32] [33] [39]. Java Virtual Machine (JVM) developers implement and test the Java Reflection API based on a Javadoc [13, p. 37], which is specified using a natural language. In fact, developers of widely used JVMs — e.g., Eclipse OpenJ9 and OpenJDK — implement test cases to verify whether their implementation is in conformance with the Javadoc.

However, there is limited knowledge on whether Java Reflection API developers systematically reveal underdetermined specifications and non-conformances in their implementations. Following Liskov [14], we say that a specification is *underdetermined* if it allows multiple implementations to return different results for the same input. In its turn, a *non-conformance* occurs when a Java Reflection API method does not follow its Javadoc specification. Empirical studies tend to only investigate the frequency [10] and the complexity on the use of the Java Reflection API [34]. They conclude that API users often have to invoke many API methods to implement recurring non-trivial tasks in their programs [10], which makes the use of reflection in Java even harder. However, the scenario may be even more worrisome. It may be the case that

each Reflection API method is poorly tested based on its specification. The lack of proper conformance testing does not help API developers to either fix bugs or improve the specification. The latter may in turn induce developers to misunderstand the behavior of API methods. Popular test cases generators, such as Randoop [28] and EvoSuite [7], heavily use and depend on the Java Reflection API in their implementations. However, these tools do not deal with complex objects in parameter objects [35], such as `Class` and `Method`, and do not focus on detecting non-conformances in the Java Reflection API [2].

To better understand the problem, we analyze test suites of two widely used JVMs — Eclipse OpenJ9 and OpenJDK. Their developers implement most test cases to check the conformance between the Javadoc specification and the Java Reflection API implementation only after a bug has been reported. Moreover, developers do not consider any strategies on choosing data to invoke methods in test cases. Second, we conduct a survey with 130 developers who use the Java Reflection API to see whether the Javadoc specified in natural language impacts on their understanding. We present some Javadoc sentences, and ask for the output of three APIs' methods used in 77% of open source Java projects. Although 67.7% of developers have more than 7 years of experience in Java and 86.9% have knowledge about the Java Reflection API, there is no consensus in the responses. Some developers' comments increase evidence that the Javadoc specification is imprecise and incomplete. Also, some developers face similar problems, as we can see in some issues reported in Randoop and EvoSuite bug trackers. The results of both investigations reinforce the need for improving systematic conformance testing of the Java Reflection API. Moreover, there is limited understanding of how often non-conformances occur and how critical they are.

To improve this scenario, we propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of the Java Reflection API (Section 3). Our technique automatically creates test cases for all possible combinations of parameter values received, and executes them in different JVMs to identify differences (i.e. underdetermined specification and non-conformance candidates). During the test cases execution, objects and primitive values yielded by methods are saved to create more test cases. The technique groups underdetermined specification and non-conformance candidates into three groups: different values, difference between exception thrown and value, and different exceptions. Then, we execute some manual steps to confirm whether an underdetermined specification or non-conformance candidate is indeed a bug, and submit to API specifiers or JVM developers.

We evaluate our technique using 439 input programs from GitHub, and four JVMs (Oracle, OpenJDK, Eclipse OpenJ9, and IBM J9) in Section 4. It identifies underdetermined specification and non-conformance candidates in 32 Java Reflection API public methods of 7 classes. Twenty-one (55.3%) candidates are detected due to test cases created using objects and primitive values saved during the test cases execution. We report underdetermined specification candidates in 12 Java Reflection API methods. The Java Reflection API specifiers accept 3 candidates (25%). We also report 24 non-conformance candidates to Eclipse OpenJ9, and 7 to Oracle JVMs. Eclipse OpenJ9 developers accept and fix 87.5% non-conformance candidates. Our technique identifies non-conformance candidates in methods, such as `Class.getMethods` that is used in 77% of Java open source projects [10]. A number of non-conformance candidates (17) are related to `Class`. Eighty percent of non-conformances candidates in `Class` are due to differences between exception and value. A number of input programs (77%) used in our technique expose at least one non-conformance candidate. Our technique helps JVM developers not only to improve the implementation but also to promote discussions about underdetermined specifications in the Java Reflection API specification. In summary, the main contributions of this work are:

- We analyze the test suites of widely used JVMs, Eclipse OpenJ9 and OpenJDK (Section 2.2);
- We conduct a survey to investigate whether the Java Reflection Javadoc specified in natural language impacts on developers understanding (Section 2.3);
- We propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of the Java Reflection API (Section 3);
- We report 12 underdetermined specification candidates to JVM specifiers detected by our technique. They accept 3 candidates. We also report 31 non-conformance candidates to JVM developers detected by our technique. They accept 26 and fix 25 of them. Twelve test cases are now part of the Eclipse OpenJ9 JVM test suite (Section 4).

2 PROBLEM

Next we characterize the problem from three perspectives.

2.1 Motivating Example

We present an example of an underdetermined specification in the Java Reflection API Javadoc. Jsprit is a Java based toolkit for solving rich traveling salesman and vehicle routing problems. Listing 1 presents the *Route* enum declaring three options. The *ConstManager* class defines the private field array *rts* to store supported routes.

Listing 1: Code snippet of a project related to routes.

```
1 public enum Route {
2   INTER_ROUTE, INTRA_ROUTE, NO_TYPE }
3 class ConstManager { private Route[] rts; }
```

Consider a test case *t1* invoking the Java Reflection API `Class.getResource` method to retrieve a resource related to *Route[]* type (Listing 2). Suppose *c* is a `Class` object representing *Route[]* type created to inspect the program of Listing 1. Using Oracle *1.8.0_151* JVM, *t1* yields a class folder URL (e.g. *file://home/classes*). However, when using Eclipse OpenJ9 JVM *0.8.0*, *t1* yields *null*. The Java Reflection API is specified in a Javadoc using a natural language. According to it, `Class.getResource` should return a resource with a given name. Javadoc presents some rules of a valid resource name, but the specification does not explain the expected result when an empty name is passed as parameter.

Listing 2: `Class.getResource` test case.

```
1 c.getResource("");
```

According to Landman et al. [10], 72% of Java open source projects use the `Class.getResource` method, such as JUnit5, Hibernate ORM, and Apache Maven. They use it mainly to retrieve files inside a jar file, load files resources used in tests, and build projects defined in a resource folder. Java applications that contain a code similar to Listing 2 may have different behaviors when running on the Oracle and Eclipse OpenJ9 JVMs. We present examples of non-conformances between the Java Reflection API implementations and the Javadoc in Section 4.4.

2.2 JVMs Test Suites

To better understand how API developers deal with the problem presented in Section 2.1, we analyze two (Eclipse OpenJ9 and OpenJDK) popular JVMs test suites. We investigate how developers reveal underdetermined specifications and how they check conformance between the specification and the implementation of the Java Reflection API. This investigation enables us to identify gaps in the tests of the Java Reflection API implementations.

