SE 465 W19 Midterm Solutions

Patrick Lam

March 5, 2019

(1)

Lots of changes are possible. I thought the obvious ones are changing something in the `if` condition but most of you changed one of the += operators.

Say you changed `vec == null` to `vec != null`. You could use the following test case:

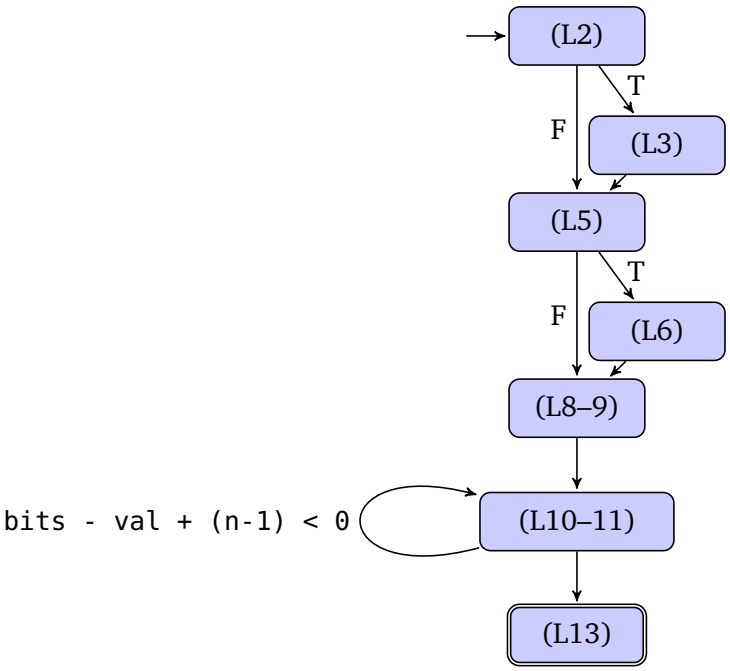
```
@Test
public void testAdd() {
  Location l1 = new Location(0.0, 4.65, 3.50, W);
  Location l2 = new Location(1.0, 2.0, 3.0, W2);
  l1 = l1.add(l2);
}
```

Best practice if you were actually writing test code would be to assert on the values in `l1` after `add`. On the other hand, best practice for writing an exam is to keep the test case as small as possible. I didn't say that you had to assert. If you do, that's great (and the exams that I looked at all did). In my case, the actual and expected behaviours for the original are successful executions with no output. The actual output for the mutant would be a `NullPointerException`. (You don't actually need expected output for the mutant).

(2)

Let's number the lines.

```
1      public int nextInt(int n) {
2          if (n<=0)
3              throw new IllegalArgumentException("n must be positive");
4
5          if ((n & -n) == n)  // i.e., n is a power of 2
6              return (int)((n * (long)next(31)) >> 31);
7
8          int bits, val;
9          do {
10             bits = next(31);
11             val = bits % n;
12         } while(bits - val + (n-1) < 0);
13         return val;
14     }
```



I don't actually care about little details like indicating the start and end nodes. The general structure is sufficient for full marks. You can omit lines 8–9 if you want. I just wanted to have an extra node for the entry to the `do` loop.

Statement coverage: Reachability of lines 1–5 depend on the input n. Because it's a do-while loop, the body of the loop is also going to be always executed as soon as line 6 is reached. And then line 10 is also unconditionally executed as soon as line 6. So there are in fact no difficulties in achieving 100% statement coverage as long as you pick the appropriate n.

Any terminating execution is going to get out of the loop, and it is reasonable to assume that this function terminates since I said it comes from the library. I guess you don't know for a fact that

`nextInt` terminates, so you should discuss potential issues with termination. Saying the assumption above is fine. Saying you don't know is fine. Not saying the assumption but relying on it is 4/5.

To achieve 100% statement coverage, you need n strictly positive (to reach line 3) and negative (line 2), and then you need `(n & -n) == n` (you can believe the comment and provide n a power of 2, 4 for instance) and n not a power of 2, say 5.

Someone pointed out that setting $n = 1$ ensures termination (you always get 0). I didn't expect students to deduce that.

Marks breakdown: 3 points for saying that 100% statement coverage is achievable by choosing n, 2 points for finding the n.

