CSE364: Software Engineering
Individual Assignment

Student Name: Nguyen Minh Duc
Student ID: 20202026

1 Detect and localize a bug (50 points)

1.1 Run Randoop and Evosuite

Both Randoop and Evosuite were executed in a similar manner to the tutorial in the assignment handout with a small modification of Evosuite options where **-seed=1684387559162** was added to make the generation deterministic. In summary, Randoop generated 1608 test cases, and all of them resulted in failures, while Evosuite generated 402 test cases and only 7 of them resulted in failure.

1.2 Failing test case that manifests the bug

The following is the failing test case that detected a bug in our library under test, taken from the Evosuite test suite.

```
1382 @Test(timeout = 4000)
1383 public void test256()  throws Throwable  {
1384     String[] stringArray0 = new String[3];
1385     stringArray0[0] = "IllegalArgumentException occurred during 1.6 backcompat code";
1386     stringArray0[2] = "IllegalArgumentException occurred during 1.6 backcompat code";
1387     // Undeclared exception!
1388     StringUtils.replaceEach("IllegalArgumentException occurred during 1.6 backcompat code
     ", stringArray0, stringArray0);
1389 }
```

This test case produces a `NullPointerException`, which is a common issue occurring when the developer forgot to create a null pointer check somewhere in the code. The following is the stack trace of the aforementioned exception.

```
1) test256(org.apache.commons.lang3.StringUtils_ESTest)
java.lang.NullPointerException
  at org.apache.commons.lang3.StringUtils.replaceEach(StringUtils.java:3676)
  at org.apache.commons.lang3.StringUtils.replaceEach(StringUtils.java:3502)
  at org.apache.commons.lang3.StringUtils_ESTest.test256(StringUtils_ESTest.java:1388)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java
    :43)
  at java.lang.reflect.Method.invoke(Method.java:498)
  at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
  at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
  at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
  at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
  at org.junit.internal.runners.statements.FailOnTimeout$StatementThread.run(
    FailOnTimeout.java:74)
```

1.3 Bug location

As one could see from the first line of the stack trace,

```
  at org.apache.commons.lang3.StringUtils.replaceEach(StringUtils.java:3676)
```

The line 3676 in the method `replaceEach` of the `StringUtils` class directly caused this bug. Below is a code snippet of the bug location in the source code.

```
3605 private static String replaceEach(String text, String[] searchList, String[]
         replacementList, boolean repeat, int timeToLive)
3606     {
                  ⋮
3607
3671         // get a good guess on the size of the result buffer so it doesnt have to double
         if it goes over a bit
3672         int increase = 0;
3673
3674         // count the replacement text elements that are larger than their corresponding
         text being replaced
3675         for (int i = 0; i < searchList.length; i++) {
3676             int greater = replacementList[i].length() - searchList[i].length();
3677             if (greater > 0) {
3678                 increase += 3 * greater; // assume 3 matches
3679             }
3680         }
3681         // have upper-bound at 20% increase, then let Java take over
3682         increase = Math.min(increase, text.length() / 5);
                   ⋮
3683
3730         return replaceEach(result, searchList, replacementList, repeat, timeToLive - 1);
3731     }
```

1.4 How this bug occurred and its root cause

Let's analyze how this bug occurred given our test case. Firstly, let's look at the input. Note that in both arrays, the first element is not initialized, hence, a potential buggy input. Secondly, the test case invoked a `replaceEach`, which in turn invoked another `replaceEach` where the main logic and the bug lies, i.e.,

```
3501 public static String replaceEach(String text, String[] searchList, String[]
         replacementList) {
3502     return replaceEach(text, searchList, replacementList, false, 0);
3503 }
```

The input took the program to the loop at the line 3675, where the method iterated through both arrays at the same time. On the buggy 3676-th line, it computed the difference between the length of two strings in the two arrays at the same index. For $i = 0$, both strings were "IllegalArgumentException occurred during 1.6 backcompat code", hence, greater $= 0$, which was still normal and nothing had happened. However, when $i = 1$, both strings were `null`, which was illegal to read the `length` attribute of a `null` object, hence, the `NullPointerException` was raised. One might suspect that this input was invalid, but in the documentation of the API, it explicitly said that if a null pointer is given, a no-op

The root cause of this bug is due to a lack of `null`-safety checks in the method. The exception would have been avoided if there had existed a simple null check before computing the length difference of the two strings, i.e.,

```
3675 for (int i = 0; i < searchList.length; i++) {
3676     // Added null check
3677     if (replacementList[i] == null || searchList[i] == null) {
3678         continue;
3679     }
3680     int greater = replacementList[i].length() - searchList[i].length();
3681     if (greater > 0) {
3682         increase += 3 * greater; // assume 3 matches
3683     }
3684 }
```

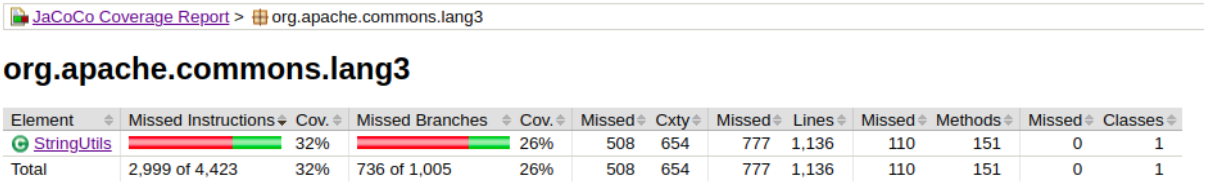
After having added this if statement, the method passed our test case, and the automated test generation tools no longer detected this error.