We consider the *master* branch commit *1d288ad* of OpenJDK source code repository and the *openj9* branch commit *c2aa034* of Eclipse OpenJ9 source code repository. As the JVMs' test suites present 1,366 source files containing test cases related to Java Reflection API methods, we analyze only the set of files related to popular methods [10]. We identify test cases invoking those popular Java Reflection API methods, and containing references to `Class` type and to `java.lang.reflect` package.

OpenJDK JVM developers implement 71% of source files in the test suite based on reported bugs (i.e. files presenting the *@bug* custom tag in code comments [26]). Some test cases invoke some Java Reflection API methods multiple times (e.g. `Class.newInstance`). JVM developers manually implement test cases using complex objects (e.g. `Method`) returned by invoking a Java Reflection API method (e.g. `Class.getMethods`). Test cases consider two oracles to check whether: i) invoking a Java Reflection API method throws an exception; and ii) the values returned by a method match the expected results.

We do not identify automatic tools used by developers to generate test cases. Input programs size range from 2 SLOC to 32 KSLOC. JVM developers consider all Java keywords in test cases but *goto*. We identify usage of enum, annotations, generics, inheritance, static initialization, inner classes, and so on in input programs. However, we do not identify usage of those Java constructs to implement test cases invoking all Java Reflection API methods. For instance, we do not find an input program that use enum to implement test cases of `Class.getResource` method. Test cases are not executed in random order, which can help identifying unexpected results.

We identify the following types used as data to invoke Java Reflection API methods: *Class[]*, *String.class*, *List.class*, *String*, *bool*, and so on. We do not find differential testing neither strategies to choose method parameters values. For instance, there is no test case invoking `Class.getResource` method with an empty string as parameter value in Eclipse OpenJ9 JVM test suite. Some Eclipse OpenJ9 test cases do not check limit values. For example, tests to create arrays using `Class.forName` method check whether the new array has at most 10 dimensions. The Javadoc specification of `Class.forName` does not specify the maximum dimension of an array. However, the Javadoc of `Array.newInstance` (another method used to create arrays) specifies *"The number of dimensions of the new array must not exceed 255."* Moreover, Section 4.4.1 (The CONSTANT_Class_info Structure) of the Java Virtual Machine Specification (JVMS) [13, p. 80] specifies *"An array type descriptor is valid only if it represents 255 or fewer dimensions."*

2.3 Survey

We also investigate whether underdetermined specifications, similar to the one presented in Section 2.1, actually impact on the understanding of programmers who use the Java Reflection API. We conduct a survey to investigate whether developers who use the Java Reflection API have the same understanding of the Javadoc. We send e-mails to 3,500 GitHub developers. Overall, 130 (3.6%) developers completed the survey, which is the usual response rate for surveys of this kind [19, 22]. A number of 88 (67.7%) developers have more than 7 years of experience in Java, and 86.9% have knowledge about the Java Reflection API.

We ask three questions about methods (`Class.getDeclaredMethods`, `Class.getMethod`, and `Class.getDeclaredFields`) used in 77% of Java open source projects [10]. Figure 1 shows a common question of our survey, in this case related to the `Class.getDeclaredMethods` method. For each question, we present a Javadoc snippet, a small program (3–13 SLOC), and we ask a question about a method call. We present some options, and an open text box in case developers have a different answer.

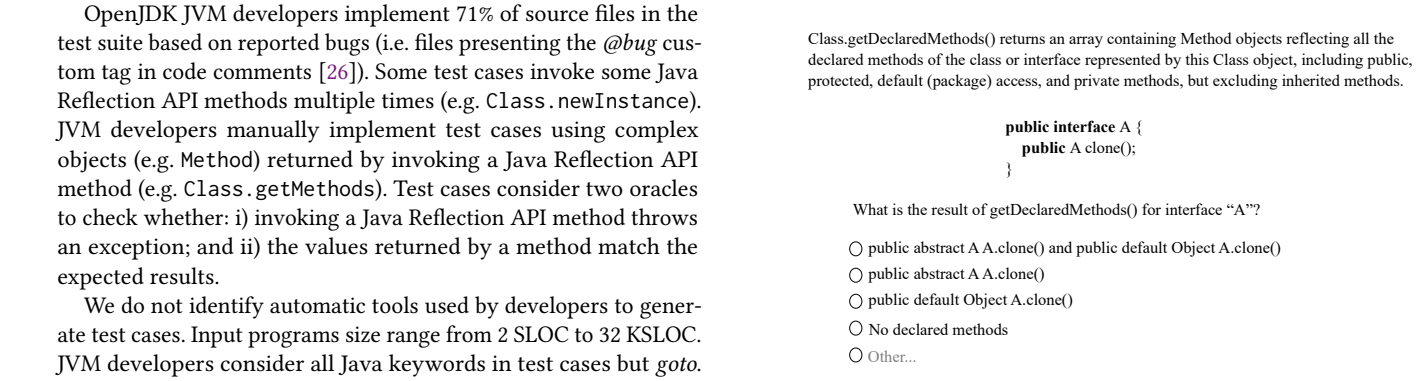

Figure 1: Question 1 about `Class.getDeclaredMethods`.

Next, we explain results of all questions in our survey. The `Class.getDeclaredMethods` method returns all declared methods of a class, excluding inherited ones [24]. Figure 1 presents the program used in Question 1. It declares an interface *A* with a public method *clone*. Question 1 asks developers the result of invoking `Class.getDeclaredMethods` on interface *A*. A number of developers (79.3%) answer *public abstract A A.clone()*. However, others (21.7%) disagree on that (Figure 2). Developers present six different answers to this question.

The `Class.getMethod` method returns a `Method` object that reflects the specified public member method of the class [25]. Question 2 presents three classes to developers (Figure 3 (a)). Class *A* extends class *B* and class *B* extends class *C*. Class *C* contains a public method *c*. We ask developers the result of invoking `Class.getMethod(''c'')` on class *A*. Some developers (57%) answer *public void*

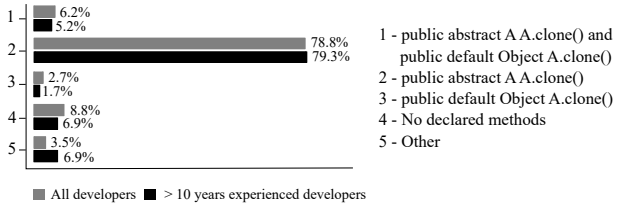

Figure 2: Results of Question 1: `A.getDeclaredMethods`.

C.c, while others (27.6%) answer *public void A.c*. Figure 3 (b) presents the percentage of developers for each answer. We obtained eight different answers to this question. Thus, the divergence here is even more worrisome than the one obtained in Question 1.

