04834580 Software Engineering (Honor Track) 2025-26

Advanced Testing Techniques

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Recap: Coverage Criteria

北京大学

Code coverage measures how thoroughly a test suite exercises the program:

- **Statement coverage** — fraction of statements executed.
- **Branch coverage** — fraction of True/False edges taken.
- **Path coverage** — fraction of execution paths taken.

Coverage is widely used as a quality target: "80% branch coverage" is a common policy.

But **does high coverage imply that tests are good at finding bugs?**

Coverage Without Assertions

Consider a buggy implementation and a passing test:

```
def abs_value(x):
    if x < 0:
        x = -x       # bug: should also handle x == INT_MIN
    return x

def test_abs_value():
    abs_value(-5)
    abs_value(5)
    abs_value(0)
```

This test achieves **100% statement and branch coverage**.
But it has **no assertions** — the function could return anything and the test would still pass. Coverage measures *execution*, not *checking*.

Coverage With Weak Assertions


```python
def max_of(a, b):
    return a   # bug: ignores b

def test_max_of():
    result = max_of(3, 1)
    assert result is not None  # too weak
```

Again 100% coverage, the assertion does fire, but the test fails to detect a clearly wrong implementation.

Coverage tells us whether code was *reached*; it cannot tell us whether the test would *notice* a fault that was reached.

Empirical Evidence

> *"Coverage is not strongly correlated with test suite effectiveness."*
>
> — *Inozemtseva and Holmes, 2014 [1]*

Large-scale study of real Java projects:

- Suites with very different bug-detection power can have nearly identical coverage.
- Suite *size* correlates with effectiveness more strongly than coverage does.
- Reaching code is a *necessary* but not *sufficient* condition for detecting a bug there.

We need a metric that measures whether tests would actually **detect** bugs — not just touch the code.

Definition (Mutation Testing [2])

A technique to assess the quality of a given test set, where faults are deliberately seeded into the original program, by a simple syntactic change, to create a set of faulty programs called **mutants**, each containing a different syntactic change. These mutants are executed against the input test set to see if the seeded faults can be detected.

Originated in the late 1970s with DeMillo, Lipton and Sayward [3] and Budd [4].

Two Foundational Hypotheses

北京大学

Mutation testing rests on two assumptions about real programs:

- **Competent Programmer Hypothesis**: programmers write code that is *close to correct*. Real bugs differ from the correct program by small syntactic changes.
- **Coupling Effect**: a test suite that detects *simple* faults will, with high probability, also detect the *complex* faults that combinations of simple faults produce.

Together they justify the strategy of generating many tiny mutants: if the suite kills the small faults, it likely catches the realistic ones too.

Killed and Live Mutants

Run the test suite T on every mutant m of program P:

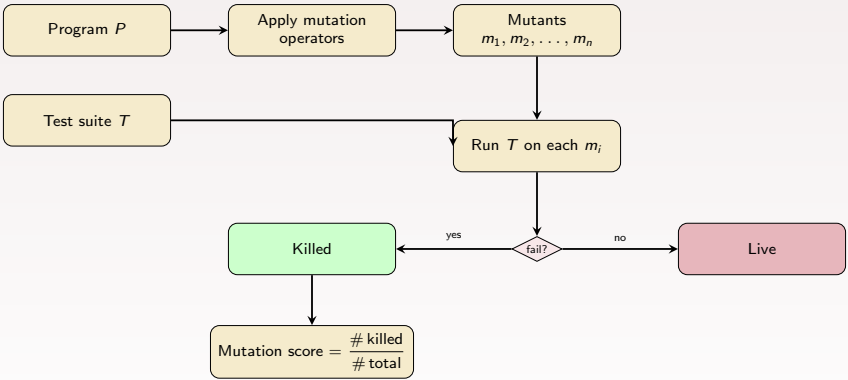
- **Killed** mutant: at least one test in T that passed on P *fails* on m. The suite *detected* the seeded fault.
- **Live** (surviving) mutant: every test still passes. The suite did not detect the change.

