ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatically Generating Java Benchmarks with Known Errors

Master's Thesis

Madalina Hurmuz

March 20, 2022

Supervised by Prof. Dr. Peter Müller, Alexandra-Olimpia Bugariu
Department of Computer Science, ETH Zürich

Abstract

With the constant increase in code development, testing has become essential. Because of the higher cost of dynamic testing, static testing is preferred. One of the existing techniques is using static analyzers. These can be tested by using benchmarks. However, generating benchmarks that expose a particular type of issue can be time-consuming if the developers of static analyzers write these benchmarks manually.

In this thesis, we present a technique to *automate* the process of generating valid benchmark programs in Java for null pointer analysis. Although we instantiate our procedure only for the null pointer analysis, the approach is also suited for generalization to other types of static analysis.

Moreover, we explore ways to ensure the quality of the generated benchmarks by defining our own lightweight definition of *equivalence* between programs. Unlike the already existing published papers and archives that describe solutions related to the generation of valid benchmark programs in Java, we focus on a solution that does not require significant human effort to be used in practice.

Acknowledgements

First and foremost, I would like to thank my supervisor, Alexandra-Olimpia Bugariu, for always being attentive and dedicated. The many prolonged discussions and rounds of feedback helped shape most of the ideas presented here. Alexandra would always make time to offer her guidance despite her other commitments, but she would also make sure I had all the resources to advance the project. For instance, she helped me connect with the developers of Facebook Infer, a discussion that would not have been possible otherwise. I also would like to thank Prof. Dr. Peter Müller for allowing me to work on this project under his supervision. From the first semester at ETH Zürich, I knew that I would highly enjoy working on a project in the Programming Methodology group.

Finally, special thanks go to my family and friends for their support and their constant hope that I will succeed.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the past years, there has been a constant interest in the generation of benchmarks [14, 27]. Benchmark programs are widely used to test various tools: e.g., compilers, static analyzers, verifiers, etc. However, it is highly challenging to automatically generate benchmarks that are *simple* (i.e., easy to understand), but also *expressive* (i.e., include various language features). We will only refer to benchmarks that contain issues, not to other types of benchmarks, i.e., that test the performance of a tool. Many of these programs are currently written manually by developers who might introduce their own biases by including previously-reported issues. Moreover, it is very challenging to write benchmarks with high reproducibility (i.e., quality of being reliable).

As our goal is to generate valid benchmark programs, we first look into the more straightforward problem of generating valid programs. For example, the work of Zhendong et al. [21] explores the generation of proper integer C programs by sketching a program's test execution on a subset of inputs and randomly pruning its unexecuted code, i.e., mutation of the source code. On the other side, regarding valid Java programs, the existing tools [14, 27] either require a high level of expertise from the programmers or significant human effort to be used in practice.

As shown in Listing 1.1, we restrict the bigger problem to the generation of benchmarks in Java. This program was written by the developers of the static analyzer Facebook Infer [3] and is a good example to emphasize what is indeed a *simple* benchmark program. The program highlights one of the analyses supported by Infer, i.e., null pointer analysis. When executed, this program is expected to throw a NullPointerException; if the analyzer is *sound* (i.e., does not miss errors), then it should identify the null pointer dereference. Considering that benchmarks can be used to test both

new static analyzers and existing static analyzers for soundness, it is important to automate the process of benchmarks generation. Thus, we focus on developing a technique that automatically generates a test suite that, by construction, exposes a particular class of issues (null pointer exception) and which systematically covers a configurable set of language features in Java.

Listing 1.1: Simple Java program that throws a NullPointerException. Such a program will not be generated, but it will serve as input.

```
class Hello {
   int test() {
      String s = null;
      return s.length();
   }
}
```

Our approach starts by designing the input programs: simple programs, called *root programs*, that throw a null pointer exception. These are *transformed* by using rewrite rules that either modify the existing code or add other pieces of code that maintain the existing behaviour (i.e., throwing the null pointer exception). We refer to such rewrite rules as *transformations*. During the generation, the benchmarks increase in length and expressiveness, and the generation stops when we reach the desired level of complexity.

In the following chapters, we illustrate our technique for creating null pointer analysis benchmarks, but our approach's main ingredients are generic. Therefore, our solution can be considered for other types of analyses. Section 8.2 explains the minimal adjustments that need to be done to consider the *generalization* of the technique.

1.2 Contributions

In this thesis, we make three core contributions:

1. We develop a technique to automatically generate Java benchmarks for null pointer analysis.
2. We improve our generation technique by defining our own definition of *equivalence* that enables us to remove redundant benchmark programs from the final test suite.
3. We manage to find soundness issues in the Facebook Infer static analyzer. Moreover, we reported these issues to the developers of Infer.

1.3 Outline

This introductory chapter is followed by presenting the necessary background knowledge (Chapter 2). It includes static analyzers, null pointer analysis, and the reasoning behind Facebook Infer. Chapter 3 presents a high-level overview of our technique, and, in Chapter 4 and Chapter 5 (implementation), we offer more details about the generation of benchmarks. We continue by presenting the results of our experimental evaluation in Chapter 6. In Chapter 7, we compare our solution with the existing ones and highlight the advantages and disadvantages of the proposed solution. Finally, we conclude in Chapter 8 and describe possible new directions for future work.

Chapter 2

Background

The purpose of this chapter is to provide the necessary theoretical and technical background information for the main ideas of this work. We discuss static analyzers (see Sec. 2.1), testing them being a practical application of generating benchmarks, then illustrate the theory behind null pointer analysis (see Sec. 2.2). Lastly, we learn more about Facebook's own static analyzer, Infer, that we use for the evaluation phase in our approach (see Sec. 2.3).

2.1 Static analyzers

Static analyzers are tools known to identify code defects before a program is actually executed. They are commonly used between the coding phase and the unit testing phase. Although manual reviews are highly used to identify bugs in source code, automated tools are preferred. Besides the advantage of automation, static analyzers cover every possible execution path, compared to traditional testing.

Static analyzers are also used in safety and security-critical systems, in areas where proving the absence of errors is essential. For example, Astree [19] is a static program analyzer aiming to prove the absence of run time errors in programs written in C. It is able to analyze complex C programs, but it does not support dynamic memory allocation and recursion. Thus, Astree is successful in analyzing many embedded programs in several areas: earth transportation, medical instrumentation, aeronautic applications, etc.

Formally, consider the following definition (from [1]):
A **static analysis tool** S analyzes the source code of a program P to determine whether it satisfies a property φ, such as "P never deferences a null pointer". However, for any nontrivial property, there is no general automated method to determine whether P indeed satisfies φ (from Rice's

theorem). In the practical static analysis setup, a static analysis tool can be wrong in one of two ways: it might say that P violates φ even though it doesn't (if soundness) or it may miss real violations of φ (if completeness).

Thus, we could examine static analyzers from two perspectives: soundness and completeness. This thesis only focuses on soundness (i.e., the analyzers should not miss errors), not completeness, i.e., the analyzers only report actual errors (not false positives).

2.2 Null pointer analysis

A null pointer analysis is a type of static analysis that detects *potential* null pointer dereferences and reports them as error messages or warnings [17]. This enables developers to add explicit checks; thus the runtime errors will not be triggered anymore. For example, some of the most popular null pointer analyzers are Infer (see details in Sec. 2.3) and the Java Null Checker [23]. Programmers are encouraged to use annotations in their code to denote which variables, fields, return values, etc., are expected to be non-null, i.e., *NotNull*, and which ones are not, i.e. *Nullable*. Even a simple flow in the static analyzer is able then to find scenarios in which a non-null variable is being set with a nullable variable.

In general, using a static null pointer analysis can bring a higher degree of reliability to the specific project. However, it is not that simple to adhere to it, as the analysis can report false positives (i.e., the locations do not trigger a null pointer dereference, but they are detected as potential warnings) by maintaining its quality of being conservative [17].

