SE 465 W19 Final Solutions

Patrick Lam

April 26, 2019

(1a) Many acceptable answers. Structured input is the key. 1) We could use hierarchical fuzzing for any Unix-like utility that accepts input. Simple levels of the hierarchy could just pass in random characters, while higher ones could provide valid input that is somehow interesting (e.g. input is very long, or has an interesting combination of option flags). 2) Word processor import functionality. 3) Media players. etc.

(1b) Sample answer: FindBugs. False positives can occur for any static rule violation that is for some reason deliberately left in the codebase. These can include things like unreachable code and unhandled exceptions.

More generally, false positives come from considering impossible executions. There is an abstraction and if the abstraction can't differentiate between a false positive and a true positive, then the tool can report a false positive. For instance, say an error condition only trips if a value is negative, but that value is computed by squaring some input (and hence is always positive).

(1c) Sample answer: AddressSanitizer (Asan). Asan aborts whenever it encounters a memory error, so it will miss any cascading memory errors. It is up to the user to re-run Asan to find any remaining errors.

Another type of answer, and more the one I was hoping for, is that dynamic analysis will miss anything that requires a particular input to trigger.

(1d) Sample answer: A logical conflict between changes (i.e. changesets C1 and C2 do not have a merge conflict, and both pass tests on their own, but cause test failures when combined)

Another answer: Forgetting to add a file to the repo, or failing to build under the specific configuration that CI tests (which may be different from the dev's build environment.)

(1e) Setup: Beginning up to `rules.addRule(rule);`
Teardown: None
Exercising: The `p.getSourceCodeProcessor()...` line
Verifying: The two asserts at the end.

(1f) State. We are using accessor methods to gather information about the final state of `r`, instead of observing how the state of `r` came to be (e.g. observing what methods were called).

(1g) Advantages:

- Can improve code quality

- Can communicate valuable lessons to the author and other contributors

Disadvantages (as far as I know I never discussed them, so that was the opportunity to think about it a bit on your own):

- Takes time and effort (opportunity cost)
- Bad code reviews can result in lots of back-and-forth over trivial details and bad inter-dev climate.

(1h) Sample answer: Firefox adds "invalid" attribute to image that seems valid.
Actual bug summary: Some images gets a spurious "invalid" ATK state attribute, leading to AT technologies to ignore them.

Talking about AT is not mandatory, although it's not a bad idea if you are trying to provide context for why someone might care about the bug.

(1i) The write to `n->data` succeeded. Some things we can deduce based on this are:

- the memory requested by `calloc (sizeof(struct node))` was allocated successfully, and we are writing to valid memory
- the node struct has a `data` field compatible with the value 5 (e.g. int, short, etc.)

(1j) Levels of test coverage may not be an indicator of the quality of tests. Although we have CRTC, the tests applied to the round trips may hardly test system behavior at all (e.g. no asserts in any tests). One thing that could be done to improve tests is to make sure each test has an assert for important values for each visited state. If we are able to edit the FSM, another thing we could do is make the states more granular.

(2.1) (2 points) a) There might be a `NullPointerException`. The open (`new Reader`) may throw an `IOException` (FileNotFound), which is caught; execution then proceeds to the `close` on the null `reader`.

b) (5 points) `@Nullable` is not necessary since `readFile()` cannot be called with a null `BufferedReader` from what is seen here. Either the instantiation of `new BufferedReader` will succeed or fail. If it succeeds, then `reader` isn't null. If it fails, then the call to `readFile()` doesn't happen.

@Nullable would be necessary when there is a possibility `BufferedReader r` passed in can be null, e.g. if we may fail to instantiate the 'reader' variable by moving the `readFile` call to after the `catch`. we will then need a null pointer exception check in the function.

c) (3 points) This code will not pass FB Infer, since FB Infer thinks `BufferedReader r` might be null when it enters the function and performing `r.readLine()` on a null `r` will fail. The code should be modified to include a `if (r == null) return;` line as the first line in the function.

(2.2) One example:

Original Program p

```
void threshold(int a, int b)
{
    int result = a+b;
    if (result > 10)
    {
        cout << "threshold not " << endl;
    }
    cout << "OK" << endl;
}
```

