272 – Homework Assignment 2

Fall 2018

Due: Tuesday, November 27th, 11:00AM in class

Do not discuss the problems with anyone other than the instructor.

1. Consider a simple class called `BufferControl` that keeps track of the number of items in a buffer, the number of items that have been inserted to the buffer, and the number of items that have been removed from the buffer. To simplify things we do not keep track of the items in the buffer in this class. `BufferControl` class has three integer fields: `numItems`, `numInserted`, and `numRemoved`. The constructor `BufferControl()` sets all three fields are equal to zero. `BufferControl` class has three methods: `void insert()`, `void remove()`, and `int getNumItems()`. `getNumItems()` returns the value of `numItems` and does not change the values of any fields. When `insert()` method is called `numItems` and `numInserted` are incremented by one and `numRemoved` remains the same. When `delete()` method can only be can only be called when `numItems` is greater than or equal to 1. When `delete()` method is called `numItems` is decremented by one and `numRemoved` is incremented by one and `numInserted` remains the same.

(a) Write the contract for the `BufferControl` class in **JML** by writing the pre and post-conditions for each method. Also write the (strongest) class invariant.

(b) Assume that there is another simple class called `BoundedBufferControl`. `BoundedBufferControl` class is very much like the `BufferControl` except that it has another integer field called `size`. The constructor `BoundedBufferControl(int s)` requires the input value `s` to be greater than or equal to 1, sets `size` to the input value `s` and initializes the rest of the variables to zero. The constructor `BoundedBufferControl()` sets `size` to a constant value (say 10) and initializes the rest of the variables to zero. The value of the variable `size` is not modified by any methods after construction. The behaviors of the methods `remove()` and `getNumItems()` for `BoundedBufferControl` are identical to those of corresponding methods in `BufferControl`. The method `insert()` can only be called when `numItems` is strictly less than `size`, otherwise it behaves exactly like the `insert()` method of the `BufferControl`.

Write the contract for the `BoundedBufferControl` class in **JML** by writing the the pre and post-conditions for each method. Also write the (strongest) class invariant.

(c) Consider the following two class structures: 1) `BufferControl` is superclass of `BoundedBufferControl`, 2) `BoundedBufferControl` is superclass of `BufferControl`. Based on the inheritance rules in design by contract explain whether these options would or would not work and why.

2. Prove the following Hoare triples. Show the axioms and the inference rules you use.

(a) $\{true\}$ `if (x>=0) then y:=x else y:=-x` $\{y \geq 0\}$

(b) $\{true\}$ `sum:=0; i:=0; while (i<10) (sum:=sum+a[i]; i:=i+1)` $\{sum = \sum_{0 \leq k < 10} a[k]\}$

3. Compute the following weakest preconditions (show each step of the derivation).

(a) wp(`i:=i+5; j:=j-3`, $i + j = 0$)

(b) wp(`if (x>y) then z:=x else z:=y`, $z \geq x$)

4. Given the following pieces of code, draw the control flow graphs and give minimal test sets for the following criteria: statement coverage, branch coverage and path coverage. If there are statements, branches or paths that are not feasible to execute, indicate them. The inputs can take any integer value. Label the basic blocks in your control flow graphs. For each test case show the set of statements, branches or paths it covers using the basic block labels (for example, for the test set for path coverage, show the paths covered for each test case). For path coverage loops should be executed zero and one time.

(a)

```
int test1(int x, int y) {
  if (x > y + 1)
     z = 1;
  else
     z = 0;
  if (x > y+10)
     w = 11;
  else
     w = 0;
 return z+w;
}
```

(b)

```
int test2(int x, int y) {
  int i, sum, out;
  sum = 0;
  for (i = 0; i < x; i++) {
    sum = sum + i;
  }
  out = sum;
  if (sum < y)
    out = -1;
  return out;
}
```

5. Consider the following code that defines a search tree data structure. A search tree is a binary tree where the data in each node has to be greater than the data in all the nodes in its left subtree, and less than the data in all the nodes in its right subtree. Also, a value cannot appear more than once in a search tree.

(a) Assume that we want to use Korat to generate test cases using the repOk method. Is the repOk method written in a way that will lead to efficient test case generation with Korat? Explain. If not, rewrite the repOk method to improve the efficiency of the test case generation with Korat.

```
class SearchTree {
  Node root; // root node
  int size; // number of nodes in the tree
  static class Node {
    Node left; // left child
```

```
    Node right; // right child
    Comparable info; // data
  }

  boolean repOk() {
    boolean result;

    // checks that empty tree has size zero
    if (root == null) result = (size == 0);

    // checks that it is a tree
    if (!isTree()) result = false;

    // checks that size is consistent
    if (numNodes(root) != size) result = false;

    // checks that data is ordered and there are no duplicates
    if (!dataOK(root)) result = false;

    return result;
  }
}
```

(b) Assume that the data values range between values 1 to 3. Draw all the non-isomorphic search trees of size 3. Given a search tree of size 3, how many trees are there that are isomorphic to that tree.

6. Answer the following questions based on the feedback-directed random test generation technique used in Randoop:

(a) What type of sequences does Randoop generate? Are the sequences generated all at once? Explain.

(b) During test generation, what are the three possible ways of generating an object argument?

(c) What are the three conditions that make a sequence un-extensible?