Branch coverage is a bit harder. The `if` statements are no harder. You need to make sure you go around the loop once as well as out of the loop. The crux is making sure that the execution goes around the loop, i.e. `bits - val + (n - 1) < 0`. You don't know what `next` is going to return, but you can say that you set `Random`'s state such that `next(31)` returns something smaller than `val - (n-1)`.

Same breakdown: 3 points for discussing the conditions, 2 points for specifically saying how to achieve branch coverage.

(3) Correct input: The easiest possible thing is to change one of the integers to a different integer.

Incorrect input: Change the URL; or, in the request, modify or delete one of the field names.

I chose the number 4 so that you would have to provide at least one of the requests that has a payload. The path of least resistance is to generate the three non-payload requests with randomly-generated data and, say, POSTing `/api/users`.

Pseudocode:

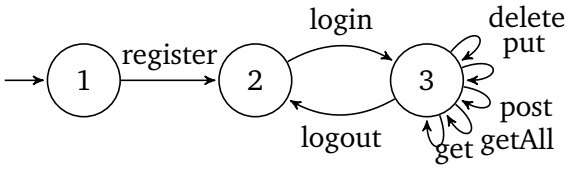
```
choose a random number between 1 and 4
case 1: generate request "GET /api/users"
case 2: generate request "GET /api/users/"+random int
case 3: generate request "DELETE /api/users/"+random int
case 4: generate request "POST /api/users"
           with payload {"name": random string, "job": random string}.
```

You could also write down a context-free grammar but then you have to write pseudocode for generating a string from the grammar, which is a lot of work. It's OK if you say how to create string from parser, 4/6 if you just give the grammar.

Any reasonable modification of the pseudocode is OK. For instance you could modify the request string, or change "random int" to "random string", or change the field names.

(4) It was in your interest to write as simple an FSM as you could get away with. I didn't actually expect everything to be in the FSM, and it was OK to abstract things away. I guess I could have been a bit more clear.

In my opinion, the best solution (but not the only one that gets full marks) includes two subparts. One of the subparts is for register/login/logout. Once logged in, then you can do all of the `users` operations. It would be fine to have only one of the subparts, and probably better for answering the exam.



For the SRTC (2 points), each node with a roundtrip needed to be represented in one of the test requirements. (While looking through the exams, I realized that the SRTC definition does not require one roundtrip per node with a cycle, i.e. multiple nodes can share the same cycle). The test suite was another 2 points, and then CRTC was another 2 points.

The safest way to answer SRTC here is the requirements $\{(2,3,2),(3,2,3),(3,3)\}$ although you can definitely argue for smaller sets which contain at least one cycle for each node, e.g. $\{2,3,2\}$ seems to meet the definition as I read it today. CRTC adds all of the round trips starting at 3. You should probably include the $(3,2,3)$ cycle, because you can argue that it's different from the $(2,3,2)$ cycle.

Note that you have to satisfy the test requirements that you give. So if you give a cycle for each node, then your test suite also has to satisfy that cycle, e.g. if you say $(a,b,c,a),(b,c,a,b)$ (among others) then you need the cycles starting at both a and b in your test suite. In that case, your test suite might miss going around a cycle from the point of view of each node in the cycle, so you would typically need to go around the cycle twice. But it's actually OK to have test requirements that only include (a,b,c,a).

(5a)

Multiple acceptable answers, but exploratory testing is most useful where human judgment is useful in identifying the correct behaviour. UI/UX bugs are a good specific example and when I was skimming answers, most of the good ones talked about UI bugs. (The lecture notes talk more about situations where exploratory testing is useful, rather than about specific bugs you might catch).

(5b) Fact 1: 55% of statements were executed by test cases. Fact 2: These statements did not throw uncaught exceptions. One answer I saw was "these statements are reachable". That is correct so it gets full marks. I was hoping for something more actionable, as in Fact 1 above.

(5c) "Nothing" is a correct answer. Also "these statements are not reachable by your testcases". Everyone I looked at got this.

(5d) Looking for "page objects". Extra sentences that contained mis-facts would trigger deductions.

(5e) I'd expect there to still be a failure. I thought that the fault would now be at the previous write to the same state. Answers saying that the fault was at the next access to that state are OK too.