2.3 Facebook Infer

Facebook Infer [4] is an open-source static program analyzer, used to identify bugs before the actual code produced by Facebook engineers is shipped. It supports multiple programming languages, such as Java, C, and Objective-C and its logic was written in OCaml.

Facebook decided to have their own static analyzer to complement traditional dynamic testing (i.e., the method to test the dynamic behaviour of software code by providing input values and checking if the output is expected or not). Therefore, dynamic testing offers the possibility to have individual runs through the code and check for correctness, and static analysis can prove the absence of errors while testing cannot.

The main idea behind Infer is using mathematical logic to do symbolic reasoning about program executions, approximating the reasoning an actual developer would do when one examines a program. Moreover, it is a

static analyzer that uses abstract interpretation [12], a general methodology for designing static analyzers that are sound by construction (even though the *design* of the analyzers provides theoretical guarantees, we still need to test them as the *implementation* may not coincide with the *design*). On top of the open-source static analysis tool, Infer's team provides a framework for developing abstract interpretation-based checkers (see [2]).

Nowadays, the analyzer is specialized in reporting issues caused by null pointer accesses and resource and memory leaks. As Facebook states, these categories of errors account for the largest percentage of application failures. Our choice of focusing the project on null pointer analysis is not necessarily correlated with Facebook's previous statement. However, we considered the constant work in the area of null pointer analysis and the availability of popular null pointer analyzers that enables us to use them during the evaluation experiments.

Chapter 3

Overview

This chapter gives a high-level overview of the technique presented in this report (Sec. 3.1). Then, we show our technique on a simple example (Sec. 3.2). We offer details about the example, the input program, and the resulted benchmark after applying one *transformation*.

We reiterate over the problem that writing benchmark programs manually by the developers is achievable, especially for simpler programs, but, once we want to support a wider range of language features and bigger programs, this becomes a more difficult process. Having an automated way of generating benchmarks is very important for simplifying this time-consuming process and thus reducing the human effort.

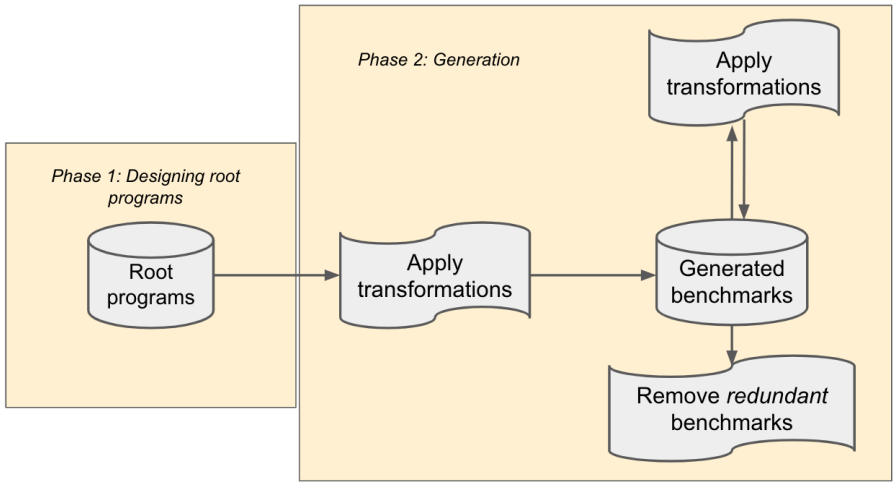
3.1 Technique overview

In Fig. 3.1, we present the high-level overview of the technique. The input consists of *root programs*, simple Java programs that trigger a null pointer exception.

These programs contain two classes: a generic type class and the public class of the program. The generic class contains fields of both primitive and reference types (the two categories of types in Java). Objects of the generic type class are created in the *main* method of the public class (i.e., the entry point of the program) and, then, we use these objects in such a way that every execution of the program ends with a null pointer exception. We clarify the structure of these root programs in Chapter 4. Regarding the number of root programs, we designed 5 programs in order to include different scenarios that could end in a null pointer exception. On this small set of root programs, we apply *transformations*, i.e., rules that can either modify or add code, while preserving the null pointer exception behaviour. Transformations are applied incrementally (see Fig. 3.2), up to a desired

complexity level of the generated benchmarks. The motivation of applying transformations in an *incremental* way is getting to programs that are bigger and more complex, and, hopefully, more challenging for the analyzer to detect potential issues. Regarding complexity, we think of a program as being *complex* if it has a relatively large number of statements and a developer will find the null pointer exception issue, but he might require more time comparing, for example, with understanding the root programs, for which it is straightforward to find the cause of the issue. Also, transformations can be instantiated by using code snippets extracted from the Java Language Specification documentation (see Sec. 4.3). During the *generation phase* (see Fig. 3.1), we *remove redundant benchmarks* in order to end up quicker with a more diverse and complex set of benchmarks. In the next section, we provide an example of such redundant benchmarks. The way in which we remove them is related to the definition of *equivalence*, explained in Section 4.5. This definition is based on the set of relevant statements corresponding to the null pointer analysis (see Sec. 4.4).

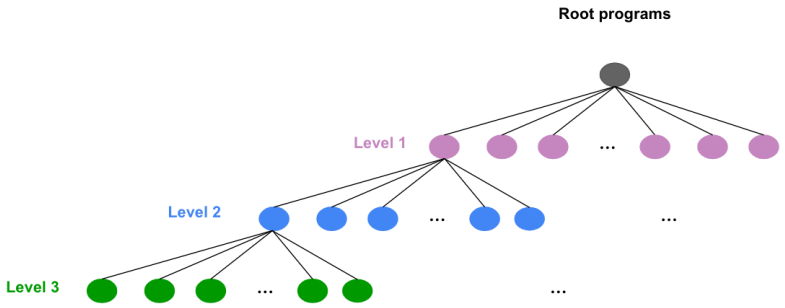
Figure 3.1: High-level overview of the technique: highlighting the first phase in which root programs were designed and the second phase that contains the design and application of transformations, the actual generation of the benchmark programs and the removal of redundant programs.



3.2 Example

Listing 3.1 shows an example of a program, that we considered as input for the technique. This is a very simple program, that makes an object point to *null* (line 8) and then tries to access a field f on the null object (line 9), statement that ends the execution of the program with a null pointer exception. It should be noted that we consider all fields and methods to

Figure 3.2: Applying transformations incrementally, starting from the root programs. In the *generation tree*, at level 1, we consider all benchmark programs obtained after the application of one transformation, at level 2, all benchmark programs obtained after the application of two transformations, etc.



be *public*, so no other errors are triggered due to inaccessibility. We have designed 5 such root programs as input.

Listing 3.1: Root program.

```
class Box<T extends Object> {
   ...
   public int f;
   public Object g;
}
public class RootProgram1 {

   public static void main(String[] args) {
       Box<Integer> box = null;
       System.out.println(box.f);
   }
}
```

In terms of transformations, we have designed and implemented 13 (these are described in Chapter 4). Moreover, all our transformations are designed to modify or add code such that the benchmark programs resulted will be compilable. It should also be stated that:

- Each root program is designed to throws **exactly** one exception.
- The inserted code does not throw exceptions itself; when executed, the resulting benchmark still raises the **same** exception as its root program (only the position may change).

As an example, we look at a transformation that adds code by creating a

new object of type *Box* and populating its fields, while preserving the behaviour of the input program. Listing 3.2 presents the resulted benchmark after applying this transformation on the program from Listing 3.1. Although this transformation does not involve branching or looping, it changes the sequential control flow by increasing the number of statements that the analyzer will need to consider.

Listing 3.2: Benchmark resulted after applying a transformation that adds a redundant object and populates its fields on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     ...
3     public int f;
4     public Object g;
5  }
6  public class Bennchmark1 {
7
8     public static void main(String[] args) {
9         Box<Integer> axcmc = new Box<Integer>();
10        axcmc.f = 1103192343;
11        axcmc.g = new Object();
12        Box<Integer> box = null;
13        System.out.println(box.f);
14    }
15 }
```

Listing 3.3: Benchmark equivalent with the one from Listing 3.1.

