Symbolic Execution and Fuzzing on Guava

Zekai Zhao, Junxiong Lin

1. Background

Guava is a well-known google java library that contains lots of algorithm implementation and data structure. In this report, we only focus on the testing on the data structure -- minMaxPriorityQueue.

Before we doing the practice, we already have some background about this test from my other testing class.

1. The minMaxPriorityQueue class file has been modified by others in ten places so that the source file is buggy.
2. We have written about 100 unit test cases that cover all the behavior of the priority queue, including add, offer, remove, poll, and etc. Those test cases reached a certain amount of test coverage and discovered four out of ten mutation bugs.

2. Plan

We plan to apply the symbolic execution and fuzzing on testing, applying what we learned from class to real-world problems. Symbolic execution is a method of white box testing which increases the coverage of the code, and unit test with fuzzing is a white box testing. Our goal is to find more mutation bugs. However, if the goal is not reachable, we want to at least improve the test coverage of this problem.

3.Symbolic execution

First, we can use symbolic execution to do some white-box testing. Symbolic execution is a method that can analyze a program to determine what inputs cause each part of a program to execute. It can be used as a tool to generate test cases and detect errors. We used Symbolic PathFinder to do the symbolic execution.

Using JAVA 8 and download jpf-core and jpf-symbc

.jpf/site.properties

jpf-core = ${user.home}/CMU/18737/jpf-core

Jpf-symbc = ${user.home}/CMU/18737/jpf-symbc

extensions=${jpf-core},${jpf-symbc}

Run a small test.

```
====================================================== search started: 19-12-8 下午10:57
New sym int x_1_SYMINT min=-2147483648, max=2147483647
New sym int y_2_SYMINT min=-2147483648, max=2147483647
numeric PC: constraint # = 1
(y_2_SYMINT + x_1_SYMINT) != CONST_0 -> false

### PCs: total:1 sat:0 unsat:1

numeric PC: constraint # = 1
(y_2_SYMINT + x_1_SYMINT) = CONST_0 -> true

### PCs: total:2 sat:1 unsat:1

string analysis: SPC # = 0
NPC constraint # = 1
(y_2_SYMINT + x_1_SYMINT) = CONST_0
```

Since Guava has a lot of dependencies and I can only run SPF in the console, but fail to execute it in Eclipse, I modeled a simple version of MinMaxPriorityQueue. It has functions of Push(), Pop(), Contains() and Peek(). We can configure the jpf file by defining the min and max of symbolic and using symbolic.lazy=on to prune some of the choices. First, we tested a sequence of push.

```
[push(-100), push(-99), push(-98), push(-97), push(-9223372036854775808)]
[push(-100), push(-99), push(-98), push(-98), push(-9223372036854775808)]
[push(-100), push(-99), push(-99), push(-100), push(-100)]
[push(-100), push(-99), push(-99), push(-100), push(-99)]
[push(-100), push(-99), push(-99), push(-98), push(-9223372036854775808)]
[push(-100), push(-98), push(-99), push(-97)]
[push(-100), push(-98), push(-99), push(-98), push(-9223372036854775808)]
[push(-100), push(-99), push(-100), push(-99), push(-99)]
[push(-100), push(-99), push(-100), push(-99), push(-100)]
[push(-100), push(-99), push(-100), push(-98), push(-100)]
[push(-100), push(-99), push(-100), push(-98), push(-99)]
[push(-99), push(-98), push(-100), push(-100), push(-99)]
[push(-100), push(-100), push(-100), push(-99), push(-99)]
[push(-100), push(-100), push(-100), push(-99), push(-100)]
[push(-99), push(-100), push(-100), push(-99), push(-100)]
[push(-99), push(-100), push(-100), push(-99), push(-99)]
[push(-99), push(-99), push(-100), push(-100), push(-99)]
[push(-100), push(-100), push(-99), push(-99), push(-9223372036854775808)]
[push(-100), push(-100), push(-99), push(-98), push(-9223372036854775808)]
```

It returns no error.

```
        @Test
        public void test143() {
                priorityqueue.push(-99);
                priorityqueue.push(-100);
                priorityqueue.push(-99);
                priorityqueue.push(-98);
                priorityqueue.push(-97);
        }

        @Test
        public void test144() {
                priorityqueue.push(-97);
                priorityqueue.push(-100);
                priorityqueue.push(-99);
                priorityqueue.push(-98);
                priorityqueue.push(-97);
        }
}

======================================================= results
no errors detected

======================================================= statistics
elapsed time:       00:00:01
states:             new=3411,visited=0,backtracked=3411,end=1942
search:             maxDepth=19,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=1469
heap:               new=1732,released=24048,maxLive=370,gcCycles=2939
instructions:       128000
max memory:         309MB
loaded code:        classes=65,methods=1434

======================================================= search finished: 19-12-9 下午1:34
```

And we tested random sequences of other methods, it shows no error too. Because it is a simple model, most subtle bugs can not be found.

For the pros and cons of symbolic execution. It can explore different kinds of workflow and analysis without real execution. It can also pre-process to eliminate unsatisfiable test cases. For the disadvantages, in heap, the symbol of the exact number isn't really related to the process and lack of documentation and it needs pruning and fully implementation

4. Fuzzing

we are trying to use fuzzer to help improve our previous unit tests. The fuzzer I chose is Radamsa. It is an open-source input generator that mutates given input by applying pre-defined mutation rules and patterns.

```
$ echo 192.168.106.103 | radamsa --count 10 --seed 0
-107.167.106.103
192.168.8407971865571866.-9?5154737306362663942413194069
191.1A1.1A1.106.1
192.129.18.106.103
192.168.0.103
192.170141183460.106.1802311213346089.104
-3402823669209.106.168.106.16.103
192093846346337460765704.192.65704.-1.?-18446744073709518847
192.106.0
191.168.106.103
$ echo 192.168.106.103 | radamsa --count 1 --seed 0 | xargs ping
ping: invalid option -- 1
```

As we can see from the above figure, by given an IP address sample, Radamsa can generate random outputs that follow the same pattern.

With those randomly generated inputs, we can modify previous unit test cases and replace the normal inputs with those generated corner inputs. An example is shown below.

```
public void testPollFirstWithIntegerMinHeapContainsNElements() {
    MinMaxPriorityQueue<Integer> q = MinMaxPriorityQueue.create();
    Collections.addAll(q, ...elements: 1,2,3,4,5,6,7,8,9,10);
    assertEquals(q.pollFirst().intValue(), actual: 1);
    assertEquals(q.peek().intValue(), actual: 2);
}
```

```
public void testPollFirstWithIntegerMinHeapContainsNElements2() {
    MinMaxPriorityQueue<Integer> q = MinMaxPriorityQueue.create();
    Collections.addAll(q, ...elements: 1,2,3,4,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,
            6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,
            5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,5,6,32769,-4,0,7,8,9,10);
    assertEquals(q.pollFirst().intValue(), actual: -4);
    assertEquals(q.peek().intValue(), actual: 0);
}
```

In this way, the line coverage and branch coverage of the tests are improved and one extra mutation bug that is related to heap size growing was found.

There are both advantages and disadvantages of using fuzzing or Radamsa. It can
Easily generate inputs with the given format and able to generate special inputs (corner cases). However, by using fuzzing, we still need to manually write assertions. And sometimes generate out-of-bound inputs. For example, there is some weird character in the IP address Example shown in the previous figure.

5. Future Direction

These two approaches improve our test robustness in different aspects, but we didn't use all the magic of them. In the future, we may explore more functions of SPF and try different fuzzers.