Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 4 — January 15, 2025

Patrick Lam *version 1*

Before we continue with operational semantics, I'm going to talk about mutation analysis.

Remember, we have this problem: when do we stop testing? How do we know that our test suite is good enough?

We talked about coverage, but we also saw how it's far from perfect.

Mutation Analysis

What if we checked whether our test suite can detect changes to the program? If it can, then it is detecting something, at least.

A *mutant* is a version of the program that has been somehow modified (in one place, usually an operator or identifier switch.)

Remember that tests contain three phases: arrange, act, assert. Running on a modified program, the test suite might trigger errors during the "act" phase. Or, it can detect where the program is giving a different output during the "assert" phase.

Another way of looking at this is that we are fuzzing the test suite (by modifying the program) and checking that the test suite is doing what it should (that is, finding broken programs).

Example Here's a program and some mutants. We can use the language grammar to create the 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. You'll find code in `code/L04/minval.c` in the `pdfs` repo.

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
$\langle a = 0, b = 1, \text{exp} = 0 \rangle$	kill		–			
$\langle a = 1, b = 0, \text{exp} = 0 \rangle$	–		–			
$\langle a = 1, b = 1, \text{exp} = 1 \rangle$			–			
$\langle a = 1, b = 349, \text{exp} = 1 \rangle$			–			

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 analysis to evaluate test suite quality/improve test suites. Good test suites ought to be effective at killing mutants.

General Concepts

Mutation analysis 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 analysis 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 analysis say about how good T is?

Mutation analysis 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.

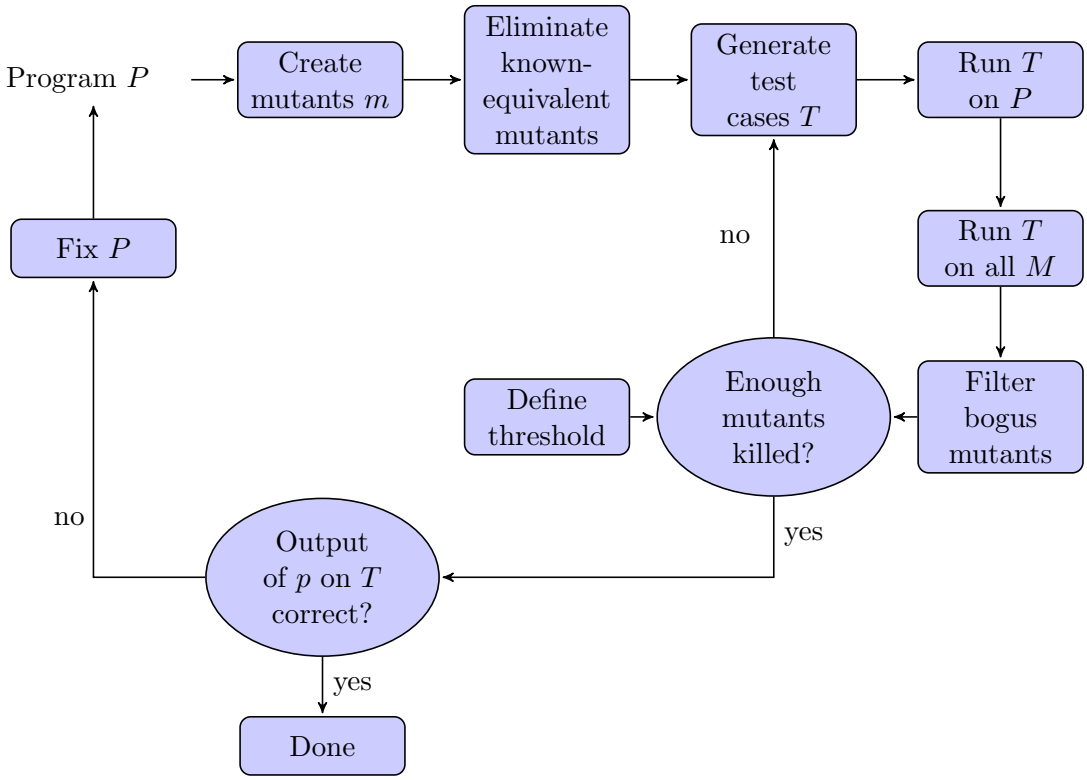
Executing the mutants can be computationally expensive, since you have to run the entire test suite on each of the mutants. This is a good time to use all the compute infrastructure available to you.

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. On the next page, "bogus mutants" denotes equivalent, stillborn and trivial mutants.

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 is similar to grammar-based fuzzing (but of the code). For mutation analysis, 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 analysis 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 = $\lrcorner$, expected $\lrcorner$), and for weak mutation $a = 7, b = 5$ (return value = $\lrcorner$, expected $\lrcorner$).

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.

Another 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*.

The other thing that can happen when you are running a test case on a mutant is that it loops indefinitely. You'd want to use a timeout when running testcases, and then you have a timeout failure, which you can presumably use to distinguish the mutant from the original.

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.

Implementing Mutation Analysis

Recall that the goals of mutation analysis are to evaluate test suites by:

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

We'll see 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.