```
1  class Box<T extends Object> {
2     ...
3     public int f;
4     public Object g;
5  }
6  public class RootProgram1 {
7
8     public static void main(String[] args) {
9         Box<String> box = null;
10        System.out.println(box.g);
11    }
12 }
```

Related to redundancy, we consider two benchmarks to be *equivalent* if they have the same sequence of statements. For example, program from Listing 3.1 is **not** *equivalent* with the program from Listing 3.2, but it is *equivalent* with the program from Listing 3.3, as, in both programs, first, we create an object *box* (either with type *Box<Integer>* or *Box<String>*) that is pointing to *null* and, afterwards, we access one of its fields (either *f* or *g*). These fields

have different types and, yet, we consider field accesses of different types as being equivalent, because, from the point of view of null pointer analysis, a field access on a *null* reference triggers the same behaviour of a null pointer exception.

To continue, on both programs from Listing 3.1 and Listing 3.2, we can apply transformations, thus any resulted benchmark program (i.e., Listing 3.2) is part of both the final test suite and can also be reused as input for generation.

Chapter 4

Technical section

In this chapter, we clarify the structure of root programs (see Sec. 4.1) and how transformations are designed and what are the basic construction steps (see Sec. 4.2). In Sec. 4.3, we highlight the code snippets extraction from the Java Language Specification documentation, continuing with defining the *relevant statements for the null pointer analysis* in Sec. 4.4. We also present the two main procedures of the technique: *the equivalence algorithm* (see Sec. 4.5) and *the generation algorithm* (see Sec. 4.6). Lastly, we present special cases when designing transformations (see Sec. 4.7).

4.1 Root programs

We created 5 root programs, that serve as input for the generation technique. All these programs share the same structure. There are two classes: *Box<T extends Object>*, generic class, and *RootProgram*, the public class of the program, whose method *main* represents the entry point.

The class *Box* was created to enable the usage of different types (i.e., to instantiate Box<Integer>, Box<String>, Box<A>, etc.). Moreover, all *boxes* can access the same fields: field *f* of type int and field *g* of type Object. There are two constructors: the default one and a parameterized constructor with parameter of type *T*. We have also redefined built-in methods in Java, such as *toString()*, to be able to call methods on objects of this generic class.

The *main* method of the public class of the program consists of a few statements: creating objects of type *Box*, followed by assignments, declarations, accessing fields or calling methods that trigger the null pointer exception. For example, in Listing 3.1, we create an object of type *Box<Integer>* that points to null and then access field *f*.

4.2 Transformations

Transformations are rules that either modify or insert code while preserving the behaviour of the program. Even though these might be correlated to mutations, a transformation is more complex, as it follows a set of clear steps, while a mutation can be as straightforward as an operator switch (i.e., from $<=$ to $>=$).

In this section, we start by explaining the design of transformations (see Subsec. 4.2.1), followed by types of transformations (see Subsec. 4.2.2).

4.2.1 Design of transformations

The motivation behind designing transformations is allowing us to construct more complex type-correct programs. The final test suite will consist of bigger programs that test the analyzer ability to find the exception.

To design transformations, we took in consideration two aspects: the need to increase the number of Java language features (except the ones already present in the input programs, *root programs*) and the need to alter the existing execution flow in the programs as much as possible (i.e., by adding branching, method calls), without changing the behaviour (i.e., ending the execution of the program with the same null pointer exception) of the resulting benchmarks.

Nowadays, Java is one of the most popular programming languages. Every six months, a new Java version is released and these constant releases bring new features to the language. For example, since Java 8, features like allowing private methods in interfaces or local-variable Type Inference were introduced [5]. Since there is this constant increase in Java language features, we decided to use a moderate set, consisting of: declarations, assignments, if-else statements, method calls, arrays, and non-null annotations (introduced with Java 8). Needless to say, this set could be at any time expanded, as the actual implementation facilitates the introduction of transformations that could introduce other features.

4.2.2 Types of transformations

We designed 13 transformations and we present the construction of a few in this subsection. In Appendix A, there are examples of other transformations.

Instantiate through method call transformation extracts the right-hand side of an assignment (arbitrary expression, except from method calls to avoid infinite recursion) into a method in the corresponding declared class of the object that is being assigned. The return type of the method will be the declared type of the object. In Listing. 4.1, object *box* of type *Box<Integer>* points to

null. After the transformation, it gets instantiated by calling method *foo* that returns null.

Listing 4.1: Resulted benchmark from applying the *instantiate through method call transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5     public Box<Integer> foo() { return null;}
6  }
7  public class RootProgram1 {
8     public static void main(String[] args) {
9         Box<Integer> box = new Box<Integer>().foo();
10        System.out.println(box.f);
11    }
12 }
```

Move null pointer exception statement in a new method transformation, as the name suggests, extracts the null pointer exception statement in a new method in a new defined class. In Listing. 4.2, the access to field *f* of object *box* that triggers the exception is moved in method *test*, in class *Hello*. This change needs to still make the final program compile, thus method *test* gets a parameter with the type of the declared type of object *box*: *Box<Integer>*. As a result, we declare a new object of type *Hello* and, on this, we call method *test* with the null reference.

Listing 4.2: Resulted benchmark from applying the *move null pointer exception statement in new method transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5  }
6  public class RootProgram1 {
7     public static void main(String[] args) {
8        Box<Integer> box = null;
9        Hello h = new Hello();
10       h.test(box);
11    }
12 }
13 class Hello {
14    public void test(Box<Integer> box) {
15       System.out.println(box.f);
16    }
17 }
```

Add if transformation is presented in Listing. 4.3. We add the method *addIf*, that based on the values of the variables *r1* and *r2*, can end the execution of the program with the exception. Variable *r1* is randomly assigned with a value from 0 to INT_MAX and, for example, for $r1 = 10$ and $r2 = 0$, the null pointer behaviour is preserved. Even though not all possible executions end with the null pointer exception, this is not relevant, as we are interested only in testing soundness of analyzers. To add *if* statements or *if-else* statements, we have designed several transformations. These transformations have the ability to break the execution flow of the programs on which we apply them. On a high level, such transformations require boolean conditions and some code on each branch. The boolean condition is either passed as parameter, explicitly set to *true* or based on two variables that gets randomly assigned (see Listing 4.3). On the true branch, we always keep the initial behaviour (i.e., the null pointer exception), respectively, on the false branch, we can add redundant code (e.g., adding a redundant object and populating its fields in Listing 3.2). The instantiation of this redundant code is enabled by using the code snippets extracted from the Java Language Specification documentation.

Listing 4.3: Resulted benchmark from applying the *add if transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5  }
6  public class RootProgram1 {
7     public static void main(String[] args) {
8        Box<Integer> box = null;
9        addIf(box);
10    }
11    public static void addIf(Box<Integer> box) {
12       int r1 = new Random().nextInt();
13       int r2 = 0;
14       if (r1 > r2) {
15          System.out.println(box.f);
16       }
17    }
18 }
```

Since applying a transformation might move the null pointer exception, we had to keep track of this information. Initially, the null pointer exception is represented by the last statement in the *main* method. During the application of transformations, we update, if needed, the position of the null pointer exception; thus, the line statement that causes the exception is always known

by construction.

In addition, the application of a transformation can be either successful or not. For example, we defined a transformation that adds *NonNull* annotations. These annotations are inserted for the redundant objects that are declared by other transformations (e.g., see Listing 3.2). Thus, if no redundant objects were added to the program, objects for which we know by construction that they do not point to *null*, the transformation of adding *NonNull* annotations is not applicable, so it results in an unsuccessful operation.

