Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 23 — March 3, 2010

Patrick Lam *version 1*

Program Based Grammars

The usual way to use mutation testing is by generating mutants by modifying programs according to the language grammar, using mutation operators.

Mutants are *valid programs* (not tests) which ought to behave differently from the ground string.

Our task, in mutation testing, is to create tests which distinguish mutants from originals.

Example. Given the ground string `x = a + b`, we might create mutants `x = a - b`, `x = a * b`, etc. A possible original on the left and a mutant on the right:

```
int foo(int x, int y) { // original     int foo(int x, int y) { // mutant
  if (x > 5) return x + y;                if (x > 5) return x - y;
  else return x;                          else return x;
}                                       }
```

Once we find a test case that kills a mutant, we can forget the mutant and keep the test case. The mutant is then *dead*.

Uninteresting Mutants. Three kinds of mutants are uninteresting:

- *stillborn*: such mutants cannot compile (or immediately crash);
- *trivial*: killed by almost any test case;
- *equivalent*: indistinguishable from original program.

The usual application of program-based mutation is to individual statements in unit-level (per-method) testing.

Mutation Example. Here are some mutants.

```
// original
int min(int a, int b) {
  int minVal;
  minVal = a;

  if (b < a) {


    minVal = b;



  }
  return minVal;
}
```

```
// with mutants
int min(int a, int b) {
  int minVal;
  minVal = a;
  minVal = b;                // Δ 1
  if (b < a) {
  if (b > a) {               // Δ 2
  if (b < minVal) {          // Δ 3
    minVal = b;
    BOMB();                  // Δ 4
    minVal = a;              // Δ 5
    minVal = failOnZero(b);  // Δ 6
  }
  return minVal;
}
```

Conceptually we've shown 6 programs, but we display them together for convenience.

Goals of mutation testing:

1. mimic (and hence test for) typical mistakes;
2. encode knowledge about specific kinds of effective tests in practice, e.g. statement coverage ($\Delta 4$), checking for 0 values ($\Delta 6$).

Reiterating the process for using mutation testing (see picture at end):

- *Goal:* kill mutants
- *Desired Side Effect:* good tests which kill the mutants.

These tests will help find faults (we hope). We find these tests by intuition and analysis.

Weak and strong mutants

So far we've talked about requiring differences in the *output* for mutants. We call such mutants **strong mutants**. We can relax this by only requiring changes in the *state*, which we'll call **weak mutants**.

In other words,

- *strong mutation*: fault must be *reachable*, *infect* state, and **propagate** to output.
- *weak mutation*: a fault which kills a mutant need only be *reachable* and *infect state*.

The book claims that experiments show that weak and strong mutation require almost the same number of tests to satisfy them.

We restate the definition of killing mutants which we've seen before:

Definition 1 Strongly killing mutants*: Given a mutant m for a program P and a test t, t is said to* strongly kill *m iff the output of t on P is different from the output of t on m.*

Criterion 1 Strong Mutation Coverage *(SMC). For each mutant m, TR contains a test which strongly kills m.*

What does this criterion not say?

Definition 2 Weakly killing mutants*: Given a mutant m that modifies a source location ℓ in program P and a test t, t is said to* weakly kill *m iff the* state *of the execution of P on t is different from the* state *of the execution of m on t, immediately after some execution of ℓ.*

How does this criterion differ from what we've tested recently in unit tests?

Criterion 2 Weak Mutation Coverage *(WMC). For each mutant m, TR contains a test which weakly kills m.*

Let's consider mutant $\Delta 1$ from before, i.e. we change `minVal = a` to `minVal = b`. In this case:

- reachability: unavoidable;
- infection: need $b \neq a$;
- propagation: wrong `minVal` needs to return to the caller; that is, we can't execute the body of the `if` statement, so we need $b > a$.

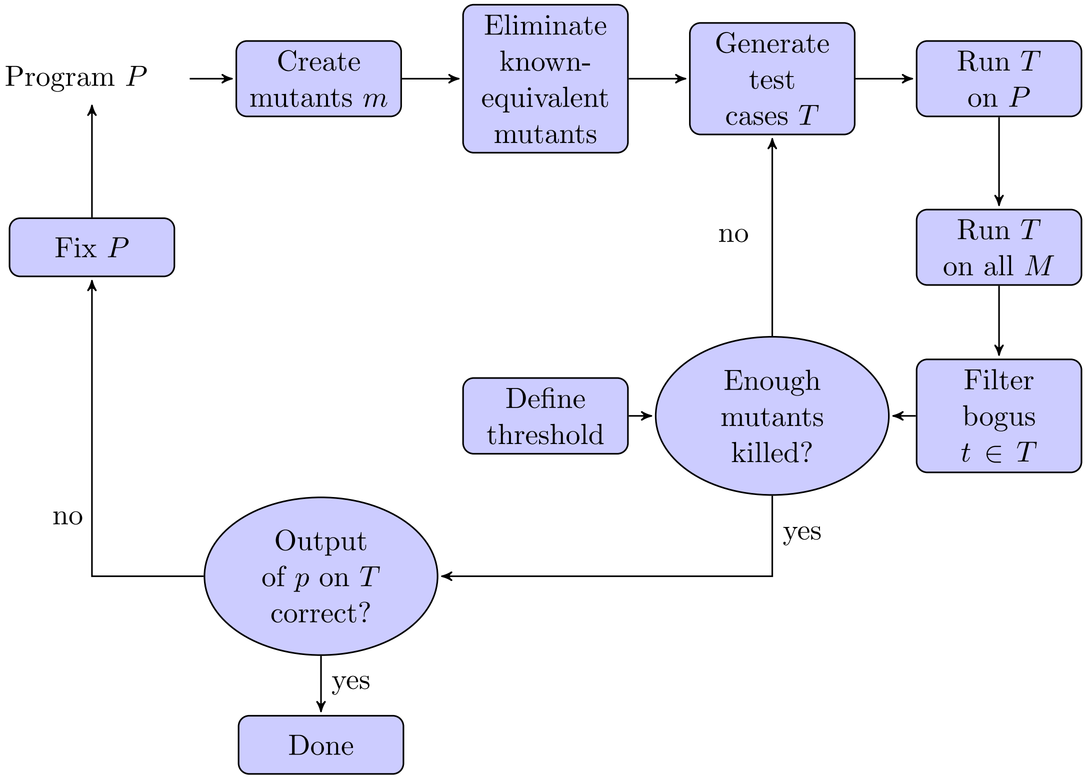
A test case for strong mutation is therefore $a = 5, b = 7$ (return value = ␣, expected ␣), and for weak mutation $a = 7, b = 5$ (return value = ␣, expected ␣).

Now consider mutant $\Delta 3$, which replaces `b < a` with `b < minVal`. This mutant is an equivalent mutant, since `a = minVal`. (The infection condition boils down to "false".)

Equivalence testing is, in its full generality, undecidable, but we can always estimate.

Testing Programs with Mutation

Here's a possible workflow for actually performing mutation testing.


Program P
Create mutants m
Eliminate known-equivalent mutants
Generate test cases T
Run T on P
Run T on all M
Filter bogus $t \in T$
Enough mutants killed?
no
Define threshold
yes
Output of p on T correct?
no
Fix P
yes
Done