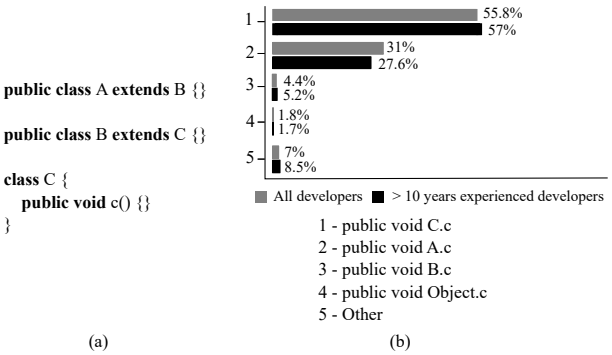

Figure 3: Results of Question 2: `A.getMethod` *c*.

The `Class.getDeclaredFields` method returns all declared fields of a class, excluding inherited ones [23]. Figure 4 (a) presents a program containing a class *B* with an *enum C*. Moreover, the class *A* extends *B*, and declares a method with a parameter of *C* type. Question 3 asks developers the result of invoking `Class.getDeclaredFields` on class *A*. A number of developers (70.7%) answer class *A* has no declared fields. However, 29.3% of developers disagree on that. Developers present nine different answers to this question.

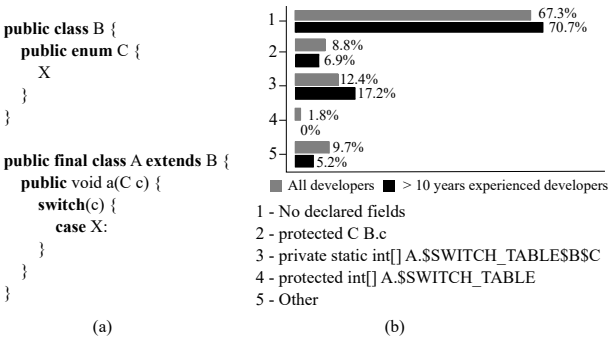

Figure 4: Results of Question 3: `A.getDeclaredFields`.

Although 58 (44.6%) participants have more than 10 years of experience in developing Java applications and knowledge about Java Reflection API, there is no consensus in their survey responses. Overall, the number of different answers varied from six to nine, and responses diverging from the most common one varied from 20.7% to 43%. Moreover, developers also send us some comments about the Java Reflection API in the open text box. An experienced developer does not recommend the use of the Java Reflection API to develop applications other than libraries and frameworks.

Popular automatic test suite generators (Randoop and EvoSuite) heavily use and depend on the Java Reflection API in their implementations. They face similar challenges when using the Java Reflection API. For example, there are some issues reported in their GitHub related underdetermined specifications in the Java Reflection API. Randoop can yield uncompilable test cases when trying to access a class field that implements multiple interfaces that define static fields with the same name. This can happen if a test case tries to access the interface field through the class that implements it because compilers do not know what interface to consider. According to the Javadoc, `Class.getFields` *"...returns the public fields of the class and of all its superclasses."* However, the Javadoc of the `Class.getFields` method does not specify anything about returning static fields inherited from multiple interfaces. To fix that bug, Randoop's developers consider generating test cases only to the classes or interfaces that declare static fields. Moreover, if we generate test cases using a JVM, and run them using another one, we may face some issues, such as flaky tests that may be due to the problems in the Java Reflection API.

In summary, developers do not have a systematic strategy to identify underdetermined specifications in the Javadoc and to test the Java Reflection API to reveal non-conformances. This way, the results presented in this section reinforce the need for such a strategy to help specifiers to improve the specification and developers to implement JVMs. Also, it can prevent users from experiencing issues.

3 TECHNIQUE

We propose a technique to detect underdetermined specifications and non-conformances in the Java Reflection API. Algorithm 1 and Figure 5 summarize the steps of our technique.

Our technique receives as input a Java program, Java Reflection API implementations (JVMs), the Java Reflection API Javadoc [27], and values for Java primitive and non-primitive types. It uses reflection to examine any Java program, such as Listing 1. For instance, we can get the fields of the *ConstManager* class (Listing 1). It tests different implementations, such as the Eclipse OpenJ9 0.8.0 and Oracle 1.8.0_151 JVMs. We can use Java Reflection API Javadoc provided by the Oracle JVM as input [9]. Moreover, our algorithm can consider the following values {*MIN_INT, -1, 0, 1, MAX_INT*} for integers, {*''''*, *'' ''*, *''gEuOVmBvn1''*, *''#A1''*, *null*} for strings, and so on. We define those values based on Equivalence Class (e.g. "gEuOVmBvn1" for *String*), Boundary Value (e.g. -1, 0, 1 for integers), and Limit Value (e.g. $2^{63} - 1$ for *long*) strategies [29].

Before creating test cases in Step 1 for each public method declared in the Javadoc, we use the input program to create `Class` objects (c) (Algorithm 1, Line 5). Consider the input program of Listing 1. We compile the *ConstManager.java* file into the *ConstManager.class* and load it in a JVM using

Algorithm 1: Detect underdetermined specification and non-conformance candidates.

```
   Input: program, implementations, specification, values
 1 testCases ← ∅;
 2 allResults ← ∅;
 3 failedTestCases ← ∅;
 4 executedTestCases ← ∅;
 5 values ← values ∪ getClassObjects(program);
   Step 1. Create test cases
 6 foreach m: specification.getPublicMethods() do
 7 |   foreach p: m.getParameters() do
 8 |   |   types ← p.getTypes();
 9 |   |   tcs ← createTestCases(m, values, types);
10 |   |   testCases ← tcs ∪ testCases;
11 |   end
12 end
   Step 2. Execute test cases
13 foreach t: testCases − executedTestCases do
14 |   tcResults ← ∅;
15 |   foreach implementation: implementations do
16 |   |   result ← implementation.execute(testCase);
17 |   |   allResults ← {(t, {result})} ∪ allResults;
18 |   |   tcResults ← {(t, {result})} ∪ tcResults;
19 |   end
20 |   executedTestCases ← {t} ∪ executedTestCases;
   |   Step 3. Identify new non-conformance candidates
21 |   if tcResults are different then
22 |   |   failedTestCases ← {t} ∪ failedTestCases;
23 |   end
24 end
   Step 4. Create new test cases using new values
25 newParams ← ∅;
26 foreach r: allResults do
27 |   newParams ← {(r.type(), r.value())} ∪ newParams;
28 end
29 if newParams <> values then
30 |   values ← newParams ∪ values;
31 |   go to Step 1;
32 end
   Step 5. Grouping failed test cases into distinct ones
33 failedTestCases ← classifier(failedTestCases);
   Output: (program, failedTestCases)
```

`Class.forName(''ConstManager.class'')` to yield a `Class` object and improve the values received as a parameter. In Step 1, we identify all Java Reflection API public methods in the Javadoc. For instance, it identifies `public URL Class.getResource(String name)` method in the Javadoc. It also identifies parameters types for each Java Reflection API public method. For example, `Class.getResource` receives a *String* as a parameter. We can use *''''* (an empty string) as parameter value for a *String*, and the type (*Route[]*) of *rts* field of the *ConstManager* class of Listing 1 to create the test case presented in Listing 2. The technique creates test cases for all possible combinations of all parameters values (Algorithm 1, Line 9). For instance, consider the `public Method Class.getMethod(String name, Class[] parameterTypes)` method. Our algorithm can create the following test cases: *c.getMethod('''', null)*, *c.getMethod('''', new Object())*, *c.getMethod(null, null)*, and so on. Our technique does not generate redundant test cases. Before generating a new test case, the technique verifies whether the input program, values, and types have already been used.