4.3 Snippets from Java Language Specification

Many of these transformations add redundant code (i.e., on the false branch of an if statement). Thus, our goal was to extract a set of code snippets that are compilable and can help us to instantiate transformations and to add more language features. Regarding this, we considered the Java Language Specification documentation, more specifically, JLS 15 [8]. We used the *HTML* version, as it is easier to extract code examples by checking *HTML* tags for *programlisting*. During the extraction, we encountered a relatively large number of code snippets that actually serve as example to highlight compile-time errors (see Listing 4.6). We had to remove such snippets and, as a result, we gathered a set of 36 compilable code snippets (see example in Listing 4.4).

Listing 4.4: Compilable code snippet from JLS.

```
1  class S         { int x = 0; }
2  class T extends S { int x = 1; }
3  class Test1 {
4    public static void main(String[] args) {
5        T t = new T();
6        System.out.println("t.x=" + t.x + when("t", t));
7        S s = new S();
8        System.out.println("s.x=" + s.x + when("s", s));
9        s = t;
10       System.out.println("s.x=" + s.x + when("s", s));
11   }
12   static String when(String name, Object t) {
13       return " when " + name + " holds a " + t.getClass() + " at
             run time.";
14   }
15 }
```

Considering the compilable code snippet from JLS from Listing 4.4, we have designed the *insert redundant method from class transformation* that randomly

selects a method from one of the classes declared in the code snippet (see Listing 4.5). Since the successful application of this transformation, as for the rest of them, should ensure by construction a resulting compilable program, we considered several issues that we had to fix. Sec. 4.7 shows these special cases.

Listing 4.5: Resulted benchmark from applying the *insert redundant method from class transformation* by using the JLS code snippet from Listing 4.4 on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5  }
6  public class RootProgram1 {
7     static String when(String name, Object t) {
8        return " when " + name + " holds a " + t.getClass() + " at
             run time.";
9     }
10
11    public static void main(String[] args) {
12       Box<Integer> box = null;
13       System.out.println(box.f);
14    }
15 }
```

Listing 4.6: Code snippet from JLS that is not compilable: *f* is declared as a *final* variable.

```
1  class Test {
2     static int v;
3     static final int f = 3;
4     public static void main(String[] args) {
5        int i;
6        i = 1;
7        v = 2;
8        f = 33; // compile-time error
9        System.out.println(i + " " + v + " " + f);
10    }
11 }
```

4.4 Relevant statements

As we specifically consider null pointer analysis, it is required to define the set of statements that are relevant for this type of analysis. This set is then used to understand and define the notion of *equivalence* in Sec 4.5.

Table 4.1 shows the statements relevant for a null pointer analysis. These statements are chosen by using the existing literature references (see [15], Fig. 2) and then extending them to fit the design of root programs and transformations, i.e., we want to match constructors for generic classes. We acknowledge that, by definition, a relevant statement for the analysis is considered as a statement that changes the abstract state [12]. However, we included in Table 4.1 statements such as, *print 'expr'*, because 'expr' might be relevant for the analysis (i.e., 'expr' represents a *new* construct) and, thus, the matching of the whole statement enables us to use a simpler matching algorithm. Examples of how program statements are matched to ids are presented in Listing 4.7.

Table 4.1: Program statements which are relevant for a null pointer analysis. *x* and *y* are objects, *f* is a field, *T* is a defined class, *m* is a method, *args* := ∅ | [arg0, arg1, ..., argn], where *argi* := new T<>(args) | T< *Type* >(args) | new T() | null | x.m'(args) | x.f | constant, *expr* := new T<>(args) | T< *Type* >(args) | new T() | null | x.m'(args) | m'(args) | x.f, and, *m'* is a non-void method.

Id	Program statement
S1	x := new T<>(args) | T< *Type* >(args) | new T()
S2	x := null
S3	x := y
S4	x := y.f
S5	x.f := y
S6	x.m(args)
S7	return 'expr'
S8	print 'expr'
S9	m(args), static method
S10	x := y.m(args)

Listing 4.7: Matching relevant statements to ids.

```
1 Box<Integer> b = new Box<Integer>(); // S1
2 System.out.println(b.f); // S8
3 Box<String> b1 = b2.m(null); // S10
```

4.5 Equivalence

In this section, we deepen the understanding of the *equivalence* notion by stating the assumptions we have made (see Subsec. 4.5.1), the lightweight definition we considered (see Subsec. 4.5.2), and, the actual algorithm that we developed (see Subsec. 4.5.3).

4.5.1 Assumptions

In this work, we do not make any decisions based on the design (except the relevant statements that are taken from the analyzer's design) and/or implementation of the static analyzers. For example, we did not annotate the code with *Nullable* or *Non-null* annotations (see Sec. 2.2 about null pointer analysis). However, our approach is not completely agnostic, as, in order to define the *equivalence* notion, we assume that, for the analyzer, order is relevant, i.e., the logic of the analyzer does consider the control flow of the program.

4.5.2 Definition

In the existing literature, there are several definitions of what can be referred as *equivalence* of two programs [22, 26]. However, there is no specific paper that defines such a property with regard to analyzers.

Currently, there is the definition of equivalence between two programs *up to a set of objects A (i.e., ignoring the values of these objects)* (see [6]). Also, there are notions that capture the meaning of equivalence at the level of the program, i.e., *path-based equivalence* [9]. Another option is to refer to the property of *structural equivalence*, but a structural equivalence is often *stronger* than a *behavioural equivalence* [29].

Consequently, we decided to give our *own* lightweight definition of *equivalence*.

We consider benchmark *P1* and benchmark *P2* equivalent if the static analysis of both benchmarks results in the analyzer processing the same sequence of relevant statements.

The intuition behind this definition is that, if we would have the implementation of the analyzer and we could instrument the code such that we can see the path that the analyzer actually takes, this path will be the same for both *P1* and *P2*. Thus, if the path is the same for the programs, it means that the analyzer behaves exactly the same from our point of view, so it is necessary to only keep one of the two benchmarks.

4.5.3 Algorithm

In Algorithm 1, we receive as input, two programs, *P1* and *P2* and return a boolean that indicates whether the two programs are *equivalent*. Initially, we find the entry point of the two programs (considering that we might have multiple classes, but only one main method as entry point) in lines 1 and 2. Then, we delete the statements that are irrelevant for the null pointer analysis, i.e., are not matched to any id from Table 4.1. In line 7, we iterate

over the lists *s1* and *s2*. The looping stops when we encounter the first different matched id in line 10. Otherwise, we return *true* in line 11.

Algorithm 1: Equivalence algorithm for two arbitrary benchmark programs.

```
   Data: P1, P2 - two distinct benchmark programs,
   RS - relevant statements from Table 4.1
   Result: Boolean that is true if the two programs are equivalent.
   // getMainMethod(p) returns the entry point method of a program p.
1  m1 = getMainMethod(P1);
2  m2 = getMainMethod(P2);
3  deleteIrrelevantStatements(m1, RS);
4  deleteIrrelevantStatements(m2, RS);
5  s1 = getStatements(m1);
6  s2 = getStatements(m2);
7  for i ∈ min(s1.length, s2.length) do
       // matchRelevantStatement(s, RS) returns id from Table 4.1.
8      id1 = matchRelevantStatement(s1[i], RS);
9      id2 = matchRelevantStatement(s2[i], RS);
10     if (id1 != id2) return false;
11 return true;
```

4.6 Generation

The entry point of our approach is the generation Algorithm 2. As initially desired, the algorithm is parametrizable: **R** - set of root programs, **T** - set of transformations, **S** - set of relevant statements (from the analyzer's design), and, **L** - integer that shows up to which level the generation runs. To run the algorithm for null pointer analysis, we can instantiate R with the root programs that trigger null pointer exception from Sec. 4.1, T with the transformations defined in Sec. 4.2, and S with the relevant statements defined in Sec. 4.4 (see Table 4.1).

Algorithm 2 starts with declaring the *currentLevel* integer in line 1 and the set of generated benchmark programs B that is empty in line 2. Then, up to level L, we read the benchmarks generated at level *currentLevel* (note: initially, for level 0, we consider the generated benchmarks to be the root programs, R.), and, on each of these programs, we try to apply once at a time a transformation from the set T. The application of a transformation on a program might result in two outcomes, depending whether the application was successful, i.e., it makes sense to apply the transformation and it results in a compilable program. If a successful application, the next check requires getting the set of all programs that were generated at the previous level or up

to now at this level. Against this set of programs, we check for equivalence in line 11. If no equivalent already generated program is found, we add program p in the result set B in line 14. In the end, we return the set of resulted benchmarks, B.

Algorithm 2: Parametric algorithm to generate benchmarks for a given type of analysis.

