Software Testing, Quality Assurance & Maintenance—Lecture 4

Patrick Lam
University of Waterloo

January 16, 2026

Part I

When to stop?
Idea 2: Mutation Analysis

How many tests?

Do you have enough tests? How do you know?

Let's fuzz the test suite (more fuzzing later).
How? Modifying the program and seeing if the test suite notices.

Mutants

A mutant is a modified version of the program being tested.

Usually we change an operator or identifier:

$$x + 5 \Rightarrow x - 5$$

Killing Mutants

The test suite should fail on the mutant.
Then the mutant is killed.

Remember: arrange, act, assert.
Mutant might trigger errors during act;
or it may detect different output during assert.

Example Mutants

Use language grammar to create mutants (code/L04/minval.c).

```
// 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;
}
```

Testing on the mutants

Here's a test suite. How do the mutants do?

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

Observe: Δ3 not killable.

Key idea for Mutation Analysis

Idea: use mutation analysis to evaluate test suite quality/improve test suites.

Good test suites ought to be effective at killing mutants.

Why should this work? (1/2)

Competent Programmer Hypothesis:

programmers usually are almost right,
except for “subtle, low-level faults”.

Mutation analysis tries to mimic this.
(Exceptions?)

Why should this work? (2/2)

Coupling Effect Hypothesis:
complex faults are the result of
simple faults combining.

Hence, detecting all simple faults will detect many complex faults.

Implication: test suites that are good at ensuring program quality also good at killing mutants.

Mutation analysis in context

Hard to apply by hand, and automation is complicated.

Mutation is a “gold standard”
against which to test other testing criteria.

Consider test suite T which ensures statement coverage.
What does mutation analysis say about T?

Part II

Using Mutation Analysis

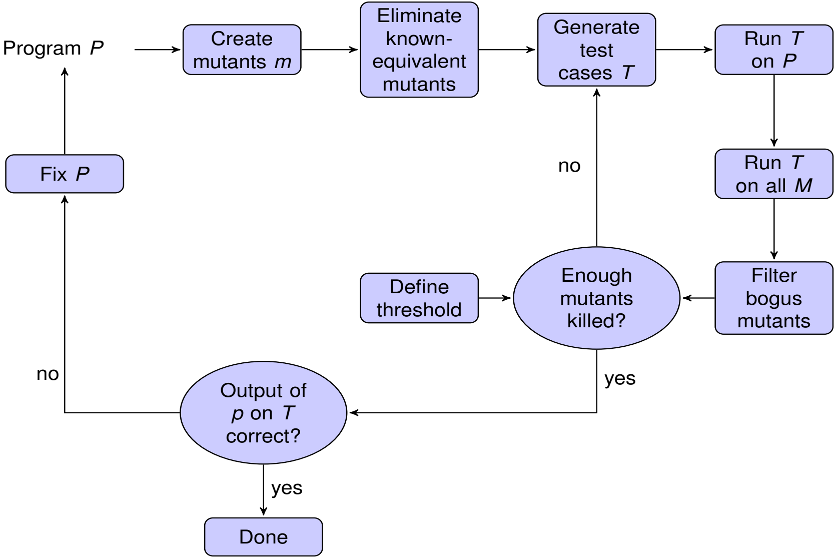
Using Mutation Analysis

Three steps:

1. Generate mutants
   (usually with a tool)
2. Execute mutants
   (computationally expensive)
3. Classify
   (manual)

Then, create new test cases to kill remaining mutants.

Mutation Analysis Diagram



Generating Mutants

We said: mutation analysis is like fuzzing the test suite.
Let's do that.

Ground string: A valid program (or fragment) that conforms to its grammar.

Mutation operator: A rule that specifies syntactic variations of strings.

Mutant: The result of one application of a mutation operator to a ground string.

Workflow: parse the ground string (original program), apply a mutation operator, unparse.

Mutation Operators

Hard to get good mutation operators.

The Δs we saw applied operators:
change identifiers, change operators, insert BOMB, insert failOnZero.

Perhaps a bad operator:
"change all boolean expressions to true".

Research says: maybe (the right) 5 operators is enough.

When to Apply Mutation

- Once at a time, to a given ground string.
- Choose where to apply the operator randomly.

Killing Mutants

One could define a mutation score
(% of mutants killed),
add tests until mutation score high enough.

HOWTO kill mutants

So far: need differences in *output* (including assertion failures).

Could relax to require changes in just the *state*.

- strong mutation: fault must be reachable, infect state, and *propagate* to output.
- weak mutation: fault must be reachable and infect state.

Supposedly: about the same in practice.