Algorithm 1 executes all test cases in all implementations (Step 2). For instance, we execute a test case in Eclipse OpenJ9 0.8.0 JVM and in Oracle 1.8.0_151 JVM. The test case execution order is random. All test case results are saved so that they can be used to create new tests. We will use them in Step 4.

Our technique compares the results using differential testing [18] (Step 3, Line 21). Differential testing requires two or more comparable systems. If the results differ or one of the systems loops indefinitely or crashes, the tester has a candidate for a bug-exposing test. Algorithm 1 detects an underdetermined specification or a non-conformance candidate in a Java Reflection API method when a test case presents different results in at least two JVMs. It verifies the return type of a test case. If the return type is primitive (e.g. *int*), it considers the == operator to compare results. When the return type is non-primitive (e.g. *String*), our algorithm invokes the *equals* method of that type to compare results. We implement the *equals* method for some Java Reflection API classes (e.g. `TypeVariable`). For example, Listing 2 presents a test case that has different results in Eclipse OpenJ9 and Oracle JVMs. Eclipse OpenJ9 0.8.0 JVM yields *null*, while Oracle 1.8.0_151 JVM returns a class folder *URL*. Since they have different values, the technique yields the input program and the failed test case representing an underdetermined specification or a non-conformance candidate in the `Class.getResource` method. In case a test case throws an exception, our algorithm analyzes whether the exceptions thrown by each JVM are different.

To improve the chances of detecting underdetermined specification and non-conformance candidates, our technique saves the objects and primitive values yielded by the test cases execution. The goal is to improve Java primitive and non-primitive values used by our technique (Step 4). It checks whether the returned data is already considered (Algorithm 1, Line 29). In case it was not considered as an input for Java types, our technique goes to Step 1 in Algorithm 1 to generate more test cases by considering all possible combinations with the new values. For instance, `Class.getFields` returns `Field` instances that Algorithm 1 uses to create tests for the `Field` class (e.g. `Field.getName`). It uses the field name returned by executing `Field.getName` test case to create a new test case for `Class.getField(String fieldName)`, and so on. The technique can generate test cases for methods that require values different of those initially considered, such as `Method`, `Field`, and `Class`. The number of generated test cases depends on the number of Java Reflection API public methods, parameters values, and class members of an input program. Then, Algorithm 1 yields the underdetermined specification and non-conformance candidates. Each one contains a program used as input and a failed test case, such as Listing 2. The technique then groups underdetermined specification and non-conformance candidates into three distinct groups: different values, difference between exception thrown and value, and different exceptions (Step 5, Line 33).

After performing the automatic steps, we perform some manual steps to check whether each underdetermined specification or non-conformance candidate is indeed a bug. To better understand each underdetermined specification and non-conformance candidate, we

remove all Java constructs from input programs that are not related to each underdetermined specification and non-conformance candidate. We simplify an input program inspired by the delta debugging technique [40] in Step 6. We remove some code snippets, and check whether the new resulting program compiles and the underdetermined specification or non-conformance candidate is still detected. Otherwise, we put back the removed code snippet. We repeat this process until we cannot remove any Java construct in the input program anymore.

We analyze each failed test case with respect to the Javadoc (Step 7) to identify false positives, and correct results. For instance, some methods may yield random values (false positive). We consider a non-conformance candidate when a Java Reflection API method throws an exception not declared in the specification, or yields a result different than specified in Section *Returns* of the specification. We consider an underdetermined specification candidate when the Javadoc of a method is incomplete, imprecise, or ambiguous. In other cases, JVM is in conformance with the specification (✓), and we exclude them. Finally, we report the remaining candidates to API specifiers and to JVM developers (Step 8).

4 EVALUATION

In this section, we evaluate our technique. The complete results and replication package are available at our website [31].

4.1 Definition

The goal of our experiment consists of analyzing our technique for the purpose of detecting underdetermined specifications and non-conformances between the Javadoc and the Java Reflection API implementations with respect to Oracle, OpenJDK, IBM J9, and Eclipse OpenJ9 JVMs from the point of view of specifiers and developers in the context of Java input programs hosted at GitHub. We address the following research questions:

RQ_1**: How many underdetermined specifications and non-conformances between Javadoc and the Java Reflection API implementations can our technique detect?**

We compute the number of underdetermined specifications and non-conformances accepted by Java Reflection API specifiers and JVM developers. The answer to this question enables us to identify issues in the specification and in the implementations of the Java Reflection API.

RQ_2**: How many input programs used by our technique yield at least one underdetermined specification or non-conformance candidate?**

We count all distinct input programs that yield at least one underdetermined specification or non-conformance candidate. The answer to this question reveals how often real input programs can detect underdetermined specification or non-conformance candidates.

4.2 Planning

We use MetricMiner [37] to retrieve a commit of 439 input programs hosted at GitHub with support to Maven and no dependency to Android SDK. Maven helps to resolve dependencies and to compile source files, which are necessary to invoke Java Reflection API methods. We consider input programs from some popular companies, such that: Apache (5), Spotify (3), Twitter (2), Google (2), Netflix (1), and Microsoft (1). Retrieved input programs contain 60,387 source files, and Maven generates 45,984 binary files. Input programs have from 85 to 399,129 SLOC, and 19,919 SLOC on average.

Spring Boot is the most popular input program (22,905 interested people and 339 developers) used in our study. Apache Maven input program has the greater number of commits (12,052). The input programs considered in Algorithm 1 do not need to use the Java Reflection API. Each test case uses an input program to invoke one Java Reflection API method. In our study, 53.5% of analyzed input programs do not use reflection. We calculate the number of executed test cases by summing all API methods calls.

We consider the Java Reflection API Javadoc provided by the Oracle JVM [27]. Algorithm 1 evaluates 237 public methods (98.75%) of classes `Class`, `ClassLoader`, and `Package`, from the `java.lang` package, and `AccessibleObject`, `AnnotatedElement`, `AnnotatedType`, `Constructor`, `Executable`, `Field`, `Method`, and `Parameter` from the `java.lang.reflect` package. We define values for the Java Reflection API methods' parameters based on Equivalence Class, Boundary Value, and Limit Value strategies [29]. We test Oracle 1.8.0_151, OpenJDK 1.8.0_141, IBM J9 8.0.5.10, and Eclipse OpenJ9 0.8.0 JVMs. We execute the experiment on Linux Deepin 15.5 64-bit (i7 3.40GHz and 32GB RAM). We use Maven 3.5, MetricMiner 2, and Git 2.12.2.