```
   Data: R - set of root programs,
         T - set of transformations,
         RS - relevant statements from Table 4.1,
         L - integer that shows up to which level the generation runs
   Result: B - set of generated benchmark programs
 1 currentLevel = 0;
 2 B = ∅;
 3 while currentLevel < L do
       // readBenchmarks(currentLevel) returns the set of benchmarks generated at level
          'currentLevel'.
       // readBenchmarks(0) returns R.
 4     P = readBenchmarks(currentLevel);
 5     for p ∈ P do
 6         for t ∈ T do
               // addTransformation(p, t) applies transformation t on program p.
 7             success = addTransformation(p, t);
 8             if success then
 9                 isRedundant = false;
                   // readBenchmarksLevels(currentLevel, currentLevel - 1) returns the set
                      of benchmarks generated up to now at the 'currentLevel', and the
                      benchmarks generated at the 'currentLevel' - 1 level.
10                 for g ∈ readBenchmarksLevels(currentLevel, currentLevel − 1) do
                       // equivalenceCheck(p, g, RS) defined in Alg. 1.
11                     isRedundant = equivalenceCheck(p, g, RS);
12                     if (isRedundant) break;
13                 if !isRedundant then
14                     B = B ∪ p;
15     currentLevel++;
16 return B;
```

For example, Listing 4.8 shows a benchmark generated by applying two transformations, *instantiate through method call* and *move null pointer exception statement in a new method*. This benchmark can result from the generation algorithm; thus, it is returned in the set B.

Listing 4.8: Resulted benchmark from applying the *instantiate through method call transformation* and the *move null pointer exception statement in a new method transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5     public Box<Integer> foo() { return null;}
6  }
7  public class RootProgram1 {
8     public static void main(String[] args) {
9        Box<Integer> box = new Box<Integer>().foo();
10       Hello h = new Hello();
11       h.test(box);
12    }
13 }
14 class Hello {
15    public void test(Box<Integer> box) {
16       System.out.println(box.f);
17    }
18 }
```

4.7 Special cases

There are a few transformations that introduce various challenges in order to produce a compilable benchmark program. One example is represented by the transformation that adds a redundant method in the declaration of an existing class (see example from Listing 4.5). This redundant method is chosen from a random class among the JLS code snippets.

The special cases that we had to overcome are:

- Cannot have multiple main methods with the same signature: *main (String [])*. In Java, we can have multiple main methods only if they have different signatures (method overloading).
- The method that we want to add might use fields declared in the class from which we extracted the method. Thus, we need to add these fields too. Moreover, we also solve the possible issue of having name conflicts between the new fields added and the existing ones.

Chapter 5

Implementation

This chapter offers details about the implementation setup in Sec. 5.1, continues by offering an overview of the most relevant classes in Sec. 5.2, and, by iterating the steps needed to add a new transformation in Sec. 5.3. It ends by discussing some of the shortcomings of the current implementation in Sec. 5.4.

5.1 Setup

The solution is implemented in Java 15.0.2, on a system with macOS Monterey Version 12.2.1. However, the solution is compatible with older versions of Java as well. The used editor is VS Code and we used Maven as a software tool to manage our Java project and automate application builds. One of the advantages of using Maven is that we can easily add package dependencies in the configuration *pom.xml* file.

5.2 Organization

The diagram from Fig.5.1 shows the most relevant classes in the implementation of the solution, and how these classes relate to each other:

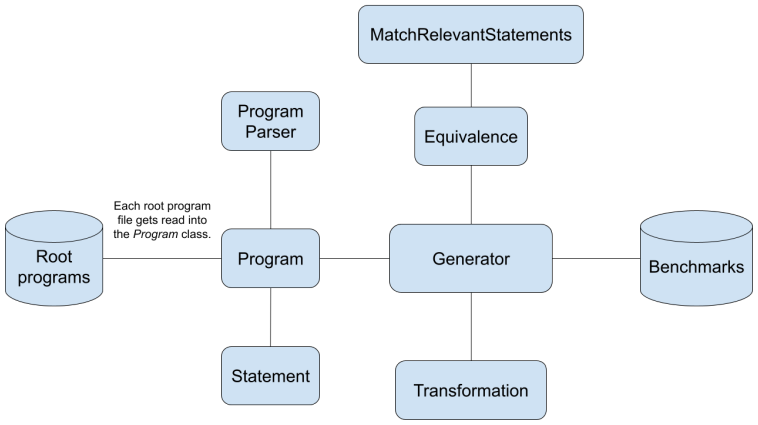
- **Program**: Each root program file gets read into the *Program* class. This class contains information, such as: the file name, the package name, the list of statements, the statement at which the null pointer exception happens. Initially, in a root program, the last statement of the *main* method is the one that triggers the exception. Then, as we add transformations, this changes and we write the line number of the null pointer exception statement as a comment on the first line of the program (i.e., after the package declaration). This information is needed as we do not store all generated benchmark programs in

memory, and decided to instead do I/O operations to either read programs from file or write programs to corresponding files.

- **ProgramParser**: The *ProgramParser* class can parse either a program given its path or its encapsulation in a *Program* variable. After parsing, it keeps three lists: a list of declared classes and/or interfaces, a list of methods, and a list of variable declaration expressions. To get these lists, we used the *Visitor Pattern*, thus we declared classes *ClassVisitor*, *MethodVisitor*, *VarDeclExprVisitor* which extend a *VoidVisitorAdapter*. We need these lists, as all the transformations use parsing information. For example, we might need to extract the beginning line of the *main* method to add redundant objects before the null pointer exception statement (see Listing 3.2).
- **Statement**: A statement keeps a field of type *String*.
- **MatchRelevantStatements**: This class contains the logic to assign corresponding relevant statement ids (i.e., S1, S2, etc.). These ids are kept in an enum *RelevantStatementID*. In case no id from Table 4.1 is assigned to a statement, we return *S0* (unknown id). Please note that this logic needs to be updated in case the solution is reused for another type of analysis, as it might use a different set of relevant statements.
- **Equivalence**: This class implements the procedure from Algorithm 1. It uses the *MatchRelevantStatements* class to go over statements in both input programs and identify possible relevant statement ids and also instantiates two variables of type *ProgramParser* to extract methods information (e.g.. getMainMethod from Algorithm 1 in line 1).
- **Transformation**: This is an abstract class that contains an abstract boolean method *addTransformation*. This returns true if the transformation was applicable and successfully applied. Moreover, it contains an abstract method *setEncoding* that needs to be overridden by each transformation. For example, for transformation *InsertIf*, the encoding is *Ii*, as the convention is to extract the uppercase letters from the transformation name in the encoding string. The encoding is then added to the name of the generated benchmark, i.e., if we apply *InsertIf* on *RootProgram1* and the application of the transformation is successful, the name of the resulted benchmark is *RootProgram1Ii*. This encoding enables us to know how many transformations were added and in which order and also shows the information at which level was the program generated.
- **Generator**: This class implements the procedure from Algorithm 2. It is given an integer that represents the level up to which we run the

generation and a path, that represents the output directory in which the generated benchmark programs are written to. Beside the main logic of generation, it creates a list with all the transformations that are used in the actual generation (i.e., the list of transformations is input for the Algorithm 2) and also contains an utility function to read from the output directory the latest generated programs (based on the encoded names of the benchmarks).

