Randoop Tutorial

December 11, 2016

1 Introduction

Randoop is a test generator for Java. Given a set of Java classes, Randoop outputs two test suites:

- Error-revealing tests that indicate errors in the Java code.
- Regression tests that capture current behavior, and that can identify when future code changes affect behavior.

Randoop creates large numbers of tests that may be too complicated or too mundane for the average programmer to write. Randoop runs automatically. The user can tune parameters and provide optional inputs to improve Randoop's performance.

This tutorial is an introduction to using Randoop to generate tests, specifically in the context of the Pascali project.

2 Getting Started

This tutorial assumes that the integration-test2 `fetch.py` and `run_dyntrace.py` scripts have been run:

```
git clone https://github.com/aas-integration/integration-test2.git
cd integration-test2
python fetch.py
python run_dyntrace.py catalano
```

The `fetch.py` script ensures that the Randoop jar file is available, and the `run_dyntrace.py` script ensures that input files for Randoop (such as the list of classes in the Pascali corpus) are there. It produces copious output, including Java stack traces — you can ignore these.

From the `integration_test2` directory, run the following commands to make links to files needed for the tutorial:

```
PASCALIROOT=`pwd`
git clone https://github.com/randoop/tutorial-examples.git
```

```
cd tutorial-examples
./gradlew -PpascaliRoot=$PASCALIROOT tutorialInit
```

3 Learning about Randoop

We will apply Randoop to a toy class, `MyInteger`.

3.1 Discovering a bug

First, run the command

```
./gradlew first build
```

Before moving on, peek in the file `src/main/java/math/MyInteger.java`. This is a simple class that creates integer-like objects that can be added and multiplied. Programmer-crafted tests are in `src/test/java/math/MyIntegerTest.java`. With a quick glance these look reasonable, but, as we'll see, this version of `MyInteger` has a bug.

When we ran the `build` Gradle task these files were compiled, and the tests were run. You can verify that the tests pass:

```
./gradlew cleanTest test
```

Now let's use Randoop to generate some tests. Copy and paste the following command into your terminal to run Randoop:

```
java -ea -cp build/classes/main:randoop.jar randoop.main.Main gentests \
  --testclass=math.MyInteger --junit-output-dir=src/test/java --outputlimit=20
```

This generates a small number of tests that call the constructors and methods of the `MyInteger` class; it writes the tests to the subdirectory `src/test/java`. Randoop has actually generated two test suites: one for error-revealing tests, and one for regression tests. Each suite consists of a JUnit4 suite file (e.g., `ErrorTest.java`) and files containing the tests themselves (e.g., `ErrorTest0.java`).

The fact that Randoop generated the error-revealing tests means that it discovered a faulty behavior. The file `src/test/java/ErrorTest0.java` contains five methods, each of which violates the same contract: if two objects are equal, then they have the same `hashCode` For example, the second test method, slightly edited, is

```
public void test2() throws Throwable {
    math.MyInteger myInteger1 = new math.MyInteger((-1));
    math.MyInteger myInteger3 = new math.MyInteger((-1));
    java.lang.String str4 = myInteger3.toString();
    math.MyInteger myInteger5 = myInteger1.multiply(myInteger3);
    int i6 = myInteger1.getIntValue();

    // Checks the contract:  equals-hashcode on myInteger3 and myInteger5
    org.junit.Assert.assertTrue(
```

```
      "Contract failed: equals-hashcode on myInteger3 and myInteger5",
      myInteger3.equals(myInteger5)
        ? myInteger3.hashCode() == myInteger5.hashCode() : true);
}
```

This is the kind of test that is likely to be overlooked by a programmer, but is critical to the proper behavior of the class in the tests that the programmer wrote.

To run the Randoop-generated tests, do:

```
./gradlew test
```

You will see that the `ErrorTests` test suite fails, and the `RegressionTests` test suite passes.

The regression tests capture sequences of code that do not violate a contract, along with assertions about the values that are generated. We'll look more closely at regression tests after we have fixed the defect.

3.2 Fixing the bug

If we look at the class (`src/main/java/math/MyInteger.java`) more closely than before, we can see that the equals method is incorrectly defined

```
public boolean equals(Object other) {
    if (other instanceof MyInteger) {
      return true;
    }
    return false;
}
```

and there is no `hashCode()` method.

