Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 22 — March 2, 2015

Patrick Lam *version 1*

Graph Coverage for Specifications

We'll move further up the abstraction chain now and talk about testing based on specifications. Specification-based testing can help ensure that a system doesn't drift too far from its intended behaviour, and tends to focus on higher-level concerns than the testing we've seen so far, which is based on properties of the source code.

Sequencing Constraints

Libraries often come with constraints on permissible actions; we call these constraints sequencing constraints.

Java Iterators. Here are two possible constraints.

(1) A good practice is to always call `hasNext()` before calling `next()`. (2) The Java API states that clients must not use an `Iterator` after they have modified the underlying `Collection`. For instance, `c = new LinkedList(); c.add(o1); i = c.iterator(); i.next(); c.add(o2); i.next(); /* bad! */`

The general format of sequencing constraints is "do X before Y", or "don't do W before A". Here is another sequencing constraint, this one implicit.

```
/* @requires size() > 0 */
public int dequeue() { ... }

/* @ensures size() = \old(size()) + 1 */
public void enqueue (int x) { ... }
```

Assuming that the only way to increase `size()` is by calling `enqueue()`, then a client had better call `enqueue()` before calling `dequeue()`.

It's generally useful to look for such constraints in code that you're developing or testing, even if it's hard to identify them. These constraints specify partial program properties.

Using Sequencing Constraints. Our approach will be to verify that sequencing constraints hold (or are never violated) on all paths through a graph modelling the system under test.

Let's consider the following three methods:

- `open(String fname)`
- `close()`
- `write(String line)`

and the following rules:

1. `open(fn)` must be executed before any call to `write(t)`;
2. `open(fn)` must be executed before any call to `close()`;
3. `write(t)` must not execute after `close()` unless `open(fn)` has executed in between;
4. `write(t)` should be executed before every `close()`;
5. `close()` must not execute after `close()` unless `open(fn)` executes in between;
6. `open(fn)` must not execute after `open(fn)` unless `close()` executes in between.

How does this simplify reality?

We will check these sequencing constraints on graphs, typically CFGs.

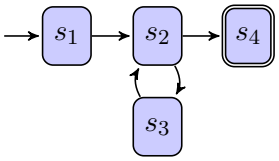

Figure 1: Simple Control-Flow Graph

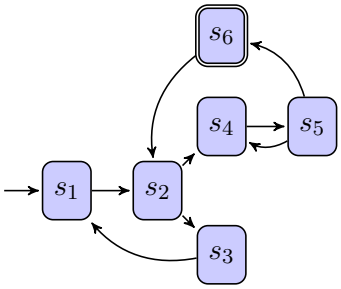

Figure 2: More Complicated Control-Flow Graph

Static approach. We can deduce, without running anything (i.e. statically), that the program in Figure 1 must satisfy all of the properties except for the one which requires that `write` occur before `close`. The model of the program in Figure 2 shows that it may violate the property that requires no writes to occur after closes. However, the back edge (s_5, s_4) might be infeasible.

Dynamic approach. By running the programs with suitable test cases, we could explicitly identify violating cases. If we are unable to find any violating cases, we can have some confidence that, in practice, the program never violates the properties.

Test Requirements

We can use test requirements to verify sequencing constraints in two ways: (1) we can create tests that satisfy various coverage criteria, thus suggesting that the program probably doesn't violate properties; or (2) we can set up test requirements, if satisfied, exhibit executions where constraints are violated. We might, for instance, set up the following test requirements:

- cover every path from the start to all `write()` nodes that don't go through any `open()` nodes;
- cover every path from the start to all `close()` nodes that don't go through any `open()` nodes;
- etc.

We are hoping that such test requirements will be infeasible; any program that meets the specification will be unable to satisfy these test requirements.

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 (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 = 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.

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've seen 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.