Software Testing, Quality Assurance and Maintenance	Winter 2015
Lecture 18 — February 13, 2015	
Patrick Lam	*version 1*

Mutation Testing

The second major way to use grammars in testing is mutation testing. Strings are usually programs, but could also be inputs, especially for testing based on invalid strings.

Definition 1 *Ground string: a (valid) string belonging to the language of the grammar.*

Definition 2 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 3 *Mutant: the result of one application of a mutation operator to a ground string.*

Mutants may be generated either by modifying existing strings or by changing a string while it is being generated.

It is generally difficult to find good mutation operators. One example of a bad mutation operator might be to change all predicates to "true".

Note that mutation is hard to apply by hand, and automation is complicated. The testing community generally considers mutation to be a "gold standard" that serves as a benchmark against which to compare other testing criteria against.

More on ground strings. Mutation manipulates ground strings to produce variants on these strings. Here are some examples:

- the program that we are testing; or
- valid inputs to a program.

If we are testing invalid inputs, we might not care about ground strings.

Credit card number examples. Valid strings:

Invalid strings:

We can also create mutants by applying a mutation operator during generation, which is useful when you don't need the ground string.

NB: There are many ways for a string to be invalid but still be generated by the grammar.

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *More work, but can subsume other criteria.*

Killing Mutants

Generate a mutant m for an original ground string m_0.

Definition 4 *Test case t* kills *m if running t on m gives different output than running t on m_0.*

(The book uses a derivation D rather than a ground string m_0.)

We can also define a mutation score, which is the percentage of mutants killed.

I'm going to list a trio of coverage criteria from the book. I don't think that the mutation coverage criteria are ever used. If one were trying to use mutation testing, one would measure the effectiveness of a test suite (the mutation score) and to keep adding tests until reaching a desired mutation score.

Criterion 1 Mutation Coverage *(MC). For each mutant m, TR contains requirement "kill m".*

(This definition does not describe the set of mutants required.)

Criterion 2 Mutation Operator Coverage *(MOC). For each mutation operator* `op`*, TR contains requirement to create a mutated string m derived using* `op`*.*

Criterion 3 Mutation Production Coverage *(MPC). For each mutation operator* `op` *and each production p that* `op` *can be applied to, TR contains requirement to create a mutated string from p.*

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
  if (x > 5) return x + y;
  else return x;
}
```

```
int foo(int x, int y) { // mutant
  if (x > 5) return x - y;
  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 (Δ4), checking for 0 values (Δ6).

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.