Figure 5.1: Diagram with the most relevant classes in the actual implementation.



5.3 Adding a new transformation

The current implementation allows one to define additional transformations. Once we understand the relations between classes from Fig 5.1, the following steps are needed in order to add a new transformation:

- Extend *Transformation* class and override *setEncoding* and *addTransformation*.
- The logic in *addTransformation* needs to update the line of the null pointer exception statement in case this gets moved. The following convention has to be fulfilled: the new line number is written as comment after the package declaration.
- The method *addTransformation* returns true **only if** the application of the transformation results in a compilable program. This could be obtained either by construction or by explicitly compiling the program that is resulted.

- This is an *optional* step, but we strongly advise that, since we add or modify source code and we have to handle many string parsing tasks, any parsing information is obtained by using the methods from the *ProgramParser* class, instead of defining other parsing utilities that might result in a highly time-consuming coding task.

5.4 Shortcomings

Besides the difficulty in defining the theoretical aspects of this technique (i.e., the equivalence notion), there were many challenges in implementing a reliable and easily extendable generation infrastructure.

Currently, during the generation, especially when the generation goes up a higher level (i.e., more than 4 levels), there are IO intensive workloads and the program might become I/O bound as the time requesting the data (i.e., reading the benchmark programs generated at the previous level) is higher than the time processing it.

Another shortcoming is that, since we do not keep generated benchmarks information in the program memory, we need to write any additional information that might be needed for future transformation applications in the benchmark file. As an example, we write the line number of the null pointer exception as a comment if this does not match the initial convention of the root program that has the null pointer exception statement last in the *main* method. This is not a shortcoming that can influence the current results. However, it represents an essential implementation detail to maintain the high success rate in applying transformations.

Chapter 6

Evaluation

The evaluation consists in two main experiments: the first, related to generation metrics (see Sec. 6.1), and, the second, related to testing the soundness of Facebook's Infer analyzer (see Sec. 6.2).

6.1 Generation experiment

For this experiment, we build and run our Java project with version 15.0.2, on a macOS Monterey 12.2.1, we used all 5 root programs, 9 defined transformations, and the set of relevant statements from Tab. 4.1.

During this experiment, we run the generation algorithm from level 1 (one transformation applied) up to level 5 (up to five transformations applied). As seen in Fig. 6.1 and Fig. 6.2, there were two cases explored depending on whether we enabled the equivalence algorithm, i.e., we removed the redundant programs from the final test suite. We were interested in the trend of two generation metrics: *generation time* and *number of generated programs*.

In terms of generation time, we measured it in seconds. Fig. 6.1 shows the generation time for the entire test suite up to a specific level. We can see up to a **24x** reduction in elapsed time, when we enable the equivalence check. This result is expected, because, even though we enable the computation of equivalence checks, we generate less benchmarks and the IO workload is reduced, i.e., we do not write all the redundant programs.

Regarding the number of generated benchmarks, in Fig. 6.2, we can see up to a **42x** reduction, when we enable the equivalence check. This is an essential improvement, as the resulting set will have the quality of being more diverse.

Figure 6.1: Results: Generation time trend up to level 5. Time difference for level 1 and 2 are relatively small. However, from level 3, generation time is extensively increasing for the case when equivalence check is not enabled. At level 5, there is up to 24x reduction in generation time, generation finishing in approx. 500 seconds.

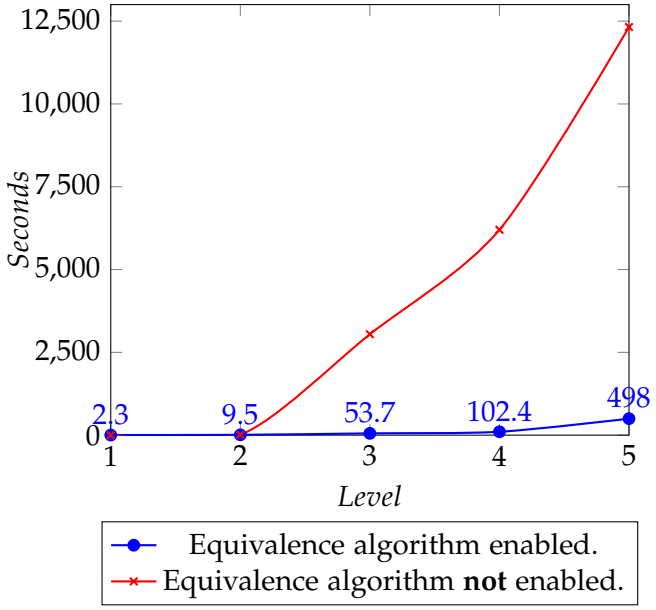
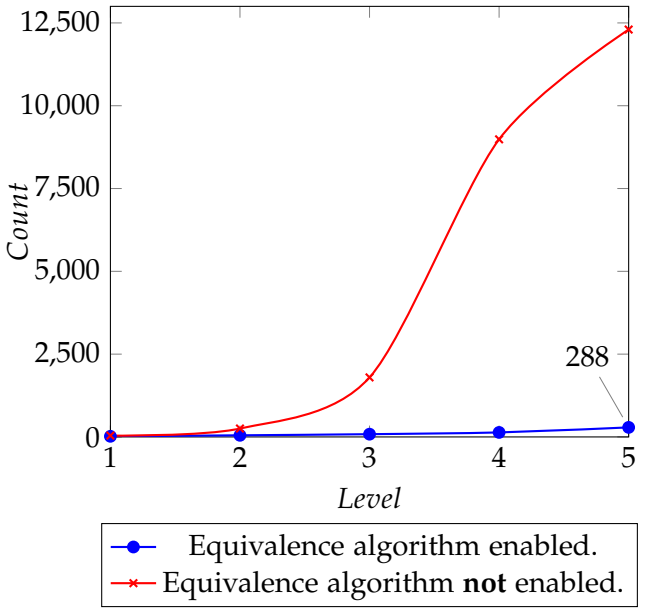


Figure 6.2: Results: Count of generated benchmarks trend up to level 5. The difference between the number of generated benchmarks is relatively small for levels 1 and 2. At level 5, we observe up to a 42x reduction, resulting in a diverse set with approx. 300 benchmarks programs.



6.2 Infer experiment

Infer is an open-source analysis tool, thus we could download it and test its soundness by running it on our set of generated benchmarks.

Regardless of the input language (Infer supports Java, Objective-C, C), when we run Infer, i.e., `infer run -- javac File.java`, there are two main phases:

- The capture phase: The file is compiled and, at the same time, it gets translated by Infer in its own intermediate language in order to enable the second phase.
- The analysis phase: The intermediate files resulted from the previous phase are analyzed by Infer. Specifically, Infer analyzes each function and method separately. When Infer finds an issue, it will stop the analysis for that particular method or function, but will continue its process for the remaining methods and functions.

For a null pointer issue, if we run it on the program from Listing 3.1, the error message reported by Infer is: RootProgram1.java:9: error: NULL_DEREFERENCE; object box last assigned on line 8 could be null and is dereferenced at line 9.

