Lecture 4 — January 11, 2010

Patrick Lam *version 2*

Some binary distinctions

Let's digress for a bit and define some older terms which we won't use much in this course, but which we should discuss briefly.

- *Black-box testing.* Deriving tests from external descriptions of software: specifications, requirements, designs; anything but the code.
- *White-box testing.* Deriving tests from the source code, e.g. branches, conditions, statements.

Our model-based approach makes this distinction less important.

Two ideologies for constructing tests:

- *Top-down testing.* Constructing tests of behaviour starting with `main()`.
- *Bottom-up testing.* Constructing tests of behaviour starting at the leaves of the call graph.

Neither of these techniques work out completely in practice; top-down is impractical because it's hard to control each leaf procedure individuall from `main()`, and bottom-up because it's hard to test methods working together. Usually we use middle-out approaches.

Two different approaches to quality assurance:

- Static: approaches that don't involve running the code, e.g. type checking, code reviews.
- Dynamic: usual testing-based approaches; we focus on these in this course.

Some mixed approaches are coming out of research labs as well.

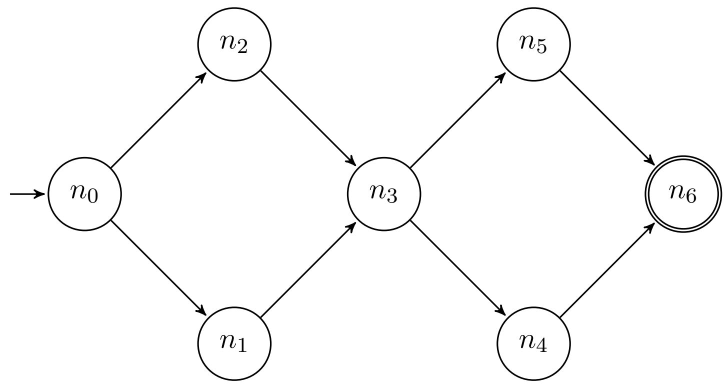
Test paths and cases

We resume our graph coverage content with the following definition:

Definition 1 *A graph is* single-entry/single-exit *(SESE) if N_0 and N_f have exactly one element each. N_f must be reachable from every node in N, and no node in $N \setminus N_f$ may be reachable from N_f, unless $N_0 = N_f$.*

The graphs that we'll be talking about in this course will almost always be SESE.

Here's another example of a graph, which happens to be SESE, and test paths in that graph. We'll call this graph D, for double-diamond, and it'll come up a few times.



Here are the four test paths in D:

$$[n_0, n_1, n_3, n_4, n_6]$$
$$[n_0, n_1, n_3, n_5, n_6]$$
$$[n_0, n_2, n_3, n_4, n_6]$$
$$[n_0, n_2, n_3, n_5, n_6]$$

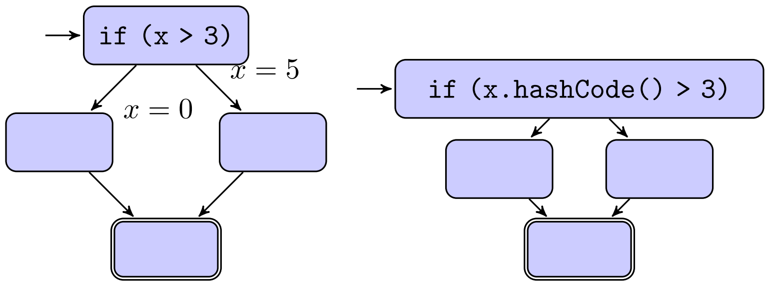
We next focus on the path $p = [n_0, n_1, n_3, n_4, n_6]$ and use it to explain several path-related definitions. We can say that p *visits* node n_3 and edge (n_0, n_1); we can write $n_3 \in p$ and $(n_0, n_1) \in p$ respectively.

Let $p' = [n_1, n_3, n_4]$. Then p' is a *subpath* of p, and conversely, p *tours* p'. Note that any path tours itself.

Test cases and test paths. We connect test cases and test paths with a mapping path_G from test cases to test paths; e.g. $\text{path}_G(t)$ is the set of test paths corresponding to test case t.

- usually we just write path since G is obvious from the context.
- we can lift the definition of path to test sets T by defining $\text{path}(T) = \{\text{path}(t) | t \in T\}$.
- each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

Here's an example of deterministic and nondeterministic control-flow graphs:



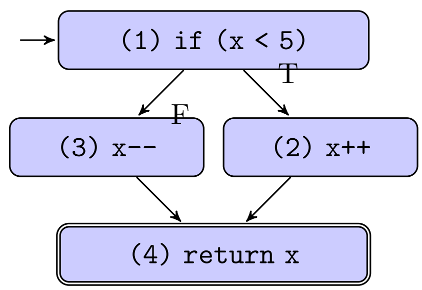
Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

Indirection. Note that we will describe coverage criteria with respect to *test paths*, but we always run *test cases*.

Example. Here is a short method, the associated control-flow graph, and some test cases and test paths.

```
int foo(int x) {
  if (x < 5) {
    x ++;
  } else {
    x --;
  }
  return x;
}
```



- Test case: $x = 5$; test path: $[(1), (3), (4)]$.
- Test case: $x = 2$; test path: $[(1), (2), (4)]$.

Note that (1) we can deduce properties of the test case from the test path; and (2) in this example, since our method is deterministic, the test case determines the test path.

Graph Coverage

Having defined all of the graph notions we'll need for now, we apply them to graphs. Recall our previous definition of coverage:

Definition 2 *Given a set of test requirements* TR *for a coverage criterion C, a test set T* satisfies *C iff for every test requirement* tr *in* TR*, at least one t in T exists such that t satisfies* tr*.*

We apply this definition to graph coverage:

Definition 3 *Given a set of test requirements* TR *for a graph criterion C, a test set T satisfies C on graph G iff for every test requirement* tr *in* TR*, at least one test path p in path(T) exists such that p satisfies* tr*.*

We'll use this notion to define a number of standard testing coverage criteria. (At this point, the textbook defines predicates, but mostly ignores them afterwards. I'll just ignore them right away.)

Recall the double-diamond graph D which we saw on page 2. For the *node coverage* criterion, we get the following test requirements:

$$\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

That is, any test set T which satisfies node coverage on D must include test cases t; the cases t give rise to test paths path(t), and some path must include each node from n_0 to n_6. (No single path must include all of these nodes; the requirement applies to the set of test paths.)

Let's formally define node coverage.

Definition 4 *Node coverage: For each node $n \in reach_G(N_0)$,* TR *contains a requirement to visit node n.*

We will state all of the coverage criteria in the following form:

Criterion 1 *Node Coverage (NC):* TR *contains each reachable node in G.*

We can then write

$$T\text{R} = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}.$$