Lecture 9 — January 23, 2015

Patrick Lam *version 1*

Review: Statements, branches and beyond

We talked about statement (length-0 paths) and branch coverage (length-1 paths) last time. I'll say this again later, but for real programs, 80% coverage is usually good enough; but also consider what is not tested. Also, Assignment 1 Question 1 should point out to you that it's possible to have 100% statement coverage but not actually test anything, if you write test cases that don't have asserts.

We could extend to paths of length 2 and beyond, but soon that gets us to Complete Path Coverage (CPC), which requires an infinite number of test requirements.

Criterion 1 Complete Path Coverage. *(CPC)* TR *contains all paths in G.*

Note that CPC is impossible to achieve for graphs with loops.

No prime paths this year. If you look at previous years' materials you'll see discussion of prime paths. I've chosen to exclude them this year.

Testing State Behaviour of Software via FSMs

We can also model the behaviour of software using a finite-state machine. Such models are higher-level than the control-flow graphs that we've seen to date. They instead capture the design of the software. There is generally no obvious mapping between a design-level FSM and the code.

We propose the use of graph coverage criteria to test with FSMs.

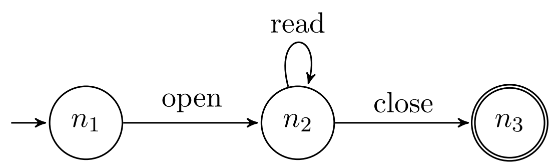
- nodes: software states (e.g. sets of values for key variables);
- edges: transitions between software states, i.e. something changes in the environment or someone enters a command.

The FSM enables exploration of the software system's state space. A software state consists of values for (possibly abstract) program variables, while a transition represents a change to these program variables. Often transitions are guarded by preconditions and postconditions; the preconditions must hold for the FSM to take the corresponding transition, and the postconditions are guaranteed to hold after the FSM has taken the transition.

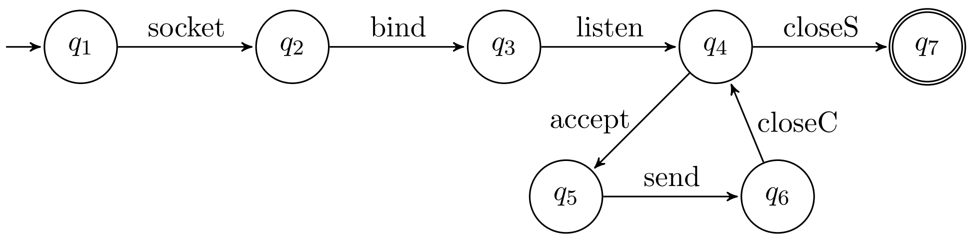
- node coverage: visiting every FSM state = state coverage;

- edge coverage: visiting every FSM transition = transition coverage;
- edge-pair coverage: actually useful for FSMS; transition-pair, two-trip coverage.

Examples. The next few graphs represent finite state machines rather than control-flow graphs. Our motivation will be to set up criteria that visit round trips in cyclic graphs.



or perhaps



The next criteria are mostly not for CFGs.

Definition 1 *A* round trip *path is a path of nonzero length with no internal cycles that starts and ends at the same node.*

Criterion 2 Simple Round Trip Coverage. *(SRTC)* TR *contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

Criterion 3 Complete Round Trip Coverage. *(CRTC)* TR *contains all round-trip paths for each reachable node in G.*

Exercise. Create a Finite State Machine for some system that you're familiar with.

Deriving Finite-State Machines

You might have to test software which doesn't come with a handy FSM. Deriving an FSM aids your understanding of the software. (You might be finding yourself re-deriving the same FSM as the software evolves; design information tends to become stale.)

We'll see some tools—iComment and Daikon—for obtaining sequencing constraints from comments/documentation and from the code.

Control-Flow Graphs. Does not really give FSMs.

- nodes aren't really states; they just abstract the program counter;
- inessential nondeterminism due e.g. to method calls;
- can only build these when you have an implementation;
- tend to be large and unwieldy.

Software Structure. Better than CFGs.

- subjective (which is not necessarily bad);
- requires lots of effort;
- requires detailed design information and knowledge of system.

Modelling State. This approach is more mechanical: once you've chosen relevant state variables and abstracted them, you need not think much.

You can also remove impossible states from such an FSM, for instance by using domain knowledge.

Specifications. These are similar to building FSMs based on software structure. Generally cleaner and easier to understand. Should resemble UML statecharts.

General Discussion. Advantages of FSMs:

- enable creation of tests before implementation;
- easier to analyze an FSM than the code.

Disadvantages:

- abstract models are not necessarily exhaustive;
- subjective (so they could be poorly done);
- FSM may not match the implementation.