For this experiment, we run Infer version v.1.1.0 on 87 benchmarks generated up to level 3 when the equivalence check is enabled. In Fig. 6.3 and Fig. 6.4, we see the results against two Java versions that we used: 1.8.0 and 15.0.2. Each version is given as a flag option to Infer, and Infer uses it to compile the programs. In both cases, on a quarter of benchmarks, Infer fails to compile them due to its inability to compile programs that use multiple packages. Our benchmarks use multiple packages because, firstly, each benchmark has its own declaration of the generic class *Box* and we need to avoid conflicts. Secondly, the transformations that need a random generator are actually using the Singleton instance used throughout the whole project to ensure that, given a seed, we can replicate experiments. From Infer's documentation, we could not find if there exists a way to provide flags that could enable Infer to compile such programs. On the remaining benchmarks that Infer manages to compile, so they can reach the analysis phase, Infer does not manage to find issues in 8% of the cases for Java 1.8.0, respectively 33.3% of the cases for Java 15.0.2. Currently, the bugs are reported to Infer's developers, and we are waiting for confirmation from their side. This difference in Infer's behaviour is unexpected, as the benchmarks do not use different language features that might not be available in one of the versions. Also, there is no clear indication in Infer's documentation that they might not be able to run Infer on newer versions of Java. In Listings 6.1 and 6.2, there are simplified examples of programs on which Infer is unsound.

Figure 6.3: Running Infer on generated benchmark programs (Java 1.8.0).

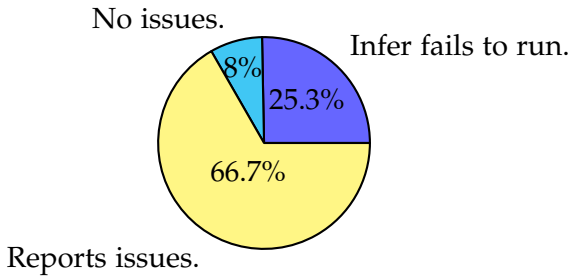
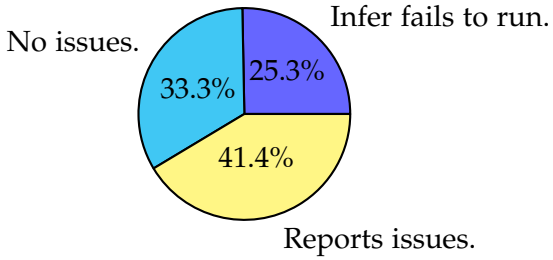


Figure 6.4: Running Infer on generated benchmark programs (Java 15.0.2).



Listing 6.1: Example of false negative by running Infer with Java 1.8.0.

```
1 public class RootProgram {
2    public static void main(String[] args) {
3        String [] array = new String[3];
4        String s = array[0];
5        System.out.println(s.toString()); // Null pointer exception.
6    }
7 }
```

Listing 6.2: Example of false negative by running Infer with Java 15.0.2. However, Infer reports issue for this example when running with Java 1.8.0.

```
1  class Box<T extends Object> {
2     ...
3     public int f;
4  }
5  public class RootProgram {
6     public static void main(String[] args) {
7         Box<Integer> box = null;
8         System.out.println(box.f); // Null pointer exception.
9     }
10 }
```

Moreover, the Infer experiment reveals other two aspects. The first is

related to the impact of transformations in finding the bugs. Regardless of the number of transformations applied to the initial root program, if Infer manages to report the bug for the initial program, it will also be able to detect the bug in the benchmarks that originated from it. The second aspect concerns how Infer behaves on two programs that we consider equivalent. As we would expect, if we run Infer on two programs that we consider equivalent, Infer proves consistency, as it behaves in the same way: either reports the issue or shows no error messages.

Chapter 7

Related work

In this chapter, we compare our technique to closely related lines of work: refactorings (see Sec. 7.1), generation of benchmarks (see Sec. 7.2), and, testing of static analyzers (see Sec. 7.3).

7.1 Refactorings

Refactoring [7] is a way to restructure an existing piece of code, by altering its internal structure without changing its external behavior. The majority of existing Java IDEs offer the possibility of refactoring code. For example, in Eclipse, there are the following options: renaming variables and methods, renaming packages, renaming classes and interfaces, extracting a class, extracting an interface, extracting a superclass, extracting a method, extracting local variables, extracting a constant, inlining, push down and pull up (parent-child relationship), changing a method signature, or moving methods to another existing class.

Our current transformations were not explicitly based on these Eclipse refactorings, but we did use the intuition behind these when constructing some transformations (e.g., see Listing 4.2). Moreover, these refactorings could be used to design new transformations, but not all refactorings would produce benchmarks that are not redundant, i.e., that will not get removed by the equivalence algorithm. For example, if we consider the renaming refactoring as transformation, the application of this transformation would result in removing the generated programs from the final test suite.

7.2 Generating benchmarks

For Java, the existing tools for generating valid programs require expert knowledge and significant human effort to be used in practice. On one

side, Daniel et al. [14] focus on an imperative approach: the programmer is responsible of *how* the test generation should proceed which has the advantage of a faster generation, but requires the expertise of the programmers to provide generators. Comparing to our technique, we do not require any expertise of the programmers related to the generation itself. Even when switching to another type of analysis, the developer will not have to change the generation logic. On the other side, Soares et al. [27] developed a Java program generator (JDOLLY) that uses Alloy and the Alloy Analyzer to create programs given the language elements (classes, fields, methods, etc.). This technique has the advantage of using a bounded-exhaustive generation method that generates programs that purely random generators might miss and, consequently, expressiveness is correlated with the constructors added to the model. In comparison, even though our technique might not produce benchmarks as expressive, it does not require such a significant expert knowledge.

Another popular line of work [16, 18, 24, 25] injects bugs into real-world programs by analyzing execution traces. Pewny and Holz developed *EvilCoder* [24], a system to automatically find vulnerable code locations and modify these locations to introduce vulnerabilities. In this work, the authors focused on the *insertion* of bugs rather than the *finding* or *fixing* of bugs, as they consider that the evaluation of this methodology of finding bugs can become very difficult without showing a realistic view of the work. Their closest related work is represented by *LAVA: Large-scale Automated Vulnerability Addition* [16], although *LAVA* specifically *adds* new vulnerabilities (buffer overread or buffer overflow), instead of transforming the code, as in *EvilCoder*. Thus, this research area shows us that inserting bugs by analyzing execution traces could produce expressive and challenging benchmarks, but, at the cost of requiring more effort on the programmer side to understand how such a bug could occur in the code.

Also, as Roy et al. [25] state, *EvilCoder* fails to inject bugs that are reproducible and *LAVA* fails to produce bugs that are deep and fair (i.e., it is possible to come across such a bug by using regular bug detection techniques). In contrast, we do not insert bugs in the code, as we initially design root programs that already throw an exception. The advantage is that, even after multiple applications of transformations, the same behaviour of the program is maintained, i.e., the program still ends with the same exception. Thus, the developer of an analyzer can determine the root cause of why the analyzer fails to report an issue by knowing the initial triggered exception and the transformations that were applied and in which order.

7.3 Testing static analyzers

Another line of work [10, 13, 28] shows the current improvements in testing static analyzers for soundness and precision. A work closely related to ours uses automatically generated benchmarks, from a given set of seed programs, to compare six analyzers, on arbitrary code [20]. Similarly, we tested Facebook's Infer static analyzer for soundness by using our set of automatically generated benchmarks, resulting in Infer missing issues, i.e., Infer fails to report issues for up to 33% of the benchmarks.

Chapter 8

Conclusions and future work

This chapter presents the final conclusions about the presented technique in this thesis (see Sec. 8.1), and ends by showing the possible lines of future work (see Sec. 8.2).

8.1 Conclusions

This thesis introduces a technique to *automatically* generate Java benchmarks, starting from a set of *root programs*, then following the application of multiple *transformations*. Our approach in generating benchmarks is making a step forward in solving current problems, such as: significant human effort to use existing tools in practice, and expert knowledge needed on the programmer side to understand the root cause of the bug in the generated benchmarks.

All the theoretical parts of the project from Chapter 4 have also an implementation correspondent in Chapter 5. Thus, we can instantiate the framework to generate a set of benchmarks for null pointer analysis.

In order to understand the possible impact of our generated test suite, we tested Facebook Infer for soundness. The results showed that Infer does not behave the same for different Java versions. Infer uses the given Java version to make its own compilation of the programs. Infer failed to report issues in 8% of the benchmarks for Java 1.8.0 and up to 33% of the benchmarks for Java 15.0.2.