Definition (Mutation Score)

$$\text{Mutation Score} = \frac{\#\,\text{killed mutants}}{\#\,\text{total non-equivalent mutants}}$$

A live mutant is a concrete suggestion: "write a test that distinguishes this mutant from the original."

The Mutation Testing Workflow



A Worked Example

Original program:

```
def max_of(a, b):
    if a > b:
        return a
    return b
```

Test suite:

```
def test_max():
    assert max_of(3, 1) == 3
    assert max_of(1, 3) == 3
```

Both tests pass on the original. Now generate mutants by perturbing single tokens.

Mutants of `max_of`

Three mutants, each a one-token change:

```
# Mutant m1: > becomes <
def max_of(a, b):
    if a < b: return a
    return b

# Mutant m2: > becomes >=
def max_of(a, b):
    if a >= b: return a
    return b

# Mutant m3: return a becomes return b
def max_of(a, b):
    if a > b: return b
    return b
```

Running the Suite Against the Mutants

北京大学

Mutant	`max_of(3,1)`	`max_of(1,3)`	Status
m_1 (`<`)	1 ($\neq 3$) **fail**	3 pass	killed
m_2 (`>=`)	3 pass	3 pass	live
m_3 (`return b`)	1 ($\neq 3$) **fail**	3 pass	killed

Mutation score: $2/3 \approx 67\%$.

m_2 survived: with our two inputs, the bug `a >= b` is indistinguishable from `a > b`. The remedy is to add a test that *exposes* the difference — e.g. `max_of(3, 3)`, where the equal-case branch matters.

A Test that Kills m_2

Add a tie-breaking case:

```
def test_max_equal():
    assert max_of(3, 3) == 3
```

This test still passes on the original (returns 3 either way), but on m_2 the function returns *a* via the >= branch — which is also 3, so it still passes! m_2 is in fact **equivalent** to the original on this input as well.

More generally, m_2 is *equivalent everywhere*: a >= b routes a == b through the "return a" branch instead of "return b", but *a* and *b* are equal, so the output is the same. **No test can kill m_2.**

Definition (Equivalent Mutant)

A mutant m is **equivalent** to P iff m and P produce the same output on *every* input. No test can kill it.

Equivalent mutants are the central nuisance of mutation testing:

- They artificially lower the mutation score.
- Detecting equivalence is **undecidable** in general (reduces to program equivalence).
- Manual classification is laborious: empirical studies report tens of person-minutes per mutant.

Equivalent Mutant Examples

```java
// Original
for (int i = 0; i < n; i++)
    sum += a[i];

// Mutant: <  ->  !=
for (int i = 0; i != n; i++)
    sum += a[i];
```

When i starts at 0 and increments by 1, `i < n` and `i != n` terminate at the same iteration. The mutant is equivalent.

```python
# Original
x = y * 1      # multiplication by 1

# Mutant: *  ->  /
x = y / 1
```

Dividing by 1 produces the same result for every numeric `y`.

Definition (Mutation Operator)

A rule that, given a program, produces a set of mutants by applying a specific syntactic transformation at every applicable location.

Operator design is language-specific. Classic families (King & Offutt, Mothra):

- **Arithmetic Operator Replacement (AOR)** — `+` → `-`, `*`, `/`, `%`
- **Relational Operator Replacement (ROR)** — `<` → `<=`, `>`, `>=`, `==`, `!=`
- **Conditional Operator Replacement (COR)** — `&&` → `||`
- **Constant Replacement** — `0` → `1`, `-1`; `true` → `false`
- **Statement Deletion (SDL)** — remove a statement
- **Negate Conditional** — `if (c)` → `if (!c)`
- **Boundary** — `<` → `<=`; off-by-one in loops

Operator Examples on a Single Statement

北京大学

Consider:

```
if (a + b > 0 && c < 10) { ... }
```

A handful of operators yields many mutants:

```
if (a - b > 0 && c < 10)  { ... }   // AOR: +  -> -
if (a + b >= 0 && c < 10) { ... }   // ROR: >  -> >=
if (a + b > 1 && c < 10)  { ... }   // CR:  0  -> 1
if (a + b > 0 || c < 10)  { ... }   // COR: && -> ||
if (a + b > 0 && c <= 10) { ... }   // BOR: <  -> <=
if (!(a + b > 0 && c < 10)) { ... } // NEG: c  -> !c
```

A handful of single-token edits typically produces **dozens of mutants per source line**.