One tool you can use for mutation analysis is PIT. It mutates your program, reruns your test suite, tells you how it went. You need to distinguish equivalent vs. not-killed. You can find a more exhaustive list of mutation operators in the PIT documentation:

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

How many mutation operators can you invent for the following code?

```
1  int mutationTest(int a, b) {
2    int x = 3 * a, y;
3    if (m > n) {
4      y = -n;
5    }
6    else if (!(a > -b)) {
7      x = a * b;
8    }
9    return x;
10 }
```

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

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))`

Making Mutation Analysis Practical

There is a paper about how to use mutation analysis in practice: Petrović et al. "Practical Mutation Testing at Scale: A View from Google".

Problems with mutation testing (aka mutation analysis):

- too many mutants!
- too many unproductive mutants!

To help with the "too many mutants" problem, solution #1 is to (1) mutate only changed (and covered) lines of code; and (2) show surviving mutants during code review.

They used 5 mutation operators:

1. arithmetic operator replacement

2. logical connector replacement
3. unary operator insertion
4. relational operator replacement
5. statement block removal

Bad (unproductive) mutants are a problem, and they describe a number of heuristics to avoid bad mutants. To avoid equivalent mutants:

- detect some clearly-equivalent mutants (`c.size() == 0, c.size() <= 0`);
- detect removal of caching;
- detect time-related calls (e.g. sleep()).

And, some mutants are killable but shouldn't be, like logging calls and `mkdir`. The test suite does not need to detect changes in logging output.

So, to apply mutation analysis to a proposed change set, they:

- generate mutants for each changed line of code (max of 7x # of files);
- filter out according to heuristics; and,
- probabilistically select mutants—choose those that are like past good mutants.

Developer should update their test sets accordingly.

Is Mutation Analysis Any Good?

We've talked about mutation analysis as a metric for evaluating test suites and making sure that test suites exercise the system under test sufficiently. The problem with metrics is that they can be gamed, or that they might measure not quite the right thing. When using metrics, it's critical to keep in mind what the right thing is. In this case, the right thing is the fault detection power of a test suite.

Researchers (including some at Waterloo) set out to determine just that. They carried out a study, using realistic code, where they isolated a number of bugs, and evaluated whether or not there exists a correlation between real fault detection and mutant detection.

Summary. The answer is **yes**: test suites that kill more mutants are also better at finding real bugs. The researchers also investigated when mutation analysis fell short—they enumerated types of bugs that mutation analysis, as currently practiced, would not detect.

Methodology. The authors used 5 open-source projects. They isolated a total of 357 reproducible faults in these projects using the projects' bug reporting systems and source control repositories. They they generated 230,000 mutants using the Major mutation framework and investigated the ability of both developer-written test suites and automatically-generated test suites (EvoSuite, Randoop, JCrasher) to detect the 357 faults.

For each fault, the authors started with a developer-written test suite T_{bug} that did not detect the fault. Then, using the source repository, they extracted a developer-written test that detects the fault. Call this suite T_{fix}. Does T_{fix} detect more mutants than T_{bug}? If so, then we can conclude that the mutant behaves like a bug.

Results. The authors found that Major-generated mutation tests could detect 73% of the faults. In other words, for 73% of faults, some mutant will be killed by a test that also detects the fault. Increasing mutation coverage thus also increases the likelihood of finding faults.

The analogous numbers for branch coverage and statement coverage are, respectively, 50% and 40%. Specifically: the 357 tests that find faults only increase branch coverage 50% of the time, and they only increase statement coverage 40% of the time. So: improving your test suite often doesn't get rewarded with a better statement coverage score, and half the time doesn't result in a better branch coverage score. Conversely, improving statement coverage doesn't help find more bugs because you're already reaching the fault, but you aren't sensitive to the erroneous state.

The authors also looked at the 27% of remaining faults that are not found by mutants. For 10% of these, better mutation operators could have helped. The remaining 17% were not suitable for mutation analysis: they were fixed by e.g. algorithmic improvements or code deletion.

Reference. René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In *Foundations of Software Engineering* 2014. pp654–665. `http://www.linozemtseva.com/research/2014/fse/mutant_validity/`

What Does (Graph) Coverage Buy You?

We've talked about graph coverage, notably statement coverage (node coverage) and branch coverage (edge coverage). They're popular because they are easy to compute. But, are they any good? Reid Holmes (a former Waterloo CS prof) and his then-PhD student Laura Inozemtseva set out to answer that question.

Answer. Coverage does not correlate with high quality when it comes to test suites. Specifically: test suites that are larger are better because they are larger, not because they have higher coverage.

Methodology. The authors picked 5 large programs and created test suites for these programs by taking random subsets of the developer-written test suites. They measured coverage and they measured effectiveness (defined as % mutants detected; we've seen that detecting mutants is good, above).

Result. In more technical terms: after controlling for suite size, coverage is not strongly correlated with effectiveness.

Furthermore, stronger coverage (e.g. branch vs statement, logic vs branch) doesn't buy you better test suites.

Discussion. So why are we making you learn about coverage? Well, it's what's out there, so you should know about it. But be aware of its limitations.

Plus: if you are not covering some program element, then you obviously get no information about the behaviour of that element. Low coverage is bad. But high coverage is not necessarily good.

Reference. Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness." In *International Conference on Software Engineering* 2014. pp435–445. `http://www.linozemtseva.com/research/2014/icse/coverage/`

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.