Report on Code Based Test Generation

Our team chose **Randoop** for test generation, as it turned out to be one of the most thorough tool, covering probably the most nontrivial cases. The SUT is SafetyLogicImpl class, covering all the decision making logic and connectivity with the SignalMapper. We planned to focus on testing the methods that closely belong to this logic, so we focused on the following public methods:

- `sectionOccupancyChanged()`,
- `turnoutDirectionChanged()`,
- `neighborStatusChanged()`.

Furthermore, Randoop only supports the testing of public methods, which also limited the available opportunities.

Two main test generation scenarios were created. The first is pretty simple, we just used the tool, and our code as they are, focusing on the SUT, and thus neglecting all methods and other classes that were irrelevant for isolated testing. Then, we turned into action what we had learnt from the outcomes, and we created some helper classes in order to "help" the test generator create more appropriate testing sequences.

We also tried using *mapcall-3.0.6.jar* for mocking the SignalMapper's implementation, but for unknown reasons the tool could not replace the substitute method implementation with the original. (We also designed some "dummy" method replacements to test that the mapper actually works, and had no problems with those – so it is quite unclear why Randoop declines to replace certain methods.)

Simple test generation

At first, we simply provided the following for the test generation:

- the classes that should be used for testing (randoop_classes.txt),
- the target class that should appear in all tests (hu.bme.mit.swsv.ris.tsm.impl.SafetyLogicImpl),
- a regexp covering some methods that do not contribute to our testing targets "(disConnect|startHeartBeat|receive*)",
- adding slfj4.jar to the classpath so that loggers can be created (avoiding NoClassDefFoundError),
- sensible time limit for generation (30 seconds).

The results were tests running into NullPointerExceptions, as Randoop initialized most of the fields with null values. It also turned out that Randoop does not recognize a proper way for initializing sidetriples, as no public constructor is available. The generated assertions are mostly about expecting NullPointerExceptions, and asserting trivial field values.

<u>The used command:</u>

```
Javatar-ris-2016\src\swsv-ris> java -classpath "Randoop\randoop-all-
3.0.6.jar;target\classes;Randoop\slf4j-api-1.7.20.jar" randoop.main.Main gentests --
classlist=Randoop\randoop_classes_simple.txt --timelimit=30 --
omitmethods="(disConnect|startHeartBeat|receive*)" --log log.txt --include-if-classname-
appears=hu.bme.mit.swsv.ris.tsm.impl.SafetyLogicImpl
```

We also tried using `--forbid_nulls=true`, so that Randoop assigns non-null values whenever it is possible, but it did not solve the SideTriple initialization issue, no tests could be generated.

A bit more complex test generation

We had the idea of creating another SignalMapper implementation class, which simply stores the decision SideTriple objects representing the decision generated by SafetlyLogicImpl. This class is named `hu.bme.mit.swsv.ris.randoop.`**`RandoopSignalMapper`**. It only implements the `sendStatus()` and `sendControl()` methods, as these will be called by the decision making logic, and the decision itself can easily be stored.

We also wanted to enforce Randoop to use this class when creating a new SignalMapperImpl object. Therefore, we created a static method that resembles the @Before method that we used in out Junit tests. By adding **`RandoopTestInitializer`** class to the list of classes used by Randoop, it will be able to initialize a `SafetyLogicImpl` object properly (without nulls). This new static method is `hu.bme.mit.swsv.ris.randoop.RandoopTestInitializer.`**`createSafetyLogicImpl()`**.

The test generating command is now the following:

```
java -classpath "Randoop\randoop-all-3.0.6.jar;target\classes;Randoop\slf4j-api-1.7.20.jar"
randoop.main.Main gentests --classlist=Randoop\randoop_classes.txt --timelimit=30 --
omitmethods="(disConnect|startHeartBeat|receive*)" --log log.txt --forbid_null=true --include-if-
classname-appears=hu.bme.mit.swsv.ris.tsm.impl.SafetyLogicImpl
```

Evaluation of the generated tests

`RegressionTest0.java` contains the tests that were generated using the previous command. This does not contain null value assignments, due to the `--forbid_null=true` clause. (The simple tests already showed that there is a possibility of initializing the isolated SafetyLogicImpl object with null fields, resulting in NullPointerExceptions.)

Many of the generated tests now contain the execution of the decision making logic, which was our goal when making these tests. These contain such method call sequences that the decision could be asserted in the end. These should happen in order (see e.g. `test01()` in `RegressionTest0.java`):

1. Create a RandoopSignalMapper
2. By using this, create a SafetyLogicImpl object:
   `RandoopTestInitializer.createSafetyLogicImpl(RandoopSignalMapper)`
3. Call `SafetyLogicImpl.***Changed()` methods
4. Read and Assert the value of the RadoopSignalMapper's fields.

The last item unfortunately never happens. Randoop reads the fields of this object only after creating them.

All in all, these generated tests by themselves cannot ensure that the decision logic works properly. The tests mostly examine the field's values and whether they are null or not, which are quite trivial and redundant to check. As a conclusion, we would not include any of the generated tests in our current test set, however, it showed that there are many cases (probably with simpler logic or less dependencies) where this tool can be quite useful.