Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 15 — February 6, 2017

Patrick Lam *version 0*

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*.

Supposedly, 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 $b > a$.

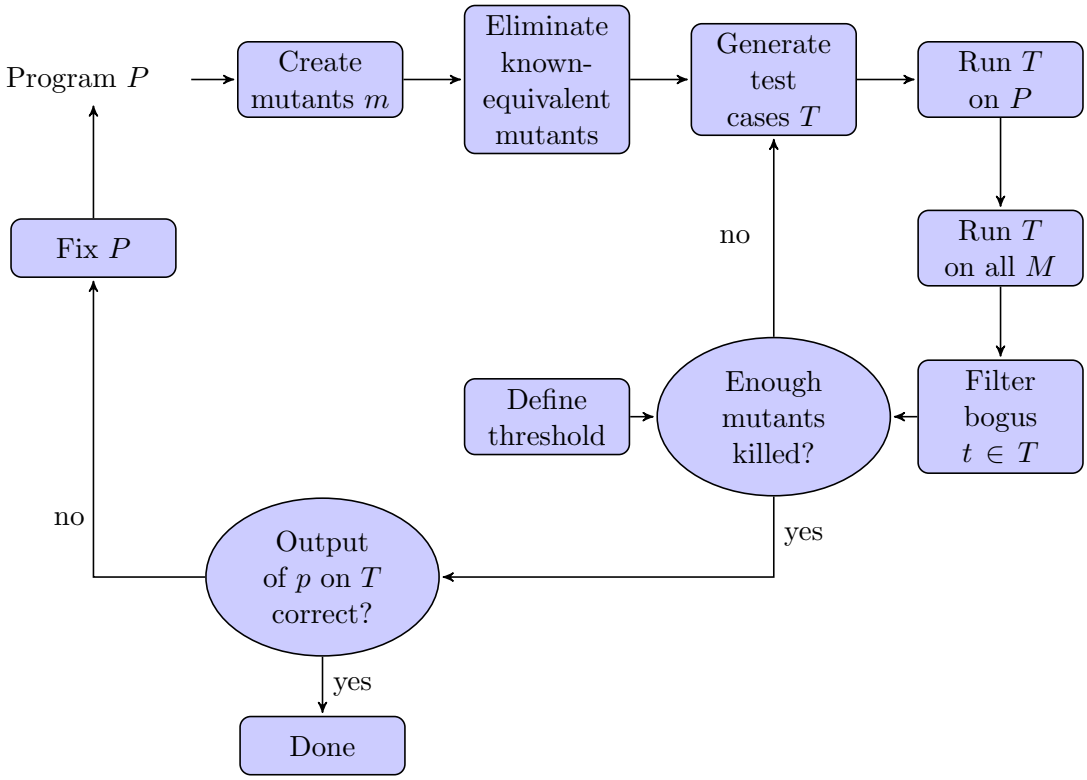
A test case for strong mutation is therefore $a = 5, b = 7$ (return value = ␣, expected ␣), and for weak mutation $a = 7, b = 5$ (return value = ␣, expected ␣).

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.



Mutation Operators

We'll define a number of mutation operators, although precise definitions are specific to a language of interest. Typical mutation operators will encode typical programmer mistakes, e.g. by changing

relational operators or variable references; or common testing heuristics, e.g. fail on zero. Some mutation operators are better than others.

The book contains a more exhaustive list of mutation operators. How many (intraprocedural) mutation operators can you invent for the following code?

```
int mutationTest(int a, b) {
  int x = 3 * a, y;
  if (m > n) {
    y = -n;
  }
  else if (!(a > -b)) {
    x = a * b;
  }
  return x;
}
```

Integration Mutation. We can go beyond mutating method bodies by also mutating interfaces between methods, e.g.

- change calling method by changing actual parameter values;
- change calling method by changing callee; or
- change callee by changing inputs and outputs.

```
class M {
  int f, g;

  void c(int x) {
    foo (x, g);
    bar (3, x);
  }

  int foo(int a, int b) {
    return a + b * f;
  }

  int bar(int a, int b) {
    return a * b;
  }
}
```

[Absolute value insertion, operator replacement, scalar variable replacement, statement replacement with crash statements...]

Mutation for OO Programs. One can also use some operators specific to object-oriented programs. Most obviously, one can modify the object on which field accesses and method calls occur.

```
class A {
  public int x;
  Object f;
  Square s;

  void m() {
    int x;
    f = new Object();
    this.x = 5;
  }
}

class B extends A {
  int x;
}
```

Exercise. Come up with a test case to kill each of these types of mutants.

- **ABS**: Absolute Value Insertion
  `x = 3 * a` $\Longrightarrow$ `x = 3 * abs(a)`, `x = 3 * -abs(a)`, `x = 3 * failOnZero(a)`;
- **ROR**: Relational Operator Replacement
  `if (m > n)` $\Longrightarrow$ `if (m >= n)`, `if (m < n)`, `if (m <= n)`, `if (m == n)`, `if (m != n)`, `if (false)`, `if (true)`
- **UOD**: Unary Operator Deletion
  `if (!(a > -b))` $\Longrightarrow$ `if (a > -b)`, `if (!(a > b))`

Summary of Syntax-Based Testing.

	Program-based	Input Space
Grammar	Programming language	Input languages / XML
Summary	Mutates programs / tests integration	Input space testing
Use Ground String?	Yes (compare outputs)	No
Use Valid Strings Only?	Yes (mutants must compile)	Invalid only
Tests	Mutants are not tests	Mutants are tests
Killing	Generate tests by killing	Not applicable

Notes:

- Program-based testing has notion of strong and weak mutants; applied exhaustively, program-based testing subsumes many other techniques.
- Sometimes we mutate the grammar, not strings, and get tests from the mutated grammar.

Tool support. PIT Mutation testing tool: `http://pitest.org`. Mutates your program, reruns your test suite, tells you how it went.