Software Verification & Validation

Natthapong Jungteerapanich

Handout 8

Acknowledgement

- Some slides in this lecture are adapted from
  - **Paul Ammann and Jeff Offutt**'s slides for their textbook **"Introduction to Software Testing"**
  - **Prof. Lori A. Clarke**'s slides for her course **"CS521-621 : Advanced Software Engineering: Analysis and Evaluation"** at University of Massachusetts Amherst

Error Seeding

- Intentionally insert <u>faults</u> into a system. Why?

1. Evaluate the effectiveness of a test set.
   - Suppose S faults have been seeded.
   - After executing a test set T, S* out of S seeded faults are caught by the test.
   - Assuming that seeded faults and actual faults are equally likely to be caught by test T, then the effectiveness of T is S*/S.

2. Estimate the number of actual faults.
   - Suppose S faults have been seeded.
   - After executing a test set T,
     - S* out of S seeded faults have been caught by the test.
     - N* actual faults have been caught by the test.
   - Assuming that seeded faults and actual faults are equally likely to be caught by test T, we may conclude (with a certain confidence) that the number of actual faults is

$$N = (N^*/S^*) \times S$$

3. Motivate developers/testers
   - Know there is something to find
   - Not looking for their own faults, so more motivated

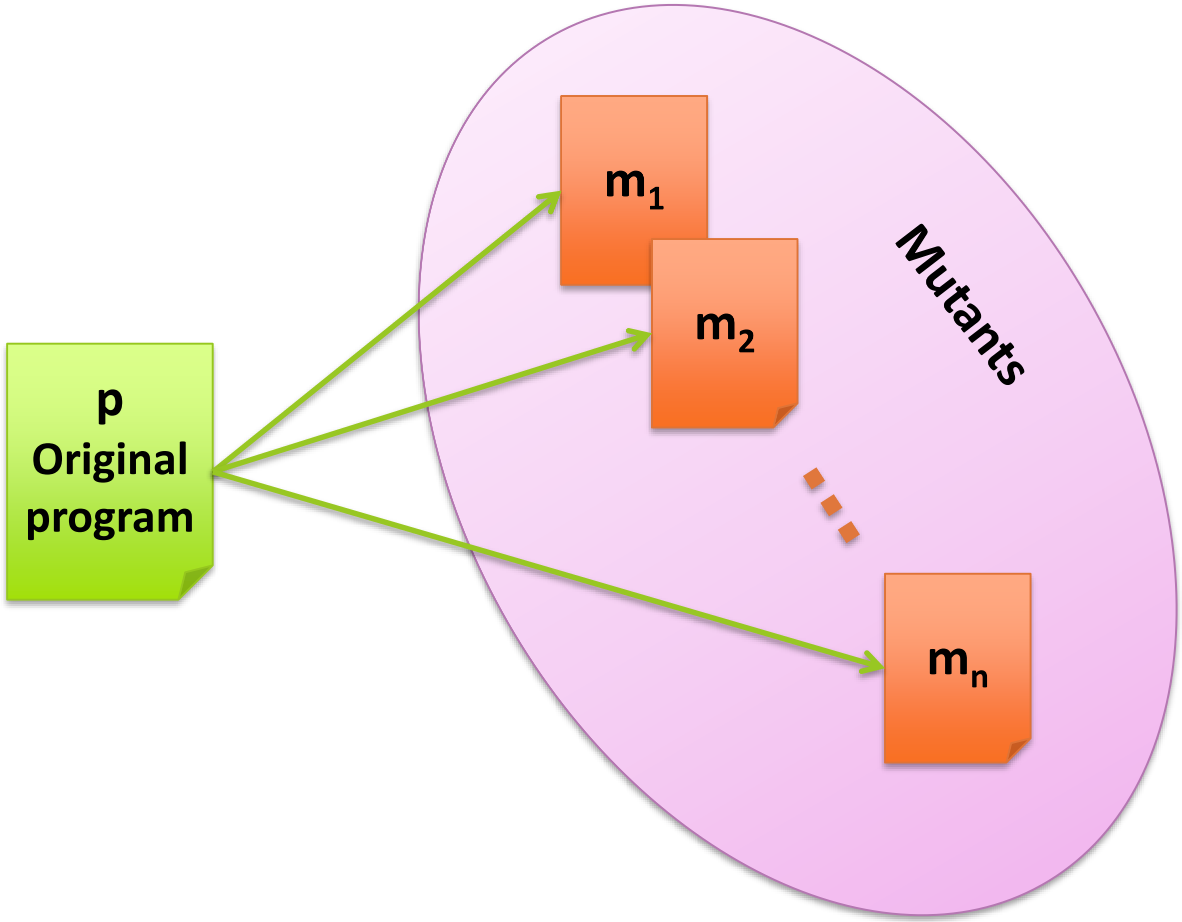
Mutation Testing

- Mutation testing is a systematic method of error seeding
  - originally proposed by Budd, Lipton, DeMillo, and Sayward in the mid 1970s

- Approach: considers all simple faults that could occur
  - introduces single faults one at a time to create "mutants" of original program
  - apply test set to each mutant program
  - "test adequacy" is measured by % "mutants killed"

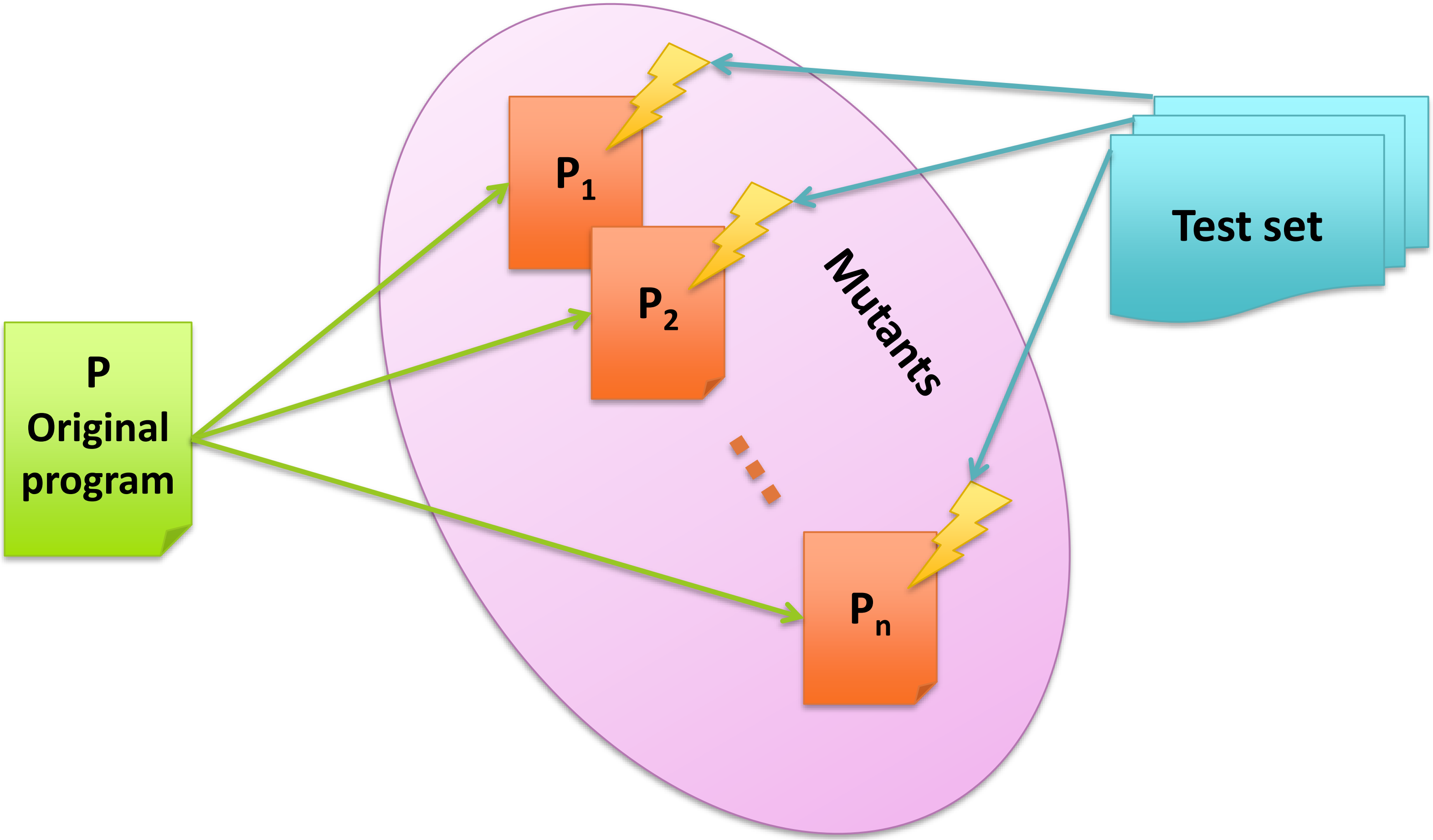
Mutation Testing

- Mutate original program by inserting a simple fault



Mutation Testing

- Design a test set to kill each mutant



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 or evaluate 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 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)

Mutating programs

Original Method

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

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);
}
```

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 $m \in M$, there is a test case that kills m.

- Mutant 1 in the Min( ) example is:

```
     minVal = A;
Δ 1  minVal = B;
     if (B < A)  minVal = B;
     return minVal
```

- Find a test case that kills Mutant 1.
- Find a test case that does not kill Mutant 1.

Equivalent Mutation Example

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

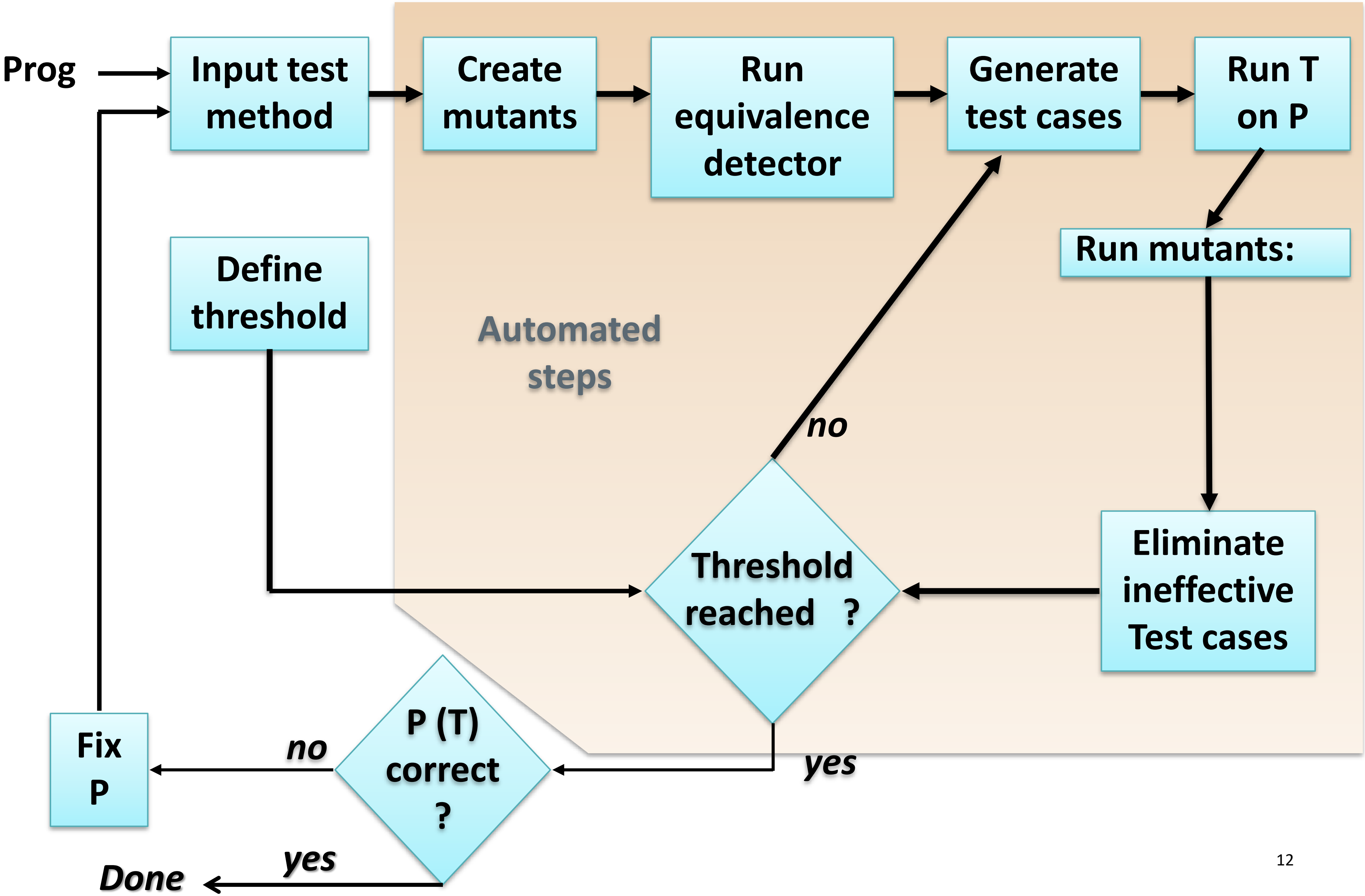
```
      minVal = A;
      if (B < A)
Δ 3  if (B < minVal)
```

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

- However, the previous statement was "minVal = A"
  - Substituting, we get: "(B < A) != (B < A)"

- <u>Thus no input can kill this mutant</u>

Testing Programs with Mutation



Why Mutation 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

- This is not an absolute !
- The mutants guide the tester to a very 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.
- The following are some example of mutation operators for Java.

Mutation Operators for Java

1. ABS — Absolute Value Insertion:

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

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*.

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.

Mutation Operators for Java (2)

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by falseOp, trueOp, leftOp, and rightOp.

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.

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.

Mutation Operators for Java (3)

7. ASR — Assignment Operator Replacement:

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

8. UOI — Unary Operator Insertion:

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

9. UOD — Unary Operator Deletion:

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

Mutation Operators for Java (4)

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.

11. BSR — Bomb Statement Replacement:

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