UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE BARROS PONTES

A TECHNIQUE TO TEST APIS SPECIFIED IN NATURAL LANGUAGE

CAMPINA GRANDE - PB
2020

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

A Technique to Test APIs Specified in Natural Language

Felipe Barros Pontes

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computaćão

Linha de Pesquisa: Linha de Pesquisa

Rohit Gheyi

Márcio Ribeiro

(Orientadores)

Campina Grande, Paraíba, Brasil

©Felipe Barros Pontes, 29/04/2020

P814t Pontes, Felipe Barros.

A technique to test apis specified in natural language/Felipe Barros Pontes. - Campina Grande, 2020.

95 f. : il. Color.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2020.

"Orientação: Prof. Dr. Rohit Gheyi, Prof. Dr. Márcio de Medeiros Ribeiro".

Referências.

1. Especificação de Requisitos. 2. Linguagem Natural. 3. Não Conformidade. 4. Especificação Incompleta. 5. Especificação Ambígua. I. Gheyi, Rohit. II. Ribeiro, Márcio de Medeiros. III. Título.

CDU 004.414.38(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA Itapuana Soares Dias CRB-15/93

A TECHNIQUE TO TEST APIS SPECIFIED IN NATURAL LANGUAGE

FELIPE BARROS PONTES

TESE APROVADA EM 05/03/2020

ROHIT GHEYI, Dr., UFCG
Orientador(a)

MÁRCIO DE MEDEIROS RIBEIRO, Dr., UFAL
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Examinador(a)

ALESSANDRO FABRICIO GARCIA, Dr., PUC-RIO
Examinador(a)

LEOPOLDO MOTTA TEIXEIRA, Dr., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Desenvolvedores de *Application Programming Interfaces* (APIs) as implementam e testam com base em documentos comumente especificados em linguagem natural. Entretanto, não se sabe até onde os desenvolvedores que implementam as APIs são capazes de sistematicamente revelar i) especificações indeterminadas; e ii) não conformidades entre suas implementações e a especificação. Este trabalho apresenta uma análise da suíte de testes da API de Reflexão de Java da máquina virtual OpenJDK e resultados de duas *surveys* para verificar se a especificação influencia no entendimento deles. Neste trabalho, propõe-se uma técnica para detectar especificações indeterminadas e não conformidades entre a especificação e implementações de uma API. A técnica automaticamente gera casos de teste e os executa usando diferentes implementações. Se os resultados forem diferentes, a técnica detecta um candidato a especificação indeterminada ou a uma não conformidade entre a especificação e pelo menos uma implementação da API. Para avaliar a técnica, foi usada a API de Reflexão de Java com 446 programas de entrada. A técnica proposta identificou especificações indeterminadas e candidatos a não conformidades em 32 métodos públicos de sete classes da API de Reflexão de Java. Foram reportados candidatos a especificações indeterminadas em 12 métodos da API de Reflexão de Java. Especificadores da API de Reflexão de Java aceitaram três candidatos a especificação indeterminada (25%). Também foram reportados 24 candidatos a não conformidades para os desenvolvedores da máquina virtual Eclipse OpenJ9 e 7 para a Oracle. Desenvolvedores da Eclipse OpenJ9 aceitaram e corrigiram 21 candidatos (87,5%). Desenvolvedores da Oracle aceitaram cinco e corrigiram quatro candidatos a não conformidades. Doze casos de teste gerados pela técnica proposta neste trabalho fazem parte da suíte de testes da Eclipse OpenJ9. A técnica proposta também foi avaliada usando a API de Collections de Java. A técnica identificou 29 candidatos a especificações indeterminadas e a não conformidades. Foram reportados 5 candidatos a especificação indeterminada para os especificadores da API de Collections de Java. Também foram reportados 9 candidatos a não conformidades na máquina virtual Eclipse OpenJ9 e 4 na Oracle. Desenvolvedores da Oracle aceitaram e corrigiram três candidatos a não conformidades. Desenvolvedores da Eclipse OpenJ9 aceitaram e corrigiram 1 candidato a não conformidade.

Abstract

Developers of widely used Application Programming Interfaces (APIs) implement and test APIs based on a document, which is commonly specified using natural language. However, there is limited knowledge on whether API developers are able to systematically reveal i) underdetermined specifications; and ii) non-conformances between their implementation and the specification. To better understand the problem, we analyze test suites of Java Reflection API, and we conduct two surveys. A survey with 130 developers who use the Java Reflection API, and a survey with 128 C# developers who use and implement the .NET Reflection API to see whether the specification impacts on their understanding. We also propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of the APIs. It automatically creates test cases, and executes them using different implementations. It saves objects yielded by methods to be used to create more test cases. If results differ, it detects an underdetermined specification or a non-conformance candidate between the specification and at least one implementation of the API. We evaluate our technique using the Java Reflection API in 446 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. Twelve test cases are now part of the Eclipse OpenJ9 JVM test suite. We also evaluate our technique using the Java Collections API. Even being a very popular Java API, our technique identifies 29 underdetermined specification and non-conformance candidates. Our technique identifies 17 candidates that cannot be detected by popular automatic test suite generators. We report 5 underdetermined specification candidates to the Java Collections API specifiers. We also report 9 non-conformance candidates to Eclipse OpenJ9 JVM, and 4 to Oracle JVM. Oracle JVM developers accept and fix 3 non-conformance candidates. Eclipse OpenJ9 JVM developers accept and fix 1 non-conformance candidate.

Agradecimentos

Agradeço primeiramente a Deus por me dar saúde, força e sabedoria para ter discernimento nos momentos necessários.

A realização deste trabalho só é possível graças a pessoas-chave na minha vida. Primeiro, minha esposa, Monike, que está sempre ao meu lado me ajudando, sendo paciente e compreensiva nos momentos em que preciso me dedicar à pesquisa. Te amo! Minha filha, Sofia, que me motiva com sua meiguice e sorrisos inocentes. Minha família, que me apoia e me ajuda a ter energia para executar o trabalho e que aguenta as saudades em momentos mais críticos. Amo vocês! Meus amigos, que de uma forma ou de outra me apoiam. Meus colegas de trabalho (Embedded/Virtus/Edge) que entendem os momentos em que preciso me ausentar para reuniões e atividades de pesquisa. Meus colegas de SPG, que enriquecem meu trabalho apresentando ideias e compartilhando conhecimentos. Meus orientadores, que dão todo o suporte necessário à condução do trabalho, questionando, apresentando ideias, cobrando resultados nos momentos certos e sendo compreensivos nos momentos mais difíceis. A todos vocês, meu muito obrigado.

Aos professores e funcionários da COPIN e do DSC; À Capes pelo apoio e suporte financeiro fornecidos a este trabalho.

Contents

1 Introduction **1**

1.1 Problem 2

1.2 Motivating Examples 4

1.3 Solution 6

1.4 Evaluation 7

1.5 Contributions 8

2 Relevance **10**

2.1 Java Reflection APIs Test Suites 12

2.2 Surveys 16

2.2.1 The Java Reflection API Survey 16

2.2.2 The .NET Reflection API Survey 22

2.2.3 Discussion 36

2.2.4 Threats to Validity 41

2.3 Conclusions 42

3 A Technique to Test APIs Specified in Natural Language **43**

3.1 Overview 43

3.2 Getting Initial Objects 49

3.3 Generating Test Cases 49

3.4 Feedback 50

3.5 Oracle 52

3.6 Classifier 53

3.7 Simplifying Input Programs 54

3.8 Reading the Specification and Reporting Bugs 55

4 Evaluation 56

4.1 Testing the Java Reflection API 56

4.1.1 Definition 56

4.1.2 Planning 57

4.1.3 Results 58

4.1.4 Answers to the Research Questions 62

4.2 Testing the Java Collections API 63

4.2.1 Definition 63

4.2.2 Planning 64

4.2.3 Results 65

4.2.4 Answer to the Research Question 67

4.3 Discussion 67

4.3.1 Report Candidates to APIs 67

4.3.2 Input Programs 71

4.3.3 False Positives 73

4.3.4 Underdetermined APIs 74

4.3.5 Automatic Test Suite Generators 75

4.4 Testing Other APIs 78

4.5 Threats to Validity 78

5 Related Work 80

5.1 Conformance Checking 80

5.2 APIs Analysis 82

5.3 Surveys 84

6 Conclusions 85

6.1 Future Work 87

List of Symbols

API - *Application Programming Interface*

SLOC - *Source Lines of Code*

SDK - *Software Development Kit*

JVM - *Java Virtual Machine*

List of Figures

2.1 Question 1 about Class.getDeclaredMethods. 17

2.2 Developers experience with Java. 18

2.3 Developers knowledge about the Java Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it. 18

2.4 Developers using the Java Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop. 19

2.5 Results of Question 1: A.getDeclaredMethods. 20

2.6 Results of Question 2: A.getMethod c. 20

2.7 Results of Question 3: A.getDeclaredFields. 21

2.8 Question 1 of our survey. 23

2.9 .NET Reflection API users experience with C#. 24

2.10 Users knowledge about the .NET Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it. 24

2.11 Users using the .NET Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop. 25

2.12 .NET Reflection API developers experience with C#. 25

2.13 Developers knowledge about the .NET Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it. 26

2.14 Developers using the .NET Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop. 26

2.15 Question 1: `MemberInfo.ReflectedType`. 27

2.16 Question 2: `ConstructorInfo.Invoke`. 28

2.17 Question 3: `Type.FindMembers`. 30

2.18 Question 4: `MethodInfo.MakeGenericMethod`. 31

2.19 Question 5: `Assembly.CreateInstance`. 33

2.20 Question 6: `Type.GetField`. 34

2.21 Question 7: `Type.GetMember`. 35

3.1 Steps of our technique to detect underdetermined specifications and non-conformances in APIs specified in natural languages. 46

4.1 Number of input programs that expose candidates after Step 7 (Figure 3.1). 62

4.2 Number of input programs exposing each candidate after Step 7 (Figure 3.1) (see *Id* column in Table 4.1). 63

List of Tables

2.1 Java Reflection API methods usage. Content derived from Landman et al. [22]. 11

2.2 Chi-squared test results comparing responses of experienced and non-experienced developers. 37

2.3 Chi-squared test results comparing responses of experienced and non-experienced users. 38

2.4 Chi-squared test results comparing responses of experienced and non-experienced developers. 38

2.5 Chi-squared test results comparing responses of users and developers. . . . 39

2.6 Results of Machine Learning classifiers for each question of the Java Reflection API Survey. Mean Accuracy considers 10-Fold Cross Validation. . . . 40

2.7 Results of Machine Learning classifiers for each question of the .NET Reflection API Survey. Mean Accuracy considers 10-Fold Cross Validation. . 41

3.1 Input data to generate test cases. 50

4.1 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. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. Status: – = Unreported bug; O = Bug Open; F = Fixed bug; A = Accepted bug; R = Rejected bug; ✓ = Correct result; D = Duplicated bug. 59

4.2 Detected Java Collections 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. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. Status: – = Unreported bug; O = Bug Open; F = Fixed bug; A = Accepted bug; R = Rejected bug; ✓ = Correct result; D = Duplicated bug. 66

4.3 Usage of Java Collections API open bugs methods in the 446 input programs of the Java Reflection API evaluation (Section 4.1). 70

4.4 Java Reflection API false positives reported by our technique. 73

4.5 Java Collections API false positives reported by our technique. 74

4.6 Java Collections API candidates detected by Randoop. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. Status: – = Unreported bug; O = Bug Open; F = Fixed bug; A = Accepted bug; R = Rejected bug; ✓ = Correct result; D = Duplicated bug. . 77

Listings

1.1 Code snippet of the Jsprit project related to routes. 4
1.2 `Class.getResource` test case. 5
1.3 `ArrayList.ensureCapacity` test case. 5
1.4 Code snippet related to `MemberInfo.ReflectedType` .NET Reflection API property. 6
2.1 Test case of Eclipse OpenJ9 JVM using two oracles. 14
2.2 `Class.getResource` OpenJDK test case. 15
2.3 Program of Question 2. 20
2.4 Program of Question 3. 21
2.5 Program used in Question 2 concerning the `ConstructorInfo.Invoke` method. 28
2.6 Program used in Question 3 concerning the `Type.FindMembers` method. 29
2.7 Program used in Question 4 concerning the `MethodInfo.MakeGenericMethod` method. 31
2.8 Program used in Question 5 related to `Assembly.CreateInstance` method. 32
2.9 Program used in Question 6 concerning the `Type.GetField` method. . . 33
2.10 Program used in Question 7 concerning the `Type.GetMember` method. . 35
3.1 Example of input program to create test cases. 49
3.2 Input program used in feedback. 52
3.3 Eclipse OpenJ9 JVM results. 52
3.4 Oracle JVM results. 53
3.5 Input program sample. 54
3.6 Simplified input program sample. 55

4.1 The SPQR input program. 68
4.2 `ConcurrentSkipListMap.put` test case. 69
4.3 `ConcurrentSkipListMap.putIfAbsent` test case. 70
4.4 `ConcurrentSkipListSet.add` test case. 71
4.5 Test case related to `Arrays.copyOfRange`. 71
4.6 The Pulsar Reporting program input. 72
4.7 Small program used as input. 76

Chapter 1

Introduction

A number of widely used Application Programming Interfaces (APIs) are specified using natural language (e.g. the Java Reflection API, the Java Collections API, and the .NET Reflection API). The Java Reflection API and the Java Collections API are two of the most used Java APIs. At least, 78% of open source Java projects use the Java Reflection API [22], 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. Most projects also use the Java Collections API [21]. In C#, the seventh most popular programming language,[1] a number of popular open source projects use the .NET Reflection API, such as ASP.NET projects (e.g. EntityFrameworkCore), and Microsoft Azure projects (e.g. SDK for .NET). These projects use reflection APIs directly, by invoking methods and properties in source code, or indirectly, by using other tools that use the .NET Reflection API.

Developers may hesitate using an API, given its inherent complexity [28]. Hence, the specification of an API plays a major role on encouraging developers to properly understand and use it in their programs. Otherwise, even experienced developers may misunderstand the behavior of API methods [56; 57; 65]. Specifiers use natural language to specify popular APIs, such as the Java Reflection API, the Java Collections API, and the .NET Reflection API [32]. We do not know whether the specification is imprecise and incomplete and how

[1] https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

those shortcomings affect developers, which may increase development time, hamper source code understanding, and induce developers to introduce non-expected results in their applications. However, developers can still be able to often implement applications that use popular APIs in spite of the specification shortcomings. We do not know whether and how the shortcomings of the specifications impact on developers' understanding.

The reliable use of an 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 [56; 57; 65]. Java Virtual Machine (JVM) developers implement and test the Java Reflection API and the Java Collections API based on a Javadoc [25, p. 37], which is specified using 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.

1.1 Problem

There is limited knowledge on whether API developers systematically reveal underdetermined specifications and non-conformances in their implementations. Following Liskov [26], 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 an API method does not follow its specification. Empirical studies tend to only investigate the frequency [22] and the complexity on the use of an API [60]. They conclude that API users often have to invoke many API methods to implement recurring non-trivial tasks in their programs [22], which makes the use of an API even harder. However, the scenario may be even more worrisome. It may be the case that each 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.

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. Popular test cases generators, such as Randoop [48] and EvoSuite [12], heavily use and depend on some APIs (e.g. Java Reflection API and Java Collections API) in their implementations. However, these tools do not deal with complex objects in parameter objects [61], such as `Class` and `Method`, and do not focus on detecting non-conformances in the Java Reflection API [2].

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 for 66.6% of questions. 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.

We also conduct a survey (Section 2.2.2) with 94 *users* of the .NET Reflection API (i.e. developers who use the API to implement applications) and with 34 *developers* of the .NET Reflection API (i.e. developers of `Mono` and `.NET Core SDK`) to investigate whether the specification of that API impacts on their understanding. We ask API users and developers about the output of seven C# small programs that use a popular API property, and six popular methods. We do not have consensus in 71.4% of questions. Respondents' responses diverging from the most common varied from 6.2% to 45.5%. In a question, only 7.3% of API users and 37.5% of API developers present the same response of `.NET Core SDK` and `Mono`, which are popular tools that implement .NET Reflection API. Some respondents state: *"The API is easy to use, but should be avoided in high performance scenarios"*. Others commented: *"Based on the quirks and gotchas the API is powerful, but of limited use in daily development"*. Those findings suggest that the .NET Reflection API is incomplete. API specifiers and developers need better ways to specify and implement them. The results of both investigations reinforce the need for improving systematic conformance testing of APIs specified in natural language. Moreover, there is limited understanding of how often non-conformances occur and how critical they are.

1.2 Motivating Examples

In this section, we present examples of underdetermined specifications in the Java Reflection API Javadoc and in the .NET Reflection API documentation, and a non-conformance in the Java Collections API. Jsprit is a Java based toolkit for solving rich traveling salesman and vehicle routing problems.[2] Listing 1.1 presents the `Route` enum declaring three options. The `ConstraintManager` class defines the private array field `routes` to store supported routes.

Listing 1.1: Code snippet of the Jsprit project related to routes.

```
1 public enum Route {
2   INTER_ROUTE, INTRA_ROUTE, NO_TYPE
3 }
4 public class ConstraintManager {
5   private Route[] routes;
6 }
```

Consider a test case *t1* invoking the Java Reflection API `Class.getResource` method to retrieve a resource related to the `Route[]` type (Listing 1.2). Suppose `c` is a `Class` object representing the `Route[]` type created to inspect the program of Listing 1.1. 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 natural language.[3] 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. So, we consider that as an underdetermined specification because the Javadoc allows multiple implementations to present different results. Although Eclipse OpenJ9 JVM implements test cases for `Class.getResource`, there is no test case evaluating the resource retrieval passing an empty string as a resource name.

