Automatically Generating Java Benchmarks with Known Errors

Master's Thesis Project Description

Mădălina Hurmuz
Supervised by Prof. Dr. Peter Müller, Alexandra Bugariu

ETH Zürich

September 2021

1 Introduction

Over the past years, there has been an increased interest in the generation of benchmarks. 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 which are *simple* (i.e., easy to understand), but also *expressive* (i.e., include various language features and different types of issues).

1.1 Example

As shown in Listing 1, our focus will be to generate benchmarks in Java. This program was written manually by the developers of the static analyzer Facebook Infer [1] and is a good example to emphasize what is indeed a *simple* benchmark program. The example focuses on 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 such simple benchmarks can be used in order to test both new static analyzers and existing static analyzers for soundness, it is of high importance to automate the process of benchmarks generation.

Listing 1: Simple Java program that throws a NullPointerException.

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

1.2 Objective

The goal of this project is to generate ***increasingly complex benchmarks*** *that, by construction, expose a* ***particular class of errors****.* We will develop a technique that automatically generates test suites, which systematically cover a configurable set of language features *in Java.* As previously stated, our practical application for these benchmarks is testing static analyzers. Among the types of static analyzers, we consider those based on abstract interpretation [4], 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*).

It should also be noted that static analyzers are usually tested for reliability in terms of soundness and precision (i.e., should report a low number of false positives). For this project, we only focus on soundness.

1.3 Related work

1.3.1 Generating benchmarks

As our goal is to generate valid benchmarks programs, we first look into the simpler problem of generating valid programs. However, it seems that even this task is not trivial. Csmith [15], unlike previous similar tools, generates programs that cover various language features of C without introducing undefined and unspecified behaviours. For Java, the existing tools require expert knowledge and significant human effort to be used in practice. On one side, Daniel et al. [6] 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. On the other side, Soares et al. [13] 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.

Another popular line of work [7, 8, 10, 12] injects bugs into real-world programs by analyzing execution traces. Pewny and Holz developed *EvilCoder* [10], 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* [7], 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. [12] 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).

1.3.2 Testing static analyzers

Another line of work [3, 5, 9, 14] shows the current improvements in testing static analyzers for soundness and precision. In the work of Taneja et al. [14], it is shown how an SMT solver is used to compute sound and maximally precise static analysis results. Another work [9], more closely related to ours, uses automatically generated benchmarks to compare six analyzers, as there was no systematic way of comparing the effectiveness of different analyzers on arbitrary code up to that point.

To sum up, the existing methods for generating benchmarks either require significant expert knowledge, or they do not allow the developers to easily understand why their tool missed a particular bug, or they do not provide any guarantees with respect to the language features covered by the test suite. With regard to testing static analyzers, the existing works either are not fully automated or they do not provide the level of generalization, randomness and expressiveness that our approach can eventually show.

In the following section, we describe our approach in order to overcome the aforementioned challenges.

2 Approach

2.1 Inputs

Our generator will be parametric in the following three inputs:

1. *the classes of errors* the benchmarks should contain (e.g., NullPointerException, ArrayIndexOutOfBoundsException, Arithmetic-Exception, etc.). Note that the error should be thrown when we execute the program, as we do not consider assertion violations.

2. *the set of language features* supported by a static analyzer (e.g., declarations, assignments, loops, method calls, etc.)

3. *the statements relevant for the analysis* (i.e., those program statements which have an effect on the abstract state).

For instance, Table 1 shows the statements relevant for a null pointer analysis (we do not show their effect on the abstract state, since this information is not required for generating the benchmarks, i.e., we do not require to know the exact implementation of the analyzer, only its design).

Table 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* is a (possibly empty) list of arguments.

Id	**Program statement**
S1	x := new T()
S2	x := null
S3	x := y
S4	x := y.f
S5	x.f := y
S6	x.m(args)

2.2 Approach overview

Initially, we will generate programs like the one from Listing 1, that compile and, when executed, are expected to throw a given type of exception. We will then define a set of transformations which allow us to construct more complex type-correct programs that preserve the incorrect behaviour, but cover different language features supported by the analyzer.