Let's assume that Randoop's tests revealed the problem to a developer, and the developer has made a fix. To obtain the fixed version of the class, run

```
./gradlew second
```

Then, compile the class and run the tests:

```
./gradlew build
```

Notice that all of the tests now pass, even the error-revealing tests that had failed before. This verifies the fix

3.3 Discovering a regression error

As we just saw, the tests generated for a previous version of the code are useful for checking the behavior of newer versions of the code. Let's create new regression tests for the `MyInteger` class. First, run

```
./gradlew cleanRandoopRegressionTests
```

to remove the existing set of tests, and then run Randoop again

```
java -ea -cp build/classes/main:randoop.jar randoop.main.Main gentests \
  --testclass=math.MyInteger --junit-output-dir=src/test/java --outputlimit=200
```

Notice that Randoop only generated new regression tests. That is because Randoop did not discover any problems in the new version of the code. Note that the command uses a larger value for `--outputlimit`, which is an upper bound on the number of tests that will generated.

The value of regression tests is that they reveal changes in behavior. Suppose that a programmer makes some modifications to improve performance or add new features, but the programmer intends that existing functionality should not be affected. To validate these changes, you can run the regression tests.

Obtain a changed version of the code with the command

```
./gradlew third
```

and run the tests:

```
./gradlew test
```

The test runner output shows that there is an `AssertionError` at line 54 of `RegressionTest0.java` (among other places). That is the last assertion in the second test:

```
public void test02() throws Throwable {

    if (debug) { System.out.format("%n%s%n","RegressionTest0.test02"); }

    math.MyInteger myInteger1 = new math.MyInteger((-1));
    math.MyInteger myInteger3 = new math.MyInteger((-1));
    java.lang.String str4 = myInteger3.toString();
    math.MyInteger myInteger5 = myInteger1.multiply(myInteger3);
    boolean b7 = myInteger5.equals((java.lang.Object)(short)100);
    int i8 = myInteger5.getIntValue();

    // Regression assertion (captures the current behavior of the code)
    org.junit.Assert.assertTrue("'" + str4 + "' != '" + "-1"+ "'", str4.equals("-1"));

    // Regression assertion (captures the current behavior of the code)
    org.junit.Assert.assertNotNull(myInteger5);

    // Regression assertion (captures the current behavior of the code)
    org.junit.Assert.assertTrue(b7 == false);

    // Regression assertion (captures the current behavior of the code)
    org.junit.Assert.assertTrue(i8 == 1);

}
```

This assertion says that $-1 \times -1 = 1$. Since the assertion fails, either the test is wrong, or there is something wrong with one of the methods that the test calls. If we look in the new `MyInteger`, the problem is that the developer performed an incorrect optimization. The optimization is to avoid multiplication of negative numbers, but the logic is wrong for correcting the sign:

```
public MyInteger multiply(MyInteger other) {
  // Always multiply positive numbers, negate later.
  boolean negative = false;
  negative = negative || this.value < 0;
  int absThis = Math.abs(this.value);
  negative = negative || other.value < 0;
  int absOther = Math.abs(other.value);
  int absProduct = absThis * absOther;
  if (negative) {
    return new MyInteger(-1 * absProduct);
  } else {
    return new MyInteger(absProduct);
  }
}
```

4 A Larger Example

What we've seen so far is that Randoop can generate tests that find important bugs, and also can help find regressions that arise between versions of code. But, Randoop can do more than build tests for a single simple class, it can build complex tests for non-toy code bases such as those included in the Pascali corpus.

As an example, consider part of the Catalano Framework (`https://github.com/DiegoCatalano/Catalano-Framework`) from the Pascali corpus. The following Gradle task runs Randoop on the files in `catalanoimage/classlist.txt`, using the classpath in `catalanoimage/classpath.txt` and redirecting standard error to a log file.

```
./gradlew runCatalanoExample 2> catalano-error-log.txt
```

Randoop runs for about 2 minutes. It generates 9 error-revealing tests and about 1062 regression tests, and writes them to `src/test/java/catalano/`.