Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 16 — February 25, 2019

Patrick Lam *version 1*

Recall that the goals of mutation testing are:

1. mimic (and hence test for) typical mistakes;
2. encode knowledge about specific kinds of effective tests in practice, e.g. statement coverage (Δ4 from Lecture 15), checking for 0 values (Δ6).

Reiterating the process for using mutation testing:

- *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.

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.

You can find a more exhaustive list of mutation operators in the PIT documentation:

`http://pitest.org/quickstart/mutators/`

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/Fuzzing
Grammar	Programming language	Input languages / XML
Summary	Mutates programs / tests integration	Input space testing
Use Ground String?	Yes (compare outputs)	Sometimes
Use Valid Strings Only?	Yes (mutants must compile)	No
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 could subsume 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. You need to distinguish equivalent vs. not-killed.