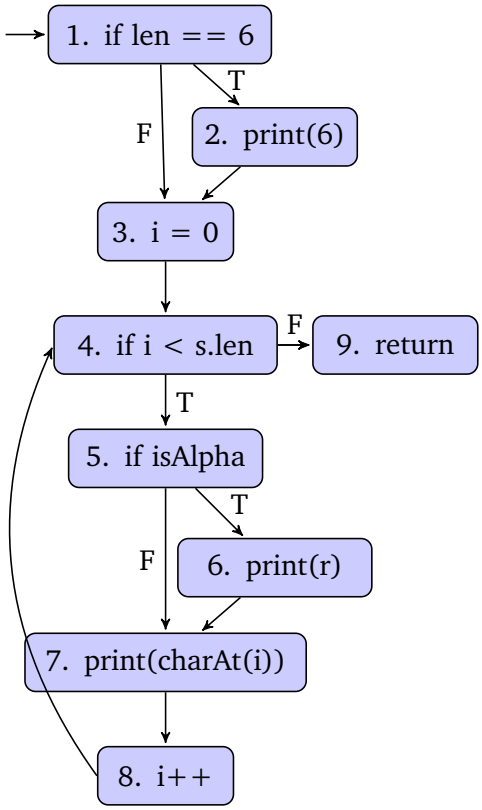
Mutated Program m

```
void threshold(int a, int b)
{
    int result = a+b;
    if (result >= 10) //added =
    {
        cout << "threshold not " << endl;
    }
    cout << "OK" << endl;
}
```

Suppose $T : \{\langle a = 4, b = 6\rangle\}$; the test case should print "OK". The test T achieves statement coverage on m as $a + b = 10$ and $4 + 6 >= 10$ enters the if block, but does not on p since $4 + 6 > 10$ does not enter the if block. I added the = sign to the if block so that running T on m will enter whereas running T on p will not.

My test suite T kills m, since running T on m will print "threshold not OK" but my test suite will be expecting the run to print "OK".

(3) The clarification was "edge coverage = branch coverage". I didn't say that you had to produce a CFG, but you probably should so that you can be sure that you achieve all of the edges. We wouldn't mark it, but it certainly helps in evaluating your solution.



Test requirements:

$$\{(1,2),(2,3),(1,3),(3,4),(4,9),(4,5),(5,6),(5,7),(6,7),(7,8),(8,4)\}$$

Let's take care of the 6 vs. not 6 by including strings "123456" and "1" in the input.

The loop definitely terminates for finite-length strings so we can be sure that $(4,9)$ is always going to execute. So that really just leaves making sure that we have an alpha and a non-alpha character in the string. So adding strings "abc123" and "123" would suffice.

The other edges are unconditionally reached assuming you enter the loop (i.e. you have an input of strictly positive length).

You could minimize the test suite, but why would you?

(4a) Oops. A significant minority of the class told me there are four mistakes. Sorry. You only have to identify 3 mistakes. And, I changed the mistakes from the ones on the 2017 final.

- So, of course, `mockTransport()` fits the definition of a mistake. I meant it to say `mockTransport`; I didn't mean it to be a mistake.
- `mockDriver` should be a mock, namely `mock(Driver)`.
- `setDestination(10)` does not match the destination of `5` that the `Person` constructor is going to set.
- There is no base price set (so it's actually impossible to say what the payment should be).
- `waitUntilArrivedAt()` should take a parameter (which should be 100).

(4b) (2 points for state vs behaviour) That is a state-based assert on the `Person` class. We should use a behaviour-based test to be consistent with `testGoHome`.

The number 405 also is incorrect somehow. I'm not sure how that happened.

(3 points for suggesting a fix): mocking Person is not a fix, because then you'd be just testing the test. Other reasonable suggestions get points. The best fix would modify everything to make the distance travelled visible through the classes here (e.g. the driver should report that.)

(5) (a) A clarification here was that I was just expecting yes-or-no; an answer doesn't have to capture the declaration itself.

```
//LocalVariableDeclaration[count(./VariableDeclarator)>1]
```

(b) The prompt was just "Write an XPath expression which detects empty catch blocks". This works.

```
//CatchStatement/Block[descendant::BlockStatement]
```

So does this:

```
//CatchStatement/Block[count(./BlockStatement)<1]
```

People responded to the "expected" thing, which was fine. Sometimes people started further up the tree. Let's not dock marks for that, unless it's too far up the tree that it is unnecessarily restrictive. I expect that many solutions wouldn't actually work, but that's fine, since students don't have a computer at hand. Just look for the concepts of locating the node and then filtering on something that seems reasonable.

Somehow a lot of students got tripped up by my statements about comments. I was looking for emptiness. I guess the question was somewhat confusingly worded. I deducted 3 points if it didn't actually check for emptiness.

(6) Oops, I forgot some closing statements. Here's what the grammar should look like.

```
desc :- building*
```

```
building :- "building" BUILDING "\n" floor* "end-building\n"
floor :- "floor" DIGIT "\n" room* "end-floor\n"
room :- "room" DIGIT DIGIT DIGIT DIGIT "\n" occupant* "end-room\n"
occupant :- "occupant" OCCUPANT "\n"
```

(3 points) In a context-free grammar it's quite hard to enforce the stated constraint. You could do it at a grammar level by having a bunch of floorN productions. But that would be weird. Better to do it at a higher level, for instance by modifying generated rooms so that they meet the stated constraint.

The `"\n"` is not mandatory. I guess I also didn't specifically say that you had to have `building*` at top level (i.e. that you could generate more than one building).