[2]https://github.com/graphhopper/jsprit

[3]https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getResource-java.lang.String-

Listing 1.2: `Class.getResource` test case.

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

According to Landman et al. [22], 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 1.2 may have different behaviors when running on distinct JVMs. We present examples of non-conformances between the Java Reflection API implementations and the Javadoc in Chapter 4.

As another example, consider an empty `ArrayList` as an input program. Listing 1.3 presents a test case *t2* invoking the Java Collections API `ArrayList.ensureCapacity` method with an empty list. Using Eclipse OpenJ9 JVM *0.8.0*, *t2* yields an `OutOfMemoryError`. It crashes the JVM. However, when using Oracle *1.8.0_151* JVM, *t2* does not throw any exception or error, as defined in the `ArrayList.ensureCapacity` method specification. The Java Collections API is also specified in a Javadoc using natural language. Specification defines that the `ArrayList.ensureCapacity` method should increase the capacity of the `ArrayList` instance. Since Javadoc does not define any expected exception when executing `ArrayList.ensureCapacity` method, we consider that as a non-conformance between the Eclipse OpenJ9 implementation and the Javadoc. Läemmel et al. [21] conclude that the Java Collections API is the Java API most used by open source projects. Java applications that contain a code similar to Listings 1.2 and 1.3 may have different behaviors when running on the Oracle and Eclipse OpenJ9 JVMs. We present examples of non-conformances between the Java Collections API implementations and the Javadoc in Chapter 4.

Listing 1.3: `ArrayList.ensureCapacity` test case.

```
1 new ArrayList().ensureCapacity(Integer.MAX_VALUE/10);
```

As a third example, consider the C# program of Listing 1.4. It defines classes `A` (Line 4) and `B` (Line 8). Class `A` defines an `Event` (Line 5) and class `B` extends from `A`. `B.Main` method (Line 9) prints the `ReflectedType` of `Event`'s `AddMethod` (Line 13). The program presents different results whether we execute it using popular .NET tools that imple-

ment the .NET Reflection API. While the result is `B` using `.NET Core SDK 2.1.401`, the program yields `A` when using `Mono 5.4.1`. We expect the result of the program of Listing 1.4 should be `A` because we use an instance of `A` to get the instance of `MemberInfo`. The `MemberInfo.ReflectedType` property returns the class object used to obtain the instance of `MemberInfo` [32]. We report a bug to *.NET Core SDK* maintainers.[4] They claim that the specification of `MemberInfo.ReflectedType` *"...doesn't apply to every API that happens to yield a member info..."*. We ask maintainers where we can find that condition in the specification. They do not answer that, highlighting the incompleteness of the .NET Reflection API specification. So, we also consider that as an underdetermined specification because the specification allows multiple implementations to present different results.

Listing 1.4: Code snippet related to `MemberInfo.ReflectedType` .NET Reflection API property.

```
1  using System;
2  using System.Reflection;
3
4  public class A {
5    public event Action Event { add { } remove { } }
6    }
7
8    public class B : A {
9      public static void Main(string[] args) {
10       Type type = typeof(B);
11       EventInfo eventInfo = type.GetEvent(nameof(A.Event));
12       MemberInfo memberInfo = eventInfo.AddMethod;
13       Console.WriteLine(memberInfo.ReflectedType);
14     }
15   }
```

1.3 Solution

To improve this scenario, we propose a technique [54] to detect underdetermined specifications and non-conformances between the specification and the implementations of APIs

[4] https://github.com/dotnet/corefx/issues/32232

specified in natural language (Chapter 3). Our technique automatically creates test cases for all possible combinations of parameter values received, and executes them in different API implementations to identify differences (i.e. underdetermined specification and non-conformance candidates). We consider a test case as a set of input data, execution commands, and conditions to determine whether a system satisfies the specification [3; 50; 38]. The technique initially considers differences as candidates because we need to check whether they are indeed an underdetermined specification or a non-conformance. 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 four groups: different values, difference between exception thrown and value, different exceptions, and implementation crash. Then, we execute some manual steps to confirm whether an underdetermined specification or non-conformance candidate is indeed a bug, and submit to the API specifiers or developers.

1.4 Evaluation

We evaluate our technique using 446 input programs from GitHub, and four JVMs (Oracle, OpenJDK, Eclipse OpenJ9, and IBM J9) in Chapter 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`, used in 77% of Java open source projects [22]. 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.

We also evaluate our technique using the Java Collections API. Our technique identifies

29 underdetermined specification and non-conformance candidates. All candidates are detected in methods that require primitive types as parameters. A number of 17 candidates cannot be detected by Randoop [48] or EvoSuite [12], popular automatic test suite generators. We report 5 underdetermined specification candidates to the Java Collections API specifiers. We also report 9 non-conformance candidates to Eclipse OpenJ9 JVM, and 4 to Oracle JVM. Oracle JVM developers accept and fix 3 non-conformance candidates. Eclipse OpenJ9 JVM developers accept and fix 1 non-conformance candidate. Our technique helps JVM developers not only to improve the implementation but also to promote discussions about underdetermined specifications in the APIs specifications.

1.5 Contributions

In summary, the main contributions of this work are:

- We analyze the Java Reflection API test suites of widely used JVMs, Eclipse OpenJ9 and OpenJDK (Section 2.1);
- We conduct surveys to investigate whether two popular reflection APIs documentations specified in natural language impacts on developers understanding (Sections 2.2.1 and 2.2.2);
- We propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of APIs specified in natural languages (Chapter 3);
- We report 17 underdetermined specification candidates detected by our technique to API specifiers. They accept 3 candidates. We also report 44 non-conformance candidates detected by our technique to JVM developers. They accept 30 and fix 29 of them. Twelve test cases are now part of the Eclipse OpenJ9 JVM test suite (Chapter 4).

This thesis is organized as follows. Chapter 2 presents an analysis of OpenJDK Java Reflection API test suite and results of two surveys that investigate the impact of developers understanding about specification. Chapter 3 describes a technique to detect underdetermined specifications and non-conformances in APIs specified in natural language. Chapter 4

evaluates our technique in the Java Reflection API and in the Java Collections API. Chapter 5 presents related work. Finally, Chapter 6 summarizes the contributions of the thesis and presents future work.

Chapter 2

Relevance

Most software development employs libraries and frameworks whose functionality is made available through APIs [65]. In Java, specifiers define most APIs in Javadocs, which are specified in natural language. The Java Persistence API is a Java specification that describes the management of relational data in applications. The Java Cryptography API enables you to encrypt and decrypt data in Java, as well as manage keys, sign and authenticate messages. In this thesis we study the Java Reflection API and the Java Collections API.

According to Landman et al. [22], a number of 78% of open source Java projects use the Java Reflection API. To conclude that, the authors perform an analysis of the frequency of using methods of the Java Reflection API in real applications. From a collection of 20 thousand projects, they applied the Software Projects Sampling (SPS) [40] tool and arrived at a set of 461 projects representing 99.47% of the entire sample. Authors consider Abstract Syntactic Trees (AST) [51] of the source code to analyze the Java Reflection API methods usage.

Table 2.1 summarizes data related to the Java Reflection API methods usage. Table 2.1 presents method category, a method example, and the usage frequency of the methods in analyzed projects. Most used Java Reflection API methods (95%) are related to getting references to meta objects (e.g. `Class.getClass`). It is an expected result because developers need to use `Class` class to get instances to other Java Reflection API classes, like `Field`, `Method`, and so on. Eighty-five percent of projects use textual representation methods (e.g. `Method.toString`). Eighty percent of projects invoke methods to inspect a program and get methods, fields, classes, and parameters.

Table 2.1: Java Reflection API methods usage. Content derived from Landman et al. [22].

Category	Method Example	Usage
References to meta objects	Class.getClass	95%
Textual representation	Constructor.toString	85%
References to other objects	Class.getDeclaredFields	80%
Signature	Method.getModifiers	80%
Resources retrieval	Class.getResource	71%
Class loading	Class.forName	70%
Objects instantiating	Class.newInstance	65%
Methods invocation	Method.invoke	60%
Field values retrieval	Field.get	45%
Access modifiers changing	Method.setAccessible	40%
Arrays manipulation	Array.set	35%
Fields values changing	Field.set	30%
Proxy instances instantiating	Proxy.newProxyInstance	20%
Annotations	Parameter.getAnnotations	19%
Security	Class.getSigners	15%
Classes assertions	Class.desiredAssertionStatus	2%

Lämmel et al. [21] describe an approach to large-scale API-usage analysis of open source Java projects. Authors use AST to get the numbers of distinct APIs used in a project, and the percentage of methods of an API used in analyzed projects. The Java Collections API is the most used API. A number of 1,374 projects use the Java Collections API. Lämmel et al. identify calls to 406 distinct methods.

In this chapter, we illustrate and characterize the problem from three perspectives: (a) an analysis of test suites of widely used JVMs showing that there is no systematic strategy to check conformance between the specification and implementations of the Java Reflection API (Section 2.1); (b) a survey of developers who use the Java Reflection API that demonstrates diverging understanding on the specification of popular Java Reflection API methods (Section 2.2.1); and (c) a survey (Section 2.2.2) with 94 developers of the most active C# projects on GitHub (users of the .NET Reflection API) and 34 developers of popular implementations (i.e. .NET Core SDK and Mono) of the .NET Reflection API to investigate whether the specification of .NET Reflection API impacts on their understanding and what is the magnitude of that.

2.1 Java Reflection APIs Test Suites

To better understand how API developers deal with the problem presented in Section 1.1, 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. We are interested in identifying strategies used by open source Java Reflection API developers to implement test cases. We look for random order and multiple test cases execution, occurrences of test cases that use a single input program to invoke all possible Java Reflection API methods, rules to choose values used to invoke them, and whether those test cases do differential testing [30]. 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[1]

[1] https://github.com/AdoptOpenJDK/openjdk-jdk8u

and the *openj9* branch commit *c2aa034* of Eclipse OpenJ9 source code repository.[2] 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 [22]. We identify test cases invoking 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 [45]). Some test cases invoke 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. Listing 2.1 presents a test case (*test_getMethod*) of Eclipse OpenJ9 JVM that tests `Class.getMethod` and `Method.invoke` Java Reflection API methods. The input program *ClassTest* (Line 4) defines a public method *pubMethod* and a private method *privMethod*. Test case *test_getMethod* gets *pubMethod* (Line 16) and asserts the returned value after invoked it (Line 17). Then, the test case tries to get *privMethod* (Line 19) and expects a `NoSuchMethodException` because method is private and can not be retrieved using `Class.getMethod` method.

[2] https://github.com/ibmruntimes/openj9-openjdk-jdk8

Listing 2.1: Test case of Eclipse OpenJ9 JVM using two oracles.

```
1
2  public class Test_Class {
3    ...
4    public static class ClassTest {
5    ...
6      public int pubMethod(){
7        return 2;
8      }
9      private int privMethod(){
10       return 1;
11     }
12   ...
13   }
14   @Test
15   public void test_getMethod() {
16     Method m = ClassTest.class.getMethod("pubMethod");
17     AssertJUnit.assertTrue(((Integer) (m.invoke(new ClassTest()))).
         intValue() == 2);
18     try {
19       m = ClassTest.class.getMethod("privMethod");
20     } catch (NoSuchMethodException e) {
21       //Correct
22       return;
23     }
24     AssertJUnit.assertTrue("Failed to throw exception accessing private
         method", false);
25   }
26   ...
27 }
```

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. On the other hand, there is a similar test in OpenJDK JVM (Listing 2.2). However, developers do not consider a strategy to implement or generate different input programs. 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) [25, p. 80] specifies *"An array type descriptor is valid only if it represents 255 or fewer dimensions."*

Listing 2.2: `Class.getResource` OpenJDK test case.

```
1  public class SourceTest {
2   ...
3   @Test
4   public void testURLReaderSource() {
5       try {
6        System.err.println(SourceTest.class.getResource(""));
7        ...
8       catch (final IOException e) {
9        fail(e.toString());
10      }
11  }
12 }
```

2.2 Surveys

In this section, we present results of two surveys that investigate whether the specification in natural language impacts on the developer's understanding. We investigate whether underdetermined specifications, similar to the one presented in Section 1.2, actually impact on the understanding of developers who use an API specified in natural language. The complete results are available at our website [53].

We consider the same choices as Nadi et al. [39] to define the background of developers who answered our surveys. The choices related to APIs knowledge are: i) *Not knowledgeable – I do not know anything about it*; ii) *Somewhat knowledgeable – I have a vague idea about it*; iii) *Knowledgeable – I am familiar with it*; and iv) *Very knowledgeable – I know all or most classes and methods of it*. To define developers APIs usage, we consider the choices: i) *Never – I never used the API to develop applications*; ii) *Sometimes – I need reflection for less than 33% of the software applications I develop*; iii) *Occasionally – I use reflection in more than 33% but less than 66% of the software applications I develop*; iv) *Frequently – I need reflection for more than 66% of the software applications I develop*.

2.2.1 The Java Reflection API Survey

We present results of a survey about the Java Reflection API we conduct with Java developers. We send e-mails to 3,500 randomly selected GitHub developers.

Planning

We ask three questions about methods (`Class.getDeclaredMethods`, `Class.getMethod`, and `Class.getDeclaredFields`) used in 77% of Java open source projects [22]. Figure 2.1 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. We do not consider responses from developers who do not have experience developing Java applications, knowledge of the Java Reflection API, or that never used the Java Reflection API to develop applications.

Class.getDeclaredMethods() returns an array containing Method objects reflecting all the declared methods of the class or interface represented by this Class object, including public, protected, default (package) access, and private methods, but excluding inherited methods.

```
public interface A {
  public A clone();
}
```

What is the result of getDeclaredMethods() for interface "A"?

- ○ public abstract A A.clone() and public default Object A.clone()
- ○ public abstract A A.clone()
- ○ public default Object A.clone()
- ○ No declared methods
- ○ Other...

Figure 2.1: Question 1 about `Class.getDeclaredMethods`.

Results

Overall, 130 (3.6%) developers answered the survey, which is the usual response rate for surveys of this kind [31; 39]. We do not consider 17 responses from developers that never used the Java Reflection API to develop applications. 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.

Figure 2.2 presents response rates related to developers experience developing Java applications. Almost 68% of the developers have more than seven years of experience with Java. Figure 2.3 shows response rates of knowledge about the Java Reflection API. Most of the survey respondents (86.9%) are familiar or know all or most of the Java Reflection API classes and methods. Figure 2.4 presents response rates of frequency of usage of the Java Reflection API to develop applications. A number of 62.3% of developers need the Java Reflection API for less than 33% of software applications their develop.

Next, we explain results of all questions in our survey. The `Class.getDeclaredMethods` method returns all declared methods of a class, excluding inherited ones [43]. Figure 2.1 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`. Most developers (79.3%) answer

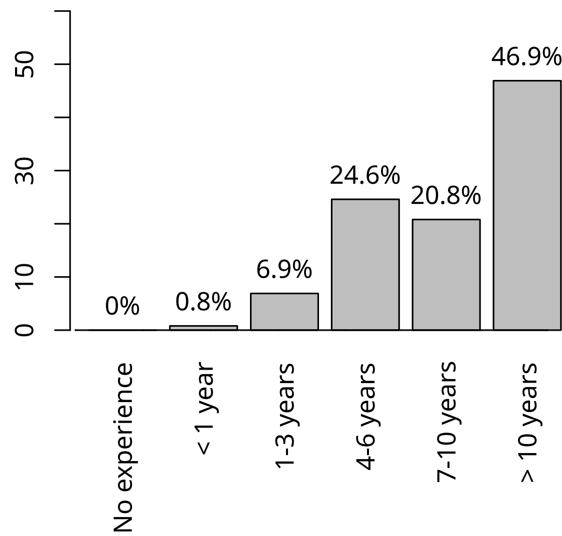

Figure 2.2: Developers experience with Java.

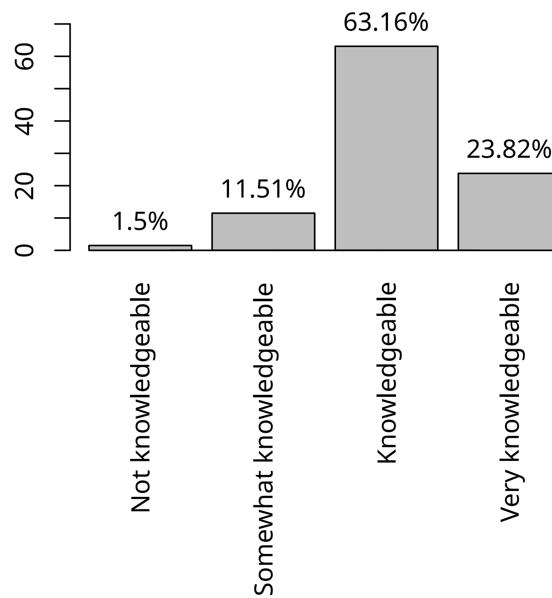

Figure 2.3: Developers knowledge about the Java Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it.

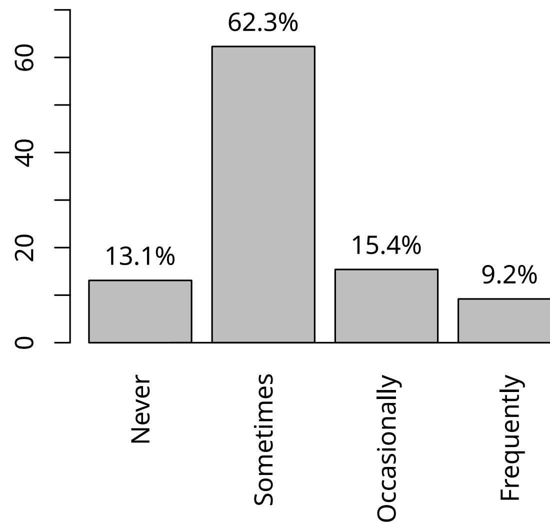

Figure 2.4: Developers using the Java Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop.