Even with a limited set of transformations, one can potentially obtain infinitely many programs (see Listing 2), thus enumerating all the programs (even with a bounded number of instructions) is not feasible. To prune the search space efficiently and to avoid generating redundant programs, we will define a notion of *behavioural equivalence* (with respect to the statements relevant for a given analyzer). As a rough general idea, if two programs map to the same sequence of statements relevant for the analysis (i.e., the same changes of the abstract state), we will consider the two programs to be behaviourally equivalent. We will also use *object histories* to ensure that our transformations do not produce behavioural equivalent programs.

Listing 2: Simple example that illustrates how we can obtain infinitely many programs by using a constructor change transformation which replaces T with any built-in Java class.

```
T t = new T(); // where T is a built-in class in Java
```

For instance, the program from Listing 3 is behaviourally equivalent to the one from Listing 1, since they both include an assignment where the right hand side is null, followed by a method call on the previously-assigned variable. Moreover, both objects s and o, on which the methods *length()* and *finalize()* are called, are constructed in the same way. To be more precise, we associate the same sequence of *events* (history) to both objects s and o. By *events*, we refer to an invocation of a method involving the object. Thus, s and o are both assigned with null and, afterwards, a method is tried to be invoked on a null receiver. By using object histories, we record this information as a sequence of events, and we should also keep track of the object allocation.

Listing 3: Simple Java program that throws a NullPointerException and is behaviourally equivalent to the program from Listing 1, thus is redundant and should not be included in the test suite.

```java
class Hello {
   int test() {
      Object o = null;
      return o.finalize();
   }
}
```

However, we acknowledge that this solution may not be sufficient, as some transformations may generate programs with *infinite histories* and *histories of different lengths, but not different behaviour*. For the *infinite histories* issue, a solution would be to consider *abstract histories* [11], i.e., a bounded representation for each object history. The latter issue is not an actual problem if we think of two programs where the second program has the same prefix as the first one, but throws an additional exception, because we focus on concrete execution, and so, both programs will stop at the first exception. For the other scenarios that we might encounter, we will try to find a solution based on the way we design our transformations.

2.3 Possible transformations

Listing 4 and Listing 5 illustrate the result of two possible transformations.

Listing 4: The program from Listing 1, transformed by inserting a method call.

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


Hello h = new Hello();
String s = null;
h.test(s);
```

Listing 5: The program from Listing 1, transformed by inserting redundant code.

```java
class Hello {
   int test() {
      String s = null;
      String hello = "Hello"; // redundant code
      return s.length();
   }
}
```

Insert method calls. The first one replaces the call to a type-correct method *m* on a null receiver with passing null to a method *m'* which calls *m* on its parameter (in Listing 4, *length()* corresponds to *m* and *test()* to *m'*).

Insert redundant code. The second transformation requires inserting blocks of code that do not modify *s*. Designing this transformation poses two challenges:
1) *how to generate the code to be inserted?*, and
2) *how to know, by construction, that it still preserves the faulty behaviour (without performing static analysis)?*.

We will address challenge 1) by using code snippets of increasing size from the Java language specification (JLS) [2]. These are usually small and target a specific language feature, thus one can choose just those language features which are supported by the analyzer. The block of code from Listing 5 was taken from JLS and inserts an assignment with a constant right hand side. For challenge 2), we will require the inserted block to preserve the incorrect behaviour with respect to a given input (not with all possible inputs), property which can be checked by concrete execution. The resulting benchmark will be *good enough* for our test suite, although it has weaker guarantees, because we only test soundness, not precision of the analyzers. For example, in Listing 6, when the parameter *valid* maps to *true*, we reach the redundant code that does not modify *s* and that allows us to preserve the incorrect behaviour, i.e., the program will still throw a NullPointerException.

Listing 6: The program from Listing 1, transformed by inserting redundant code and preserving the incorrect behaviour with respect to *valid* being *true*.

