CS 510 Software Engineering
Project 1 (100 Points), Version 1

Instructors: Lin Tan

Release Date: August 29, 2020

Due: 11:59 PM Monday, September 14, 2020
Submit: An electronic copy on D2L Brightspace

The entire project 1 is meant to be 10% of your final grade Please download proj1-skeleton.tar.gz from the course repository to get the necessary source code and test cases needed for finishing this project.

I expect each of you to do this project independently.

Submission Instructions:

Please read the following instructions carefully. **If you do not follow the instructions, you may be penalized up to 5 points.** Illegible answers receive no points.

Go to D2L Brightspace → Project 1 to submit your answer. Submit only one file in .zip format. Please name your file

<FirstName>-<LastName>-<Username>.zip

For example, use John-Smith-jsmith.zip if you are John Smith with username jsmith. The .zip file should contain the following items:

- a single pdf file "proj1_sub.pdf". The first page must include your full name, and your Purdue email address.
- a directory "q1" that contains your code (source code only; no binaries or object files) for Question 1
- "q2/sll_fixed.c" for Question 2
- "q4/TestM.java" for Question 4, and
- "q5/CFGTest.java" and "q5/CFG.java" for Question 5.

You can submit multiple times. After submission, **please view your submissions to make sure you have uploaded the right files/versions**.

Question 1 (10 points)

Write one simple program that prints out the value of -5 modulo 2 (e.g., -5%2 in C) in each of the following 4 programming languages: C/C++, Java, Perl, and Python. You will write 4 programs in total. Report what you have found and the reason for this discrepancy. Discuss at least 2 different strategies to cope with such a problem. Hint: think about what can change: developers, compilers, standards, etc.

Question 2 (20 points)

Compile program *sll_buggy.c* with the `-O0 -g` options[1], execute it with Valgrind, and answer the following questions:

(a) Run Test Case 1 and report the memory problem you find.
Paste the Valgrind outputs. Briefly explain the problem. We've made a copy of *sll_buggy.c*, renamed to "*sll_fixed.c*". Fix the bug in "*sll_fixed.c*", and use comments, e.g. "`// Fix for Q2 (a)`", to mark the bug fixes in your program. Explain how you fix this bug in the report. (6 points)

(b) Run Test Case 2 and report the memory problem you find.
Briefly explain the problem. Fix the bug in "*sll_fixed.c*". Use comments, e.g. "`// Fix for Q2 (b)`", to mark the bug fixes in your program. Explain how you fix this bug in the report. (6 points)

(c) Generate a test case that would cause a new bug, e.g., a buffer overflow bug.
Attach the test case, and briefly explain the problem. Fix the bug in "*sll_fixed.c*". Use comments, e.g. "`// Fix for Q2 (c)`", to mark the bug fixes in your program. Explain how you fix this bug in the report. (8 points)

Note that you only submit **one** *sll_fixed.c* file with all the fixes. For instructions on downloading Valgrind and installing it, please refer to the tutorial slides. You should do this exercise in Unix/Linux. You may find '`gdb`' very useful. Our CS Linux machines (mc01.cs.purdue.edu–mc18.cs.purdue.edu) already have Valgrind installed.

`Test Case 1`

```
insert>insert>delete>exit

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>111
enter the name:>Mary

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:d
enter the tel:>111

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
```

[1] According to Valgrind's documentation, "Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers. Using `-O0` is also a good idea, if you can tolerate the slowdown. With `-O1` line numbers in error messages can be inaccurate, although generally speaking running Memcheck on code compiled at `-O1` works fairly well, and the speed improvement compared to running `-O0` is quite significant. Use of `-O2` and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist."

```
Test Case 2

insert>insert>insert>edit>delete all>exit

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>111
enter the name:>Mary

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>112
enter the name:>John

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:e
enter the old tel :>111
enter the new tel :>111
enter the new name:>Mary

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:a

[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
```

Question 3 (5 points)

Consider the following program:

```
void bubbleSort(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

Using the minimal number of nodes (hint: 9, including the exit node), draw a Control Flow Graph (CFG) for method `bubbleSort()` and include it in your proj1_sub.pdf. The CFG should be at the level of basic blocks. See the lecture notes on *Structural Coverage* and *CFG* for examples.

Question 4 (25 points)

Consider the following (contrived) program and the control flow graph of method `M.m()`.