`public abstract A A.clone()`. However, others (21.7%) disagree on that (Figure 2.5). 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 [44]. Question 2 presents three classes to developers (Listing 2.3). 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 C.c`, while others (27.6%) answer `public void A.c`. Figure 2.6 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.

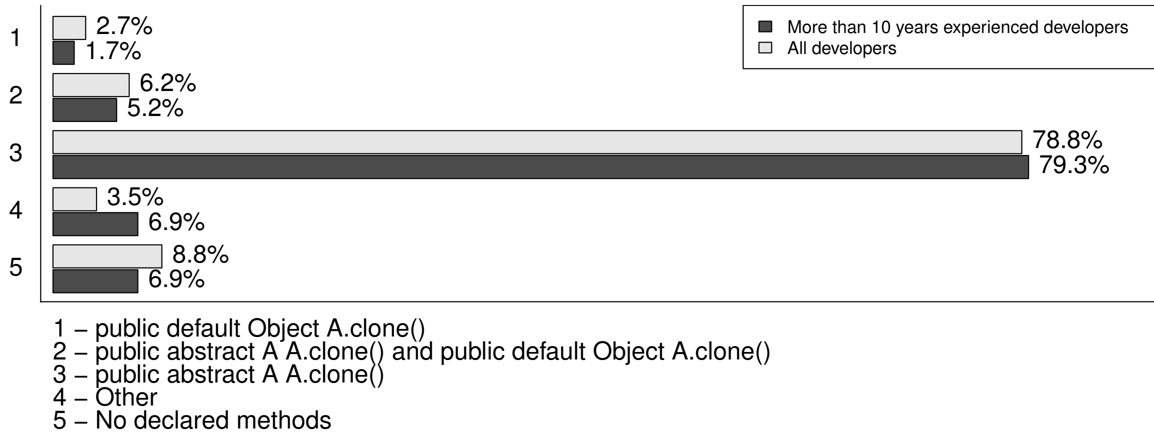

Figure 2.5: Results of Question 1: `A.getDeclaredMethods`.

Listing 2.3: Program of Question 2.

```
1  public class A extends B {
2  }
3
4  public class B extends C {
5  }
6
7  class C {
8      public void c() {
9      }
10 }
```

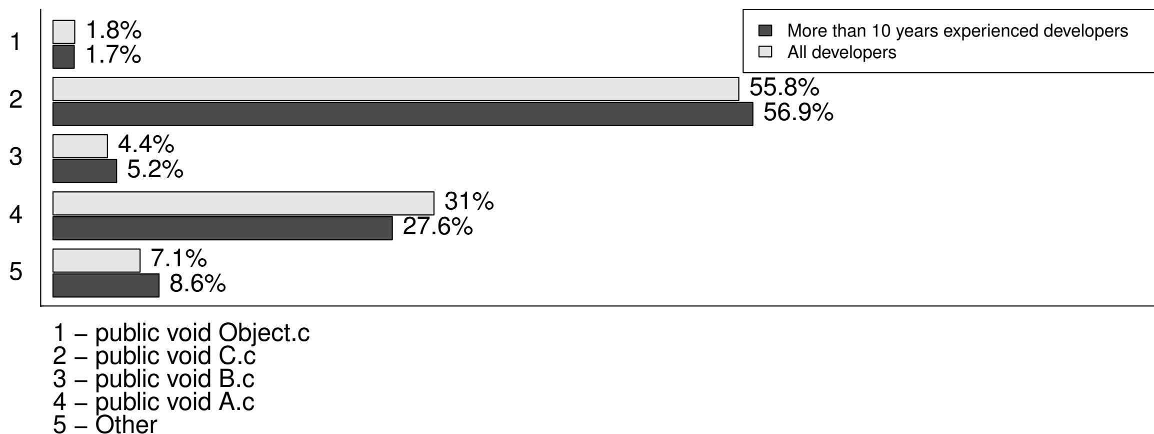

Figure 2.6: Results of Question 2: `A.getMethod c`.

The `Class.getDeclaredFields` method returns all declared fields of a class, ex-

cluding inherited ones [42]. Listing 2.4 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.

Listing 2.4: Program of Question 3.

```
1  public final class A extends B {
2      public void a(C c) {
3          switch (c) {
4              case X:
5          }
6      }
7  }
8
9  public class B {
10     public enum C {
11         X
12     }
13     protected C c;
14 }
```

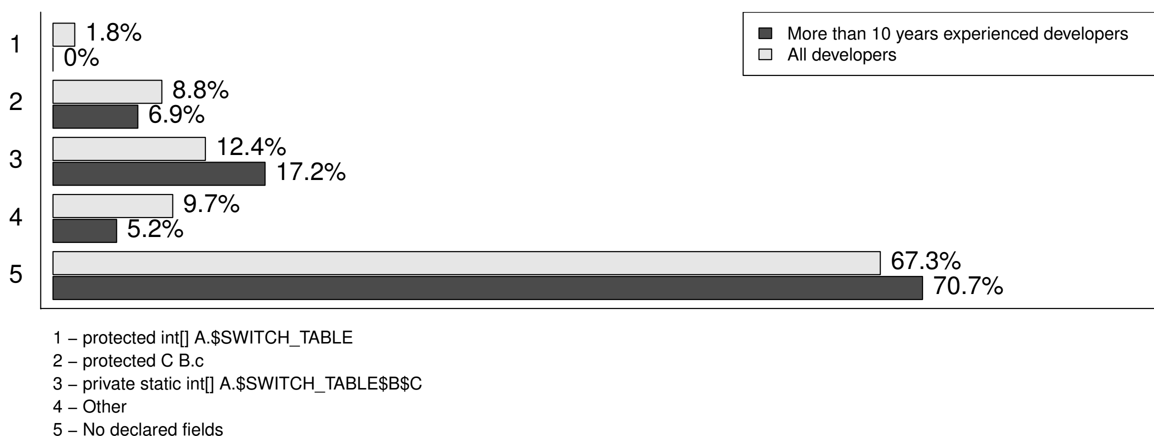

Figure 2.7: Results of Question 3: A.getDeclaredFields.

2.2.2 The .NET Reflection API Survey

We present results of a survey about the .NET Reflection API we conduct with C# developers. We invite API users from the 25 most active C# projects of GitHub, including projects from Unity, Microsoft, and Azure. We also invite developers of `Mono` and `.NET Core SDK`, which are popular tools that implement the .NET Reflection API. We send e-mails to 1,507 randomly selected users, and to 1,134 API developers.

Planning

We divide the survey into three sections. The first section asks API users and developers about their C# experience, and .NET Reflection API knowledge. The second section presents questions related to a popular property and some popular methods of .NET Reflection API. For each question, we present a small program (13–20 source lines of code, SLOC), a link to a .NET Reflection API method or property specification, and we ask the output of that program. Figure 2.8 shows a question of our survey, in this case related to `MemberInfo.ReflectedType` property. The third section asks API users and developers for additional comments. We do not consider responses from developers who do not have experience developing C# applications, knowledge of the .NET Reflection API, or that never used the .NET Reflection API to develop applications.

Results

Overall, 128 (4.8%) API users and developers answered the survey, which is the usual response rate for surveys of this kind [31]. We have 94 responses from users, and 34 responses from developers of the .NET Reflection API. We do not consider 11 users' responses and 7 developers' responses that never used the .NET Reflection API to develop applications. Figure 2.9 presents response rates related to developers experience developing C# applications. Forty-four (46.8%) API users have more than 10 years of experience in C#, and 77.6% of them have knowledge about the .NET Reflection API. About 69% of API users have more than seven years of experience with C#. Figure 2.10 shows response rates of API users knowledge about the .NET Reflection API. Most API users (60.6%) are familiar with the .NET Reflection API. Figure 2.11 presents response rates of frequency of usage of the

Consider the following program:

```
1  using System;
2  using System.Reflection;
3
4  public class A {
5    public event Action Event { add { } remove { } }
6  }
7
8  public class B : A {
9    public static void Main(string[] args) {
10     Type type = typeof(B);
11     EventInfo eventInfo = type.GetEvent(nameof(A.Event));
12     MemberInfo memberInfo = eventInfo.AddMethod;
13     Console.WriteLine(memberInfo.ReflectedType);
14   }
15 }
```

What is the output of the above program? See documentation (https://goo.gl/TD3aMz).

◯ A

◯ B

◯ Event

◯ Null

◯ Other: [ ]

Figure 2.8: Question 1 of our survey.

.NET Reflection API to develop applications. A number of 50% of API users need the .NET Reflection API for less than 33% of software applications they develop.

Figure 2.12 presents response rates related to API developers experience developing C# applications. About 73.5% of developers have more than seven years of experience with C#. Seventeen (50%) API developers have more than 10 years of experience in C#. Figure 2.13 shows response rates of developers knowledge about the .NET Reflection API. Fourteen API developers (44.1%) are familiar with the .NET Reflection API, and 85.3% of them have knowledge about the .NET Reflection API. Figure 2.14 presents response rates of frequency of usage of the .NET Reflection API to develop applications. A number of 41.1% of developers need the .NET Reflection API at least for 33% of software applications they develop.

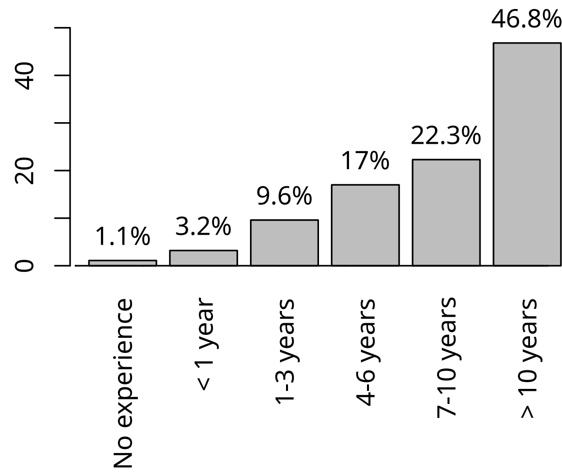

Figure 2.9: .NET Reflection API users experience with C#.

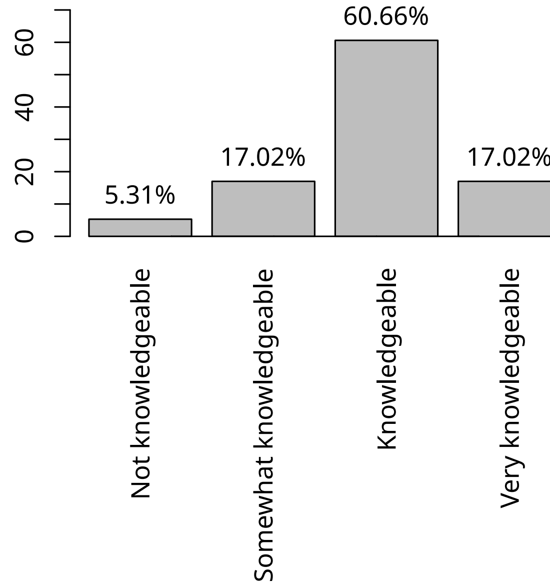

Figure 2.10: Users knowledge about the .NET Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it.

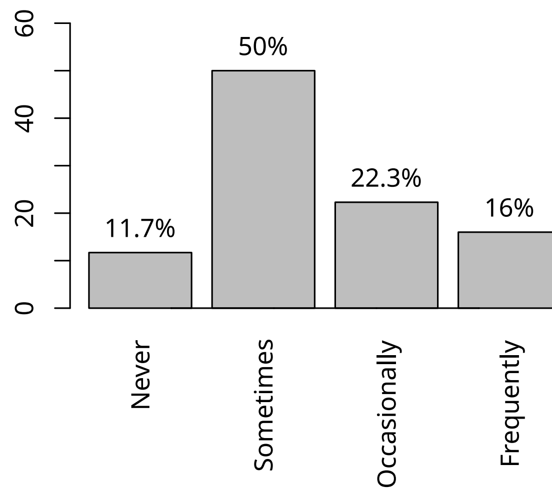

Figure 2.11: Users using the .NET Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop.

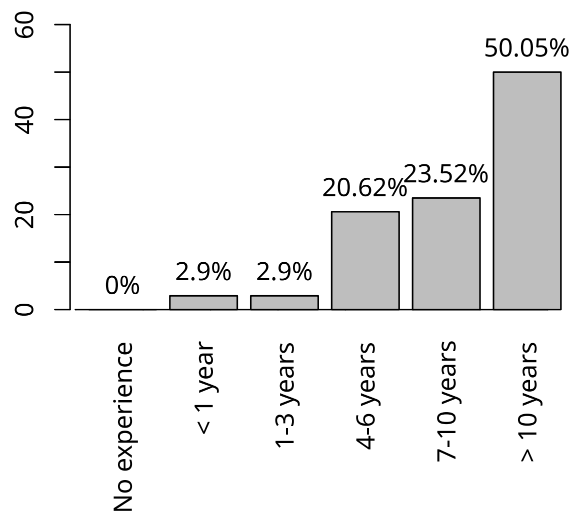

Figure 2.12: .NET Reflection API developers experience with C#.

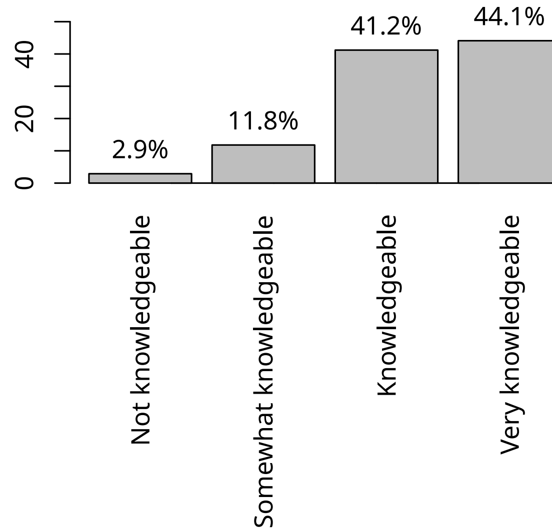

Figure 2.13: Developers knowledge about the .NET Reflection API. Not knowledgeable - I do not know anything about it; Somewhat knowledgeable - I have a vague idea about it; Knowledgeable - I am familiar with it; Very knowledgeable - I know all/most classes and methods of it.

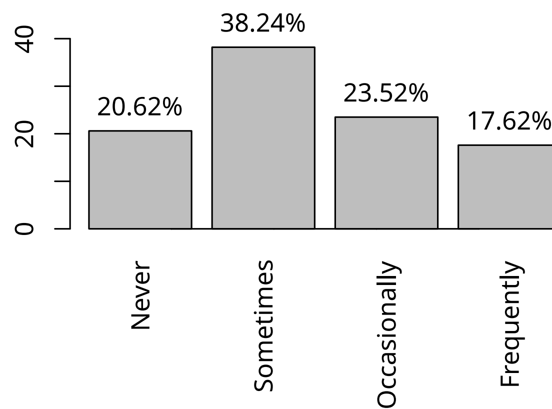

Figure 2.14: Developers using the .NET Reflection API to develop applications. Sometimes - I need reflection for less than 33% of the software applications I develop; Occasionally - I use reflection in more than 33% but less than 66% of the software applications I develop; Frequently - I need reflection for more than 66% of the software applications I develop.

The `MemberInfo.ReflectedType` property returns the class object used to obtain the instance of `MemberInfo` [32]. The program used in Question 1 of survey (Figure 2.8) defines classes `A` (Line 4) and `B` (Line 8). Class `A` defines an `Event` (Line 5) and class `B` extends from `A`. `Main` method (Line 9) in class `B` prints the `ReflectedType` of `Event`'s `AddMethod` (Line 13). Figure 2.15 presents results for responses of the .NET Reflection API users and developers. Almost two thirds of the API users (65.9%) answer `B`, while others (26.8%) answer `A`. Most API developers (56.2%) answer `B`, while others (25%) answer `A`. The program presents `B` as result when we execute it using `.NET Core SDK 2.1.401`, and `A` when using `Mono 5.4.1`. We expect the result of the program of Figure 2.8 should be `A` because we use an instance of `A` to get the instance of `MemberInfo`. We report a bug to *.NET Core SDK* maintainers.[3] They claim that the specification of `MemberInfo.ReflectedType` *"...doesn't apply to every API that happens to yield a member info..."*. We can not find that condition in the specification, highlighting the incompleteness of the .NET Reflection API specification.

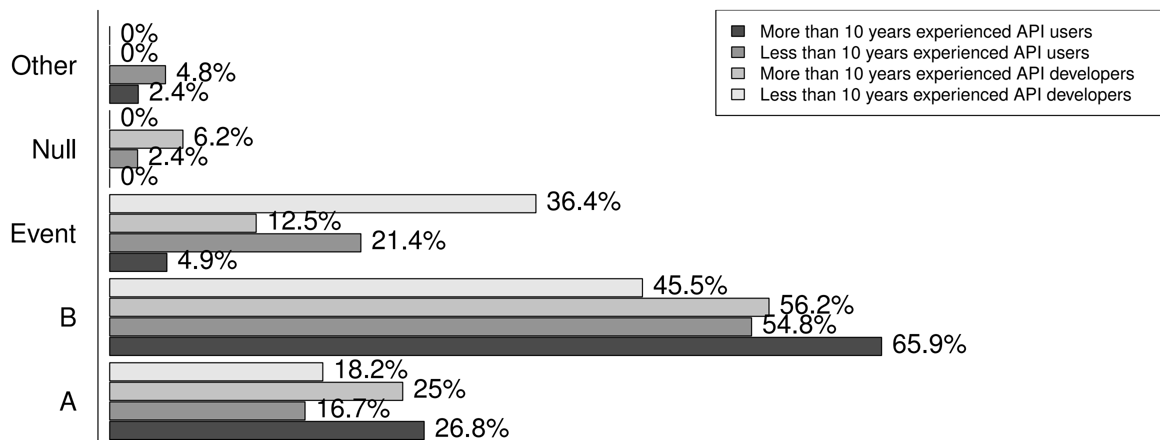

Figure 2.15: Question 1: `MemberInfo.ReflectedType`.

