Refactoring-based Benchmarks Generation

September 16, 2024

Problem description

Benchmarks programs are required in a variety of contexts: for testing or comparing the performance of compilers, static analyzers, or program verifiers, as data sets for machine learning algorithms, for evaluating program repair techniques or Large Language Models (LLMs), etc. However, systematically generating programs that are *simple* (i.e., easy to understand) but also *expressive* (i.e., include various language features) is very challenging. For example, Csmith [8], a widely-used generator for random C programs, relies on complex static analyses and run time checks to ensure that the programs do not have undefined and unspecified behavior. Enumeration-based approaches [9] produce all the C programs with k variables that match a small, syntactic skeleton. Techniques for synthesizing Java programs either start from user-provided generators [2] or require an Alloy [5] model of the constraints that should be satisfied by the programs [7]. Recent works on evaluating LLMs for code generation use manually-written benchmarks: HumanEval [1] (which includes 164 functions, together with docstrings and test cases) or ClassEval [3] (which consists of 100 class-level Python code generation tasks and required a 500 person-hours effort).

To overcome some of these limitations, Hurmuz [4] automatically constructs incrementally complex Java programs that throw null pointer exception. The approach uses simple, manually written seed programs (when executed, these always raise the exception) and then applies syntactic transformations that preserve the faulty behavior for at least one set of inputs. However, extending this technique to generate benchmarks with other properties requires expert knowledge in designing the transformations. We believe that this process can be automated by constructing the transformations using code refactoring.

The goal of this project is to develop a generic framework for synthesizing diverse but increasingly complex benchmarks programs that fulfill a given set of properties. The benchmarks will be constructed by applying (reverse) refactoring techniques. Our framework should also provide theoretical guarantees (similar to [9]) about the space of the generated benchmarks, with respect to a property-based notion of equivalence between benchmarks that we will define. (Hurmuz defines the equivalence with respect to a static analysis [4], while Zhang et. al

[9] include in the generated benchmarks suite only those programs which are not α-equivalent, where α is a variable renaming transformation).

Example

Let us assume that our goal is to automatically generate syntactically-correct Java benchmarks in which all the methods have maximum two instructions. That is, the desired set of properties is:

$P = \{language = Java, \neg syntax\_errors, \#instructions/method \leq 2\}$ (*).

To synthesize them, we can start with methods with no parameters and exactly one, type correct return instruction, as shown in Figure 1. We can then increase the number of instructions by replacing the returned constant (0 in Figure 1) with an expression that evaluates to the same value. A possible result of this transformation is shown in Figure 2 and has the inverse effect of the refactoring for inlining temporary variables.

Approach

We will first design a domain specific language to express properties about the benchmarks (e.g., as those in (*)). We will then automatically synthesize the simplest (shortest) programs that fulfills all the required properties. Note that if they exist, these programs do not have to be unique. We can use a compiler to check if they are syntactically correct.

Next, we will collect various code refactorings that, when applied directly or reversed on the simplest programs, preserve each of the properties. We will then apply these refactorings in a systematic way (potentially multiple times), to obtain increasingly complex but *non-redundant* benchmarks. The notation of redundancy is application-dependent. If the benchmarks are used, for example, for testing the Java compiler, then the program from Figure 3 is equivalent with the one from Figure 1, and thus it is redundant. However, if the goal of the benchmarks is to validate an algorithm for inferring method names, then both programs should be kept, as they represent a positive and a negative example.

As a possible extension, we could also automatically generate test cases (input-output pairs) for each benchmark program that accepts input parameters or reads its data, e.g., from a file or a database. Depending on the concrete type

```
class Benchmark {
 int foo() {
  return 0;
 }
}
```

Figure 1: Simple Java program with one instruction per method.

```
class Benchmark {
 int foo() {
  int tmp = 0;
  return tmp;
 }
}
```

Figure 2: The program from Figure 1, transformed by introducing a temporary variable.

```
class Benchmark {
 int zero() {
  return 0;
 }
}
```

Figure 3: Potentially redundant Java program with one instruction per method, obtained from the program from Figure 1 by renaming the method.

of tests we would like to consider, we can use various fuzzing tools, as well as state-of-the-art LLM-based approaches (such as [6]).

Prerequisites

The student is expected to have good programming skills. Prior experience with refactoring techniques is a plus.

Opportunities

The student will have the chance to learn about state-of-the-art test case generation techniques.

Contact

Alexandra Bugariu: bugariua@mpi-sws.org

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[2] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 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*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. Association for Computing Machinery.

[3] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Madalina Hurmuz. Automatically generating Java benchmarks with known errors. Master's thesis, ETH Zürich, 2022.

[5] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[6] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[7] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[8] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI

'11, pages 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 347–361, New York, NY, USA, 2017. Association for Computing Machinery.