SOFTWARE TESTING: SAMPLE ANSWERS TO TUTORIAL 3

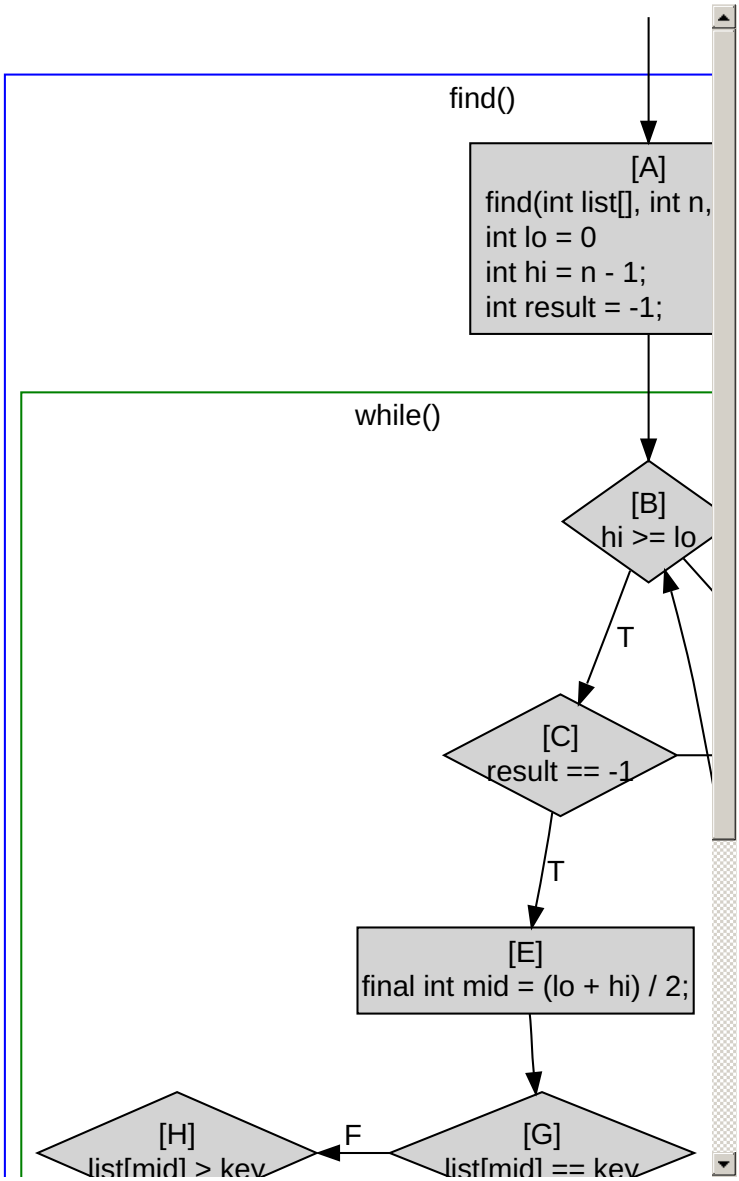
Here are sample answers to the activities for tutorial 3, on data flow testing.

The method find() implements a binary search on an ordered list of integers, in order to find the index of any element with value *key*. For example, on these inputs:

```
list[] = { 0, 3, 5, 8, 8, 17 }
n = 6
key = 5
```

… find() should return 2 (the index at which we find the key value 5). Given *key* = 6, the return value should be -1 (not found), and given *key* = 8, the return value may be 3 or 4 (since 8 appears twice in *list[]*).

CONTROL FLOW GRAPH



Note how I've broken the two basic conditions in the while loop out into separate nodes; this makes it easier to track the behaviour of short-circuiting. You don't need to do this in diagrams for statement or branch coverage, but it helps when you're thinking about the different types of condition coverage, as well as data flow.

DEFS, AND DU PAIRS

Here's a table presenting all the variables, where they're defined, and where they're used.

Variable	Definitions	DU pairs
list[]	A (parameter)	AH, AG

n	A (parameter)	AA
key	A (parameter)	AH, AG
mid	E	EH, EG, EL, EM, EK
lo	A, M	AB, AE, MB, ME
hi	A, L	AB, AE, LB, LE
result	A, K	AC, AD, KC, KD

PATHS FOR “ALL DU PAIRS”, “ALL DU PATHS”

To cover “all DU pairs”, the test suite must include a def-clear path from each variable definition to all uses of that variable that are reachable from the definition.

To cover “all DU paths”, we're looking again at covering all DU pairs def-use pairs, but this time we don't just want one path for each pair, we want **all** paths between each pair, modulo loop iterations.

TEST FOR “ALL DU PAIRS”

As with previous coverage criteria, let's start with a simple test and see how well it satisfies the “all DU pairs” criterion:

```
list[] = { 0, 3, 5, 8, 8, 17 }
n = 6
key = 8
```

Executing this, we start off on the first iteration with *lo* = 0 and *hi* = 5. *mid* will be 2 (since Java rounds ints down), and since *list[2]* = 5 (< *key*), we'll execute node M and start the next iteration with *lo* = 3 and *hi* = 5. *mid* will now be 4, and since *list[4]* = 8 (== *key*), we'll set *result* in node K, and execution will terminate via node C on the next iteration. So our execution path will have been as follows:

```
A BCEGHM BCEGK BCD
```

We can then cross off DU pairs covered in our table (for convenience I've used the HTML <s> tag to strike through DU pairs we've covered; if you can't see any struck-through text, please let me know and I'll try to come up with an alternate way of doing this). Note that because *lo*, *hi* and *result* can be defined in more than one place, we have to be careful that the paths we're looking at are def-clear paths, e.g. for *result*'s A→C path, there must be no instance of node K redefining *result* between A and C.