1.5 The tool that reveals the bug

In my case, only Evosuite can detect this error, while Randoop could not detect any `NullPointerException` but some expected exceptions, i.e., `IllegalArgumentException`, `IllegalStateException`, and some `ArrayIndexOutOfBoundsException`. Even after re-running 10 times, it failed to obtain the same bug as Evosuite.

2 Coverage of generated tests (20 points)

By running `jacoco` in the same manner as the tutorial, I obtained coverage reports of Randoop and Evosuite tests. Please refer to Figure 1 and 2 for coverage reports of Randoop and Evosuite respectively.

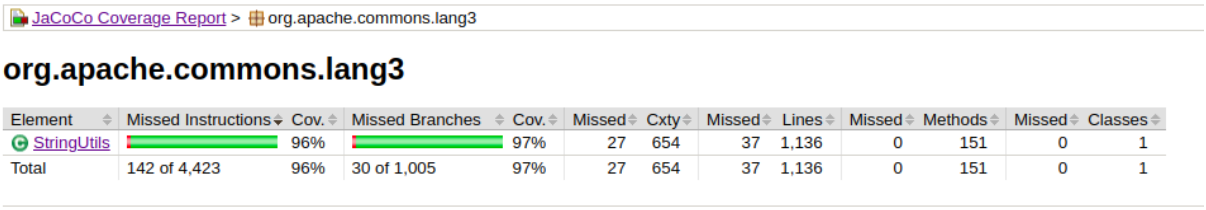


JaCoCo Coverage Report > org.apache.commons.lang3

org.apache.commons.lang3

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
StringUtils		32%		26%	508	654	777	1,136	110	151	0	1
Total	2,999 of 4,423	32%	736 of 1,005	26%	508	654	777	1,136	110	151	0	1

Figure 1: Randoop coverage report



JaCoCo Coverage Report > org.apache.commons.lang3

org.apache.commons.lang3

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
StringUtils		96%		97%	27	654	37	1,136	0	151	0	1
Total	142 of 4,423	96%	30 of 1,005	97%	27	654	37	1,136	0	151	0	1

Figure 2: Evosuite coverage report

3 Strengths and weaknesses (30 points)

3.1 Randoop

By referring to the previous sections, one could obtain the following strengths from Randoop:

- Randoop is simple and easy to use.
- Randoop's speed is fast. It took 67 seconds to generate test cases with given command line options in the assignment handout.
- Randoop only keeps failing test cases, which could be helpful for detecting bugs.

Despite having good strengths, Randoop suffers from the following weaknesses:

- Test data are too complicated due to Randoop's randomness. This could be solved by reducing the `--string-maxlen` option when executing.
- Randoop has a poor branch coverage of 26%.
- Randoop only keeps failing test cases, which reduces the branch coverage and increase the probability of false alarm.

3.2 Evosuite

By referring to the previous sections, one could obtain the following strengths from Evosuite:

- Evosuite is able to achieve a very high branch coverage of 97% as it is guided by coverage.

- It has shorter test cases compared to Randoop.
- Evosuite utilizes mocking frameworks so that it can handle different complex scenarios.

Despite having good strengths, Evosuite suffers from the following weaknesses:

- Slower than Randoop. With the given command line options in the assignment handout, it took Evosuite 91 seconds to terminate, an 1.4 times increase from that of Randoop.
- String values generated from Evosuite seem to have less diversity compared to Randoop as it lacks random generation and seems to only contain 1) Exception messages 2) Java class names 3) Some random strings.

4 Improve Test Generation (20 points)

4.1 Randoop

The main problem that affects the performance of Randoop is generating too many false alarm test cases. Randoop spends most of the time generating random inputs that have high probabilities of being invalid, which is not useful when it comes to finding bugs. To fix this issue, one could add some basic constraints before the generation of inputs. For example, when there are some arrays, try to produce some test cases in which other integers lie on the range of the arrays' length, or allow some null values inside an array, etc.

Another improvement could be randomly saving some valid inputs that do not raise errors, which is beneficial for improving the branch coverage of the algorithm.

4.2 Evosuite

Although Evosuite outperforms Randoop on most of the criteria, there is something that could be improved. Firstly, the diversity of string values in Evosuite's test cases can be improved by introducing more mutations, and with a higher probability of generating random strings, not just taken from the method under test's exception messages or some pre-defined values. This could help the algorithm to investigate more behaviors from the API, which can potentially reveal more buggy codes.

Additionally, since most of the failures come from invalid inputs, we can allow the user to manually select which exception is expected, and the algorithm will ignore those input that causes a false alarm. However, this will likely reduce branch coverage. To solve this, the tool could first scan the method under test, and count all of the expected throws in the control flow of the method. The probability of generating an invalid input will gradually decrease as more and more unique exceptions have been discovered.

4.3 Apply Natural Language Processing techniques

Since most of the API has extensive documentation, which contains useful information such as the expected range of the input, the expected exception that will be thrown, etc. One can leverage the different NLP techniques, especially Large Language Models since they have impressive knowledge, to extract that information to guide the automated testing tools away from generating too many invalid inputs.