Ch2 Code Unit Test

Automatic Unit Tests Design(3)

Instructor: Haiying SUN

E-mail: hysun@sei.ecnu.edu.cn

Office: ECNU Science Build B1104

Available Time: Wednesday 8:00 -12:00 a.m.

Overview

- Control Based Tests Generation
  - Prime Path Testing
- Data Flow Based Testing
  - Data Flow Graph (DFG)
  - Data Flow Testing Coverage Criteria
- Mutation Testing
  - Related Concepts
  - Mutation based test generation
  - Mutation operators design

Overview

- Control Based Tests Generation
  - Prime Path Testing
- Data Flow Based Testing
  - Data Flow Graph (DFG)
  - Data Flow Testing Coverage Criteria
- Mutation Testing
  - Related Concepts
  - Mutation based test generation
  - Mutation operators design

Mutation Testing

Although satisfying with the same coverage criteria, which test set would you like to select? Why?

```
7        if(op1 <= op2)
8            return op1 + op2;
9        else
10           return op1 * op2;
11   }
```

Test set 1

- test inputs: op1 = -1, op2 = 2, expected result: 1
- test inputs: op1 = -1, op2 = -2, expected result: 2

Test set 2

- test inputs: op1 = -1, op2 = -1, expected result: -2
- test inputs: op1 = -1, op2 = -2, expected result: 2

Test set 3

- test inputs: op1 = -1, op2 = 2, expected result: 1
- test inputs: op1 = -1, op2 = -2, expected result: 2
- test inputs: op1 = -1, op2 = -1, expected result: -2

Mutation Testing

- In case test set1, how about this bug?

```
    //if (op1 <= op2)
    if(op1 < op2)
        return op1 + op2;
    else
        return op1 * op2;
}
```

Test set 1

- test inputs: op1 = -1, op2 = 2, expected result: 1
- test inputs: op1 = -1, op2 = -2, expected result: 2

Mutation Testing

- In case test set2, how about this bug?

```
    //if (op1 <= op2)
    if(op1 == op2)
        return op1 + op2;
    else
        return op1 * op2;
}
```

Test set 2

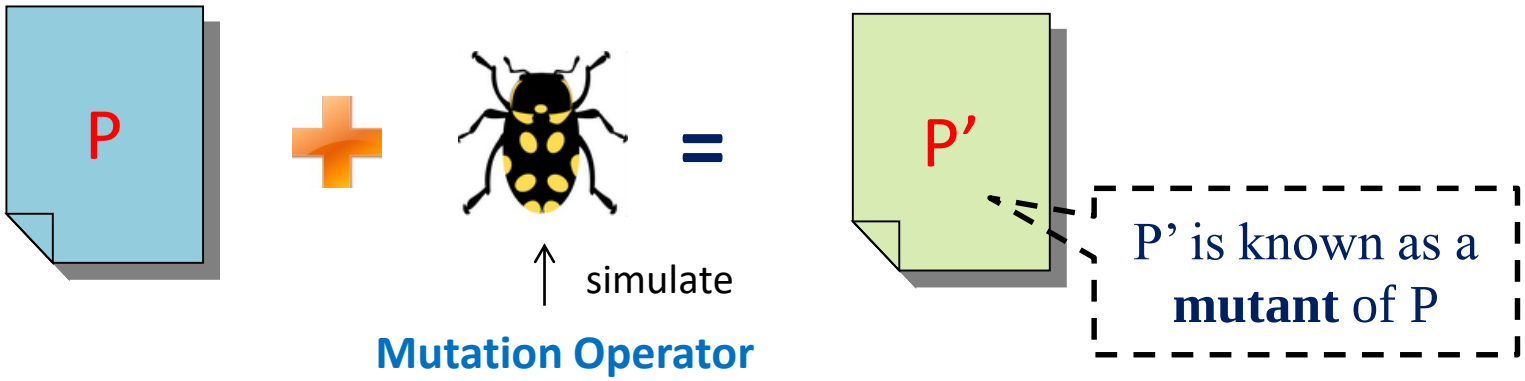
- test inputs: op1 = -1, op2 = -1, expected result: -2
- test inputs: op1 = -1, op2 = -2, expected result: 2

