Mutation Testing

Moonzoo Kim
School of Computing
KAIST

The original slides are taken from Chap. 9 of Intro. to SW Testing 2nd ed by Ammann and Offutt

2 Hypotheses of Mutation Testing

- The first is the **competent programmer** hypothesis.
  - This hypothesis states that most software faults introduced by experienced programmers are due to small syntactic errors.
    - These days, this hypothesis is not considered important
- The second hypothesis is called the **coupling effect**.
  - The coupling effect asserts that simple faults can cascade or couple to form other emergent faults
  - This hypothesis is at the heart of mutation analysis

Mutation Testing

- Operators modify a program under test to create mutant programs
  - Mutant programs must compile correctly
  - Mutants are not tests, but used to find good tests
- Once mutants are defined, tests must be found to cause mutants to fail when executed
  - This is called "killing mutants"

Killing Mutants

Given a mutant *m* $\in$ *M* for a ground string program *P* and a test *t*, *t* is said to kill *m* if and only if the output of *t* on *P* is different from the output of *t* on *m*.

- If mutation operators are designed well, the resulting tests will be very powerful
- Different operators must be defined for different programming languages and goals
- Testers can keep adding tests until all mutants have been killed
  - *Dead mutant* : A test case has killed it
  - *Trivial mutant* : Almost every test can kill it
  - *Equivalent mutant* : No test can kill it (equivalent to original program)
  - *Stubborn mutant*: Almost no test can kill it (a.k.a hard-to-kill mutants)
  - *Stillborn mutant*: An uncompilable mutant (i.e., syntax error)

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min'(int A, int B)
{
        int minVal;
        minVal = A;
Δ 1     minVal = B;
        if (B < A)
Δ 2     if (B > A)
Δ 3     if (B < minVal)
        {
            minVal = B;
Δ 4         Bomb ();
Δ 5         minVal = A;
Δ 6         minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing

6 mutants

Each represents a separate program

Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each *m* ∈ *M*, TR contains exactly one requirement, to kill *m*.

- **The RIP model**
  - ***Reachability* : The test causes the faulty statement to be reached (in mutation – the mutated statement)**
  - ***Infection* : The test causes the faulty statement to result in an incorrect state**
  - ***Propagation* : The incorrect state propagates to incorrect output**
- **The RIP model leads to two variants of mutation coverage …**

Strong v.s. Weak Mutants

1) Strongly Killing Mutants:

Given a mutant $m \in M$ for a program *P* and a test *t*, *t* is said to *strongly kill* *m* if and only if the output of *t* on *P* is different from the output of *t* on *m*

2) Weakly Killing Mutants:

Given a mutant $m \in M$ that modifies a location *l* in a program *P*, and a test *t*, *t* is said to *weakly kill* *m* if and only if the state of the execution of *P* on *t* is different from the state of the execution of *m* immediately on *t* after *l*

- **Weakly killing satisfies reachability and infection, but not propagation**

Equivalent Mutation Example

- Mutant 3 in the Min() example is equivalent:

```
      minVal = A;
      if (B < A)
Δ 3  if (B < minVal)
```

- **The infection condition is "(B < A) != (B < minVal)"**

- **However, the previous statement was "minVal = A"**
  - **Substituting, we get: "(B < A) != (B < A)"**
  - **This is a logical contradiction !**

- **Thus no input can kill this mutant**

Strong Versus Weak Mutation

```
1    boolean isEven (int X)
2    {
3        if (X < 0)
4            X = 0 - X;
Δ 4          X = 0;
5        if (double) (X/2) == ((double) X) / 2.0
6            return (true);
7        else
8            return (false);
9    }
```

<u>Reachability</u> : X < 0

<u>Infection</u> : X != 0

(X = -6) will kill mutant 4 under <u>weak mutation</u>

<u>Propagation</u> :

