COMP 587
Spring 2020

Final Review with Answers

The following review, **in conjunction with the midterm, midterm review, assignments, and in-class handouts**, are intended to be comprehensive for the final exam. The questions below are representative of the sort of questions you'll see on the exam.

Property-Based Testing

1.) Consider the following code representing a list of integers in Java:

```
public class MyList {
  // creates a new, empty list
  public MyList() { ... }

  // returns a copy of this list
  public MyList copy() { ... }

  // returns the number of elements in this list
  public int getLength() { ... }

  // adds this element to the end of the list
  public void addElement(final int element) { ... }

  // gets the element at the given index (0-indexed)
  public int getElementAt(final int index) { ... }

  // adds all the elements of the given list to this list
  public void appendAll(final MyList other) { ... }
}
```

The property below checks that the list's length is non-negative:

```
@Property
public void list_has_non_negative_length(final MyList list) {
  assertTrue(list.getLength() >= 0);
}
```

Write at least **three** additional properties below which collectively involve getLength, addElement, getElementAt, and appendAll. You don't have to call all methods in each property. The specific syntax doesn't matter much; this isn't for any one particular library. You may put constraints on the input if you wish.

```
@Property
public void added_element_is_at_end(final MyList list,
                                    final int element) {
  list.addElement(element);
  assertEquals(element,
               list.getElementAt(list.getLength() - 1));
}

@Property
public void adding_element_increases_length(final MyList list,
                                            final int element) {
  int originalLength = list.getLength();
  list.addElement(element);
  assertEquals(originalLength + 1, list.getLength());
}

@Property
public void appending_increases_length(final MyList a,
                                       final MyList b) {
  int aLen = a.getLength();
  int bLen = b.getLength();
  a.appendAll(b);
  assertEquals(aLen + bLen, a.getLength());
  assertEquals(bLen, b.getLength());
}
```

2.) Consider the following property, which has a precondition (`assume`):

```
@Property
public void add_prop(final int x, final int y) {
  assume(x > 0 && x < 2);
  assertEquals(y + 1, y + x);
}
```

When we attempt to run this property, our unit testing framework reports that the test was skipped. Why might this be happening?

The precondition is very specific, to the point where it won't be true for most inputs. We only apply the property to a fixed number of random inputs, and it's likely the case that the `assume` didn't hold for all of the inputs we tried. As such, the test is effectively skipped - we never got to run the `assertEquals`.

3.) The properties below are intended to test `min`, with the usual intention for `min`:

```
@Property
public void result_is_an_input(final int x, final int y) {
  final int result = min(x, y);
  assertTrue(result == x || result == y);
}

@Property
public void result_is_lte_first(final int x, final int y) {
  assertTrue(min(x, y) <= x);
}
```

3.a.) The above properties are not sufficient for a correct `min` implementation. Write an incorrect definition of `min` below which will nonetheless satisfy the above two properties.

```
public static int min(final int x, final int y) {
  return x;
}
```

3.b.) Write an additional correct property of min, where the property would fail (in general) underneath your above definition.

```
@Property
public void result_is_lte_second(final int x, final int y) {
  assertTrue(min(x, y) <= y);
}
```

Fully Automated Testing Tools and Techniques

4.) You decide to run AFL on your SUT. After 5 days, AFL finds no crashes, but it achieves 100% code coverage. How confident are you that the system is bug-free? Explain.

Not very. AFL can only find crashes, not general correctness bugs. For example, with the prior incorrect definition of `min`, AFL would never find a crash in this definition of `min`, but the `min` is nonetheless incorrect. While the SUT produces output under AFL, nothing checks that this output is correct; it only checks that the output wasn't a crash.

5.) In general, fully automated tools (e.g., AFL, concolic execution) can only find crash bugs, not arbitrary bugs. Why do these tools give up on other kinds of bugs?

Fundamentally, these tools generate arbitrary inputs to a system, and behave in system-agnostic ways. These tools don't have any understanding of what these inputs semantically mean, and they similarly don't understand what semantically correct output is for a given input. Without this additional information, they can only reason about simple things like crashes. This is referred to as the **oracle problem**, that is, knowing what the system should do underneath a given input, as opposed to what the system actually does underneath a given input.

6.) Why is concolic execution well-suited to test code with lots of `if` statements?

Concolic execution systematically explores different program paths. With this in mind, it can explore all the various branches through `if` statements, including nested `if` statements.

7.) Why is concolic execution poorly-suited to test code with lots of loops?

Each iteration through a loop yields a different program path, and concolic execution tries to explore all paths. This can lead to path explosion with loops, where it effectively gets "stuck" generating inputs that go through progressively more and more iterations of loops.

Verification

8.) Alice has verified that her SUT correctly implements its specification. Should Alice still bother with tests? Why or why not?

Alice should still test her SUT. There could be bugs in the specification itself, and testing will help find specification-level bugs.

9.) Theoretically, can you ever be 100% sure a system works correctly? Explain.

No; some assumptions always must be made. However, we can shrink those assumptions down significantly with verification efforts (e.g., "we assume the flight navigation software works correctly" can become "we assume first-order logic holds").

10.) Consider the Dafny code below, which implements some basic operations on natural numbers:

```
datatype Nat = Zero | Succ(n: Nat)

predicate lessThan(n1: Nat, n2: Nat)
{
  (n1.Zero? && n2.Succ?) ||
  (n1.Succ? && n2.Succ? && lessThan(n1.n, n2.n))
}

function method add(n1: Nat, n2: Nat): Nat
{
  if (n1.Zero?) then n2 else Succ(add(n1.n, n2))
}
```

10.a.) Write a lemma that would prove that `lessThan` is transitive. You only need to worry about preconditions and/or postconditions.

```
lemma lessThan_is_transitive(n1: Nat, n2: Nat, n3: Nat)
  requires lessThan(n1, n2);
  requires lessThan(n2, n3);
  ensures lessThan(n1, n3);
{}
```

10.b.) Write a lemma that would prove that `add` is commutative. You only need to worry about preconditions and/or postconditions.

```
lemma add_is_commutative(n1: Nat, n2: Nat)
  ensures add(n1, n2) == add(n2, n1);
{}
```