Variable	Definitions	Def-use pairs
list[]	A	~~A→G~~, ~~A→H~~
n	A	~~A→A~~
key	A	~~A→G~~, ~~A→H~~
mid	E	~~E→K~~, E→L, ~~E→M~~, ~~E→G~~, ~~E→H~~
lo	A, M	~~A→E~~, ~~A→B~~, ~~M→E~~, ~~M→B~~
hi	A, L	~~A→E~~, ~~A→B~~, L→E, L→B
result	A, K	A→D, ~~A→C~~, ~~K→D~~, ~~K→C~~

Looking at the left-over DU pairs here, we need to exercise the DU pairs involving L and to also exercise the AD pair which can only occur when the list is empty. To exercise the DU pairs involving node L, we can add the following test:

```
list[] = { 0, 3, 5, 8, 8, 17 }
n = 6
key = 3
```

The execution path will be (*lo* = 0, *hi* = 5, *mid* = 2; *lo* = 0, *hi* = 1, *mid* = 0; *lo* = 1, *hi* = 1, *mid* = 1; *result* = 1 in node K, and execution will terminate via node C on the next iteration.):

```
A BCEGHL BCEGHM BCEGK BCD
```

This enables us to cross off the following remaining DU pairs:

Variable	Definitions	Def-use pairs
list[]	A	none remaining
key	A	none remaining
mid	E	~~E→L~~
lo	A, M	none remaining
hi	A, L	~~L→E~~, ~~L→B~~
result	A, K	A→D

The only uncovered DU pair is A→D for *result*. This can be covered with a test that provides an empty input array : *n* will be 0, *hi* will be initialised to -1, and execution will terminate straight off (path A BD).

So, our final test suite to satisfy all DU pairs is as follows:

	list[]	*n*	*key*
Test 1	{ 0, 3, 5, 8, 8, 17 }	6	8
Test 2	{ 0, 3, 5, 8, 8, 17 }	6	3
Test 3	{ }	0	1

TEST FOR “ALL DU PATHS”

Start with the above all DU pairs test suite.

Since all DU paths requires that the def-clear paths not include more than one iteration of any loops, we need to take care that we've not used any def-clear paths which cover more than one iteration of the main loop. This would mean, from Test 1, that we can't have any DU pairs running from the opening A to anything in the closing BCD (because two loop iterations intervene); from Test 2, that we can't have any DU pairs running from the A to the closing BCEGHL BD, or from the first loop iteration BCEGHL to the closing BD. And there's a problem there: we ticked off *result*'s A→D path in Test 2. That's got three iterations of the loop in the middle, so we need to get it some other way. But in Test 3, we have execution path A BD — which includes A→D without an intervening K *or* any loop iterations, just as we'd like. So the above test doesn't break our “not more than one loop” rule, but does it contain all possible paths?

The only DU pairs between which there are more than one loop-free path are the pairs A→D and K→D for variable*result*: with both of these, the D can be reached either by BD or BCD. Are these covered by our current test suite?

- Test 2 covers A→BCD *but* there are too many loop iterations in between the pair; Test 3 covers A→BD again, but with no intervening loops.
- Going back to our impossible pairs notes above, A→BCD is impossible: you can't have a def-clear path for *result* from setting `result = -1` to a decision in which `result == -1` is false.
- What about K→BD? This too is impossible: if we hit node K, we know that *result* is no longer -1, so we will definitely exit the loop via CD on the next iteration; in order to exit via BD before that, `hi >= lo` would have to be false — however, it was true on the previous iteration of the loop, and since we took the path BCEG to node K, we know that *lo* and *hi* haven't changed since we last compared them. So it will still be true on the next iteration, and we'll exit via CD.
- Test 1 covers K→BCD (but nothing for A since the presence of K means that we don't have a def-clear path from A to D).

So actually our existing test suite already covers all of the DU paths that it's possible to cover.

TEST THAT SATISFIES “ALL DU PAIRS”, BUT FAILS “ALL DU PATHS”

This is a little tricky, since we've established above that there's no DU pair between which there exists more than one feasible path.

Consequently the only approach to this would be to break the “no more than one loop” rule. For example, we could replace Test 3 with something which is good enough for all uses, but involves lots of loop iterations so it won't satisfy all DU paths. Here's a new suite doing just that: Test 3 gets replaced with two new tests, one of which gets us A→BD for *lo*, and the other for *hi* (noting that for *lo* we need to avoid the def in node M, and for *hi* we need to avoid the def in node L):

	list[]	*n*	*key*	Path
Test 1	{ 0, 3, 5, 8, 8, 17 }	6	8	(as before)
Test 2	{ 0, 3, 5, 8, 8, 17 }	6	3	(as before)
Test 3	{ 0, 3, 5, 8, 8, 17 }	6	-1	A BCEGHL BCEGHL BD
Test 4	{ 0, 3, 5, 8, 8, 17 }	6	18	A BCEGHM BCEGHM BCEGHM BD

Version 1.1, 2018/01/09 13:42:03

Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, Scotland, UK
Tel: +44 131 651 5661, Fax: +44 131 651 1426, E-mail: school-office@inf.ed.ac.uk
Please contact our webadmin with any comments or corrections.