Mutation Testing

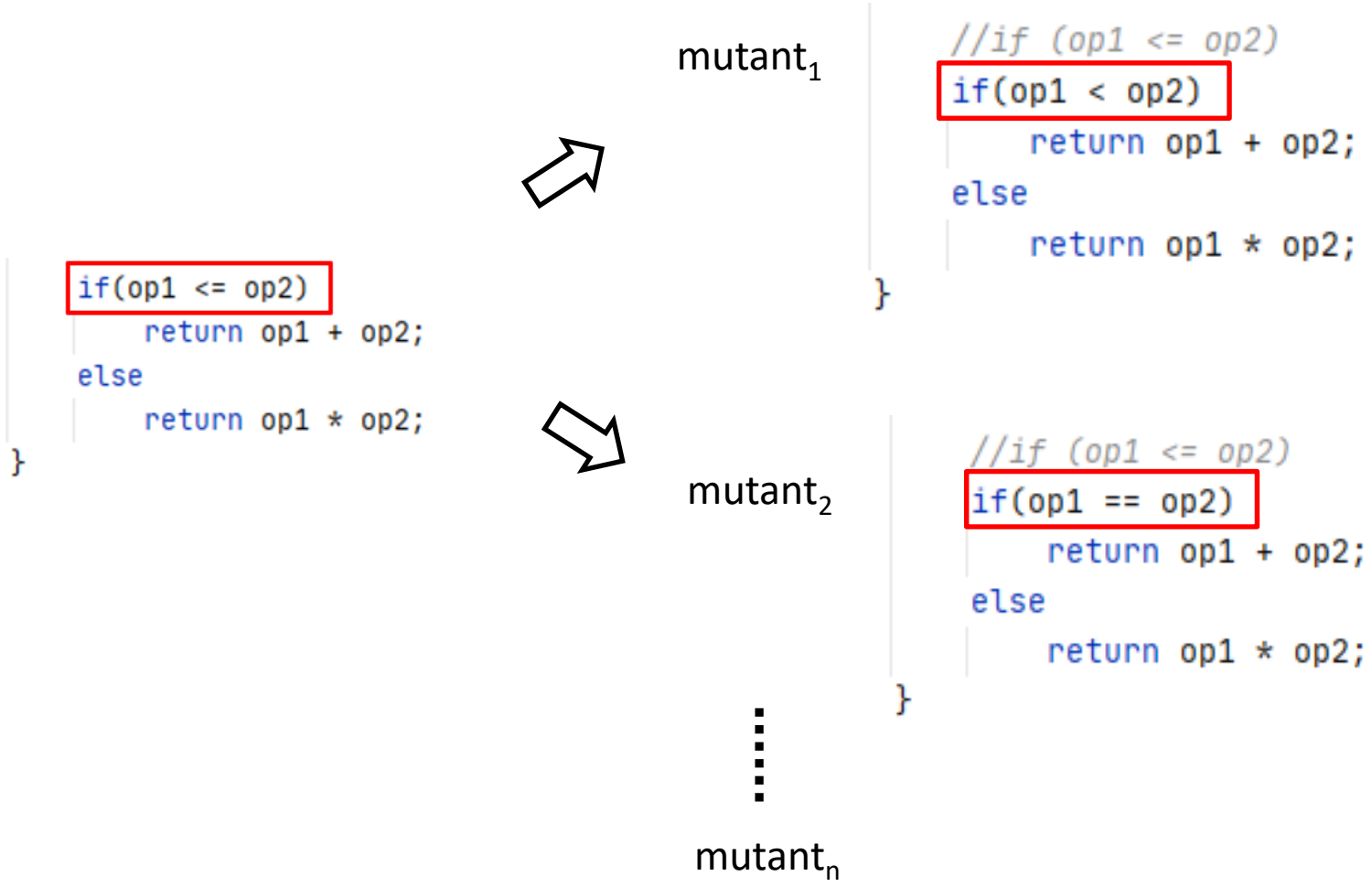
- Mutation Testing is used to evaluate the defect-detected ability of a test set by introducing bugs into the object under test and calculating how many bugs are found
  - [mutation operator] method used to introduce bugs
  - [mutation score] method used to calculate found bugs

Mutation Operator

- Mutation Operator
  - A rule that specifies syntactic variations of object under test generated from a grammar
  - Mutant programs must compile correctly
  - First order : Only one change
  - High order: More than one changes



Mutation Operator



Mutation Operators of PIT

Mutators		
Conditionals Boundary	Remove Conditionals	Arithmetic Operator Replacement
Increments	Experimental Switch	Arithmetic Operator Deletion
Invert Negatives	Inline Constant	Constant Replacement
Math	Constructor Calls	Bitwise Operator
Negate Conditionals	Non Void Method Calls	Relational Operator Replacement
Return Values	Remove Increments	Unary Operator Insertion
Void Method Calls	Experimental Argument Propagation	
Empty returns	Experimental Big Integer	
False Returns	Experimental Member Variable	
True returns	Experimental Naked Receiver	
Null returns	Negation	
Primitive returns		