`ConstructorInfo.Invoke` invokes the constructor reflected by an instance. The method throws a `TargetInvocationException` when the invoked constructor throws an exception [32]. We ask developers the result of invoking a constructor of `object[]` method informing `-1` as parameter (Listing 2.5). Figure 2.16 presents results for responses of the .NET Reflection API users and developers. Some API users (19.5%) answer `OverflowException`, while others (58.5%) an-

[3] `https://github.com/dotnet/corefx/issues/32232`

swer `TargetInvocationException`. A number of API developers (25%) answer `OverflowException`, while others (50%) answer `TargetInvocationException`. Executing the program of Listing 2.5 in `.NET Core SDK 2.1.401` yields `OverflowException`. However, when using `Mono 5.4.1`, the program yields `TargetInvocationException`. We report a bug to *.NET Core SDK* maintainers.[4] They rejected it and claim that: *"The constructors exposed by arrays have special code paths and the TargetInvokeException wrapping was overlooked for those paths. We keep the behavior now for compatibility sake"*. We are not able to find that behavior in the .NET Reflection API specification.

Listing 2.5: Program used in Question 2 concerning the `ConstructorInfo.Invoke` method.

```
1  using System;
2  using System.Linq;
3  using System.Reflection;
4  class A {
5    static void Main() {
6      TypeInfo t = typeof(object[]).GetTypeInfo();
7      ConstructorInfo c = t.DeclaredConstructors.ToArray()[0];
8      Console.WriteLine(c.Invoke(new object[]{-1}));
9    }
10 }
```

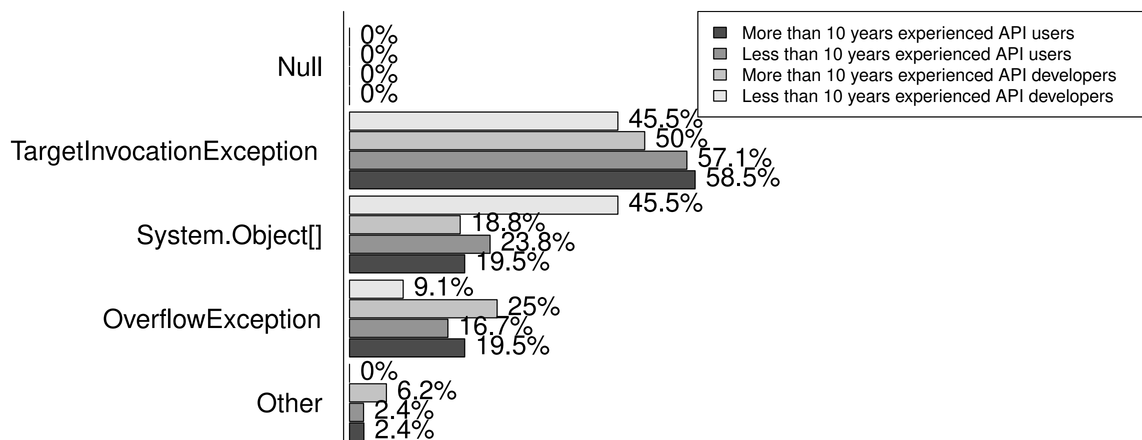

Figure 2.16: Question 2: `ConstructorInfo.Invoke`.

[4] https://github.com/dotnet/corefx/issues/32231

Type.FindMembers returns a filtered array of MemberInfo objects of the specified member type. API specifiers do not define the order of members returned by Type.FindMembers [32]. Figure 2.17 presents results for responses of the .NET Reflection API users and developers. Listing 2.6 presents the program used in Question 3 of the survey. It defines a class A (Line 4) containing Member1 (Line 5) field and Member2 (Line 6) method. Main method (Line 8) invokes Type.FindMembers (Line 10) method to retrieve members of class A. We expect returned order should be the same as declared in source code (i.e. *Member1, Member2*). Some API users (7.3%) answer Member2, Member1, while 92.7% of users answer a different result. Most API developers (62.5%) answer Member1, Member2. Both .NET Core SDK 2.1.401 and Mono 5.4.1 present the same result (Member2, Member1). Results ordering seems to matter to some people.[5] Some other .NET Reflection API methods do not return results considering an specific order (e.g. Type.GetFields). Respondents of our survey with more than 10 years of experience in C# send us comments stating that there is an absence of the members ordering policy in the specification. *"I would hope that this would be implementation specific, since the documentation doesn't give an order"* and *"Particular order is not guaranteed"*.

Listing 2.6: Program used in Question 3 concerning the Type.FindMembers method.

```
1  using System;
2  using System.Reflection;
3  class A {
4    public static int Member1 = 0;
5    public static void Member2() { }
6    static void Main() {
7      Type type = typeof(A);
8      MemberInfo[] members = type.FindMembers(...);
9      for (int i = 0; i < members.Length; i++) {
10       Console.WriteLine(members[i]);
11     }
12   }
13 }
```

MethodInfo.MakeGenericMethod substitutes the current generic method definition, and returns a MethodInfo object representing the resulting constructed method [32].

[5] https://github.com/dotnet/corefx/issues/14606

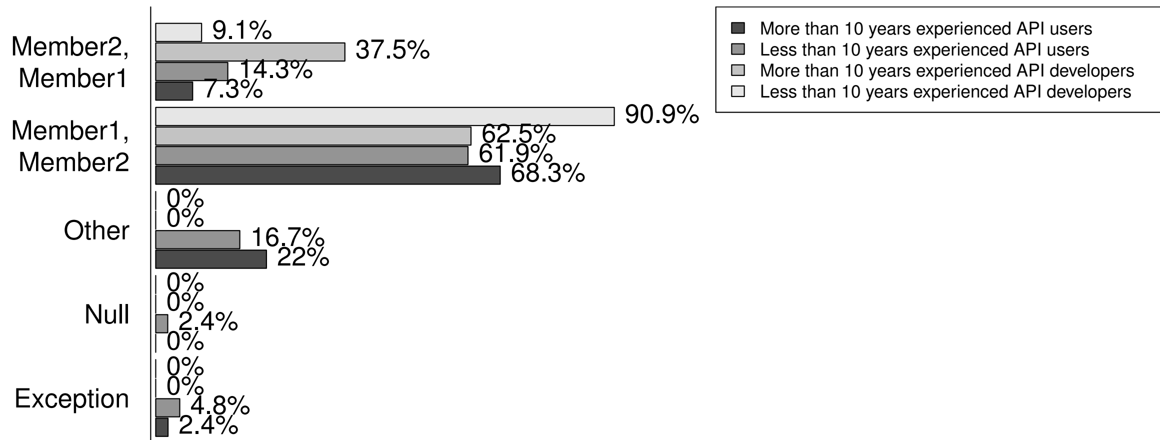

Figure 2.17: Question 3: `Type.FindMembers`.

Listing 2.7 presents the program used in Question 4 of the survey. Figure 2.18 presents results for responses of the .NET Reflection API users and developers. It defines a class `A` (Line 3) containing `M<T>` (Line 5) method and a class `B` (Line 8) extending from class `A` and containing `Main` method (Line 10). `Main` method invokes `MethodInfo.MakeGenericMethod` (Line 14) method to substitute the generic definition of `M<T>` method. We expect result should be *M[Int32]*. Most API users (90.2%) answer `M[Int32]`, while other users answer a different result. Most API developers (93.8%) answer `M[Int32]`. Both `.NET Core SDK 2.1.401` and `Mono 5.4.1` present the same result (`M[Int32]`).

Listing 2.7: Program used in Question 4 concerning the `MethodInfo.MakeGenericMethod` method.

```
1  using System;
2  using System.Reflection;
3  public class A
4  {
5      public void M<T>() { }
6  }
7
8  public class B : A
9  {
10     public static void Main(string[] args)
11     {
12         Type t = typeof(B);
13         MethodInfo method = t.GetMethod("M");
14         Console.WriteLine(method.MakeGenericMethod(typeof(int)));
15     }
16 }
```

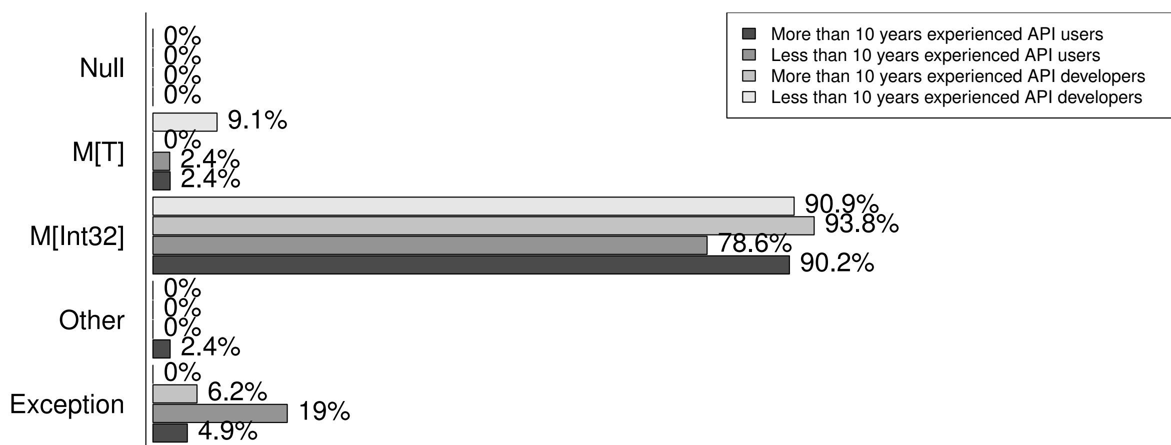

Figure 2.18: Question 4: `MethodInfo.MakeGenericMethod`.

`Assembly.CreateInstance` method locates a type from this assembly and creates an instance of it. The specification defines that the method should throw an `ArgumentException` if argument informed is an empty string (“”). However, API specifiers do not specify what to return when one passes a blank character (“ ”) as argument [32]. Listing 2.8 presents the program used in Question 5 of the survey. It defines a class `A` (Line

4) containing a Main method (Line 5). The program tries to create an instance of class A invoking Assembly.CreateInstance method with a blank character (Line 7). Figure 2.19 presents results for responses of the .NET Reflection API users and developers. Some API users (46.3%) state the result is Null, while others (51.2%) think It throws an exception. On the other hand, a half of API developers state the result is Null, while the other half think It throws an exception. Both *.NET Core SDK 2.1.401* and *Mono 5.4.1* yield Null for program of Listing 2.8. According to the specification, that program should throw an ArgumentException, similar to the result returned when we pass an empty string as argument. Question 5 presents the highest disagreement rate among developers.

Listing 2.8: Program used in Question 5 related to Assembly.CreateInstance method.

```
1  using System;
2  using System.Reflection;
3  public class A {
4    public static void Main() {
5      Assembly assem = typeof(A).Assembly;
6      A a = (A)assem.CreateInstance(‘‘ ’’);
7      if (a == null) {
8        Console.WriteLine(‘‘Null’’);
9      } else {
10       Console.WriteLine(a);
11     }
12   }
13 }
```

Type.GetField searches for the public field with the specified name [32]. Listing 2.9 presents the program used in Question 6 of the survey. It defines a class A (Line 3) containing a Field field (Line 5) and a Main method (Line 6). The program tries to get the field "" of class A (Line 9). Figure 2.20 presents results for responses of the .NET Reflection API users and developers. Some API users (73.2%) state the result is Null, while others (24.4%) think It throws an exception. On the other hand, a number of 93.8% of API developers state the result is Null, while only 6.2% think It throws an exception. Both *.NET Core SDK 2.1.401* and *Mono 5.4.1* yield Null for program of Listing 2.9. According to the

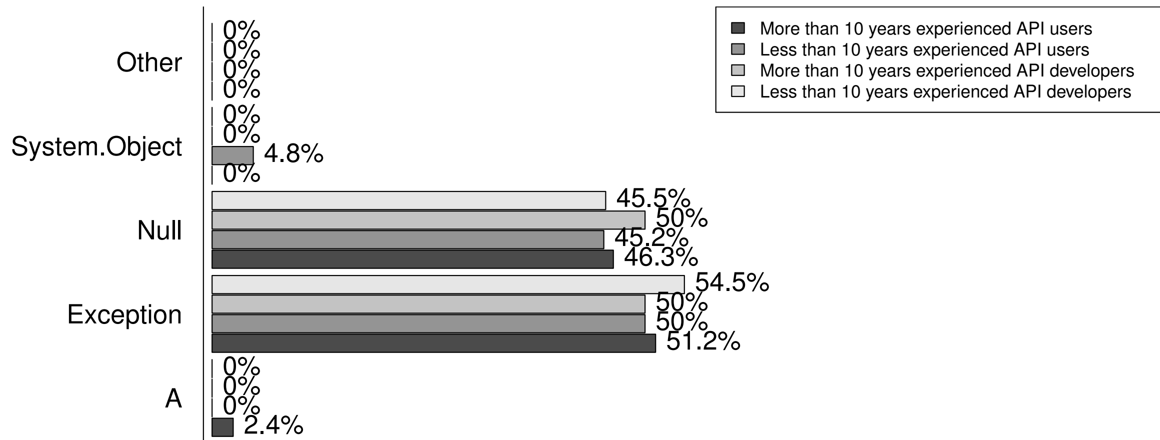

Figure 2.19: Question 5: `Assembly.CreateInstance`.

specification, that program should print `Null`.

Listing 2.9: Program used in Question 6 concerning the `Type.GetField` method.

```
1  using System;
2  using System.Reflection;
3  public class A
4  {
5      public int Field;
6      public static void Main()
7      {
8          Type type = typeof(A);
9          FieldInfo field = type.GetField("");
10         if (field == null)
11         {
12             Console.WriteLine("Null");
13         }
14         else
15         {
16             Console.WriteLine(field);
17         }
18     }
19 }
```

`Type.GetMember` searches for the public members with the specified name [32]. Listing 2.10 presents the program used in Question 7 of the survey. It defines a class `A` (Line 3) containing a `M(int p)` method (Line 5), and a class `B` extending from class `A` and defin-

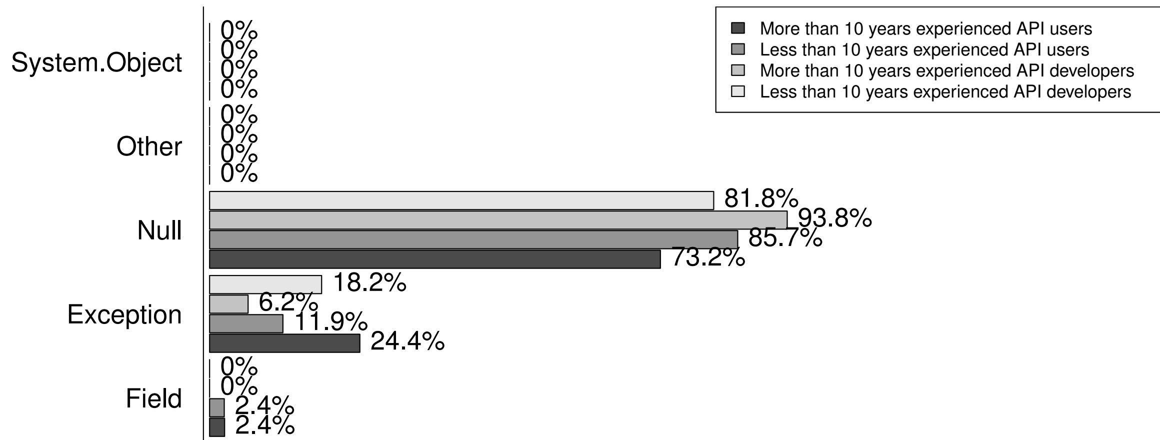

Figure 2.20: Question 6: `Type.GetField`.

ing a `M(string p)` method (Line 9). The program tries to get the method "`M`" of class `B` (Line 13). Figure 2.21 presents results for responses of the .NET Reflection API users and developers. Most API users (65.9%) state the result is `M(String) and M(Int32)`, while others (26.8%) think it is `M(String)`. A number of 62.5% of API developers state the result is `M(String) and M(Int32)`, while 31.2% think it is `M(String)`. Both *.NET Core SDK 2.1.401* and *Mono 5.4.1* yield `M(String) and M(Int32)` for program of Listing 2.10. The specification defines nothing on whether the members of a superclass must also be returned.