```java
class Hello {
   int test(boolean valid) {
      String s = null;
      if (valid) {
         // redundant code & does not modify 's'
         String hello = "Hello";
      } else {
         s = "S";
      }
      return s.length();
   }
}
```

3 Core goals

3.1 Assumptions

We will first instantiate our approach for a particular type of analysis (i.e., null pointer analysis) and for a particular analyzer (i.e., Facebook Infer). We will assume for simplicity that the only statements that affect the abstract state are the ones from Table 1. We will also assume that the only supported language features are: declarations, assignments, if-else statements, and method calls.

3.2 Overview of the core goals

1. **Define root programs**: Write initial programs (like the one from Listing 1) whose execution will end in a null pointer exception. These should be simple programs on top of which we generate our test suite.

2. **Identify relevant statements**: Design and implement an algorithm for identifying the statements of a program that can modify its abstract state (see Table 1). We will use a pattern matching algorithm that has as input the program and returns these statements (i.e., we return the id from Table 1 corresponding to the statement).

3. **Define transformations**: Define transformations to systematically introduce into the root programs language features which are not yet covered (as stated later in goal 7, we do measurements in terms of language features to see what is covered), from the following set: declarations, assignments, if-else statements, and method calls. Express these transformations as templates. For instance, we could construct the following templates for assignments: x := new T(), x := null, x := x, x := y, x := constant, x := y.m(args), x := y.f, where x and y are objects, f is a field and m is a method. It should be noted that we ignore the types and choose a more lightweight approach where we use the compiler to check for the validity of the code.

4. **Template instantiation with JLS code**: Read from the Java Language Specification web pages pieces of code that increase in size. We plan to use the JLS code to instantiate as many templates as possible. Thus, the pieces of code will not be used only for the *insert redundant code transformation* (as shown in Section 2.2). In order to extract the actual example code, we can read the content in a HTML format and look for the *blockquote* tag. Inside this tag, there is quoted text from different sources and, in order to distinguish the code examples from other citations, we search in the beginning of the text for *import* statements, *package* declarations, *class* declarations or *interface* declarations.

5. **Behavioural equivalence**: Definite the notion of *behavioural equivalence*, with respect to the statements relevant for the analysis and to object histories (i.e., sequences of operations in which the object is created/used), for the statements which do not change the abstract state.

6. **Generate benchmark programs**: Systematically generate programs (with increasing number of instructions, up to a predefined bound) starting from the root programs, followed by the application of the transformations. Remove all the redundant programs (those that are behaviourally equivalent, as defined in the previous goal).

7. **Comparison with Facebook Infer**: With every iteration of our solution, we want to compare our generated test suite with the one from Facebook Infer. We measure how many of the statements from Table 1

and how many languages features (declarations, assignments, if-else statements, and method calls) we support compared with Infer. We will then ensure that our generated test suite covers all these statements and the restricted set of language features, and will measure the combinations of statements-language features covered. Additionally, we will discuss what completeness guarantees our approach can provide with respect to different classes of behaviour (determined based on the notion of behavioural equivalence from goal 5).

4 Extension goals

- **Search-space pruning**: Create a mechanism to ensure that each applied transformation will only create programs that should be behaviourally different (i.e., prune the search space before generation).
- **Challenging behaviour detection**: We would like to include in our test suite examples which have been shown to be challenging for the analyzers (e.g., which contain aliasing, write effects, etc.).
- **Generalization**: It would be relevant to look into the generalization of the approach to other types of analyses (e.g., buffer overrun, division by zero).

5 Schedule

To meet our core goals, we plan to roughly adhere to the following schedule:

Week	Goal
2021-09-20	Prepare initial presentation and define root programs
2021-09-27	Identify relevant statements
2021-10-04	Identify relevant statements
2021-10-11	Identify relevant statements
2021-10-18	Evaluation: collect existing tests from Infer and measure their coverage
2021-10-25	Evaluation: collect existing tests from Infer and measure their coverage
2021-11-01	Define transformations
2021-11-08	Define transformations
2021-11-15	Define transformations
2021-11-22	Template instantiation with JLS code
2021-11-29	Template instantiation with JLS code
2021-12-06	Template instantiation with JLS code
2021-12-13	Behavioural equivalence
2021-12-20	Behavioural equivalence
2021-12-27	Behavioural equivalence
2022-01-03	Generate benchmark programs
2022-01-10	Generate benchmark programs

2022-01-17	Evaluation: measure coverage
2022-01-24	Evaluation: measure coverage
2022-01-31	Evaluation: define completeness and measure it
2022-02-07	Write thesis
2022-02-14	Write thesis
2022-02-21	Write thesis
2022-02-28	Prepare final presentation

References

[1] Facebook Infer. `https://fbinfer.com/`.

[2] Java Language Specification. `https://docs.oracle.com/javase/specs/jls/se6/html/j3TOC.html`.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 419–428.

[12] 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.

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

[14] 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.

[15] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (2011), pp. 283–294.