Algorithm 1 executes Maven to compile input programs and generate *.class* files (*bytecodes*). It creates a `Class` instance representing a *bytecode* invoking `Class.forName` method. Algorithm 1 uses public methods signatures, parameters values, and the `Class` instance to create test cases. It randomly executes test cases in a JVM and logs results to files identified by the JVM name (e.g. *eclipse-openj9.txt*). Each line of the result files contains a key-value pair. Keys contain a reference to the input program (e.g. *spring-boot*), to the class and method of the Java Reflection API (e.g. `Class.getResource`), and to parameters values (e.g. ''''). Values contain results of the test case execution (e.g. *null*). Algorithm 1 considers the same key for all results files and compares values. If values differ for the same key, it detects an underdetermined specification or a non-conformance candidate. The technique analyzes the results of `Class` test cases to create new test cases with new values to other Java Reflection API methods. For instance, `Class.getMethods` returns `Method` instances that Algorithm 1 uses to create tests for `Method` (e.g. `Method.getName`). Algorithm 1 uses the method name returned by `Method.getName` to create a new test case for `Class.getMethod`, and so on.

4.3 Results

Algorithm 1 executes a total of 288M test cases. Some of them (0.03%) failed. The technique takes about 12 hours to execute all steps presented in Algorithm 1. It yields 10 underdetermined specification and 17 non-conformance candidates. We detect manually 11 candidates during Step 6. We take approximately one hour to manually analyze each of them in Steps 6 and 7. Experienced JVM developers may take less time. We identified 10 candidates as false positives in Step 7. Some JVMs yield results in conformance with the specification in Step 7 (✓). Then, we submit the remaining underdetermined specification and non-conformance candidates only to JVMs that provide bug trackers open to the community in Step 8. Table 1 presents methods with detected candidates, number of test cases, number of failures, and the status of a candidate reported to

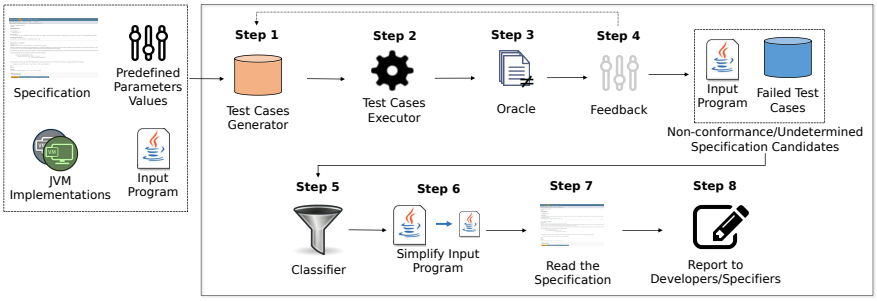

Figure 5: Steps of our technique to detect underdetermined specifications and non-conformances.

the Java Reflection API specifiers, and to Eclipse OpenJ9, and Oracle JVMs' bug trackers. Twenty-one (55.3%) candidates are detected due to test cases created using objects and primitive values saved during the test cases execution.

Eclipse OpenJ9 and IBM J9 JVMs throw an unexpected exception on candidates Ids: *14-25*, *26-27*, and *30-32*. All JVMs return expected exception but with different messages on candidate Id *28*. IBM J9 JVM returns a result different than expected on candidates Ids: *13*, and *29*. Oracle and OpenJDK JVMs return a result different than expected on candidates Ids *1-12*. We consider all JVMs that do not throw an unexpected exception, and returns expected results as correct. As IBM J9 and OpenJDK do not provide open bug trackers, and some JVMs follow the specification for some non-conformance candidates, we submitted 12 underdetermined specifications to Javadoc specifiers and 31 reports to JVM developers. Java Reflection API specifiers and JVM developers accepted 67.4% as real bugs. Eleven bugs are open. So far, we have no answer to them.

`Class.getMethod` is executed in more test cases (8,205,417). `Class.getMethods` yields more test failures (160,213). Algorithm 1 detects 7 non-conformance candidates in Oracle and in OpenJDK, and 26 in Eclipse OpenJ9 and in IBM JVMs. JVM developers answered to 72.7% of them. Oracle JVM developers accepted 5 non-conformance candidates, and Eclipse OpenJ9 JVM developers accepted and fixed 87.5% of the non-conformance candidates. We do not report non-conformances to OpenJDK and IBM J9 JVMs. Their bug trackers can only be accessed by registered developers.

A number of input programs (73.1%) used in our technique expose underdetermined specifications and non-conformance candidates accepted by JVM developers. Cubeqa is the input program that exposed most candidates (23). It is also the input program that most accepted the reported bugs (17). Figure 6 presents a histogram with the number of input programs according to the number of detected candidates, except false positives.

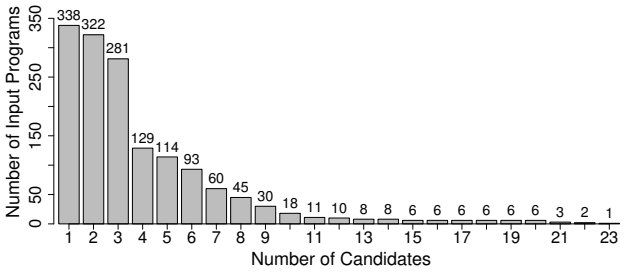

Figure 6: Number of input programs that expose candidates after Step 7 (Figure 5).

4.4 Discussion

4.4.1 Report Candidates to JVM Developers. In RQ_1, in some cases JVM developers do not agree on how to fix some bugs related to the Java Reflection API. For instance, we report a non-conformance candidate in the `Class.getResource` method (Listings 1 and 2) to Oracle JVM and Eclipse OpenJ9 developers. Oracle JVM developers consider the non-conformance as a specification issue because the Java Reflection API Javadoc does not specify what to return when a resource name is empty. On the other hand, developers of Eclipse OpenJ9 do not agree: *"I don't think it is an spec issue. The javadocs define: The rules for searching resources associated with a given class are implemented by the defining class loader of the class."* Eclipse OpenJ9 developers fix the reported non-conformance candidate by changing the `Class.getResource` method result to have the same behavior of the Oracle JVM.

In other cases, developers initially did not accept some of our reported bugs. For instance, when we reported a non-conformance candidate in `Class.getDeclaredFields`, developers stated we must initialize a class instance before accessing declared fields. Javadoc specification for `Class.getDeclaredFields` is not clear about class initialization. Section Structural Constraints (4.9.2) of JVMS [13] states: *"When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized."* However, since `Class` declares `getDeclaredFields`, we should initialize *ManagerServer.class* instance instead of *ManagerServer* instance. Developers agreed on that, fixed the non-conformance and asked us to add our test case to their test suite. We submit a pull request containing that test case and other 11 test cases exposing non-conformances.