Listing 2.10: Program used in Question 7 concerning the `Type.GetMember` method.

```
1  using System;
2  using System.Reflection;
3  public class A
4  {
5      public void M(int p) {}
6  }
7  public class B : A
8  {
9      public void M(string p) {}
10     public static void Main()
11     {
12         Type type = typeof(B);
13         MemberInfo[] membersArray = type.GetMember("M");
14         for (int index = 0; index < membersArray.Length; index++)
15         {
16             Console.WriteLine(membersArray[index].ToString());
17         }
18     }
19 }
```

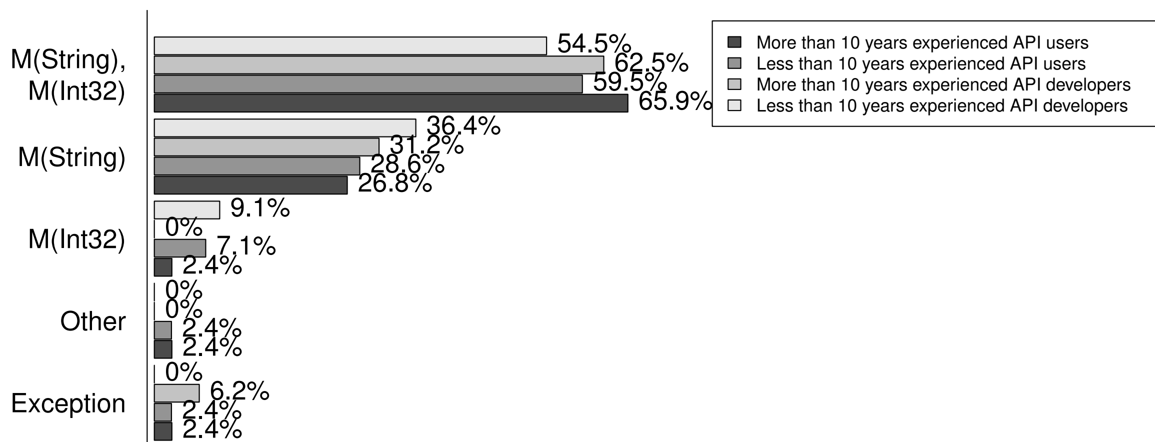

Figure 2.21: Question 7: `Type.GetMember`.

2.2.3 Discussion

Overall, the number of different answers of the Java Reflection API Survey 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 Java Reflection API in the open text box. A developer states *"I would strongly recommend not using the reflection API in an application. It is a useful API to build a library or framework but isn't something you would want to use directly"*. An experienced developer does not recommend the use of the Java Reflection API to develop applications other than libraries and frameworks. We send the results of our survey to developers. They highlighted the importance of fixing bugs in the Java Reflection API.

`Assembly.CreateInstance` is the method that has more disagreement among users and developers (Figure 2.19) of the .NET Reflection API Survey. The question about `MethodInfo.MakeGenericMethod` method provides more agreement among users and developers (Figure 2.18). API users provide the highest number of different responses (nine) to the question concerning the members ordering returned by `Type.FindMembers` method (Figure 2.17). API developers present at most four different responses for the questions related to `MemberInfo.ReflectedType` and `ConstructorInfo.Invoke` methods.

Impact of Developers Experience in Surveys Responses

We conduct a Chi-Squared Test [14] with 0.05 as significance level to investigate whether developers experience impacts on their responses. We use `chisq.test` R [55] function using default parameters to execute Chi-Squared test.

Reflection API. To investigate on whether developers experience with Java are related to their survey responses, we divide responses in two groups, such as: experienced developers (more than 10 years of experience with Java), and non-experienced developers (less than or equal to 10 years of experience with Java). To test whether the experience with Java is associated with their responses in all questions of our survey, we consider that *responses do not depend on developers experience with Java* as null hypothesis (H_0), and that *responses do depend on developers experience* as alternative hypothesis (H_1).

Table 2.2 presents the *p–value* returned by Chi-Squared test for each question. We statistically conclude that experience of API users is not associated with their responses to our survey. We can not reject the H_0 null hypothesis ($p - value > 0.05$) for all questions. We detect some underdetermined specifications in the Java Reflection API (Chapter 4). In this case, experience with Java of respondent of our survey do not help them to read the specification of Java Reflection API and to answer our survey.

Table 2.2: Chi-squared test results comparing responses of experienced and non-experienced developers.

Question	p–value
1	0.2414
2	0.2202
3	0.2202

Collections API. We are interested in investigate whether: i) experience with C# of API users is associated with their responses; ii) experience with C# of API developers is associated with their responses; and iii) API users' responses are associated with API developers' responses. To test whether the experience with C# of API users and developers is associated with their responses in all questions of our survey, we consider *API Users/API developers responses are independent of their experience (H_0a/H_0b)* as null hypothesis, and *API Users/API developers responses are dependent of their experience (H_1a/H_1b)* as alternative hypothesis. We consider *API users and developers responses are independent (H_0c)* as null hypothesis, and *API users and developers responses are dependent (H_1c)* as alternative hypothesis, to test whether responses of API users are associated with responses of API developers in all questions of our survey.

Tables 2.3, 2.4, and 2.5 present the *p–value* returned by Chi-Squared test for each question. We statistically conclude that experience of API users is not associated with their responses to our survey for all questions. We can not reject the H_0a null hypothesis ($p - value > 0.05$). Chi-Squared Test results also show that experience of API developers is not associated with their responses in five of the survey questions. We reject the H_0b null hypothesis ($p - value < 0.05$) for developers' responses to Questions 3, and 6 of our sur-

vey. Finally, we conclude that API users' responses are not associated with API developers' responses. Results of statistical test do not reject the H_{0C} null hypothesis for all questions of our survey. We have some evidence that specification of the .NET Reflection API is incomplete. In this case, experience with C# and the kind of respondent of our survey do not help them to read the specification of .NET Reflection API and to answer our survey.

Table 2.3: Chi-squared test results comparing responses of experienced and non-experienced users.

Question	p–value
1	0.22020
2	0.24140
3	0.22020
4	0.25900
5	0.25900
6	0.09094
7	0.09094

Table 2.4: Chi-squared test results comparing responses of experienced and non-experienced developers.

Question	p–value
1	0.24140
2	0.26500
3	0.04043
4	0.23490
5	0.08208
6	0.04043
7	0.25900

Table 2.5: Chi-squared test results comparing responses of users and developers.

Question	p–value
1	0.2414
2	0.2202
3	0.2650
4	0.2650
5	0.2650
6	0.1247
7	0.2590

Consensus in Surveys Responses

To better investigate the extent of consensus with developers responses, we train a model to predict responses to our surveys using six supervised machine learning classifiers (i.e. SVM, Decision Tree, Random Forest, kNN, Naive Bayes, and Logistic Regression) [13]. Independent variables are experience and whether the respondent is an API developer or user. Dependent variable is respondent's answer. We consider that there is consensus in the responses of a question if the mean accuracy presented by Machine Learning classifiers is greater than 80% [20]. On the other hand, if mean accuracy of a model is less than 80%, we find evidence that there are issues on the API specification. In this case, API specifiers should not present all possible parameters and programming language constructs when defining a specification of a method. The lack of consensus may help understand that experience with the programming language and the kind of a respondent do not affect responses to questions of our survey.

We use *scikit-learn 0.20.3* Python library[6] using default parameters. We consider 10-Fold Cross Validation technique [13] to mitigate the risk of introducing bias in the results. That technique execute each Machine Learning algorithm 10 times changing the testing and training sets and present a mean accuracy. We use 75% of the data set to train and 25% to test each classifier.

Reflection API. Table 2.6 presents the mean accuracy returned by each classifier for all ques-

[6] https://scikit-learn.org

tions of the Java Reflection API Survey when varying developer experience with Java. Classifiers have a good mean accuracy only in responses of Questions 1, and a worst mean accuracy for responses of Question 2. SVM, Random Forest, Logistic Regression and Decision Tree returns the better mean accuracy for all questions, and Naive Bayes returns the worst mean accuracy for Question 1. Although 58 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 for Questions 2 and 3.

Table 2.6: Results of Machine Learning classifiers for each question of the Java Reflection API Survey. Mean Accuracy considers 10-Fold Cross Validation.

Classifier	Mean Accuracy		
	Q1	Q2	Q3
SVM	83.8%	57.8%	68.2%
Random Forest	83.8%	57.8%	68.2%
kNN	80.0%	43.2%	68.2%
Logistic Regression	83.8%	57.8%	68.2%
Naive Bayes	3.7%	56.0%	38.8%
Decision Tree	83.8%	57.8%	68.2%

Collections API. Table 2.7 presents the mean accuracy returned by each classifier for all questions of the .NET Reflection API Survey when varying the type of the respondent and its experience with C#. Classifiers have a good mean accuracy only in responses of Questions 4 and 6, and a worst mean accuracy for responses of Question 1. SVM and Decision Tree returns the better mean accuracy for all questions, but for Question 2, which kNN classifier had a better mean accuracy, and Naive Bayes returns the worst mean accuracy for all questions.

The mean accuracy of the models gives more evidence that the .NET Reflection API users and developers do not have a consensus in 5 out of 7 questions of the survey. Respondents of our survey present some general comments about the .NET Reflection API. A respondent comment *"...since the API is cumbersome the chance of a mistake is super high. In real world application I would spent quite some time in debugger (or preferably writing unit tests) to*

Table 2.7: Results of Machine Learning classifiers for each question of the .NET Reflection API Survey. Mean Accuracy considers 10-Fold Cross Validation.

Classifier	Mean Accuracy						
	Q1	Q2	Q3	Q4	Q5	Q6	Q7
SVM	42.3%	49.3%	75.8%	85.8%	57.6%	88.0%	57.2%
Random Forest	42.3%	49.3%	75.8%	85.8%	50.9%	88.0%	53.9%
kNN	38.6%	51.3%	75.8%	85.8%	57.6%	88.0%	57.2%
Logistic Regression	42.3%	49.3%	75.8%	85.8%	47.6%	88.0%	57.2%
Naive Bayes	9.1%	10.0%	75.8%	32.5%	39.2%	84.6%	19.6%
Decision Tree	42.3%	49.3%	75.8%	85.8%	57.6%	88.0%	57.2%

make sure everything works as expected". Another respondent says *"...[some questions of the survey] are very technical and almost everyone I know will just test the code until they get the results they need"*.

2.2.4 Threats to Validity

Internal. Respondents could not understand the programs that we present in our surveys. To mitigate that threat, we present small programs (3–20 SLOC), and a snippet or a link of the specification. At least 46.9% of respondents of our survey have more than 10 years of experience with Java, and 63.1% of them have knowledge about the Java Reflection API. Moreover, at least 46.8% of respondents of our survey have more than 10 years of experience in C#, and 77.6% of them have knowledge about the .NET Reflection API. Also, respondents of our survey can execute the programs used in our survey to get the answers of our questions. As we see in practice, in some cases the answers depend on the tool used. If some respondents present responses after executing the program, we have more evidence that there are problems in the specifications of the Java Reflection API and the .NET Reflection API. Respondents can not be motivated to answer the questions of our survey. We think it is not a threat because in case respondents provide answers without motivation, answers should be distributed almost equally among all options. Machine Learning classifiers can present biased results when we choose testing and training sets. We consider Fold Cross

Validation technique with 10 repetitions to mitigate that threat. Survey questions may ask different things than what we want to ask. We conducted two pilots and improved the questions. Developers should be concerned about providing a wrong answer. However, all the data collected from the survey is anonymous. We inform developers that the results of the survey may be reported in academic publications.

External. We conduct our survey presenting programs in Java, and based on the Java Reflection API specification. However, we face similar results in a study conducted in the .NET Reflection API (Section 2.2.2). In that study, we also do not have consensus in responses of most questions of experienced developers. So, we think problems we find in the Java Reflection API specification can happen in other APIs specified in natural language.

2.3 Conclusions

Surveys results and respondents' comments suggest that APIs specified in natural language allows users and developers to create a significantly different mental model of the API behavior, thereby introducing additional effort for everyone intending to safely use reflection in Java and .NET projects and to implement the Java Reflection API and the .NET Reflection API. Moreover, developers do not have a systematic strategy to test popular APIs specified in natural language and to detect the problems in the specification identified in our surveys. The results presented in this chapter reinforce the need for improving systematic conformance testing of the APIs specified in natural language. Also, it can prevent users from experiencing issues. In the following chapter, we present a technique to detect non-conformances and underdetermined specifications in APIs specified in natural language, and we evaluate our technique in Chapter 4.

Chapter 3

A Technique to Test APIs Specified in Natural Language

In this chapter we propose a technique [54] to detect underdetermined specifications and non-conformances in APIs specified in natural language. Section 3.1 presents an overview of our technique. We summarize the algorithm to get initial data in Section 3.2. We present algorithms and input data to generate test cases in Section 3.3. Section 3.4 explains how our technique uses feedback to generate new test cases. Section 3.5 describes the oracle based on differential testing. We present the classifier in Section 3.6. Sections 3.7 and 3.8 detail manual steps of our technique.

3.1 Overview

Algorithm 1 and Figure 3.1 summarize the steps of our technique. Next, we explain our technique using an example. Our technique receives as input a Java program, API implementations (e.g. JVMs), the API specification (e.g. Javadoc [47]), values for Java primitive and non-primitive types and whether test cases should be sorted before execution. 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 [17]. 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 [50].

Before creating test cases in Step 1 for each public method and constructor declared in the specification, we use the input program to get initial objects (e.g. Java Reflection API `Class` objects) (Algorithm 1, Line 5). For instance, consider the input program of Listing 1.1. To test the Java Reflection API, our technique compiles the `ConstraintManager.java` file into the `ConstraintManager.class` and loads it in a JVM using `Class.forName("ConstraintManager.class")` to yield a `Class` object (`c`). Object `c` is added to the values received as input.

In Step 1, we identify all API public methods and constructors in the specification. For instance, it identifies `public URL Class.getResource(String name)` method in the Javadoc. It also identifies parameter types for each API public method and constructor. 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 `routes` field of the `ConstraintManager` class of Listing 1.1 to create the test case presented in Listing 1.2. The technique creates test cases for all possible combinations of all parameters values (Algorithm 1, Line 8). 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. Algorithm 1 returns a default parameter value (e.g. `new Object()`) to previously undefined types. Our technique does not generate redundant test cases. Redundant test cases invoke the same API method using the same parameters values and the same input program. Before generating a new test case, the technique verifies whether the input program, values, and types have already been used to generate a test case.

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 depends on the user input for `sort` parameter. In general, executing test cases randomly improves chances of detecting bugs. However, some APIs (e.g. Java Collections API) require a established order to avoid increasing the rate of false positives. Invoking Java Collections API methods changes the input program (i.e. the col-

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

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

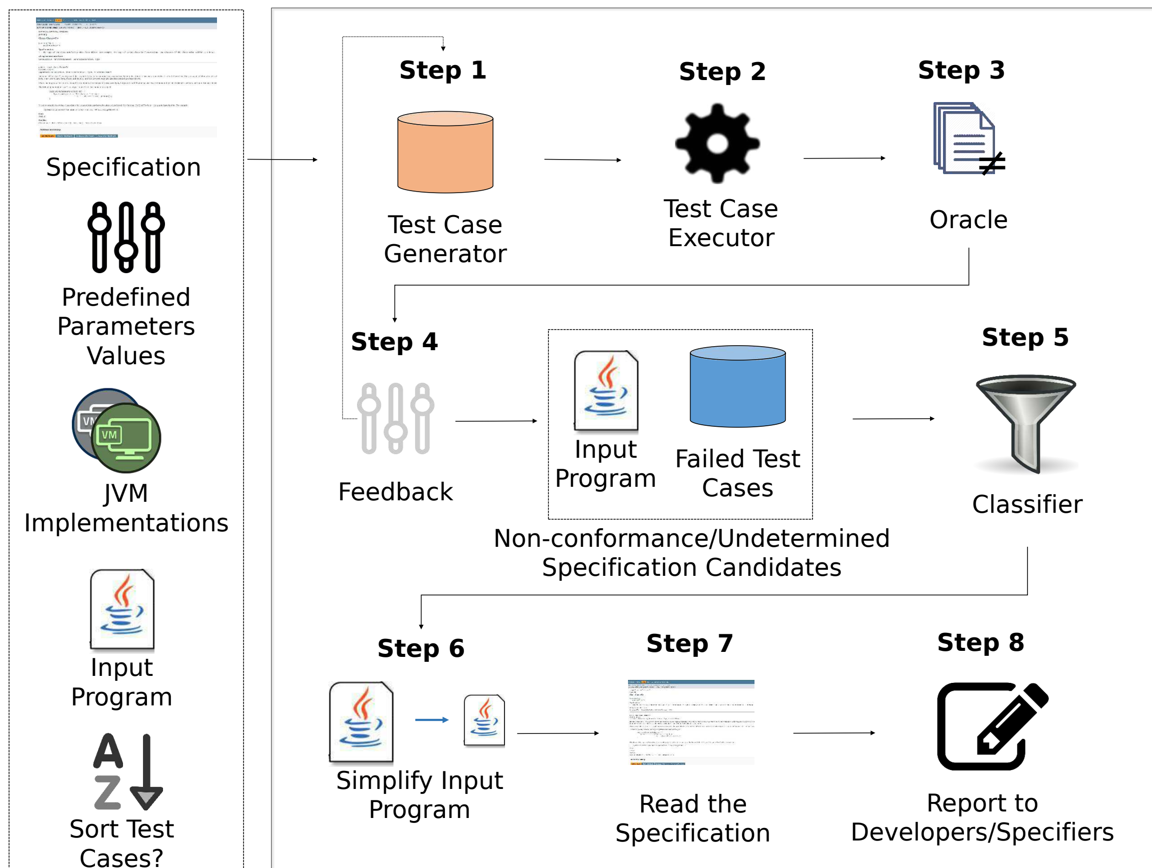

Figure 3.1: Steps of our technique to detect underdetermined specifications and non-conformances in APIs specified in natural languages.

lection). For instance, executing `Vector.add` method adds an object to a vector instance and `Vector.remove` removes it. In a case we execute `Vector.remove` before `Vector.add`, we have a vector containing an object. On the other hand, executing `Vector.add` before `Vector.remove` yields an empty vector. So, if our technique executes test cases in different order in different implementations, it can yield false positives. All test case results are saved so that they can be used to create new tests. We use them in Step 4.

Our technique compares the results using differential testing [30] (Step 3, Line 24). 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 an API method when a test case presents different results in at least two implementations. 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 API classes (e.g. Java Reflection API `TypeVariable` class). For example, Listing 1.2 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 implementation 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 case 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 33). 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 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 1.2. The technique then groups underdetermined specification and non-conformance candidates into four distinct groups: different values, difference between exception thrown and value, different exceptions, and implementation crashes (Step 5, Line 38).