```
class M {
        public static void main(String [] argv){
                M obj = new M();
                if (argv.length > 0)
                        obj.m(argv[0], argv.length);
        }

        public void m(String arg, int i) {
                int q = 1;
                A o = null;
                Impossible nothing = new Impossible();
                if (i == 0)
                        q = 4;
                q++;
                switch (arg.length()) {
                        case 0: q /= 2; break;
                        case 1: o = new A(); new B(); q = 25; break;
                        case 2: o = new A(); q = q * 100; // no break
                        default: o = new B(); break;
                }
                if (arg.length() > 0) {
                        o.m();
                } else {
                        System.out.println("zero");
                }
                nothing.happened();
        }
}
```

```
class A {
        public void m() {
                System.out.println("a");
        }
}

class B extends A {
        public void m() {
                System.out.println("b");
        }
}

class Impossible{
        public void happened() {
                // "2b||!2b?", whatever the answer nothing happens here
        }
}
```

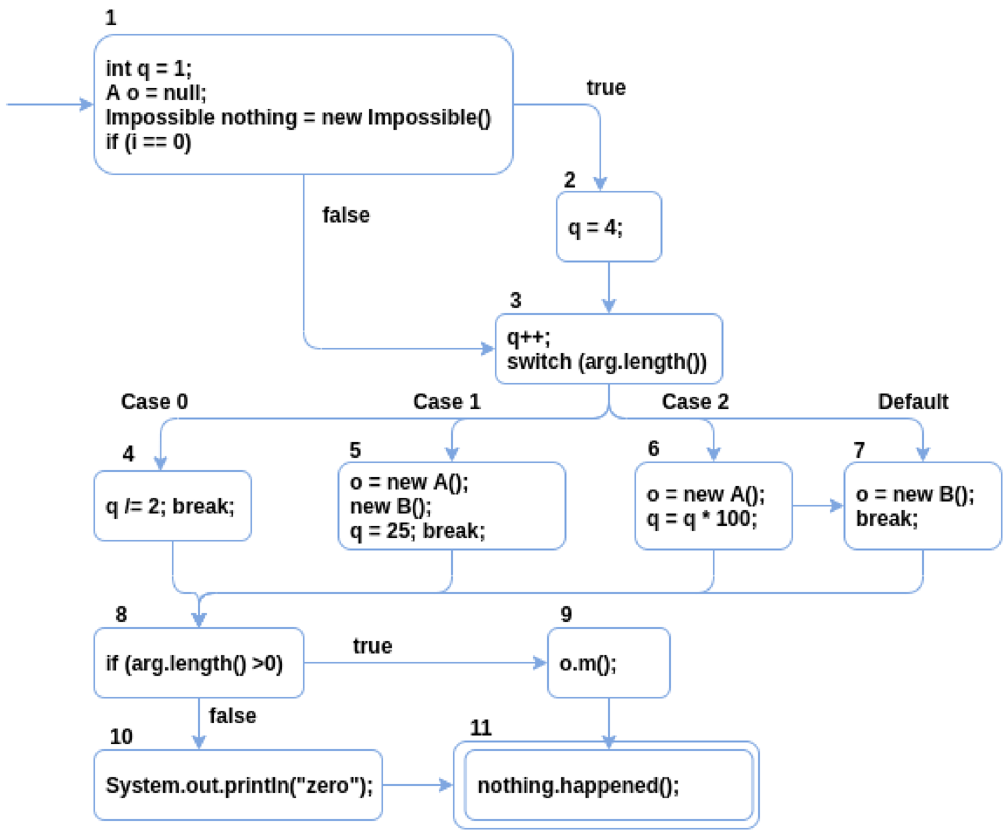

Figure 1: CFG of method M.m()

(a) List the sets of Test Requirements (TRs) with respect to the CFG for each of the following coverages: node coverage; edge coverage; edge-pair coverage; and prime path coverage. In other words, write four sets: TR_{NC}, TR_{EC}, TR_{EPC}, and TR_{PPC}. If there are infeasible test requirements, list them separately and explain why they are infeasible.

(b) Using `TestM-skeleton.java` as a starting point, write one JUnit Test Class that achieves, for method `M.m()`, each of the following coverages: (1) node coverage but not edge coverage; (2) edge coverage but not edge-pair coverage; (3) edge-pair coverage but not prime path coverage; and (4) prime path coverage. In other words, you will write four test sets (groups of JUnit test functions) in total. One test set satisfies (1), one satisfies (2), one satisfies (3), and the last satisfies (4), if possible. If it is not possible to write a test set to satisfy (1), (2), (3), or (4), explain why. For each test written, provide a simple documentation in the form of a few comment lines above the test function, listing which TRs are satisfied by that test. Consider feasible test requirements only for this part. The code template provides three import declarations for PrintStream, OutputStream, and ByteArrayOutputStream, which is a hint for how to handle the System.out.println() outputs created by function M.m().

Our CS Linux machines have JUnit installed (in /homes/cs510/jars/junit/). If you use your own machine and you want to run from terminal, you can get the JUnit jar file here: `https://github.com/junit-team/junit4/wiki/Download-and-Install`. JUnit is included in standard Java code development environments such as Eclipse and IntelliJ.

Question 5 (40 points)

You are to implement and test a class that will construct a partial[2] control-flow graph from the bytecode[3] of a given Java class using the ASM framework, version 7.3.1 (`http://asm.ow2.org`). Download all jars listed below and include them in your project's classpath.

- asm.jar
- asm-commons.jar
- asm-tree.jar
- asm-analysis.jar

If you use CS Linux machines, the jar file is in directory /homes/cs510/jars/asm/. You will also need JUnit for this question.

To illustrate the CFG functionality, consider the following class C:

```
public class C {
     int max(int x, int y) {
         if (x < y) {
             return y;
         } else return x;
     }
}
```

which can be represented in bytecode as (e.g. the output of
`java -cp path/to/asm/*.jar org.objectweb.asm.util.Textifier C.class, modified for readability`):

```
class C {
     Code:
     max(II)I
     0: ILOAD 1
     1: ILOAD 2
     2: IF_ICMPGE L0
     3: ILOAD 2
     4: IRETURN
     5: L0
     6: FRAME SAME
     7: ILOAD 1
     8: IRETURN
 }
```

You can easily draw the corresponding control-flow graph.

Graph representation of control-flow. Implement the class `CFG` to model control-flow in a Java bytecode program. A `CFG` object has a set of nodes that represent bytecode statements and a set of edges that represent the flow of control (branches) among statements. Each node contains:

- an integer that represents the position (bytecode line number) of the statement in the method.
- a reference to the method (an object of `org.objectweb.asm.tree.MethodNode`) containing the bytecode statement; and
- a reference to the class (an object of class `org.objectweb.asm.tree.ClassNode`) that defines the method.

Represent the set of nodes using a `java.util.HashSet` object, and the set of edges using a `java.util.HashMap` object, which maps a node to the set of its neighbors. Ensure the sets of nodes and edges have values that are consistent, i.e., for any edge, say from node a to node b, both a and b are in the set of nodes. Moreover, ensure that for any node, say, n, the map maps n to a non-null set, which is empty if the node has no neighbours.

You can find a partial implementation of the `CFG` class in `CFG-skeleton.java`. You'll need to rename it to `CFG.java` to continue. Your first task is to fill in the implementation of this class. Then you will write JUnit test cases to achieve some coverage criterion.

(a) Adding a node (4 points). Implement the method `CFG.addNode` such that it creates a new node with the given values and adds it to `nodes` as well as initializes `edges` to map the node to an empty set. If the graph already contains a node that is equivalent (under `.equals()`) to the new node, `addNode` does not modify the graph.

[2] This assignment ignores the labels that are traditionally annotated on nodes and edges, as well as the edges that correspond to method invocations or `jsr[ w]` bytecodes.

[3] `https://docs.oracle.com/javase/specs/jvms/se6/html/VMSpecTOC.doc.html`

(b) Adding an edge (4 points). Implement the method `CFG.addEdge` such that it adds an edge from the node `(p1, m1, c1)` to the node `(p2, m2, c2)`. Your implementation should update `nodes` to maintain its consistency with `edges` as needed. If the node does not exist in the graph, add the node.

(c) Deleting a node (4 points). Implement the method `CFG.deleteNode` such that it deletes a node with the given values, and all edges connected to the node, if any. If the graph does not contain a node that is `.equals` to the new node, `deleteNode` does not modify the graph. Your implementation should update `nodes` to maintain its consistency with `edges` as needed.

(d) Deleting an edge (4 points). Implement the method `CFG.deleteEdge` such that it deletes an edge from the node `(p1, m1, c1)` to the node `(p2, m2, c2)`.

(e) Reachability (6 points). Implement the method `CFG.isReachable` such that it traverses the control-flow graph starting at the node represented by `(p1, m1, c1)` and ending at the node represented by `(p2, m2, c2)` to determine if there exists any path from the given start node to the given end node. If the start node or the end node are not in the graph, the method returns false. You must not use recursion for this question.

(f) Testing (18 points). I have also provided a class `CFGTest` in `CFGTest.java`, which exercises your `CFG` class. Examine this test class. For each method you write, answer the following questions. (1)Does the test class satisfy NC and EC for the method? (2) Justify your answer. You should explain why it does/doesn't satisfy NC or EC. You can use graphs to explain it if necessary. (3) If it does not satisfy either coverage criterion, write additional JUnit test cases to satisfy one of them. Recall that to run JUnit tests in Junit 4.x, run `java org.junit.runner.JUnitCore CFGTest`, with the necessary jar and class files on your classpath. Running from an IDE (Eclipse or Intellij) might be easier as it handles the dependencies for you.