Eclipse OpenJ9 developers rejected three reported non-conformance candidates. Since the bug in `Method.invoke` is present in a code imported from the OpenJDK JVM, they asked us to report the bug to the OpenJDK JVM developers. Moreover, Oracle JVM developers have doubts about the specification of `Class.getDeclaredAnnotations`, `Class.getResource`, and `Class.getResourceAsStream`. The bugs related to these methods (Ids: 2, 3, and 4 in Table 1) are still unfixed.

Table 1: Detected Java Reflection API candidates. Test Cases: number of test cases executed by Algorithm 1 calling the method. Failures: number of test cases exposing a candidate in the method. Status: – = Unreported bug; O = Bug Open; F = Fixed bug; A = Accepted bug; R = Rejected bug; ✓= Correct result; D = Duplicated bug.

Id	Method	Test Cases	Failures	Specification	Oracle	OpenJDK	Eclipse OpenJ9	IBM J9
1	Class.getAnnotations	937,860	3,602	O	–	–	–	–
2	Class.getDeclaredAnnotations	939,368	3,490	D	–	–	–	–
3	Class.getResource	1,178,325	200	A	–	–	–	–
4	Class.getResourceAsStream	1,172,325	200	A	–	–	–	–
5	Executable.getAnnotations	177,220	41	O	–	–	–	–
6	Executable.getDeclaredAnnotations	177,412	48	O	–	–	–	–
7	Executable.getParameterAnnotations	177,212	7	O	–	–	–	–
8	Field.getAnnotations	603,028	120	O	–	–	–	–
9	Field.getDeclaredAnnotations	615,960	122	O	–	–	–	–
10	Method.getAnnotations	965,740	293	O	–	–	–	–
11	Method.getDeclaredAnnotations	976,304	338	O	–	–	–	–
12	Method.getParameterAnnotations	963,708	45	O	–	–	–	–
13	Class.getPackage	473,436	471	–	✓	✓	✓	–
14	Class.getConstructor	470,384	59,934	–	✓	✓	F	–
15	Class.getConstructors	468,663	979	–	✓	✓	F	–
16	Class.getDeclaredConstructor	469,624	15,389	–	✓	✓	F	–
17	Class.getDeclaredConstructors	466,646	1,013	–	✓	✓	F	–
18	Class.getDeclaredField	2,350,808	298	–	✓	✓	F	–
19	Class.getDeclaredFields	467,522	449	–	✓	✓	F	–
20	Class.getDeclaredMethod	8,184,703	480	–	✓	✓	F	–
21	Class.getDeclaredMethods	467,347	1,265	–	✓	✓	F	–
22	Class.getField	2,359,361	1,768	–	✓	✓	F	–
23	Class.getFields	470,437	880	–	✓	✓	F	–
24	Class.getMethod	8,205,417	40	–	✓	✓	F	–
25	Class.getMethods	467,821	160,213	–	✓	✓	F	–
26	Constructor.getAnnotatedParameterTypes	356,884	7,657	–	F	–	F	–
				–	F	–	F	–
				–	✓	✓	F	–
27	Executable.getAnnotatedParameterTypes	88,702	435	–	F	–	F	–
				–	F	–	F	–
				–	✓	✓	F	–
28	Method.invoke	1,468,228	240	–	O	–	R	–
				–	O	–	R	–
				–	D	–	R	–
29	Package.getImplementationTitle	59,014	117	–	✓	✓	✓	–
30	Parameter.getAnnotatedType	88,806	1,135	–	✓	✓	F	–
31	Parameter.getParameterizedType	88,764	1,131	–	✓	✓	F	–
32	Parameter.toString	88,664	1,154	–	✓	✓	F	–

4.4.2 Input Programs. RQ_2 is important to better understand the Java input programs that yield candidates. For instance, JVM developers can use Cubeqa input program to detect 23 candidates. Some candidates can be detected by more than 70% of our input programs. Those results indicate that JVM developers consider simpler Java programs as inputs than us. Real input programs contain more Java constructs, which increases the probability of detecting candidates, since we can create more complex objects.

Even small input programs expose candidates. We manually simplify programs in Step 6. They have 6.8 SLOC on average, and use 14 (28%) Java keywords (4.5 on average). Only one simplified input program contains a method body. Listing 3 presents a simplified input program from the Pulsar Reporting. When invoking the `Parameter.getAnnotatedType` method to get the annotated type of the *d* parameter, the Oracle JVM yields *Class BytesBoundedLinkedQueue*. Eclipse OpenJ9 JVM, however, yields an exception.

Listing 3: The Pulsar Reporting program input.

```
1 public class BytesBoundedLinkedQueue<E> {
2   private class Itr implements
3     Iterator<E> { Itr(Iterator<E> d){} } }
```

As a feasibility study, we used an automatic program generator (JDolly [36]) to generate input programs to identify candidates. JDolly generated 197,530 input programs. They contain one package, at most two classes and two methods, inheritance between classes, interface, and one field. However, it does not contain some popular Java constructs, like enum, static blocks, inner classes, generics, or annotations. We used the programs generated by JDolly in Algorithm 1. Our technique detects one candidate in the `Class.getMethods` method. The *Object.wait* method is implemented by the Eclipse OpenJ9 JVM as a native method and it is implemented by the Oracle JVM as a non-native method. We reported that candidate. However, developers of both JVMs claimed that the Java Reflection API specification does not specify whether a method should be native, and rejected it. The technique also reported one false positive in `Class.hashCode`. Step 6 is much easier to be done in small programs generated by JDolly. To detect more candidates using programs generated by JDolly, we must include other Java constructs (e.g. generics, inner classes) in Alloy theory. However, we can face problems related to state explosion. We can improve this scenario by skipping some similar programs [21]. We decided to use real programs because they use a number of Java constructs, and we can create a number of complex objects, and reuse the saved ones.

4.4.3 False positives. In Step 7, we identify 10 false positives in the following methods: `Class.hashCode`, `Class.newInstance`, `Constructor.isAccessible`, `Constructor.newInstance`, `Field.getInt`, `Field.getLong`, `Field.isAccessible`,

`Method.isAccessible`, `Parameter.getDeclaringExecutable`, and `Parameter.hashCode`. The Java Reflection API represents parameters identifiers of a method as *arg0*, *arg1*, and so on. So, two methods can have different parameters types represented by the same identifier. Methods returning hash codes of an object (e.g. `Class.hashCode`) must return the same value more than once just during an execution of a Java application. Hash codes calculated by different JVMs do not necessarily have to be equal. We must invoke the `Field.setAccessible` method before trying to access the value of a private, protected or package-private field. Otherwise it yields an exception. If the running order of the `Field.setAccessible` and `Field.getInt` test cases is different between JVMs, results are also different.

4.4.4 Underdetermined APIs. Recent studies indicate that incompleteness in an API specification can avoid developers to use an API [32] [33] [39]. In fact, some developers who answered our Survey suggest to use other Java Reflection APIs. Other developers state that it is difficult to read the specification and get coding [31]. Moreover, developers that implement APIs must assume some particular constraints in underdetermined APIs, which can lead different implementations to present different results.