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 [68] 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 specification (Step 7) to identify false positives and correct results. For instance, some methods may yield random values as specified. When our technique executes a test case in two different implementations results may differ (false positive). We consider a non-conformance candidate when an 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 specification of a method is incomplete, imprecise, or ambiguous. In other cases, implementation is in conformance with the specification (✓), and we exclude them. Finally, we report the remaining candidates to API specifiers and developers (Step 8).

3.2 Getting Initial Objects

Reflection is the ability to examine a program and to change its structure and behavior at run time [28; 10]. To create test cases invoking the Java Reflection API methods we need an input program. Algorithm 1 (Line 5) compiles input programs and generate `.class` files (*bytecodes*). It creates a `Class` instance representing a *bytecode* invoking `Class.forName` method. The Java Reflection API does not provide public constructors to instantiate objects of most classes (e.g. `Method`). `Class` class is the entry point to other classes. For instance, we can get objects representing methods of a class by invoking `Class.getMethods` method.

3.3 Generating Test Cases

A test case is a set of input data, execution commands, and conditions to determine whether a system satisfies the specification [3; 50; 38]. Our technique uses public methods and constructors signatures, parameters values, and initial objects (e.g. `Class` instance) to create test cases. Our technique consider a differential testing oracle to decide whether a test should pass or fail.

When testing the Java Reflection API test case, result depends on the input program. For example, `Class.getMethods` returns all public methods of a class. Our technique creates new test cases for each `Method` instance. Number of test cases depends on the quantity and accessibility (i.e. public, protected, or private) of methods of a class. Listing 3.1 presents a program containing a class `A` (Line 1) which defines a public method `a` (Line 2). Result of invoking `Class.getMethods` is a `Method` instance representing `a` method. Our technique creates new test cases by invoking methods of `Method` class (e.g. `a.getModifiers()`, `a.getName()`, and so on).

Listing 3.1: Example of input program to create test cases.

```
1 public class A {
2   public void a() {};
3 }
```

Some methods of popular APIs, like Java Reflection API and Java Collections API, require parameters. Our technique considers Equivalence Class, Boundary Values, and Limit

Values [67] to generate input data of test cases. Table 3.1 presents some considered data. For instance, our technique considers minimum and maximum Java integers as limit values for `int` type; empty, valid ([^A-Za-z0-9]), and invalid (#A1) strings as values for equivalence class for `String` type; and -1, 0, 1 as boundary values for `int`, `long`, and `double` types.

Table 3.1: Input data to generate test cases.

Type	Data
`int`	-2147483648, -1, 0, 1, 2147483647
`long`	2^{63}, -1, 0, 1, $2^{63} - 1$
`double`	4.9x10^{-324}, -1.0, 0, 1.0, 1.7x10^{308}
`boolean`	`false,true`
`char`	'', 'a'
`String`	"", "[^A-Za-z0-9]", "#A1", `null`
`Collection`	`new ArrayList(),new ArrayList("", "abc", ...),null`
Other	`new Object(),null`

Algorithm 2 presents the algorithm of method that returns data based on a type. That algorithm receives the parameter types of a method or constructor and returns a set of values accordingly to a specific type. Algorithm 3 presents the algorithm to generate test cases (Algorithm 1, Line 8). Our technique gets parameter types of all public methods and constructors to get data using Algorithm 2. Our technique generates a test case for each implementation and for all possible combinations of parameters. Finally, Algorithm 3 returns generated test cases.

3.4 Feedback

Algorithm 1 executes generated test cases in all implementations. It considers value of `sort` input parameter to decide whether test cases should be sorted before execute them. Our technique executes test cases in a implementation and logs results to files identified by the implementation name (e.g. `eclipse-openj9-jvm.txt`). Each line of the result files contains a key-value pair (e.g. `A|Class|isArray|void:false`). Keys contain a ref-

```
   Input: parameterTypes
1  data ← ∅;
2  foreach p: parameterTypes do
3      if p.isInt() then
4          data.add({MIN_INT, −1, 0, 1, MAX_INT});
5      end
6      else if p.isLong() then
7          data.add({MIN_LONG, −1, 0, 1, MAX_LONG});
8      end
9      else if p.isDouble() then
10         data.add({MIN_DOUBLE, −1.0, 0.0, 1.0, MAX_DOUBLE});
11     end
12     else if p.isBoolean() then
13         data.add({false, true});
14     end
15     else if p.isChar() then
16         data.add({′′,′ a′});
17     end
18     else if p.isString() then
19         data.add({“”, < valid_string >, < invalid_string >, null});
20     end
21     else if p.isCollection() then
22         data.add({ArrayList(), ArrayList(“”, < valid_string >, < invalid_string >), null});
23     end
24     else
25         data.add({Object(), null});
26     end
27 end
   Output: data
```

Algorithm 2: Input data generation algorithm.

```
   Input: specification
1  cases ← ∅;
2  foreach member : specification.getPublicMethods() ∪ specification.getPublicConstructors() do
3      inputData ← getData(member.getParameterTypes());
4      foreach data : inputData do
5          case ← member(data);
6          if cases.contains(case) then
7              cases.add(case);
8          end
9      end
10 end
   Output: cases
```

Algorithm 3: Test case generation algorithm.

erence to the input program (`A`), to the class and method of the API (`Class.isArray`), and to parameters values (`void`). Values contain results of the test case execution (`false`). Our technique analyzes the results of test cases to create new test cases with new values to other API methods. For instance, consider the input program of Listing 3.2. It defines a class `A` containing a method `m`. Invoking `Class.getMethods` on class `A` returns one `Method` instance representing method `m`. Our technique uses that `Method` instance to create tests for `Method` class (e.g. `Method.getParameters`). Our technique uses the `Parameter` instance returned by `Method.getParameters` method to create a new test case for `Parameter.getType` method, and so on.

Listing 3.2: Input program used in feedback.

```
1 public class A {
2   public void m(int p) {
3   }
4 }
```

3.5 Oracle

Algorithm 4 presents the oracle algorithm. It 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. Our technique detects a non-conformance or underdetermined specification when a test case presents different results in at least two implementations. For example, Listings 3.3 and 3.4 present some results of test cases generated using program of Listing 1.1 as input executed in Eclipse OpenJ9 and Oracle JVMs. For the key `A[]|Class|getResource|""`, Algorithm 4 detects different results (i.e. `null` and `file:/.../target/classes`). In this case, our technique detects an underdetermined specification or a non-conformance candidate.

Listing 3.3: Eclipse OpenJ9 JVM results.

```
1 A[]|Class|getMethods|void:[]
2 A[]|Class|getResource|"":null
3 A[]|Class|isArray|void:true
4 ...
```

```
Input: testCases
1 candidates ← ∅;
2 foreach testCase : testCases do
3     results ← testCase.getResults();
4     if compare(results) then
5         candidates.add(testCase, results);
6     end
7 end
Output: candidates
```

Algorithm 4: Oracle algorithm.

Listing 3.4: Oracle JVM results.

```
1 A[]|Class|getMethods|void:[]
2 A[]|Class|getResource|"":file:/.../target/classes
3 A[]|Class|isArray|void:true
4 ...
```

3.6 Classifier

Our technique groups all failed test cases considering the API method, the input program, data used to generate the test case, and results into four groups: difference between values, difference between exceptions, difference between exception and value, and system crash. Algorithm 5 presents the classifier algorithm.

```
Input: failedTestCase
1 type ← NULL;
2 if failedTestCase.result1.isSystemCrash() or failedTestCase.result2.isSystemCrash() then
3     type ← SystemCrashType;
4 end
5 else if failedTestCase.result1.isValue() and failedTestCase.result2.isValue() then
6     type ← DifferentValuesType;
7 end
8 else if failedTestCase.result1.isException() and failedTestCase.result2.isException() then
9     type ← DifferentExceptionsType;
10 end
11 else if (failedTestCase.result1.isException() and failedTestCase.result2.isValue()) or (failedTestCase.result1.isValue() and
   failedTestCase.result2.isException()) then
12     type ← ValueExceptionType;
13 end
Output: type
```

Algorithm 5: Classifier algorithm.

3.7 Simplifying Input Programs

To better understand each underdetermined specification and non-conformance candidate, we manually simplify input programs based on delta debugging [68]. Program of Listing 3.5 defines an interface `C`, a class `A`, which implements `C`, and another class `B`, which extends from `A`. Oracle JVM yields `public void m()` when executing a test case for getting methods of class `B` (i.e. `Class.getMethods`). On the other hand, Eclipse OpenJ9 JVM throws a `NullPointerException` (Section 4.3.1).

Listing 3.5: Input program sample.

```
1  interface C {
2      void c();
3  }
4  public class A implements C {
5      private String a;
6
7      static {
8          String x = null;
9          x.getClass();
10     }
11
12     void c() {
13     }
14 }
15
16 public class B extends A {
17     private int b;
18     public void m() {}
19 }
```

Some program constructs are not related to the detected candidate. We first remove interface `C` and run the `Class.getMethods` test case again to get methods of class `B`. We continue detecting the candidate to non-conformance or underdetermined specification. As test case is related to getting methods of class `B`, we remove all fields of classes, and method `c` of class `A`. We execute the test case again, and our technique still detects that candidate. We finally, remove the static block of class `A` and execute the test case again. Eclipse OpenJ9

JVM does not throw the exception anymore. We stop simplifying the input program and Listing 3.6 presents the resulting simplified input program.

Listing 3.6: Simplified input program sample.

```
1  public class A {
2      static {
3          String x = null;
4          x.getClass();
5      }
6  }
7
8  public class B extends A {
9      public void m() {}
10 }
```

3.8 Reading the Specification and Reporting Bugs

Javadoc specification for `Class.getMethods` method does not define a `NullPointerException` as an expected exception. Javadoc defines that `Class.getMethods` method can throw only a `SecurityException`. In this case, we detect a non-conformance between the specification and the Eclipse OpenJ9 implementation of `Class.getMethods` method. We report a bug to Eclipse OpenJ9 JVM developers. They accept and fix that bug. Moreover, they ask us to send a pull request containing the generated test case. Twelve test cases generated by our technique are now part of the Eclipse OpenJ9 JVM test suite.

Chapter 4

Evaluation

In this chapter, we consider two APIs to evaluate our technique. Both APIs are widely used [22; 21] and testing them allow us to stress our technique in different ways. Section 4.1 evaluates our technique using the Java Reflection API, and Section 4.2 evaluates our technique using the Java Collections API. Section 4.3 presents discussions about the evaluations. Section 4.4 shows the steps to adapt our technique to test other APIs. Finally, Section 4.5 details the threats to validity of the evaluations. The complete results and replication package are available at our website [53].

4.1 Testing the Java Reflection API

We evaluate our technique using the Java Reflection API.

4.1.1 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.

4.1.2 Planning

We use MetricMiner [63] to retrieve the commit (2017-03-02) of 446 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 as: 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 greatest 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 reference Javadoc provided by the Oracle JVM [47]. 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 [50]. 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 and constructors signatures, parameters values, and the `Class` instance to create test cases. It randomly executes test cases in a JVM (*sort* parameter as false in input of Algorithm 1) 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.

4.1.3 Results

Algorithm 1 executes a total of 278M 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 manually detect 11 candidates during Step 6. Our technique cannot detect them automatically because of the random test case execution. 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. Ta-

ble 4.1 presents methods with detected candidates, number of test cases, number of failures, and the status of a candidate reported to 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 candidate 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.

Table 4.1: 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. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. 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	S	J1	J2	J3	J4
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.get-Annotations	177,220	41	O	✓	✓	✓	✓
6	Executable.getDeclared-Annotations	177,412	48	O	✓	✓	✓	✓

Continue on next page

Table 4.1 – *Continued from previous page*

Id	Method	Test Cases	Failures	S	J1	J2	J3	J4
7	Executable.getParameter-Annotations	177,212	7	O	✓	✓	✓	✓
8	Field.getAnnotations	603,028	120	O	✓	✓	✓	✓
9	Field.getDeclared-Annotations	615,960	122	O	✓	✓	✓	✓
10	Method.getAnnotations	965,740	293	O	✓	✓	✓	✓
11	Method.getDeclared-Annotations	976,304	338	O	✓	✓	✓	✓
12	Method.getParameter-Annotations	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.getDeclared-Constructor	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.getAnnotated-ParameterTypes	356,884	7,657	✓	F	–	F	–
				✓	F	–	F	–
				✓	✓	✓	F	–
				✓	F	–	F	–

Continue on next page

Table 4.1 – *Continued from previous page*

Id	Method	Test Cases	Failures	S	J1	J2	J3	J4
27	Executable.getAnnotated-ParameterTypes	88,702	435	✓ ✓	F ✓	– ✓	F F	– –
28	Method.invoke	1,468,228	240	✓ ✓ ✓	O O D	– – –	R R R	– – –
29	Package.getImplementa-tionTitle	59,014	117	✓	✓	✓	✓	–
30	Parameter.getAnnotated-Type	88,806	1,135	✓	✓	✓	F	–
31	Parameter.getParameteri-zedType	88,764	1,131	✓	✓	✓	F	–
32	Parameter.toString	88,664	1,154	✓	✓	✓	F	–

`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. Figure 4.1 presents a histogram with the number of input programs according to the number of detected candidates, except false positives. That histogram presents data similar to an exponential distribution. There are many input programs used to detect few candidates. On the other hand, there are few input programs used to detect many candidates. Cubeqa is the input program that exposed most candidates (23). It is also the input program that most accepted by the reported bugs (17). Figure 4.2 shows the number of input

programs that expose each candidate. `Executable.getParameterAnnotations` and `Method.getParameterAnnotations` have less input programs exposing them. `Class.getConstructor`, `Class.getConstructor`, and `Class.getMethods` are the methods that have most input programs exposing them.

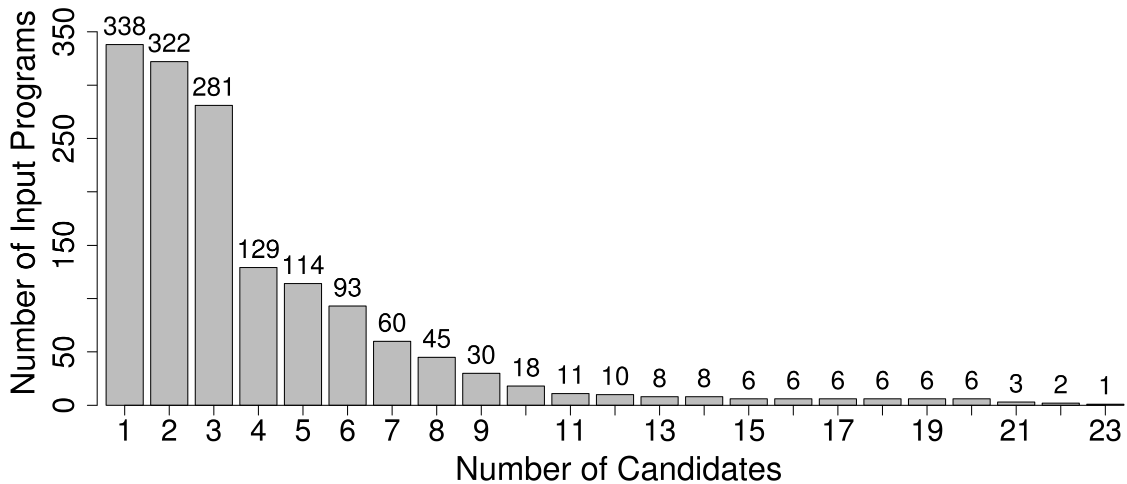

Figure 4.1: Number of input programs that expose candidates after Step 7 (Figure 3.1).

4.1.4 Answers to the Research Questions

Next, we answer our research questions.

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

We find non-conformances in `Class`, `Constructor`, `Executable`, `Field`, `Method`, `Package`, and `Parameter` classes, and in 32 public methods out of 237 methods tested. We manually identify 10 detected non-conformances as false positives. We report 33 non-conformances to Oracle and Eclipse OpenJ9 JVMs. Eclipse OpenJ9 developers accept and fix 29 non-conformances, and ask us to send a pull request containing 12 test cases.

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

A number of input programs (76.9%) can be used to detect at least one non-conformance in

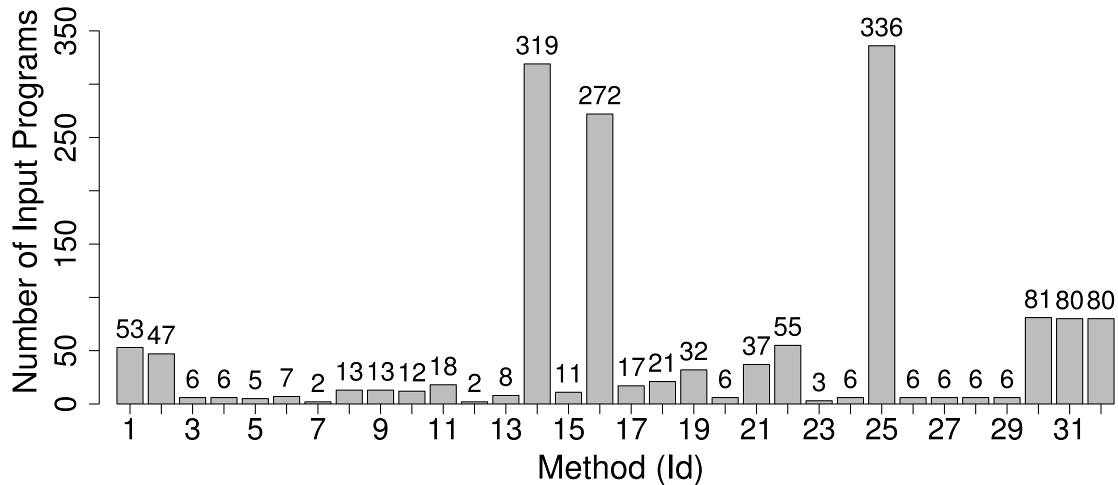

Figure 4.2: Number of input programs exposing each candidate after Step 7 (Figure 3.1) (see *Id* column in Table 4.1).

our technique. Some of them (73.1%) expose non-conformances accepted by JVM developers. Cubeqa input program exposes most non-conformances (23). It also exposes the most accepted number of non-conformances (17). We identify 76.5% of input programs exposing a non-conformance in `Class.getMethods` method.