The Cost of Mutation Testing

Naive mutation testing has cost roughly

$$O\big(|\text{mutants}| \times |\text{tests}| \times t_{\text{run}}\big).$$

- For a project of L lines of code, $|\text{mutants}| = O(L)$ to $O(L^2)$.
- Each mutant requires recompiling and re-running the suite.
- For a 10 000-line project this can mean millions of test executions.

Most of the research on mutation testing since the 1980s has been about **making it affordable**.

The Test Oracle Problem

Mutation testing assumes we have an oracle: tests already *know* the right answer for each input. But what is the "correct" output of:

- a numerical solver computing sin(1.234) to 12 digits?
- a search engine query?
- a machine-learning classifier on an unseen image?
- a compiler optimization pass?

When the expected output is unknown, expensive to compute, or only loosely defined, the program is called **non-testable** (Weyuker, 1982 [5]). This is the *oracle problem*, and it limits every technique we have seen so far.

Definition (Metamorphic Testing [6])

A technique that alleviates the oracle problem by checking **relations between the outputs** of multiple related executions of the program, instead of checking the output of a single execution against a known answer.

Introduced by Chen, Cheung and Yiu in 1998. Key idea:

Even if we don't know $f(x)$, we may know how $f(x)$ should relate to $f(g(x))$ for some transformation g.

Definition (Metamorphic Relation (MR))

A property over multiple inputs and their corresponding outputs of the program under test: a relation

$$R\big(x_1, \ldots, x_k,\ f(x_1), \ldots, f(x_k)\big)$$

that must hold for every choice of $x_1, \ldots, x_k$ in the input domain.

An MR turns one input into a follow-up test:

1. Pick a **source input** x.
2. Construct a **follow-up input** $x' = g(x)$ via the transformation prescribed by the MR.
3. Run f on both.
4. Check that the relation between $f(x)$ and $f(x')$ holds.

We never need to know $f(x)$ exactly — only how outputs should relate.

Example: Testing `sin`

Suppose we want to test a numerical implementation of `sin(x)`, but we don't have an oracle for arbitrary x. Use trigonometric identities as MRs:

- $\sin(x) = \sin(x + 2\pi)$ *(periodicity)*
- $\sin(-x) = -\sin(x)$ *(odd function)*
- $\sin(\pi - x) = \sin(x)$ *(reflection)*

```
def test_sin_periodicity():
    for x in random_inputs(1000):
        assert abs(sin(x) - sin(x + 2*pi)) < 1e-10
```

We assert that the relation holds, not that the value is any particular number.

Example: Testing a Sort Function

北京大学

Properties (MRs) of `sort`:

- Permuting the input does not change the output.
- Sorting twice yields the same result: `sort(sort(L)) == sort(L)`.
- Length is preserved: `len(sort(L)) == len(L)`.
- Adding v to L then sorting puts v at the position determined by the comparison.

```
def test_sort_permutation_invariance():
    for L in random_lists(1000):
        assert sort(L) == sort(shuffle(L))
```

No reference implementation is required — the relation *is* the oracle.

Beyond Numerics: Search Engines

北京大学

Search engines [7]: tested major search engines using MRs such as

- Adding a constraint to a query should produce a *subset* of results.
- "A OR B" should return at least as many hits as "A".
- Reordering keywords should produce a similar result set.

Found numerous inconsistencies without ever defining what the "correct" result is.