Example 1

```
// with mutants
int min(int a, int b)
{
  int minVal;
  minVal = a;
  minVal = b;                    // Δ 1
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

Reachability: unavoidable; infection: $b \neq a$;
propagation: can't execute then case, so need $b > a$.

Strong mutation test case: $a = 5, b = 7$;
weak mutation: $a = 7, b = 5$.

Example 2

```
// with mutants
int min(int a, int b)
{
  int minVal;
  minVal = a;
  if (b < a) {
  if (b < minVal) {          // Δ 3
    minVal = b;
  }
  return minVal;
}
```

This is an equivalent mutant, since `a = minVal;` infection condition is “false”.

(Equivalence testing is generally undecidable.)

Example 3

```
// original
int foo(int x, int y) {
if (x > 5)
  return x + y;
else
  return x;
}
```

```
// mutant
int foo(int x, int y) {
if (x > 5)
  return x - y;
else
  return x;
}
```

Test case $\langle 6, 2\rangle$ kills the mutant, while $\langle 6, 0\rangle$ will not.

Once we find a mutant-killing test case, forget the mutant and keep the test case (like fuzzing).

Uninteresting Mutants

Sometimes the mutant will loop indefinitely. Use a timeout.

Other uninteresting mutants:

- stillborn: can't compile
- trivial: killed by any input
- equivalent

Implementing Mutation Analysis

What mutation analysis does:

- mimic (and hence test for) typical mistakes;
- encode knowledge re: specific kinds of effective tests:
  e.g. statement coverage, checking for 0.

Choosing the right mutation operators is key.

Mutation Analysis Tool

PIT:

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

Mutates your program, reruns your test suite, tells you how it went.

Up to you: distinguish equivalent, not-killed.

Exercise: Find Mutation Operators

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

Exercise: Killing Mutants

For the `mutationTest` code on the previous slide, find a test case to kill each of these types of mutants:

- **ABS**: Absolute Value Insertion

  `x = 3 * a` $\Longrightarrow$ `x = 3 * abs(a)`

- **ROR**: Relational Operator Replacement

  `if (m > n)` $\Longrightarrow$ `if (m >= n)`

- **UOD**: Unary Operator Deletion

  `if (!(a > -b))` $\Longrightarrow$ `if (a > -b)`

Making Mutation Analysis Practical

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!

Practicality via Incrementality

Solution #1:
mutate only changed (and covered) lines of code;
show surviving mutants during code review.

Mutation operators used

arithmetic operator replacement
logical connector replacement
unary operator insertion
relational operator replacement
statement block removal

Heuristics to avoid bad mutants: equivalent

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

Heuristics to avoid bad mutants: unproductive

Some mutants are killable but shouldn't be:

- logging calls
- mkdir

The test suite does not need to detect changes in logging output.

Applying mutation testing

On a proposed change set:

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

Developer should update their test sets accordingly.

Part III

Is Mutation Analysis Any Good?

Paper: Are Mutants a Valid Substitute for Real Faults in Software Testing?

Answer: Yes! Test suites that kill more mutants are also better at finding real bugs.

Also identified types of bugs that then-current mutation analysis would not detect.

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/`

Mutation Effectiveness: Methodology

5 open-source projects.
357 faults from these projects.
230,000 mutants (using Major mutation framework).

Can developer-written and automatically-generated test suites detect these faults?

Mutation Effectiveness: Methodology

For each fault:

- developer-written suite T_{bug} that did not detect the fault;
- extracted from the source repo a developer-written test that detects the fault, add it to T_{bug} to obtain T_{fix}.

Question: Does T_{fix} detect more mutants than T_{bug}?

If so, then the mutant behaves like a bug.

Results

Major-generated mutation could detect 73% of the faults:
that is, for 73% of faults, some mutant will be killed by a test that also detects the fault.

So: ↑ mutation coverage also
↑ likelihood of finding faults.

Results: Branch and Statement Coverage

Analogous numbers for branch and statement coverage: 50%, 40%.

The 357 tests that find faults only increase branch coverage 50% of the time.

Improving your test suite doesn't get rewarded with a better coverage score.

Conversely, improving statement coverage doesn't help find more bugs—you're not sensitive to erroneous state.

Results: Faults not found by mutants

27% of remaining faults were not found by mutants.

- For 10%, better mutation operators could have helped.
- Remaining 17% not suitable for mutation analysis (algorithmic improvements or code deletion).

Another Paper: Coverage is Not Strongly Correlated with Test Suite Effectiveness

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/`

Coverage paper summary

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

5 large programs.
Test suites: random subsets of developer-written suites.

Measured coverage & effectiveness (% mutants detected).

Result

After controlling for suite size,
coverage not strongly correlated
with effectiveness.

Stronger coverage (e.g. branch vs statement)
doesn't buy you better test suites.