Random Test Generation

Gage Tylee
Simon Fraser University

Abstract

This project outlines an approach used to implement random test generation by closely following the RANDOOP implementation by Pacheo et al. Emphasis is placed on topics such as testing, software design, and dynamic analysis. The source code can be found here.

1 Introduction

As one of many students who has worked on the repetitive task of writing numerous unit tests for a company's code-base, the idea of unit test generation sounded very intriguing.

Traditional white-box testing typically involves writing tests with specific input-output pairs in mind. The developer understands the internal structure of the system and has direct access to the source code. This can allow for a comprehensive and thorough test suite, but it does have its flaws. It can be rather time consuming, and may not cover all scenarios.

Some forms of test generation such as random test generation and fuzz testing, help patch these holes. They test the software from a more objective point of view which can lead to the discovery of new bugs or vulnerabilities. Generated tests also offer the advantage of scalability, and can generate large amounts of test cases very quickly.

Random test generation is an approach to testing that has been shown to outperform systematic test generation in terms of coverage and error detection [1]. A problem with uniform random testing, however, is the extraneous amount of illegal or redundant tests that can be generated. This can negatively impact performance, while also reducing the amount of code coverage. A solution to this problem is to use feedback heuristics that guide test generation based off of previously created tests.

2 Objective

The original goal of this project was to recreate a version of RANDOOP that included user defined resource constraints. After some further research, I realized there was still a lot for me to learn before I'd be able to achieve this. For this reason, the scope of my project is focused solely on the random test generation component. I closely studied the RANDOOP implementation and used it as means of guidance during development RANDOOP

3 Development Process

To develop RANDUPL, I started by outlining the process the program would follow.

1. Take a Java class as a runtime argument.
2. Parse the input class, storing every method and constructor.
3. Generate the input data for the methods to test.
4. Generate the code for the unit tests.

In order to do this a few key components were needed.

3.1 Operation

The Operation interface is a way to represent method calls, constructors, a field accesses. This hierarchy makes it easy to store the relevant details for the operations in a class and later convert them back to code for our unit test generation. An OperationVisitor is used to add extra functionality to each operation type (more on that soon).

3.2 Reflection

I initially thought that one of the most challenging parts of this project would be the dynamic analysis required to parse a class and get the function execution outputs at runtime. Luckily Java's Reflection library simplified this process. The Reflection package in this project runs through the input class and converts every method/constructor into an Operation. Each operation is kept in an OperationStore which will be used as in put for the Generator.

3.3 Generator

The generator is where the inputs are generated for each method. Currently, it only supports random uniform generation, but this is an area that I aim to improve later using the feedback heuristics. Generating random objects for reference types includes the use of the object pool design pattern.

3.4 Output

The TestCaseManager is responsible for creating new TestCase instances from the OperationStore. In doing so it uses the CodeBuilder class which makes it easy to generate multiple lines of code.

An opportunity to make use of the visitor design pattern quickly presented itself in this section. Since the strategies for code generation will change in the future, using the visitor pattern would help separate the logic behind code generation from its respective operation model. The benefit of using this is that it makes the code more extensible while adhering to the single responsibility principle. Rather than individually adding additional functions to each type of operation for future code generation strategies, we'd just be able to create a new visitor.

4 Results

RANDUPL provides a strong foundational infrastructure for Random Test Generation and leaves it extensible to future improvements. It successfully outputs randomly generated test cases for basic input classes. There are quantitative metrics I can add in the future that would help in optimizing the runtime of the program and quality of tests.

Below are some examples of the tests generated for a simple Calculator class that contains add, sub, mult, and div methods, using the equals contract.

```
TEST_add() {
   Calculator obj = new Calculator();
   int res = c.add(-2146421632, -1847989750);
   assertNotNull(res);
}
TEST_sub() {
   Calculator obj = new Calculator();
   int res = c.sub(-1516470074, -1697724814);
   assertNotNull(res);
}
TEST_mult() {
    Calculator obj = new Calculator();
    int res = c.add(750529863, 1302163724);
    assertNotNull(res);
}
TEST_div() {
    Calculator c = new Calculator();
    int res = c.add(979623757, -1989790190);
    assertNotNull(res);
}
```

5 Future plans

While the current state of the project helped in understanding the building blocks of a unit test generator there are many improvements to be made in both functionality and design.

5.1 Class Support

As of now, RANDUPL is only compatible with basic classes that use methods with primitive types. In most scenarios this wouldn't be very useful as classes are typically much more complex. I'd like to add full support to any type of class and or field. This includes reference classes, generic classes, and static variables.

5.2 Contracts

One of the most useful components of a test generator is the use of contracts. In RANDUPL the test assertions are currently hard-coded using CodeBuilder. If instead the different types of assertions (contracts) are made into their own classes, it makes it possible for the generator to choose which contract it wants to use.

5.3 Input Generation

Like I previously mentioned, RANDUPL only supports primitive types. A big reason for this is because it can be harder to implement random generation for unknown reference types. Some reference types may depend on other reference types as well. This case is normally handled by recursively traversing the constructors of each class until it reaches a constructor that takes only primitive types. Instantiated objects of these classes are then be stored and selected randomly by the generator to be used as a random input. The object pool design pattern can help with this task as it can reduce memory consumption and improve execution.

6 Conclusion

Creating RANDUPL was a fun project and in the process I learned a lot about the different approaches to testing. Having previously not known anything about unit test generation, there was a steep learning curve at first, but I now feel that I have a deep enough understanding to plan out the development of future features to this project. Applying some of design patterns learned in class helped me realize their benefits, and made what once looked like highly advanced code, much more digestible to me. I'd appreciate any feedback as I look forward to designing the rest of the system.

References

[1] Michael D. Ernst Carlos Pacheco, Shuvendu K. Lahiri and Thomas Ball. Feedback-directed random test generation. page 1, 2007.