Definition (EMI)

Given a program P and an input I, an **EMI variant** P' is any program that behaves identically to P on I (but may differ on other inputs). For a correct compiler C, the outputs of $C(P)$ and $C(P')$ on input I must agree.

Recipe (Le, Afshari, Su, PLDI 2014):

1. Run P on I under a profiler; record which statements are *not executed* (*dead* for I).
2. Mutate the dead code: delete, insert, or replace it. The result P' still produces the same output on I.
3. Compile both P and P', run on I, compare outputs.
4. A divergence is a **compiler bug**: optimization mishandled the surrounding code.

No oracle needed — the metamorphic relation *is* the test. EMI and successors found 1600+ bugs in GCC and LLVM.

EMI Example

北京大学

Original program P, run with input `x = 5`:

```
int main() {
    int x = 5;
    int y = 0;
    if (x > 0) {
        y = x * 2;              // executed
    } else {
        y = -x;                 // DEAD on this input
        for (int i = 0; i < 10; i++) y += i;  // DEAD
    }
    printf("%d\n", y);          // expects 10
}
```

EMI variant P' — inject arbitrary code into the dead branch:

```
    } else {
        y = -x;
        for (int i = 0; i < 10; i++) y += i;
        int *p = NULL; *p = 42;          // injected: would crash
        while (1) y++;                   // injected: infinite loop
    }
```

Both must print 10 on x=5. If `gcc -O3` produces a different output (or crashes) on P', the optimizer mis-analyzed the live branch.

Example: Invariance Test for an Image Classifier

北京大学

```python
import numpy as np
from scipy.ndimage import shift

def test_translation_invariance(model, images, max_shift=3):
    violations = 0
    for img in images:
        pred = model.predict(img).argmax()
        for dx in range(-max_shift, max_shift + 1):
            for dy in range(-max_shift, max_shift + 1):
                shifted = shift(img, (dy, dx, 0), mode='nearest')
                if model.predict(shifted).argmax() != pred:
                    violations += 1
    return violations / (len(images) * (2*max_shift+1)**2)
```

We never assert "the picture is a cat." We assert *shifting it by 2 pixels does not turn it into a dog*. A high violation rate signals brittleness even on a held-out test set where labels are unknown.

Example: Fairness Test for a Hiring Classifier

北京大学

```python
def test_gender_fairness(model, resumes):
    """Swapping gendered tokens must not change the hire decision."""
    swap = {"he": "she", "she": "he", "his": "her", "her": "his",
            "Mr.": "Ms.", "Ms.": "Mr."}
    flips = 0
    for r in resumes:
        original_decision = model.predict(r)
        swapped = " ".join(swap.get(w, w) for w in r.split())
        if model.predict(swapped) != original_decision:
            flips += 1
    return flips / len(resumes)
```

Run on 10 000 real resumes: every flip is a fairness violation, surfaced *without* a ground-truth "correct hiring decision."

The same template generalizes to NLP robustness: `model(x) == model(paraphrase(x))`, `sentiment(x) == sentiment(synonym_swap(x))`, etc.

References I

[1] Laura Inozemtseva and Reid Holmes.
Coverage is not strongly correlated with test suite effectiveness.
International Conference on Software Engineering (ICSE), pages 435–445, 2014.

[2] Yue Jia and Mark Harman.
An analysis and survey of the development of mutation testing.
IEEE transactions on software engineering, 37(5):649–678, 2010.

[3] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward.
Hints on test data selection: Help for the practicing programmer.
Computer, 11(4):34–41, 1978.

[4] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward.
Mutation analysis.
Yale University. Department of Computer Science, 1979.

[5] Elaine J Weyuker.
On testing non-testable programs.
The Computer Journal, 25(4):465–470, 1982.

[6] Tsong Yueh Chen, Shing Chi Cheung, and Shiu Ming Yiu.
Metamorphic testing: A new approach for generating next test cases.
Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[7] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen.
Metamorphic testing for software quality assessment: A study of search engines.
IEEE Transactions on Software Engineering, 42(3):264–284, 2016.