Prof. Dr. Peter Thiemann
Sergio Feo-Arenis
Sergiy Bogomolov

Summer Term 2014

Software Engineering

http://proglang.informatik.uni-freiburg.de/teaching/swt/2014/

Exercise Sheet 8

Exercise 1: DART (15 Points)

Recall the DART[1] [2] technique from the lecture and consider the following program:

```
int maxOf3(int x, int y, int z) {
  int m;
  if (x > y)
    if (x > z)
      m = x;
    else
      m = z;
  else if (y > z)
    m = y;
  else
    m = x;
  return m;
}
```

i. Apply *DART* on method `maxOf3`.

   Compute a set of tuples of input values (x, y, z) that covers all paths of `maxOf3`. Each tuple (x, y, z) is a test case which covers one path of `maxOf3`. Provide the concrete execution, the symbolic execution and the path constraints.

ii. Additionally generate test cases to ensure all possible combinations of relationships between x, y and z are covered. I.e., $x < y, x = y, x > y, y < z, y = z, y > z, \ldots$

iii. For each generated test case, determine *your* expected return value of `maxOf3` (i.e. the test oracle is you). Is method `maxOf3` faulty? If so, name the test case generated in i, that reveals the bug, if possible.

iv. Which kind of coverage is achieved by DART. what is your opinion of coverage criteria, in general? Is it guaranteed for *DART* to reveal the bug in this particular example? Justify your answer.

v. Consider extending DART to programs with loops and function calls. which problems do you expect? How would you deal with impure functions that return different results for the same parameters (for example, a random number function or a function returning the current time)?

[1] Paper: http://research.microsoft.com/en-us/um/people/pg/public_psfiles/pldi2005.pdf
[2] Talk: http://research.microsoft.com/en-us/um/people/pg/public_psfiles/talk-pldi2005.pdf

. . . . Solution . . . .

i. Test cases: consider the path constraints as annotated in the code:

```
1  int maxOf3(int x, int y, int z) {
2    int m;
3    if (x > y)
4      if (x > z)
5        m = x; // x > y ∧ x > z
6      else
7        m = z; // x > y ∧ x ≤ z
8    else if (y > z)
9      m = y; // x ≤ y ∧ y > z
10   else
11     m = x;  // x ≤ y ∧ y ≤ z (should be m = z)
12   return m;
13 }
```

We generate test cases that satisfy those constraints:

Line	Test Case	Exp. Value	Concrete Execution	Symbolic Execution
5	(2, 1, 1)	2	$2 > 1, 2 > 1, m = 2$	$x > y, x > z, m = x$
7	(2, 1, 2)	2	$2 > 1, \neg 2 > 2, m = 2$	$x > y, \neg x > z, m = z$
9	(1, 2, 1)	2	$\neg 1 > 2, 2 > 1, m = 2$	$\neg x > y, y > z, m = y$
11	(1, 1, 1)	1	$\neg 1 > 1, \neg 1 > 1, m = 1$	$\neg x > y, \neg y > z, m = x$

ii. The method `maxOf3` is faulty, line 11 should be `m = z;`. Unfortunately, no test case revealed the bug. The test case (1, 2, 3) would have revealed the fault.

iii. It is not guaranteed to find bugs using DART, test cases can be generated that satisfy the path constraints and produce the expected result.

Complete path coverage does not guarantee finding all bugs in general. Every path can be executed with many different variable valuations.

iv. In general, for DART one has to limit the number of paths to a finite number. Possible enhancements include:

Loops In case of loops, one could use path coverage (acyclic) rather than full path coverage (including cycles). Furthermore, one could also set a fixed maximum number of loop unrollings to consider.

Non-deterministic function calls Random values based on previous runs could be assumed for non-deterministic function calls. Limit attempts for DART to achieve full coverage to a finite number of runs (it can take infinitely many runs for DART to come up with a new (so far uncovered) path).

Exercise 2: Random Testing (5 Points)

Consider a (black box) function "boolean leapYear(int year)" that returns `true` iff the year input is a leap year[3].

- How would you set up random testing for this function?
- Assuming that the function's implementation just contains a single return statement without function calls, give a minimum set of test cases to validate this implementation.

[3] `http://en.wikipedia.org/wiki/Leap_year`

.... Solution ....

- A possible test environment for the function would be:

```java
import java.util.Random;
import java.util.GregorianCalendar;
import static org.junit.Assert.assertEquals;

public class TestLeapYear {
// perform n random tests
void testLeapYear(int n) {
    Random rand = new Random();
    // create a new calendar
    GregorianCalendar cal =
            (GregorianCalendar) GregorianCalendar.getInstance();
    for (int i=0; i<n; i++) {
        int x = rand.nextInt();
        int result = leapYear(x);
        assertEquals(result, cal.isLeapYear(x));
    }
}

@Test
public void doRandomTesting() {
  testLeapYear(1000);
}
}
```

  Here, we assume a reference implementation that serves the purpose of test oracle.

- Consider the following implementation:

```java
boolean leapYear(int year){
  return (year % 100 == 0) ? (year % 400 == 0) : (year % 4 == 0);
}
```

  It is equivalent to the implementation

```java
1  boolean leapYear(int year){
2    if (year % 100 == 0)
3      if (year % 400 == 0)
4        return true; // year is divisible by 100 and 400
5      else
6        return false; // year is divisible by 100 but not by 400
7    else if (year % 4 == 0)
8      return true; // year is not divisible by 100 but by 4
9    else
10     return false; // year is not divisible by 100 and not divisible
           by 4
11 }
```

  Now one can readily extract path conditions to generate test cases that achieve complete path coverage:

Line	Test Case	Expected result
4	400	true
6	300	false
8	104	true
10	1001	false

Submission

- Submit this sheet *before* the lecture of Thursdays.
- Late submissions will not be accepted.
- Deadline: Thursday 11:59 a.m.