Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 13 — February 1, 2010

Patrick Lam *version 2*

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

Dataflow coverage doesn't apply very well to FSMs. We could associate defs and uses with FSM edges; however, even if we make sure to achieve edge coverage, we might not be able to individually control all of the abstract variables underlying the FSM.

Exercise. Create a Finite State Machine for some system that you're familiar with. (In class, we saw a traffic light (AM) and for baking cookies (PM); with the baking cookies example, we saw both an imperative FSM which is CFG-like and a more state-based FSM. We had a question about whether we could split the state-based FSM into an oven FSM and an ingrediate FSM. Yes, we could, but then we'd have to understand how to synchronize different FSMs, which is beyond the scope of this class.)

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 four techniques.

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.