COSC345 Software Engineering

Property-based testing, and coverage

Richard A. O'Keefe

July 18, 2017

Test Coverage

- Q. How do we evaluate a set of tests?
- A. Untested code is untrusted code. We evaluate *how much* of the code is exercised by tests.
- This is what *test coverage* measures.

Levels of Coverage

- Class/module
- Method/function
- Statement
- Branch
- Path

Class/module coverage

- Is there are least one test case for each class (or module)?
- Is there a specification you *can* write a test case for?
- Can you tell whether an object/module is in a consistent state? (Invariant.)
- Who is responsible for testing?

Method/function coverage

- Is each method or function called in at least one test?
- Is there a specification you *can* write a test case for?
- What must you do to set up a test for that function? *E.g.*, to test adding to a set, you must start by creating a set.
- Watch out for #ifdef.

Statement coverage

- Is each statement executed at least once by some test?
- This is what gcov does. Example pop2lex.txt
- (gcov also gives us function coverage.)
- If a statement isn't executed, construct a test case to force it to be executed.
- Error handling code is particularly likely to be untested.
- Watch out for #ifdef.

Branch coverage

- Consider "if (a && b) c()".
- It has two statements, statement coverage.
- But there are three cases:
  - a false, b who cares?
  - a true, b false
  - a true, b true
- Each branch should go both ways in some test.
- Fixed.hs example

Path coverage

- Is every *path* through a function tested?
- When there are loops, the set of paths may be infinite, so not always practical.
- Branch coverage gets part way.
- Watch out for off-by-one errors.

Unit testing

- XUnit family of testing frameworks, began with SUnit for Smalltalk. See https://en.wikipedia.org/wiki/SUnit and especially http://wiki.c2.com/?SmalltalkUnit
- Set up a *fixture*, which is a collection of data *etc* used by the tests. Run the *cases*. Tear down the fixture.
- Test case ensures method precondition satisfied. Calls method. Checks result.
- Need precondition, normal postcondition, exception postconditions.

Property-based testing

- Began with Koen Claessen's QuickCheck for Haskell.
- See www.cs.tufts.edu/∼nr/cs257/archive/john-hughes/quick.pdf
- Programmer states *properties* that should be true.
- System generates *test data* at random.
- Uses types to choose generator.

Simple examples

```
prop_RevUnit x =
  reverse [x] == [x]

prop_RevApp xs ys =
  reverse (xs++ys) == reverse xs ++ reverse ys

prop_RevRev xs =
  reverse (reverse xs) == xs

Main> quickCheck prop_RevApp
OK: passed 100 tests.
```

What does that do?

- The type of prop_RevApp is $[x] \to [x] \to$ Bool
- Making a random list, not so hard.
- Making a random element of unknown type, can't do that.
- Need an explicit type.
- prop_RevApp :: $[\text{Int}] \to [\text{Int}] \to$ Bool
- Generates 100 pairs of random lists, calls prop_RevApp, checks that it comes out True each time.

Without types

- You have to tell the library *somehow* what generator to use. If you can't use the types, you must explicitly call a generator.
- PropEr for Erlang does this.
  See http://proper.softlab.ntua.gr/Tutorials/PropEr_introduction_to_Property-Based_Testing.html
- Extended example in class.