Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 15 — February 13, 2019

Patrick Lam *version 1*

Mutation Testing

The second major way to use grammars in testing is mutation testing. Let's start with an example. Here is a program, along with some mutations to the program, which are derived using the grammar.

```
// 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. You'll find code in `live-coding/minval.c`.

We're generating mutants m for the original program m_0.

Definition 1 *Test case t* kills *m if running t on m gives different output than running t on m_0.*

We use these mutants to evaluate test suites. Here's a simple test suite. I'm abstractly representing a JUnit test case by a tuple; assume that it calls `min()` and asserts on the return value. You can fill out this table.

	Δ 1	Δ 2	Δ 3	Δ 4	Δ 5	Δ 6
$\langle a = 0, b = 1, \exp = 0\rangle$	kill		–			
$\langle a = 1, b = 0, \exp = 0\rangle$	–		–			
$\langle a = 1, b = 1, \exp = 1\rangle$			–			
$\langle a = 1, b = 349, \exp = 1\rangle$			–			

Note that, for instance, Δ 3 is not killable; if you look at the modification, you can see that it is equivalent to the original.

The idea is to use mutation testing to evaluate test suite quality/improve test suites. Good test suites ought to be effective at killing mutants.

General Concepts

Mutation testing relies on two hypotheses, summarized from [DPHG$^+$18].

The *Competent Programmer Hypothesis* posits that programmers usually are almost right. There may be "subtle, low-level faults". Mutation testing introduces faults that are similar to such faults. (We can think of exceptions to this hypothesis—if the code isn't tested, for instance; or, if the code was written to the wrong requirements.)

The *Coupling Effect Hypothesis* posits that complex faults are the result of simple faults combining; hence, detecting all simple faults will detect many complex faults.

If we accept these hypotheses, then test suites that are good at ensuring program quality are also good at killing mutants.

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. For example, consider a test suite T which ensures statement coverage. What can mutation testing say about how good T is?

Mutation testing proceeds as follows.

1. *Generate mutants:* apply mutation operators to the program to get a set of mutants M.
2. *Execute mutants:* execute the test suite on each mutant and collect suite pass/fail results.
3. *Classify:* interpret the results as either killing each mutant or not; a failed test suite execution implies a killed mutant.

Although you could generate mutants by hand, typically you would tend to use a tool which parses the input program, applies a mutation operator, and then unparses back to source code, which is then recompiled.

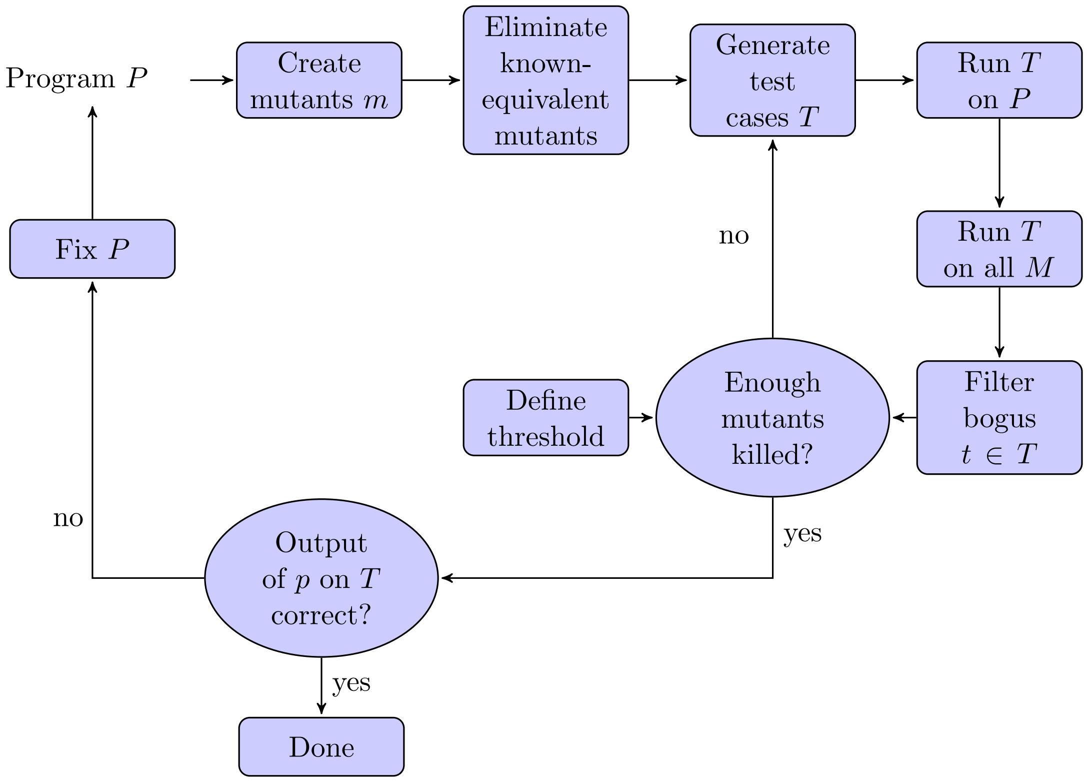
Executing the mutants can be computationally expensive, since you have to run the entire test suite on each of the mutants. This is a good time to use all the compute infrastructure available to you.

Generating and executing are computationally expensive, but classifying is worse, because it requires manual analysis. In particular, a not-killed result could be due to an equivalent mutant (like Δ 3 above); compilers can help, but the problem is fundamentally undecidable. Alternately, not-killed could be because the test suite isn't good enough. It's up to you to distinguish these cases.

You would normally want to then craft new test cases to kill the non-equivalent mutants that you found.

Testing Programs with Mutation

Here's a picture that illustrates a variant of the above workflow.



Generating Mutants

Now let's see how to generate mutants; this ties in to the grammar-based material I talked about last time. For mutation testing, strings will always be programs.

Definition 2 *Ground string: a (valid) string belonging to the language of the grammar (i.e. a programming language grammar).*

Definition 3 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 4 *Mutant: the result of one application of a mutation operator to a ground string.*

The workflow is to parse the ground string (original program), apply a mutation operator, and then unparse.

It is generally difficult to find good mutation operators. One example of a bad mutation operator might be to change all boolean expressions to "true". Fortunately, the research shows that you don't need many mutation operators—the right 5 will do fine.

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *Too much work; choose randomly.*

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

To use mutation testing for generating test cases, one would measure the effectiveness of a test suite (the mutation score), and keep adding tests until reaching a desired mutation score.

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.

Let's consider mutant $\Delta 1$ from above, 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.

Program Based Grammars

The usual way to use mutation testing for generating test cases 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.

Mutation testing looks for 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;
}
```

In this example, the test case $\langle 6, 2 \rangle$ will kill the mutant, since it returns 8 for the original and 4 for the mutant, while the case $\langle 6, 0 \rangle$ will not kill the mutant, since it returns 6 in both cases.

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.

References

[DPHG+18] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. In *IEEE Transactions on Reliability*, pages 1406–1419, 2018.