4.2 Testing the Java Collections API

In this section, we evaluate our technique using the Java Collections API [46].

4.2.1 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 Java Collections API implementations with respect to Oracle, OpenJDK, IBM J9, and Eclipse OpenJ9 JVMs from the point of view of specifiers and developers. We address the following research question:

- RQ_3**: How many underdetermined specifications and non-conformances between Javadoc and the Java Collections API implementations can our technique detect?**

We compute the number of underdetermined specifications and non-conformances accepted by Java Collections 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 Collections API. We do not count the number of input programs used by our technique yield at least one underdetermined specification or non-conformance candidate since we do not need to yield complex objects, such as `Class`.

4.2.2 Planning

We do not consider real input programs to test the Java Collections API because they are simpler than the input programs required to test the Java Reflection API. To test the Java Reflection API we use real input programs because we have to create interesting complex objects (`Class`, `Method`, and so on). On the other hand, we do not need real input programs to test the Java Collections API because we can just instantiate collections using default parameters. We use `null`, empty collections (e.g. `HashMap`), and an `ArrayList` of `String` as input to generate test cases. We consider the Java Collections API reference Javadoc provided by the Oracle JVM [46]. Algorithm 1 evaluates all public methods of 25 classes (i.e. `AbstractCollection`, `AbstractList`, `AbstractMap`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, `ArrayDeque`, `ArrayList`, `Arrays`, `Collections`, `EnumMap`, `EnumSet`, `HashMap`, `HashSet`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `LinkedHashSet`, `LinkedList`, `Objects`, `PriorityQueue`, `TreeMap`, `TreeSet`, `Vector`, and `WeakHashMap`) of the `java.util` package (e.g. `ArrayList` class), and 13 classes (i.e. `ArrayBlockingQueue`, `ConcurrentHashMap`, `ConcurrentSkipListSet`, `ConcurrentSkipListMap`, `ConcurrentLinkedQueue`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedTransferQueue`, `PriorityBlockingQueue`, and `SynchronousQueue`) of the `java.util.concurrent` package (e.g. `ConcurrentHashMap`). Our technique cannot test methods `LinkedBlockingQueue.take`, `ArrayBlockingQueue.take`, `PriorityBlockingQueue.take`, `LinkedBlockingDeque.takeFirst`, `LinkedBlockingDeque.takeLast`, `LinkedBlockingDeque.take`,

`DelayQueue.take`, `DelayQueue.take`, `SynchronousQueue.take`, `LinkedTransferQueue.take`, `SynchronousQueue.put`, `BlockingQueue.put`, and `LinkedTransferQueue.transfer` because they wait until some element becomes available. So, execution blocks when executing test cases invoking those methods.

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). Our technique executes all steps of Algorithm 1 in a similar way of the Java Reflection API evaluation. However, we have different approaches to get initial objects in the two APIs. In the Java Reflection API, we need to load binary files. As the Java Reflection API consider more complex objects, like `Parameter`, feedback is more important than when generating test cases for the Java Collections API. On the other hand, we can invoke `new` statement to create objects of the Java Collections API. It does not need to inspect a program (Algorithm 1, Line 5) to yield classes, fields, parameters, and so on. We just instantiate objects using default parameters values. Before executing Java Collections API test cases, our technique alphabetically sorts generated test cases by method names to mitigate detecting false positives (Algorithm 1, Line 13).

4.2.3 Results

Algorithm 1 executes a total of 118,251 test cases. Some of them (0.73%) failed. The technique takes about five minutes to execute all automatic steps presented in Algorithm 1. It yields 5 underdetermined specification and 24 non-conformance candidates. Eight non-conformance candidates result in a JVM crash due to `OutOfMemoryError`. Since programs are small, it is easy to execute Step 6. We take approximately 15 minutes to manually analyze each of them in Steps 6 and 7. Experienced JVM developers may take less time. We identified three 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 4.2 presents methods with detected candidates, number of test cases, number of failures, and the status of a candidate reported to the Java Collections API specifiers, and to Eclipse OpenJ9, and Oracle JVMs' bug trackers.

Eclipse OpenJ9 JVM throws an unexpected exception on candidates Ids: *1*, and *7–9*. IBM J9 JVM throws an unexpected exception on candidate Ids: *1*, and *8*. All JVMs return a result different than expected on candidates Ids *3–6*. The Java Collections API specification allows multiple implementations to return different results for the same input on candidate Ids: *1-2*, and *7–9*. 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 candidates are false positives, we submitted 5 underdetermined specifications to Javadoc specifiers and 13 reports to JVM developers.

Table 4.2: Detected Java Collections 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. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. 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	S	J1	J2	J3	J4
1	ArrayDeque	72	6	O	✓	✓	R	–
2	ArrayList.ensureCapacity	40	1	R	✓	✓	F	–
3	Arrays.copyOfRange	4,900	180	✓	R	–	O	–
4	ConcurrentSkipListMap.put	44	1	✓	F	–	O	–
5	ConcurrentSkipListMap.put-IfAbsent	48	1	✓	F	–	O	–
6	ConcurrentSkipListSet.add	20	1	✓	F	–	O	–
7	Hashtable	264	72	O	✓	✓	R	✓
8	IdentityHashMap	160	108	O	✓	✓	R	–
9	WeakHashMap	577	498	O	✓	✓	R	✓

`Arrays.copyOfRange` method is executed by more test cases (4,900). `WeakHashMap`'s constructor yields more test failures (498). Algorithm 1 detects four non-conformance candidates in Oracle and in OpenJDK, nine in Eclipse OpenJ9, and seven in IBM JVMs. JVM developers answered to 61.5% of them. Oracle JVM developers accepted and fixed three non-conformance candidates. Eclipse OpenJ9 JVM developers

accept and fix one non-conformance candidate, rejected four of them and four are still open.

4.2.4 Answer to the Research Question

Next, we answer our research question.

RQ_3: **How many underdetermined specifications and non-conformances between Javadoc and the Java Collections API implementations can our technique detect?** Our technique identifies 29 underdetermined specification and non-conformance candidates. A number of 17 candidates cannot be detected by Randoop [48] or EvoSuite [12], popular automatic test suite generators. We report 5 underdetermined specification candidates to the Java Collections API specifiers. We also report 9 non-conformance candidates to Eclipse OpenJ9 JVM, and 4 to Oracle JVM. Oracle JVM developers accept and fix 3 non-conformance candidates. Eclipse OpenJ9 JVM developers accept and fix 1 non-conformance candidate.

4.3 Discussion

In this section, we discuss our results.

4.3.1 Report Candidates to APIs

Reflection API. 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 1.1 and 1.2) to Oracle JVM[1] and Eclipse OpenJ9[2] 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*

[1] https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8202687
[2] https://github.com/eclipse/openj9/issues/1848

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. After discussing with them, they accepted some of them as bugs. For instance, Listing 4.1 defines `ManagerServer` class extending from class `App`, which invokes a method (`LoggingFactory.bootstrap()`) in a static block. Eclipse OpenJ9 JVM throws an exception executing `Class.getDeclaredFields` test case to retrieve declared fields of `ManagerServer` because class `App` throws an exception and we can not get a `ManagerServer` instance. 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 [25] 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.

Listing 4.1: The SPQR input program.

```
1 public class App {
2   static {
3     LoggingFactory.bootstrap();
4   }
5 }
6 public class ManagerServer extends App {
7   private NodeManager nodeManager = null;
8 }
```

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 4.1) are still unfixed.

Collections API. Developers of Oracle JVM accept and fix non-conformances between the specification and implementation of `ConcurrentSkipListMap.put`, `ConcurrentSkipListMap.putIfAbsent`, and `ConcurrentSkipListSet.add` methods. Those methods present nondeterministic results when executing test cases of Listings 4.2, 4.3, and 4.4 multiple times. Sometimes the result is `null`, sometimes is `ClassCastException`. The non-conformance is fixed by developers and those test cases always return `ClassCastException`. Developers of Eclipse OpenJ9 JVM accept and fix a non-conformance between the specification and the implementation of `ArrayList.ensureCapacity` method (Section 1.2). Our technique also detects non-conformances between the specification and implementation of `Arrays.copyOfRange`, `ConcurrentSkipListMap.put`, `ConcurrentSkipListMap.putIfAbsent`, and `ConcurrentSkipListSet.add` methods in Eclipse OpenJ9 and IBM J9 JVMs. Eclipse OpenJ9 developers claim that those non-conformances are present in a code imported from the OpenJDK JVM. They asked us to report the bug to the OpenJDK JVM developers. However, those bugs are still open. Non-conformances detected in classes `ConcurrentSkipListMap` and `ConcurrentSkipListSet` may be explained by intrinsic difficulty to test concurrent code.

Listing 4.2: `ConcurrentSkipListMap.put` test case.

```
1 new ConcurrentSkipListMap().put(new Object(), new Object());
```

We identify the usage of the Java Collections API methods, related to candidates reported to developers and specifiers that are still open (Table 4.2), in the 446 input programs used in the Java Reflection API evaluation. Table 4.3 presents the results. `Hashtable` is the most used. On the other hand, less than 10% of the input programs use all other methods. That low usage rate can help to explain why those reported candidates are still open.

Our technique detects eight non-conformances in methods of the Java Collections API that crash Eclipse OpenJ9 and IBM J9 JVMs. Those non-conformances throw an `OutOfMemoryError` because test cases use more memory than defined on JVM heap size. Eclipse OpenJ9 developers claim that we should increase the heap size on JVM initialization. However, Oracle and OpenJDK JVMs dynamically allocate heap size at run-time

Table 4.3: Usage of Java Collections API open bugs methods in the 446 input programs of the Java Reflection API evaluation (Section 4.1).

Method	Usage (%)
ArrayDeque	8.1%
Arrays.copyOfRange	6.3%
ConcurrentSkipListMap	2.2%
ConcurrentSkipListSet	1.8%
Hashtable	48.2%
IdentityHashMap	4%
WeakHashMap	4.5%

based on system configuration to avoid crashing the JVM.

Listing 4.3: `ConcurrentSkipListMap.putIfAbsent` test case.

```
1 new ConcurrentSkipListMap().putIfAbsent(new Object(), new Object());
```

JVM developers reject five non-conformances, and one underdetermined specification. We do not agree with all of them. For instance, we report an underdetermined specification in the `ArrayList.ensureCapacity` method. Javadoc does not restrict the capacity of the `ArrayList`. Oracle JVM developers confirm that they limit the capacity of `ArrayList` in the implementation. However, specifiers reject the reported underdetermined specification claiming that the implementation works as designed. Listing 4.5 presents a test case of `Arrays.copyOfRange` method. `Arrays.copyOfRange(byte[] original, int from, int to)` method copies the specified range of the specified array into a new array. Specification defines that `Arrays.copyOfRange` method must throw a `NullPointerException` when `original` parameter is `null`. The test case presented in Listing 4.5 throws an `OutOfMemoryError` when executed in Oracle JVM. We report a bug to Oracle JVM developers.[3] Developers claim that code behaves according to the specification, and reject it. However, the specification of `Arrays.copyOfRange` does not define a scenario, in which an `OutOfMemoryError` should be thrown.

[3] https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8227674

Listing 4.4: ConcurrentSkipListSet.add test case.

```
1 new ConcurrentSkipListSet().add(new Object());
```

Developers also reject non-conformances in ArrayDeque, Hashtable, IdentityHashMap, and WeakHashMap constructors. All of them receives as parameter the maximum capacity of the collection. Our technique generates test cases with Integer.MAX_VALUE as parameter value. Those test cases throw an OutOfMemoryError in the Eclipse OpenJ9 JVM. Developers claim that we are trying to create an array of 2GB elements. In this case, we should configure the maximum heap size on the JVM startup. However, Oracle JVM increases heap size dynamically and does no throw any exception when executing test cases invoking those constructors.

4.3.2 Input Programs

Reflection API. 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.

Listing 4.5: Test case related to Arrays.copyOfRange.

```
1 byte[] newArray = Arrays.copyOfRange(null, 0, 2147483647);
```

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 4.6 presents a simplified input program from the Pulsar Reporting.[4] It declares an inner class (Itr, Line 2) inside a concrete class used to iterate over elements of a linked queue. It uses four Java keywords and has 6 SLOC. 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.

[4]Data visualization and reporting framework designed to provide real-time insights from Pulsar analytics platform.

Listing 4.6: The Pulsar Reporting program input.

```
1 public class BytesBoundedLinkedQueue<E> {
2   private class Itr implements Iterator<E> {
3     Itr(Iterator<E> d) {
4     }
5   }
6 }
```

As the simplified input programs that expose a candidate are small, we used an automatic program generator (JDolly [62]) to increase the quantity of input programs. JDolly generated 197,530 input programs. We used the same Alloy [19] theory used by Mongiovi et al. [36]. 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.[5] [6] 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. JDolly may generate millions of programs. We can improve this scenario by skipping some similar programs [37]. 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. The goal was to improve chances to identify non-conformance candidates.

Collections API. Differently from the Java Reflection API evaluation, we use small input programs to evaluate the Java Collections API. We do not need them to create complex objects, such as `Class`, `Method`, and `Field`. We consider `null`, empty collections, and collections of `String` as input programs to detect all 29 candidates. Our technique detects

[5] https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8202688
[6] https://github.com/eclipse/openj9/issues/803

86.2% of candidates using an empty collection, and 13.8% using `null` as input program. We detect 17 candidates not detected by automatic test suite generators.

4.3.3 False Positives

Reflection API. In Step 7, our technique reports 10 false positives (Table 4.4). The Java Reflection API represents parameters identifiers of a method as `arg0`, `arg1`, and so on. So, two different methods of an input program can have different parameters types represented by the same identifier (e.g. `arg0`). 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. Java assign a new reference to each new object instance. Methods that instantiate new objects, like `Class.newInstance`, return different instances references to the same input program. 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.

Table 4.4: Java Reflection API false positives reported by our technique.

Id	False Positive	Cause
1	Parameter.getDeclaringExecutable	Generic parameters names
2	Class.hashCode	Hash code values do not have to be equal
3	Parameter.hashCode	
4	Class.newInstance	Different instance references
5	Constructor.newInstance	
6	Constructor.isAccessible	Different execution order
7	Field.isAccessible	
8	Method.isAccessible	
9	Field.getInt	
10	Field.getLong	

Collections API. Table 4.5 presents the false positives reported by our technique when test-

ing the Java Collections API. Our technique detects candidates in `HashMap.put` and `HashMap.putIfAbsent` because they insert elements in different orders. However, Javadoc of `HashMap` class defines that it does not guarantee that the order of elements will remain constant over time. According to specification of methods `Arrays.fill`, `HashSet.toArray`, and `LinkedHashSet.toArray` implementations must throw an `ArrayStoreException` if the specified value is not of a run-time type that can be stored in the specified array. Our technique generates test cases, such as `Arrays.fill(new Integer[1], new Object())`, which does not follow the specification.

Table 4.5: Java Collections API false positives reported by our technique.

Id	False Positive	Cause
1	Arrays.fill	Test cases do not follow specification
2	HashSet.toArray	
3	LinkedHashSet.toArray	
4	*.hashCode	Hash code values do not have to be equal
5	HashMap.put	Order of elements can change
6	HashMap.putIfAbsent	

4.3.4 Underdetermined APIs

Reflection API. Recent studies indicate that incompleteness in an API specification can avoid developers to use an API [56; 57; 65]. 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 [53]. 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 some specifications that do not explain what to do when considering some Java constructs, such as annotations (e.g. underdetermined specification 2 in Table 4.1), or input values, such as empty string (Listing 1.1) (underdetermined specification 3 in Ta-

ble 4.1). Algorithm 1 also detects a non-conformance candidate in the `Class.getFields` method.[7] Oracle JVM returns the class fields while Eclipse OpenJ9 JVM yields an exception. `Class.getFields` should return all public fields of a class. However, Javadoc does not specify what to return when a class inherits fields with the same name.

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.[8] We hope that the results presented here can help the Java Reflection API specifiers and developers to better understand and improve the specification and, consequently, the JVM's implementations.

Collections API. Our technique also detects other underdetermined specifications in 19 Java Collections API methods related to exception's message. Specifications of those methods (e.g. `ArrayList.remove`) do not define the message that should be returned in case an exception is thrown. Eclipse OpenJ9 and IBM J9 JVMs return `-1` as message when executing `ArrayList.remove(-1)` test case. On the other hand, Oracle and OpenJDK JVMs return `null` as message. We report a bug to Java Collections API specifiers but we have no response yet. As another example, the following test case `Stack.add(-1, new Object())` yields `Array index out of range: -1` exception message when executed in Eclipse OpenJ9 and IBM J9 JVMs, and `null` when executed in Oracle and OpenJDK JVMs. The specification also does not state the output in this case.

4.3.5 Automatic Test Suite Generators

Reflection API. Tools like Randoop [48] and EvoSuite [12] 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

[7] https://github.com/eclipse/openj9/issues/1845

[8] https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8202687

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 4.7. 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 [61]. 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 [22]. So, these tools do not focus on testing programs using the Java Reflection API [2], differently from our technique.