However, it is not an easy task to find underdetermined APIs. Our evaluation gives evidence that our technique can help developers to detect specifications excerpts that are incomplete. We found parts of the specification that do not explain what to do when considering some Java constructs, such as annotations (e.g. underdetermined specification 2 in Table 1) , or input values, such as empty string (Listing 1). In our evaluation, first we assumed the problem was in at least one JVM implementation. We reported candidates to all possible implementations. In some cases, they accepted them in at least one implementation. In other cases, developers indicated underdetermined specifications. We send them to developers in charge of the Java Reflection API specification. We hope that the results presented here can help the Java Reflection API specifiers and the JVM's developers to better understand and improve the specification and, consequently, the JVM's implementations.

4.4.5 Automatic Test Suite Generators. Tools like Randoop [28] and EvoSuite [7] can be used to aid developers on improving tests coverage, and finding bugs in widely-deployed commercial and open-source software. However, we cannot use these tools since most Java Reflection API classes do not expose a public constructor to allow instantiating an object and invoking methods directly. We execute Randoop and EvoSuite to generate tests for the `Method` class. Since `Method` defines only methods that need an instance to be invoked, Randoop does not generate tests to `Method`. So, Randoop does not generate tests invoking Java Reflection API methods. EvoSuite throws an exception when trying to generate tests to the `Method` class. Our technique finds non-conformance bugs in the Oracle JVM, like `Method.invoke`.

We also evaluate Randoop and EvoSuite in small programs that use the Java Reflection API. For example, consider the program of Listing 4. It defines a class *A* and a method *m* containing a `Method` parameter *p*. Randoop does not generate tests, while EvoSuite consider *null* for *p*. Randoop and EvoSuite do not deal with these complex objects. It is a complex and challenging task for them, since it requires a certain sequence of method calls prior to exercising the target method [35]. For instance, to generate a `Method` object, an automated tool must consider accessibility, parameters, body, return type, and so on. Moreover, the software behavior using the Java Reflection API is fundamentally hard to predict by analyzing the code [10]. So, these tools do not focus on testing programs using the Java Reflection API [2] differently from our technique.

Listing 4: Small program used as input.

```
1 public class A {
2   public int m(Method p) {return 0;} }
```

4.4.6 Testing Other APIs. As future work, we intend to adapt our technique to test other API implementations than Java Reflection API. Algorithm 1 can receive the Javadoc to identify all public methods, and their parameters types. Other APIs, like the Java Collections API, do not require a Java input program. In this case, we can remove this parameter. We can test the Java Collections API implemented by the Oracle JVM and Eclipse OpenJ9 JVMs. We also have to implement *equals* methods for some classes (e.g. *ArrayList*) (Step 3 in Algorithm 1). We do not need to change the other steps of our technique to test the Java Collections API implementations.

4.5 Threats to Validity

Algorithm 1 does not detect an underdetermined specification or a non-conformance candidate if a Java Reflection API method presents the same results in different JVMs. To reduce this threat, we evaluated four JVMs. Our classifier may miss some candidates in Step 5. Due to manual reasoning, we can incorrectly classify a candidate as a false positive, or having a correct result in Step 7.

We selected all Maven input programs hosted at GitHub. Although this set of Java input programs may not be representative, we need input programs with support to resolving dependencies because we need to build input programs to execute our analysis. Although input programs are from one code repository, GitHub has almost 20M contributors and more than 1M Java input programs. We consider input programs with different SLOC, amount of developers, and from different domains, including: build automation tools, Web frameworks, and JDBC drivers.

5 RELATED WORK

To the best of our knowledge, there is no study aiming to detect underdetermined specifications and to check conformances between the specification and the implementations of the Java Reflection API. Tan et al. [38] present a technique, called @TCOMMENT, to check conformance between the Javadoc and Java programs implementations. @TCOMMENT analyzes Javadoc of Java input programs to infer properties about *null* values and related exceptions. Then, it generates random tests for the input programs, checks the inferred properties, and reports inconsistencies. We have two non-conformance candidates related to the *null* normal property [38]. @TCOMMENT can be useful to confirm them as implementation bugs. Considering other types of non-conformances may be a challenge [38], but we will investigate other heuristics.

Gyori et al. [8] propose NonDex, a tool for detecting and debugging wrong assumptions on Java APIs. Some APIs have underdetermined specifications to allow the implementations to achieve different goals, e.g., to optimize performance. When clients of such APIs

assume stronger-than-specified guarantees, the resulting client code can fail. NonDex tool executes application's test suite, then it changes the API implementation (e.g. the order of elements in a collection), and it executes tests again. If results differ, it detects an underdetermined specification candidate and presents an API code snippet. NonDex uses a binary search to locate invocations that cause a failure. We manually reduced the programs inspired by the delta debugging technique [40]. We can automate that reduction by adapting our technique based on binary search [8].

Chen et al. [4] present the Classfuzz tool that creates Java programs by using mutations, and uses differential testing to identify bugs in JVMs' startup processes. Classfuzz uses predefined binary files to generate mutants used as input programs to create test cases. Chen et al. [3] evaluate some differential testing techniques (Randomized Differential Testing – RDT and Different Optimization Levels – DOL) in compilers. RDT randomly tests compilers, and DOL looks for optimization bugs. We can use their tool to mutate the input programs considered in our evaluation. Our technique considers all combinations of input programs, values, and Java Reflection API methods to create test cases. Algorithm 1 applies differential testing to check whether the tests results are different when running in more than one JVM.

Pham et al. [30] propose Java StarFinder (JSF), a tool that uses symbolic execution to generate test cases. JSF handles dynamically-allocated linked data structures (e.g. trees) as input. Those tools do not focus on testing the Java Reflection API. Algorithm 1 generates test cases to identify underdetermined specification and non-conformance candidates in the Java Reflection API.

Legunsen et al. [12] perform a study of the bug-finding effectiveness of formal specifications by using JavaMOP to check whether test suite execution results are in conformance with formal specifications. Ahrendt et al. [1] propose a tool (KeYTestGen) that generate test cases for a real-time Java API. KeYTestGen uses JML specifications as input of a prover and uses each proof branch to generate test inputs satisfying each constraint. Cheon and Leavens [5] propose a tool that automatically generates test cases from JML formal specifications. Milanez [20] proposes a tool to automatically check conformance of Java programs annotated with JML [11] specifications based on automatic test generation using Randoop [28]. Massoni et al. [17] propose a formal approach to semi-automatically refactor Java programs in a model-driven manner. They explain their approach in a case study considering three refactorings [6]. They show evidences about issues with keeping object models and the implementation in conformance during refactoring. Our technique checks conformance between the specification of Java Reflection API, which is described in natural language.