Moreover, the work in this thesis can be replicated for other types of analysis (see next section for more details), and we are confident that this approach in generating benchmarks to test soundness can prove successful, beyond the scope of null pointer analysis.

8.2 Future work

Test the *equivalence* definition

We present our own *lightweight* definition of equivalence in Subsec. 4.5.2, by also highlighting the intuition behind the notion definition. For future work, we can choose a static analyzer for which we have the implementation and try to instrument the code to check our assumption: *two programs are equivalent if the analyzer follows the same path*. It should be noted that it is not relevant which static analyzer we chose for this experiment, one already implemented or one that is currently being written, because we did not define the notion making implementation assumptions.

Extend approach to consider completeness

To test the precision of the analyzers, we would need to check that the blocks of code inserted during transformations preserve the incorrect behaviour with respect to *all inputs*. For a given input, we can check this property by using concrete execution. However, to check for *all inputs* is completely impractical. Favourably, in the current implementation, the actual transformations do not use any input read from console or user defined, but a few transformations declare variables that gets assigned to random values and, based on this assignations, the behaviour of null pointer exception might not be triggered. Finally, we could extend our approach to consider completeness by either modifying or deleting these few transformations that we mentioned above.

Search-space pruning

In the current implementation, we first apply a transformation and then check if the resulted benchmark program is equivalent or not to other program already generated. As a future improvement, we could create a mechanism to ensure that each applied transformation will only create programs that are not equivalent to any previously-generated program, i.e., prune the search space before generation.

Challenging behaviour detection

There are some behaviours that are known to be challenging for analyzers to detect, such as: aliasing, write effects, etc [11]. The resulted test suite could include examples that reflect the previously mentioned behaviours by designing additional transformations.

Generalization

This technique could be easily extendable to other types of analyses, taking into consideration the generality of the generation algorithm. For future work, we could consider: buffer overrun or division by zero. The current Java solution can be reused, as the logic from the generation algorithm does

not have to change. We would only need to provide appropriate inputs for the root programs, i.e., which target the particular type of exception, and for the set of relevant statements. Moreover, considering that the set of relevant statements is different, the logic that matches statements with ids has to be updated (i.e., *matchRelevantStatement* from Algorithm 1).

Reduce IO workload

Sec. 5.4 mentions that, for a higher level of generation, there are IO intensive workloads. A future improvement would be to reduce this IO workload to spend less time to request the data, i.e., reading the benchmarks already generated.

Handle more special cases

In Sec. 4.7, we described some of the most common scenarios that we fixed in order to output compilable programs after applying the *insert redundant method from random class transformation*. However, this list with scenarios is not exhaustive. For example, parameter types, returns types or locally declared variables types in the method might be undefined in the current program. Another example is that we could add a method that calls another method that is undefined from our point of view. For missing types, these special cases could be solved by either finding their declarations in the initial code snippet that we selected, or, by our own definition of these types.

Appendix A

Other examples of transformations

Listing A.1: Resulted benchmark from applying an *add if-else transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5  }
6  public class RootProgram1 {
7      public static void main(String[] args) {
8          Box<Integer> box = null;
9          addBranch788(box);
10     }
11     public static void addBranch788(Box<Integer> box) {
12         int r1 = project.SingleRandom.getInstance().nextInt();
13         int r2 = project.SingleRandom.getInstance().nextInt(1);
14         if (r1 > r2) {
15             System.out.println(box.f);
16         } else {
17             Box<Integer> sdzsp = new Box<Integer>();
18             sdzsp.f = 1789481587;
19             sdzsp.g = new Object();
20         }
21     }
22 }
```

Listing A.2: Resulted benchmark from applying the *insert redundant class transformation* on the program from Listing 3.1.

```
1  class Box<T extends Object> {
2     public int f;
3     public Object g;
4     ...
5  }
6  public class RootProgram1 {
7      public static void main(String[] args) {
8          Box<Integer> box = null;
9          System.out.println(box.f);
10     }
11 }
12 class Test1 {
13   public static void main(String[] args) {
14       int i = 4;
15       int ia[][] = new int[i][i=3];
16       System.out.println(
17           "[" + ia.length + "," + ia[0].length + "]");
18   }
19 }
```

Bibliography

[1] About static analysis. https://courses.cs.washington.edu/courses/cse403/15sp/lectures/L19.pdf.

[2] Building checkers with the Infer.AI framework. https://fbinfer.com/docs/absint-framework/.

[3] Facebook infer. https://fbinfer.com/.

[4] Facebook Infer: Identify bugs before you ship. https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

[5] Language enhancements after Java 8. https://advancedweb.hu/new-language-features-since-java-8-to-17/.

[6] Program Equivalence through Trace Equivalence. http://www.lexicalscope.com/blog/wp-content/uploads/2014/10/fool2014-traceequivalence.pdf.

[7] Refactoring in Eclipse. https://www.baeldung.com/eclipse-refactoring.

[8] The Java Language Specification 15. https://docs.oracle.com/javase/specs/jls/se15/jls15.pdf.

[9] Bandyopadhyay, S., Sarkar, S., Sarkar, D., and Mandal, C. Samatulyata: An efficient path based equivalence checking tool. In *International Symposium on Automated Technology for Verification and Analysis* (2017), Springer, pp. 109–116.

[10] Bugariu, A., Wüstholz, V., Christakis, M., and Müller, P. Automatically testing implementations of numerical abstract domains.

In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), pp. 768–778.

[11] Christakis, M., Müller, P., and Wüstholz, V. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 8931* (Berlin, Heidelberg, 2015), VMCAI 2015, Springer-Verlag, p. 336–354.

[12] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), pp. 238–252.

[13] Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., and Yang, X. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium* (2012), Springer, pp. 120–125.

[14] Daniel, B., Dig, D., Garcia, K., and Marinov, D. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007), pp. 185–194.

[15] Das, A., and Lal, A. Precise null pointer analysis through global value numbering. *CoRR abs/1702.05807* (2017).

[16] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 110–121.

[17] Estep, S., Wise, J., Aldrich, J., Tanter, É., Bader, J., and Sunshine, J. Gradual program analysis for null pointers. *CoRR abs/2105.06081* (2021).

[18] Kashyap, V., Ruchti, J., Kot, L., Turetsky, E., Swords, R., Pan, S. A., Henry, J., Melski, D., and Schulte, E. Automated customized bug-benchmark generation. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2019), IEEE, pp. 103–114.

[19] Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X., et al. Astrée: Proving the absence of runtime errors. *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)* (2010), 9.

[20] Klinger, C., Christakis, M., and Wüstholz, V. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 239–250.

[21] Le, V., Afshari, M., and Su, Z. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, Association for Computing Machinery, p. 216–226.

[22] Lopes, N. P., and Monteiro, J. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer 18*, 4 (Aug. 2016), 359–374.

[23] Papi, M. M., Ali, M., Correa, T. L., Perkins, J. H., and Ernst, M. D. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, Association for Computing Machinery, p. 201–212.

[24] Pewny, J., and Holz, T. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), pp. 214–225.

[25] Roy, S., Pandey, A., Dolan-Gavitt, B., and Hu, Y. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 224–234.

[26] Simpson, A., and Voorneveld, N. F. W. Behavioural equivalence via modalities for algebraic effects. *CoRR abs/1904.08843* (2019).

[27] Soares, G., Gheyi, R., and Massoni, T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering 39*, 2 (2012), 147–162.

[28] Taneja, J., Liu, Z., and Regehr, J. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (2020), pp. 81–93.

[29] Yücesan, E., and Schruben, L. Structural and behavioral equivalence of simulation models. *ACM Trans. Model. Comput. Simul. 2*, 1 (1992), 82–103.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Automatically Generating Java Benchmarks with Known Errors

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hurmuz	Madalina

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 20.03.2022	Madalina Hurmuz

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.