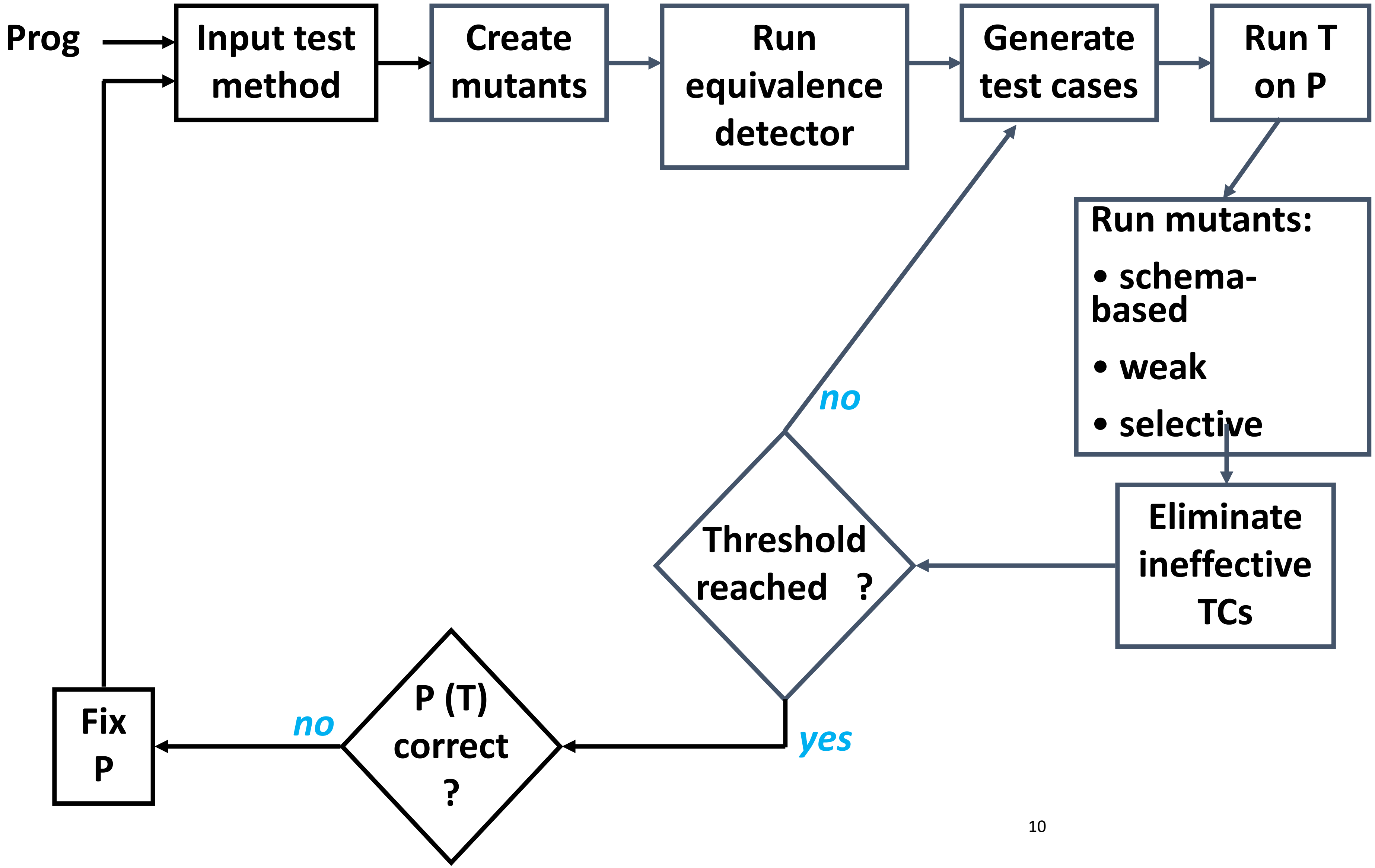
((double) ((0-X)/2) == ((double) 0-X) / 2.0)

!=   ((double) (0/2) == ((double) 0) / 2.0)

That is, X is <u>not</u> even …

Thus (X = -6) does <u>*not*</u> kill the mutant under strong mutation

Testing Programs with Mutation



Why Mutation Testing Works

Fundamental Premise of Mutation Testing

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- Also known as "Coupling Effect"
  - "a test data set that distinguishes all programs with simple faults is so sensitive that it will also distinguish programs with more complex faults"
    - R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. Computer, 11(4), April 1978.
- The mutants guide the tester to an effective set of tests
- A very challenging problem :
  - Find a fault and a set of mutation-adequate tests that do not find the fault
- Of course, this depends on the mutation operators …

Designing Mutation Operators

- At the method level, mutation operators for different programming languages are similar
- Mutation operators do one of two things :
  - Mimic typical programmer mistakes (incorrect variable name)
  - Encourage common test heuristics (cause expressions to be 0)
- Researchers design lots of operators, then experimentally *select* the most useful

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators *O* = {*o1, o2, …*} also kill mutants created by all remaining mutation operators with very high probability, then *O* defines an *effective* set of mutation operators

Mutation Operators

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

Examples:

a = m * (o + p);

Δ1 a = abs (m * (o + p));

Δ2 a = m * abs ((o + p));

Δ3 a = failOnZero (m * (o + p));

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, −, *, /, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

Examples:

a = m * (o + p);

Δ1 a = m + (o + p);

Δ2 a = m * (o * p);

Δ3 a = m *leftOp* (o + p);

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by *falseOp* and *trueOp*.

Examples:

if (X <= Y)

Δ1 if (X > Y)

Δ2 if (X < Y)

Δ3 if (X *falseOp* Y) // always returns false

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - ||) is replaced by each of the other operators; in addition, each is replaced by *falseOp*, *trueOp*, *leftOp*, and *rightOp*.

Examples:

if (X <= Y && a > 0)

Δ1 if (X <= Y || a > 0)

Δ2 if (X <= Y *leftOp* a > 0) // returns result of left clause

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, >>>, and <<< is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.

Examples:

byte b = (byte) 16;

b = b >> 2;

Δ1 b = b << 2;

Δ2 b = b *leftOp* 2; // result is b

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Examples:

int a = 60; int b = 13;

int c = a & b;

Δ1 int c = a | b;

Δ2 int c = a *rightOp* b; // result is b

7. ASR — Assignment Operator Replacement:

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

Examples:

a = m * (o + p);

Δ1 a += m * (o + p);

Δ2 a *= m * (o + p);

8. UOI — Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

Examples:

a = m * (o + p);

Δ1 a = m * -(o + p);

Δ2 a = -(m * (o + p));

9. UOD — Unary Operator Deletion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Examples:

if !(X <= Y && !Z)

Δ1 if (X > Y && !Z)

Δ2 if !(X < Y && Z)

10. SVR — Scalar Variable Replacement:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

Examples:

a = m * (o + p);

Δ 1 a = o * (o + p);

Δ 2 a = m * (m + p);

Δ 3 a = m * (o + o);

Δ 4 p = m * (o + p);

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Example:

a = m * (o + p);

Δ1 *Bomb*() // Raises exception when reached

Summary : Subsumption of Other Criteria

- Mutation is widely considered the strongest test criterion
  - And most expensive !
  - By far the most test requirements (each mutant)
  - Not always the most tests
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption can only be defined for weak mutation – other criteria impose local requirements, like weak mutation
  - Node coverage
  - Edge coverage
  - Clause coverage
  - All-defs data flow coverage
- Reference:
  - An Analysis and Survey of the Development of Mutation Testing by Y.Jia et al.
    - IEEE Transactions on Software Engineering Volume: 37 Issue: 5
  - Design Of Mutant Operators For The C Programming Language by H.Agrawal et al.
    - Technical report

Bug Observability/Detection Model: **R**eachability, **I**nfection, **P**ropagation, and **R**evealation (RIPR)

- Terminology
  - Fault: static defect in a program text (a.k.a a bug)
  - Error: dynamic (intermediate) behavior that deviates from its (internal) intended goal
    - A fault causes an error (i.e. error is a symptom of fault)
  - Failiure: dynamic behavior which violates a ultimate goal of a target program
    - Not every error leads to failure due to error masking or fault tolerance

- Graph coverage
  - Test requirement satisfaction == Reachability
    - the fault in the code has to be reached
- Logic coverage
  - Test requirement satisfaction

    == Reachability +Infection
    - the fault has to put the program into an error state.
      - Note that a program is in an error state does not mean that it will always produce the failure
- Mutation coverage
  - Test requirement satisfaction

    == Reachability +Infection + Propagation
    - the program needs to exhibit incorrect outputs
- Furthermore, test oracle plays critical role to reveal failure of a target program (Revealation)