Listing 4.7: Small program used as input.

```
1 public class A {
2   public int m(Method p) {
3     return 0;
4   }
5 }
```

Collections API. To detect bugs in the Java Collections API, we consider two setups to evaluate Randoop and EvoSuite. First, we generate test cases and execute them in the same JVM. Second, we generate test cases in a JVM and execute them in other one. We consider Randoop 4.1.2 with *–time-limit: 60* and *–flaky-test-behavior: OUTPUT* parameters, and EvoSuite 1.0.6 with *-generateTests* parameter. Table 4.6 presents candidates detected using Randoop. Randoop detects two underdetermined specifications in two Java Collections API methods (Candidate Ids *1* and *5*), and six non-conformances in three Java Collections API methods (Candidate Ids *2–4*). EvoSuite does not generate test cases using default parameters.

Randoop detects three candidates when generating and executing test cases using the same JVM. However, Randoop detects two other candidates (Ids: *2* and *4* of Table 4.6) only when generating test cases using a JVM (e.g. IBM J9) and executing them in another JVM (Eclipse OpenJ9). Our technique also detects those candidates.

Our technique detects candidates in seven Java Collections API methods that Randoop does not detect. Randoop generates test cases for `ArrayList.ensureCapacity(int`

Table 4.6: Java Collections API candidates detected by Randoop. S: Specification. J1: Oracle JVM. J2: OpenJDK JVM. J3: Eclipse OpenJ9 JVM. J4: IBM J9 JVM. Status: – = Unreported bug; O = Bug Open; F = Fixed bug; A = Accepted bug; R = Rejected bug; ✓= Correct result; D = Duplicated bug.

Id	Method	S	J1	J2	J3	J4
1	Arrays.binarySearch	O	✓	✓	R	–
2	Arrays.copyOfRange	✓	✓	✓	O	–
3	Collections.addAll	✓	A	–	R	–
4	Hashtable	✓	F	–	O	–
5	Objects.hash	R	✓	✓	✓	✓

`capacity)` using 10 as the maximum integer value. Our technique detects a non-conformance in `ArrayList.ensureCapacity` method considering `Integer.MAX_INTEGER` as parameter. In that case, Eclipse OpenJ9 and IBM J9 throw an unspecified `OutOfMemoryError`, which crashes the JVMs. Randoop does not consider `Object` instances to generate test cases for `ConcurrentSkipListMap.put`, `ConcurrentSkipListMap.putIfAbsent`, and `ConcurrentSkipListSet.add`. Our technique detects non-conformances using `Object` instances to generate test cases to those methods.

Our technique does not detect candidates in methods `Arrays.binarySearch`, `Collections.addAll`, and `Objects.Hash` (Table 4.6). It generates test cases for `Collections.addAll` considering `Object[] p = null` as parameter. Randoop detects a candidate considering `int[] = null` as parameter. Our technique should consider `Object[] p = new Object[1]` as parameter instead of `Object[] p = new Object[]new Object(), null` to detect candidates in the `Arrays.binarySearch` method.

4.4 Testing Other APIs

Initially, we should decide to use real or toy input programs. That decision depends on the characteristics of the new API. For instance, we should consider real input programs to test Abstract Syntax Tree (AST) APIs because they need to inspect an input program. The more Java constructs we have, the more the probability of detect underdetermined specifications or non-conformances between the specification and the implementation. On the other hand, we can use toy input programs to test Java Cryptography Architecture APIs. We may need to provide some predefined parameters values for new API types. For example, we define an `ArrayList` for `Collection` type to test the Java Collections API. Another input that we should evaluate to test other APIs is whether a test case can be randomly executed. We must sort test cases before executing them when invoking API methods changes the input programs. We also have to implement `equals` methods for some classes that do not provide a default implementation (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 an API method presents the same results in different implementations. To reduce this threat, we evaluated four implementations. 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 execute Randoop with *–time-limit: 60* and *–flaky-test-behavior: OUTPUT*, and EvoSuite with *-generateTests* command line arguments. We consider default arguments for all other parameters. We cannot find any argument in Randoop and EvoSuite documentation to increase likelihood of detecting bugs.

We consider data available on the Google BigQuery[9] to find all input programs that contain the `pom.xml` file to select 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.

[9] https://medium.com/google-cloud/github-on-bigquery-analyze-all-the-code-b3576fd2b150#.9e3g8sdb9

Although input programs are from one code repository, GitHub has almost 20M contributors[10] 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. We consider only `null`, empty collections, and collections of `String` as input programs to test the Java Collections API. Although they are simple, our tool can test all non-blocking public methods and constructors. We evaluate only specifications and implementations of Java 8 APIs. We consider that the specifications and the implementations of the Java Reflection API and Java Collections API do not significantly change in recent Java versions.

[10]https://github.blog/2017-04-10-celebrating-nine-years-of-github-with-an-anniversary-sale/

Chapter 5

Related Work

In this chapter we present 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 and of the Java Collections API. Section 5.1 describes works related to conformance checking. Section 5.2 presents works related to statically and dynamically analyzing APIs. Section 5.3 reports works related to surveys.

5.1 Conformance Checking

Tan et al. [64] 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 some non-conformance candidates in the Java Reflection API and Java Collections API methods related to the `null` normal property [64]. @TCOMMENT can be useful to confirm them as implementation bugs. Considering other types of non-conformances may be a challenge [64], but we will investigate other heuristics.

Legunsen et al. [24] 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. Miranda et al. [35] propose RVPRIO, an approach to automatically prioritize Runtime Verification violations and increase the likelihood to identify true

bugs. RVPRIO considers properties manually formalized from the Javadoc of a subset of four Java packages, runs JavaMOP to collect violations of 11 open source projects, and uses Machine Learning classifiers to prioritize those violations. 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 [7] propose a tool that automatically generates test cases from JML formal specifications. Milanez [33] proposes a tool to automatically check conformance of Java programs annotated with JML [23] specifications based on automatic test generation using Randoop [48]. Massoni et al. [29] 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 [11]. They show evidences about issues with keeping object models and the implementation in conformance during refactoring. Khurshid and Marinov [58] propose TestEra – a tool that considers a formal specification for a method and uses the method precondition to automatically generate test inputs. When a program violates a correctness property, TestEra generates concrete counterexamples. Khurshid and Marinov test some methods of the Java Collections API. Corazza et al. [8] describe a protocol to manually checking the coherence between comments and implementation of Java methods. This protocol maps a dataset programs' results to Javadoc specifications. Our technique checks conformance between the specification of Java Reflection API and Java Collections API, which are described in natural language.

Cheon [6] proposes a tool (JET) that generates test cases to check conformance between implementations and specifications of applications defined in Java Modeling Language (JML). JET generates oracles based on contracts and uses genetic algorithms to generate input data to test cases. Our technique considers Equivalence Class, Limit Values, and Boundary Values as strategies to generate input data to test cases. We can consider genetic algorithms to improve quality of generated test cases, increasing probability of detecting non-conformances and underdetermined specifications.

Milanez et al. [34] present a tool (JMLOK2) to check conformance in applications based on contracts defined using JML. JMLOK2 considers a random strategy to generate test cases. That tool automatically generates oracles based on JML contracts assertions. It is not easy to automatically generate oracles from specifications defined in natural language. Our tech-

nique considers differential testing to compare results of different API implementations. JMLOK2 considers some heuristics to categorize non-conformances. Our technique groups non-conformances and underdetermined specifications based on differences types.

5.2 APIs Analysis

Gyori et al. [15] 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 [68]. We can automate that reduction by adapting our technique based on binary search [15].

Chen et al. [5] present the Classfuzz tool that creates Java programs by using mutations, and uses differential testing to identify bugs in JVMs' start-up processes. Classfuzz uses pre-defined binary files to generate mutants used as input programs to create test cases. Chen et al. [4] 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 and Java Collections 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. [52] 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. JSF does not focus on testing the Java Reflection API and Java Collections API. Algorithm 1 generates test cases to identify underdetermined specification and non-conformance candidates in both APIs.

Sen and Agha [59] propose a tool (jCUTE) to automatically test Java concurrent programs. jCUTE detects bugs in six Java Collections API synchronized classes (e.g. `Vector`). Our technique does not test concurrent programs. Pacheco et al. [49] propose Randoop, a tool to generate unit tests for object-oriented programs. Randoop randomly generates test cases for static methods and for concrete methods of any Java class that provides a public constructor. They found bugs in the Java Collections API. Fraser and Arcuri [12] introduce a tool (EvoSuite) to generate test cases for Java classes. EvoSuite uses evolutionary search to generate test cases until it reaches a coverage target. Randoop and EvoSuite heavily depends on a reflection API in their implementations. Developers of Randoop and EvoSuite can also face similar problems that we identify in this work.

Visser et al. [66] use symbolic execution to generate test inputs and increase branch coverage of Java TreeMap API implementation [9] [41]. They propose generating test inputs based on white-box technique while we generate them using a black-box technique. does not focus on increase code coverage. However, a study shows evidences that coverage is not strongly related to test suite effectiveness [18]. Moreover, we do not require source code of Java Reflection API as input.

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. Our tool uses Limit Values, Boundary Value and Equivalence Class strategies to generate test inputs. It does not check Javadoc specification formally. KeYTestGen requires formal specifications and source code to generate test inputs.

Livshits et al. [27] 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. [22] 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.

5.3 Surveys

Uddin and Robillard [65] present results of two surveys that investigate issues on 72 distinct APIs from six types of programming languages. They request developers for provide good and bad examples of API specifications. After analyzing those examples, they identify ambiguity, incompleteness, and incorrectness as the three severest issues. In the second survey, researchers ask developers the issues' frequency, their severity, and the necessity to solve them. Head et al. [16] report results of a survey that asks developers what they miss from C++ API specifications. They conclude that in 5% to 25% of the cases developers do not find how to use the API. Our approach complements those works, since we focus on investigating in practice developers' responses about the output of some popular .NET Reflection API methods and properties. There is no consensus in responses of all questions. We find that some responses diverge from 48.8% of the most common.

Robillard [56] conducts a survey to investigate the obstacles faced by Microsoft developers when learning how to use APIs. He concludes that obstacles caused by inadequate or absent resources for learning the API (for example, documentation) are the most indicated by developers. Nadi et al. [39] perform a survey about obstacles developers face while using Java cryptography APIs. They conclude that the lack of documentation impacts on how correctly developers use a Java cryptography API. We find evidence that the .NET Reflection API specification is incomplete and imprecise. It can impact developers' understanding about some methods and properties.

Chapter 6

Conclusions

This work presents empirical investigations and a new technique that enables us to detect underdetermined specifications and non-conformances in APIs specified in natural language. 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 for 66.6% of questions. We also conduct a survey with 128 users and developers of the .NET Reflection API. In general, experience with C# of API users and developers does not affect their responses to questions of our survey. Moreover, we can not find statistical differences between responses of API users and developers. There is no consensus in responses to 71.4% of questions of our survey. Some responses diverge from 45.5% of the most common. In a question, only 7.3% of API users present the same response of popular tools that implement the .NET Reflection API. In the same question, most API developers (62.5%) present a response diverging from those tools.

To improve this scenario, we propose a technique to detect underdetermined specifications and non-conformances between the specification and the implementations of APIs specified in natural language. It automatically creates test cases and executes them in different implementations. We evaluate our technique in 446 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 Chapter 2 cannot detect the candidates found by our technique. We also evaluate our technique using the Java Collections API, a popular Java API [21]. Our technique identifies 29 underdetermined specification and non-conformance candidates. Our technique identifies 17 candidates that cannot be detected by popular automatic test suite generators (using default parameters). We report 5 underdetermined specification candidates to the Java Collections API specifiers. We also report 9 non-conformance candidates to Eclipse OpenJ9 JVM, and 4 to Oracle JVM. Oracle JVM developers accept and fix 3 non-conformance candidates. Eclipse OpenJ9 JVM developers accept and fix 1 non-conformance candidate. So far, we do not find any patterns in the bugs found.

The results of applying our technique helped both API developers to improve the implementation and promote discussions about underdetermined specifications in the Java Reflection API and Java Collections API Javadocs, confirming the lack of consensus in most questions found by our surveys. The Java Reflection API and Java Collections 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 both API 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 API methods; ii) use strategies to choose values (like Limit Value); iii) use differential testing; iv) use real programs to richer complex objects for some kinds of APIs; and v) consider more complex objects for `Class`, `Method`, and so on.

6.1 Future Work

As future work, we intend to give more evidences that our technique can be applied in other APIs. We aim at considering new parameters values and methods, and implement test cases creation considering more than one API method per test case. We intend to evaluate Algorithm 1 in different operating systems, and using new versions of JVMs that fixed the detected non-conformance candidates found. We intend to collect bugs from JVM bug trackers, and see whether our technique can detect them. We plan to measure classes, lines, and blocks coverage of executing generated test cases APIs implementations. We will extend our technique to other programming languages (e.g. C#).

Bibliography

[1] Wolfgang Ahrendt, Wojciech Mostowski, and Gabriele Paganelli. Real-time Java API Specifications for High Coverage Test Generation. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 145–154, 2012.

[2] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated Unit Test Generation for Classes with Environment Dependencies. In *Proceedings of the Automated Software Engineering*, pages 79–90, 2014.

[3] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Corporation, 2nd edition, 1990.

[4] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An Empirical Comparison of Compiler Testing Techniques. In *Proceedings of the International Conference on Software Engineering*, pages 180–190, 2016.

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

[6] Yoonsik Cheon. Automated random testing to detect specification-code inconsistencies. In *Proceedings of the International Conference on Software Engineering Theory and Practice*, pages 112–119, 2007.

[7] Yoonsik Cheon and Gary Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 231–255, 2002.

[8] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of Comments and Method Implementations: a Dataset and an Empirical Investigation. *Software Quality Journal*, 26(2):751–777, 2018.

[9] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[10] Ira Forman and Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1st edition, 1999.

[12] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the Foundations of Software Engineering*, pages 416–419, 2011.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 1st edition, 2016.

[14] P. E. Greenwood and M. S. Nikulin. *A Guide to Chi-Squared Testing*. Wiley-Interscience, 1st edition, 1996.

[15] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications. In *Proceedings of the Foundations of Software Engineering*, pages 993–997, 2016.

[16] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects. In *Proceedings of the International Conference on Software Engineering*, pages 643–653, 2018.

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

[18] Laura Inozemtseva and Reid Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the International Conference on Software Engineering*, pages 435–445, 2014.

[19] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT press, 2006.

[20] Ron Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *International Joint Conferences on Artificial Intelligence*, pages 1137–1143, 1995.

[21] Lämmel, Ralf and Pek, Ekaterina and Starek, Jürgen. Large-Scale, AST-Based API-Usage Analysis of Open-Source Java Projects. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1317–1324, 2011.

[22] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study. In *Proceedings of the International Conference on Software Engineering*, pages 507–518, 2017.

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

[24] Owolabi Legunsen, Wajih Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How Good Are the Specs? A Study of the Bug-finding Effectiveness of Existing Java API Specifications. In *Proceedings of the Automated Software Engineering*, pages 602–613, 2016.

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

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

[27] Benjamin Livshits, John Whaley, and Monica Lam. Reflection Analysis for Java. In *Proceedings of the Asian Conference on Programming Languages and Systems*, pages 139–160, 2005.

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

[29] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal Model-driven Program Refactoring. In *Proceedings of the Fundamental Approaches to Software Engineering*, pages 362–376, 2008.

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

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

[32] Microsoft. .NET Reflection API Specification. https://docs.microsoft.com/en-us/dotnet/api/system.reflection?view=netcore-2.1.

[33] Alysson Milanez. *Fostering Design By Contract by Exploiting the Relationship between Code Commentary and Contracts*. PhD thesis, UFCG, 2018.

[34] Alysson Milanez, Dênnis Souza, Tiago Massoni, and Rohit Gheyi. JMLOK2: A Tool for Detecting and Categorizing Nonconformances. In *Proceedings of the Brazilian Conference on Software*, pages 69–76, 2014.

[35] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. Prioritizing runtime verification violations. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2020.

[36] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, 2018.

[37] Melina Mongiovi, Gustavo Wagner, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling Testing of Refactoring Tools. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 371–380, 2014.

[38] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[39] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping Through Hoops: Why do Java Developers Struggle with Cryptography APIs? In *Proceedings of the International Conference on Software Engineering*, pages 935–946, 2016.

[40] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the Foundations of Software Engineering*, pages 466–476, 2013.

[41] Oracle. Class TreeMap Javadoc Specification. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html.

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

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

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

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

[46] Oracle. Java Collections API Javadoc Specification, 2014. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html.

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

[48] Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering*, pages 75–84, 2007.

[49] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-directed Random Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 87–96, 2008.

[50] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 1st edition, 2007.

[51] F. Pfenning and C. Elliott. Higher-order abstract syntax. *SIGPLAN Notices*, 23(7):199–208, 1988.

[52] Long Pham, Quang Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. Testing Heap-based Programs with Java StarFinder. In *Proceedings of the International Conference on Software Engineering*, pages 268–269, 2018.

[53] Felipe Pontes, Rohit Gheyi, and Márcio Ribeiro. A Technique to Test APIs Specified in Natural Language (Artifacts), 2020. http://www.dsc.ufcg.edu.br/~spg/thesis.html.

[54] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. Java Reflection API: Revealing the Dark Side of the Mirror. In *Proceedings of the Foundations of Software Engineering*, page 636–646, 2019.

[55] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

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

[57] Martin Robillard and Robert Deline. A Field Study of API Learning Obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[58] Sarfraz Khurshid and Darko Marinov. TestEra: A Novel Framework for Testing Java Programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 22–31, 2003.

[59] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the Computer Aided Verification*, pages 419–423, 2006.

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

[61] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the Automated Software Engineering*, pages 201–211, 2015.

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

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

[64] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

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

[66] Willem Visser, Corina Păsăreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. *SIGSOFT Software Engeneering Notes*, 29(4):97–107, 2004.

[67] Michal Young and Mauro Pezzè. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, Hoboken, NJ, USA, 2005.

[68] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers, 2nd edition, 2009.