Livshits et al. [15] propose an algorithm to statically analyze programs that use Java Reflection. It does not cover all reflection usage because some classes are available only at run time, specially in dynamic class loading scenarios. Landman et al. [10] perform studies to identify challenges related to static analysis of programs using Java Reflection API. They suggest that combining both static and dynamic analysis of programs using Java Reflection API may improve the existing solutions for the challenges found. Algorithm 1 performs dynamic analysis to identify candidates.

6 CONCLUSIONS

This work presents empirical investigations and a new technique that enables us to understand "the dark side of the mirror." We analyze test suites of popular JVMs, and we find that their developers implement most test cases to reveal underdetermined specifications and to check the conformance between the Javadoc specification and the Java Reflection API implementation only after a bug has been reported. We conduct a survey with 130 developers who use the Java Reflection API to see whether the Javadoc impacts on their understanding. Although 67.7% of developers have more than 7 years of experience in Java and 86.9% have knowledge about the Java Reflection API, there is no consensus in their responses.

To improve this scenario, we propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of the Java Reflection API. It automatically creates test cases and executes them in different JVMs. We evaluate our technique in 439 input programs hosted at GitHub. We find underdetermined specifications and non-conformance candidates in 32 public methods of 7 Java Reflection API classes. We report underdetermined specification candidates on 12 Java Reflection API methods. Java Reflection API specifiers accept 3 underdetermined specification candidates (25%). We also report 31 non-conformance candidates to JVM developers. Oracle developers accept 5 and fix 4 non-conformance candidates and Eclipse OpenJ9 developers accept and fix 21 non-conformance candidates, and include 12 test cases in their suite. Twenty-one (55.3%) candidates are detected due to test cases created using objects and values saved during previous test case execution. The test suites described in Section 2.2 cannot detect the candidates found by our technique. So far, we do not find any patterns in the bugs found.

The results of applying our technique helped JVM developers to improve the implementation and promote discussions about underdetermined specifications in the Java Reflection API Javadoc, confirming the lack of consensus found by our survey. Java Reflection API specifiers should propose a formal specification to avoid underdetermined specifications and consider more Java constructs and values for method parameters when specifying method results. We recommend JVM developers to improve their testing strategies to identify underdetermined specifications and non-conformances: i) create test cases using more combinations between input programs and parameter values for all Java Reflection API methods; ii) use strategies to choose values (like Limit Value); iii) execute test cases in random order; iv) use differential testing; v) use real programs to richer complex objects; and vi) consider more complex objects for `Class`, `Method`, and so on.

As future work, we intend to consider new parameters values and methods, and implement test cases creation considering more than one Java Reflection API method per test case. We aim at evaluating Algorithm 1 in different operating systems, and using new versions of JVMs that fixed the detected non-conformance candidates found. We also aim to extend our technique to other APIs (e.g. Java Collections API), and to other languages (e.g. C#).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers. This work was partially supported by CNPq and CAPES grants.

REFERENCES

[1] Wolfgang Ahrendt, Wojciech Mostowski, and Gabriele Paganelli. 2012. Real-time Java API Specifications for High Coverage Test Generation. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'12)*. Copenhagen, Denmark, 145–154.

[2] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated Unit Test Generation for Classes with Environment Dependencies. In *Proceedings of the 2014 29th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE'14)*. New York, NY, USA, 79–90.

[3] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. Austin, USA, 180–190.

[4] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. *ACM SIGPLAN Notices* 51, 6 (2016), 85–99.

[5] Yoonsik Cheon and Gary Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*. Malaga, Spain, 231–255.

[6] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley.

[7] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. Szeged, Hungary, 416–419.

[8] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications. In *Proceedings of the 24th SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. Seattle, USA, 993–997.

[9] IBM. 2015. API Reference. https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.api.80.doc/api_overview.html.

[10] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. 2017. Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. Buenos Aires, Argentina, 507–518.

[11] A. Leavens, G. Baker and C. Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38.

[12] Owolabi Legunsen, Wajih Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How Good Are the Specs? A Study of the Bug-finding Effectiveness of Existing Java API Specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. Singapore, Singapore, 602–613.

[13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[14] Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* (1st ed.). Addison-Wesley Professional.

[15] Benjamin Livshits, John Whaley, and Monica Lam. 2005. Reflection Analysis for Java. In *Proceedings of the 3rd Asian Conference on Programming Languages and Systems (APLAS'05)*. Tsukuba, Japan, 139–160.

[16] Pattie Maes. 1987. Concepts and Experiments in Computational Reflection. *ACM SIGPLAN Notices* 22, 12 (1987), 147–155.

[17] Tiago Massoni, Rohit Gheyi, and Paulo Borba. 2008. Formal Model-driven Program Refactoring. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08/ETAPS'08)*. Berlin, Heidelberg, 362–376.

[18] William McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[19] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469.

[20] Alysson Milanez. 2018. *Fostering Design By Contract by Exploiting the Relationship between Code Commentary and Contracts*. Ph.D. Dissertation. UFCG.

[21] Melina Mongiovi, Gustavo Wagner, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. 2014. Scaling Testing of Refactoring Tools. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME) (ICSME'14)*. Victoria, BC, Canada, 371–380.

[22] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. Austin, USA, 935–946.

[23] Oracle. 2014. Class.getDeclaredFields Method Javadoc Specification. https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getDeclaredFields--.

[24] Oracle. 2014. Class.getDeclaredMethods Method Javadoc Specification. https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getDeclaredMethods--.

[25] Oracle. 2014. Class.getMethod Method Javadoc Specification. https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getMethod-java.lang.String-java.lang.Class...-.

[26] Oracle. 2014. How to Write Doc Comments for the Javadoc Tool. http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

[27] Oracle. 2014. Java Reflection API Javadoc Specification. https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html.

[28] Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. 2007. Feedback-directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, USA, 75–84.

[29] Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles and Techniques* (1st ed.). Wiley.

[30] Long Pham, Quang Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2018. Testing Heap-based Programs with Java StarFinder. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. Gothenburg, Sweden, 268–269.

[31] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java Reflection API: Revealing the Dark Side of the Mirror (Artifacts). http://www.dsc.ufcg.edu.br/~spg/fse2019.html.

[32] Martin Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34.

[33] Martin Robillard and Robert Deline. 2011. A Field Study of API Learning Obstacles. *EMSE* 16, 6 (2011), 703–732.

[34] Zalia Shams and Stephen Edwards. 2013. Reflection Support: Java Reflection Made Easy. *The Open Software Engineering Journal* 7, 1 (2013), 38–52.

[35] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE'15)*. Washington, DC, USA, 201–211.

[36] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

[37] Francisco Sokol, Mauricio Aniche, and Marco Gerosa. 2013. MetricMiner: Supporting Researchers in Mining Software Repositories. *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* 1, 1 (2013), 142–146.

[38] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST'12)*. Washington, USA, 260–269.

[39] Gias Uddin and Martin Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (2015), 68–75.

[40] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers.