CS 5154

Mutation Testing

Owolabi Legunsen

The following are modified versions of the publicly-available slides for Chapter 9 in the Ammann and Offutt Book, "Introduction to Software Testing"
(http://www.cs.gmu.edu/~offutt/softwaretest)

Some announcements

- All assigned work so far has been graded
  - See scores on CMS
  - Daniel (TA): announce office hours for questions/requests on Ed
- The scores so far represent ~35% of your course grade
  - ~5 out of 40 percentage points for project
  - ~30 out of 50 percentage points for homework
- There is still a LOT more points to work for in the course
  - At least 2 more homework (~20%)
  - Course project gets in full swing today (~35%)
  - Keep working to participate in class and in your group (~10%)
- Past performance is not necessarily a good predictor of future performance

Food for thought…

- You have written tests that satisfy "strong" coverage criteria
- But, how do you know how good those tests are in terms of their fault-detection capability?

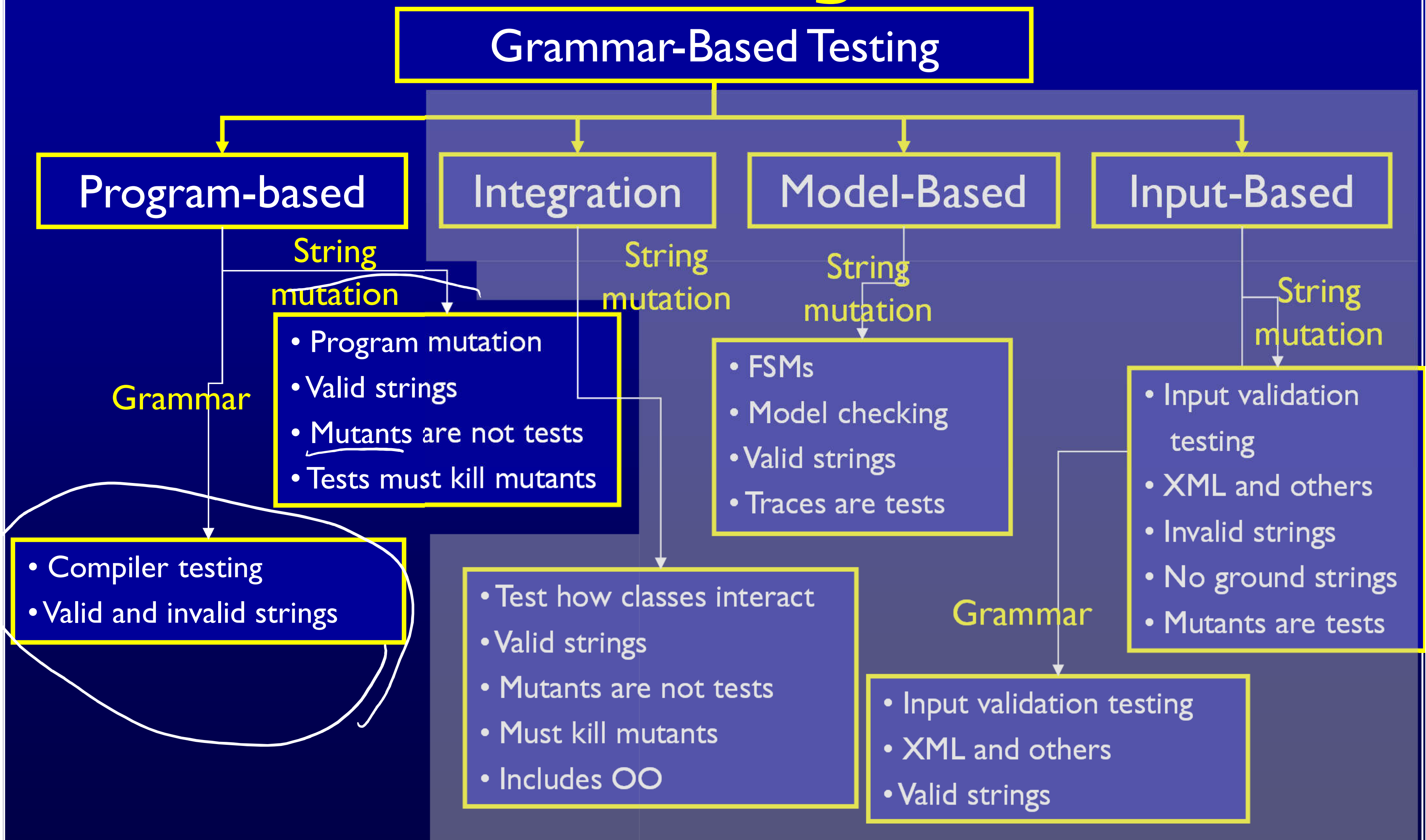
how to test your tests?

Mutation Testing

Applying Syntax-based Testing to Programs

- Syntax-based criteria originated with programs and have been used mostly with programs
- BNF criteria are most commonly used to test compilers
- Mutation testing criteria are most commonly used for unit testing and integration testing of classes

Instantiating Grammar-Based Testing



BNF Testing for Compilers

- Testing compilers is very complicated
  - Millions of correct programs !
  - Compilers must recognize and reject incorrect programs
- BNF criteria can be used to generate programs to test all language features that compilers must process
- A very specialized application; not discussed in CS5154

Some work on compiler testing

Finding and Understanding Bugs in C Compilers

Taming Compiler Fuzzers

A Survey of Compiler Testing

JUNJIE CHEN, College of Intelligence and Computing, Tianjin University, China
JIBESH PATRA and MICHAEL PRADEL, Department of Computer Science, University of Stuttgart, Germany
YINGFEI XIONG, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
HONGYU ZHANG, School of Electrical Engineering and Computing, University of Newcastle, Australia
DAN HAO and LU ZHANG, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China

- **Articles should be available for FREE @ Cornell Library**
- **Do NOT pay to read these articles**

Program-based Grammars

- The original and most common application of syntax-based testing is to modify programs
- Operators modify a ground string (program under test) to create mutant programs

- Mutant programs must compile correctly (valid strings)
- 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 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 different goals
- Testers can keep adding tests until all mutants are killed

Some types of Mutants

- *Dead mutant* : A test case has killed it
- *Stillborn mutant* : Syntactically illegal
- *Trivial mutant* : Almost every test can kill it
- *Equivalent mutant* : No test can kill it (same behavior as original)

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

Replace one variable with another

Replaces 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*, TR contains exactly one requirement, to kill *m*.

- The RIPR model from chapter 2:
  - *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
  - *Revealability* : The tester must observe part of the incorrect output
- The RIPR model leads to two variants of mutation coverage …

Syntax-Based Coverage Criteria

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 on t immediately after l

- Weakly killing satisfies reachability and infection, but not propagation

Weak Mutation

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

- "Weak mutation" is so named because it is easier to kill mutants under this assumption
- Weak mutation also requires less analysis
- A few mutants can be killed under weak mutation but not under strong mutation (no propagation)
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants

Weak Mutation In-class Exercise

Mutant 1 in the Min( ) example is:

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

In your breakout room :

1. Find a test that **weakly kills** the mutant, but not strongly
2. Generalize : What **must be true** to weakly kill the mutant, but not strongly?
3. Try to write down the **conditions** needed to (i) **reach** the mutated statement, (ii) **infect** the program state, and (iii) **propagate** to output

Weak Mutation In-class Exercise

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

1. Find a test that weakly kills the mutant, but not strongly

   A = 5, B = 3

2. Generalize : What must be true to weakly kill the mutant, but not strongly?

   B < A // minVal is set to B for both

3. RIP conditions

   Reachability : *true* // we always reach

   Infection : *A ≠ B // minVal has a different value*

   Propagation : *(B < A) = false // Take a different branch*

Equivalent Mutation In-class Exercise

Mutant 3 in the Min() example is equivalent:

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)
Δ 3   if (B < minVal)
      {
          minVal = B;
      }
      return (minVal);
} // end Min
```

With one or two partners

1. Convince yourselves that this mutant is equivalent
2. Briefly explain why
3. Try to prove the equivalence

Hint : Think about what must be true to kill the mutant

Equivalent Mutation In-class Exercise

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

1. Convince yourselves that this mutant is equivalent
2. Briefly explain why

> A and minVal have the same value at the mutated statement

3. Try to prove the equivalence

Hint : Think about what must be true to kill the mutant

> Infection : (B < A) != (B < minVal)
>
> Previous statement : minVal = A
>
> Substitute : (B < A) != (B < A)
>
> Contradiction … therefore, **equivalent**

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    }
```

Reachability : X < 0

Infection : X != 0

(X = -6) will kill mutant 4 under weak mutation

Propagation :

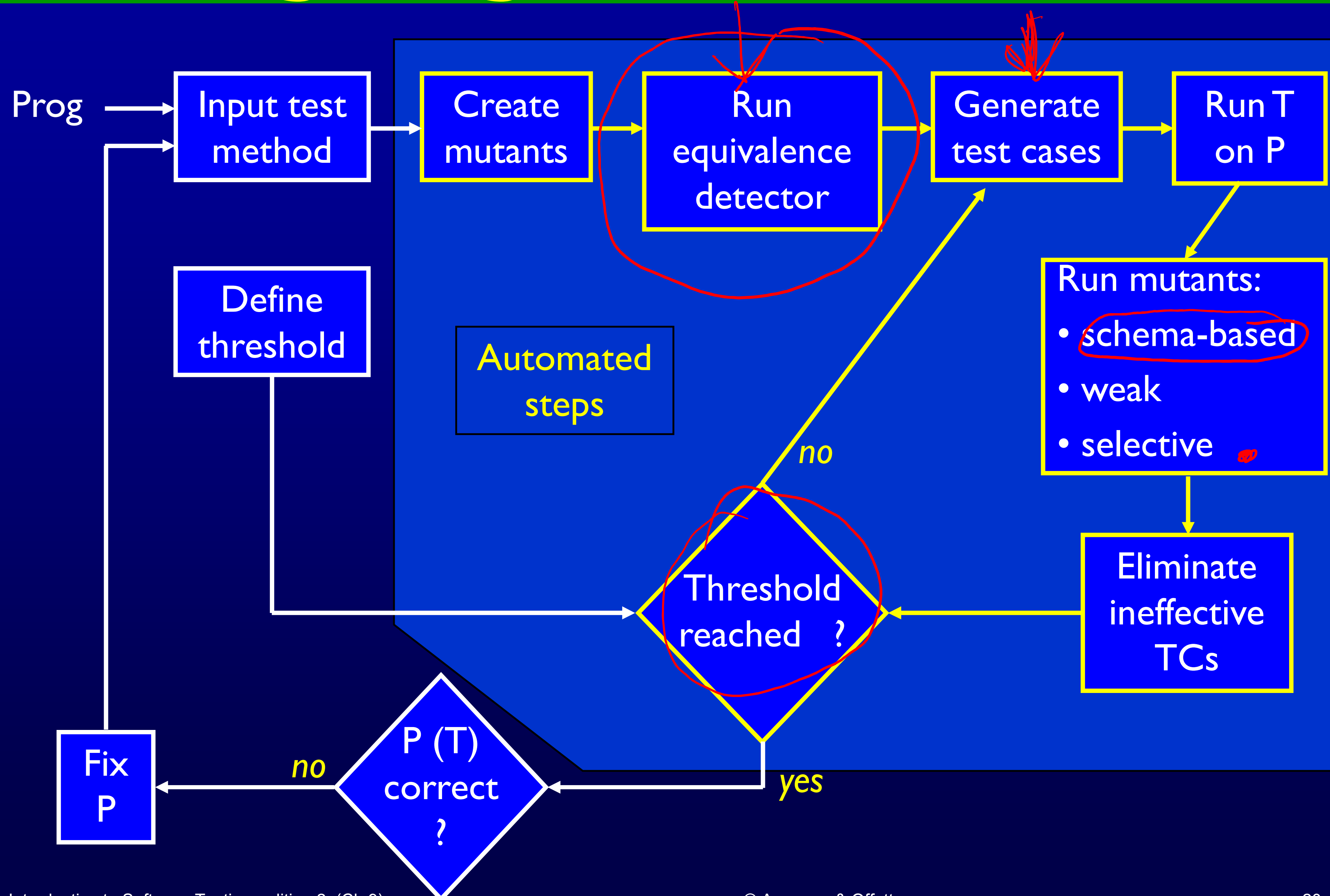
((double) ((0-X)/2) == ((double) 0-X) / 2.0)

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

That is, X is not even …

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

Testing Programs with Mutation



Industry is using mutation testing

An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions

Goran Petrović Marko Ivanković
Google Switzerland GmbH
Zurich, Switzerland
{goranpetrovic, markoi}@google.com

Bob Kurtz Paul Ammann
George Mason University
Fairfax, VA, USA
{rkurtz2, pammann}@gmu.edu

René Just
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice

State of Mutation Testing at Google

Goran Petrović
Google Inc.
goranpetrovic@google.com

Marko Ivanković
Google Inc.
markoi@google.com

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 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 for Java

1. ABS — Absolute Value Insertion
2. AOR — Arithmetic Operator Replacement
3. ROR — Relational Operator Replacement
4. COR — Conditional Operator Replacement
5. SOR — Shift Operator Replacement
6. LOR — Logical Operator Replacement
7. ASR — Assignment Operator Replacement
8. UOI — Unary Operator Insertion
9. UOD — Unary Operator Deletion
10. SVR — Scalar Variable Replacement
11. BSR — Bomb Statement Replacement

Full definitions …

Mutation Operators for Java

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);
```

Mutation Operators for Java (2)

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 - || , 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*.

Examples:

if (X <= Y && a > 0)
Δ1 if (X <= Y || a > 0)
Δ2 if (X <= Y *leftOp* a > 0) // returns result of left clause

Mutation Operators for Java (4)

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
```

Mutation Operators for Java (5)

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));
```

Mutation Operators for Java (6)

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);
```

Mutation Operators for Java (7)

11. BSR — Bomb Statement Replacement:

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

Example:

```
      a = m * (o + p);
∆1    Bomb() // Raises exception when reached
```

Summary : Subsuming Other Criteria

- Mutation is widely considered the strongest test criterion
  - And most expensive !
  - By far the most test requirements (each mutant)
  - Usually the most tests
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption can only be defined for weak mutation – other criteria only impose local requirements
  - Node coverage, Edge coverage, Clause coverage
  - General active clause coverage: Yes–Requirement on single tests
  - Correlated active clause coverage: No–Requirement on test *pairs*
  - All-defs data flow coverage

Next

- Demo of a tool that does Mutation Testing
- Project Sprint starts today
  - Progress report due in ~3 weeks
  - There will be one homework in that time span
- The rest of the course…