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

Mutation Operator of PIT

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

For example

```
if (a < b) {
  // do something
}
```

will be mutated to

```
if (a <= b) {
  // do something
}
```

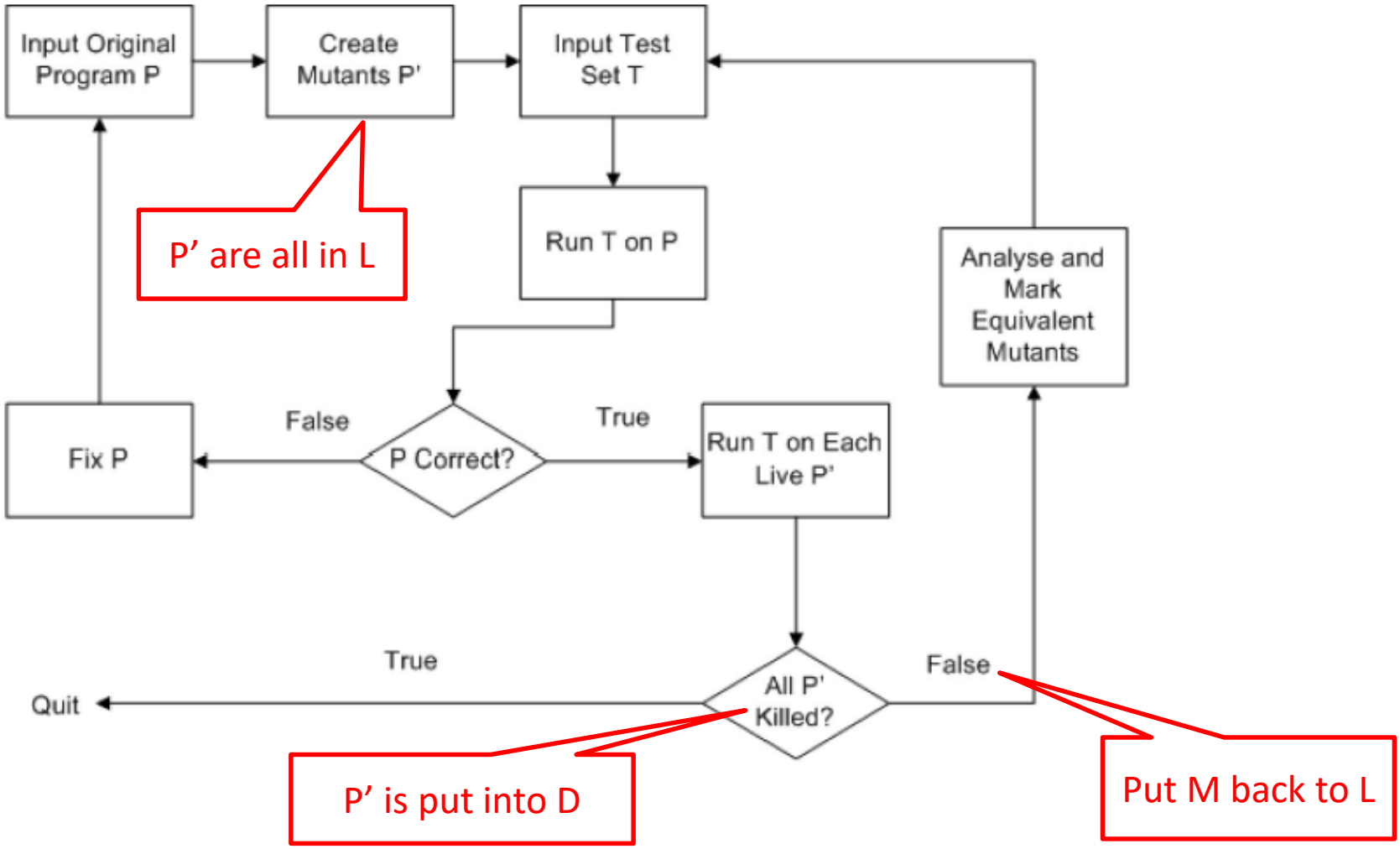
Mutation Score

- MS(T) = |D|/(|L|+|D|)
  - |D|: the number of killed mutants
  - |L| : the number of lived mutants
  - Mutation score for a test set is computed with respect to a given set of mutants. Therefore, mutation score may be different for a test set if the set of mutants changed

Mutation Testing

- Mutant
  - The result of one application of a mutation operator
  - Live mutant: The mutant that has the same results as that of P against all test cases of T, denoted as L
  - Killed mutant : The mutant that has different result to P against at least one test case of T, denoted as D
  - Equivalent mutant: The mutant that has the same results as that of P against all input domain, denoted as E

Mutation Testing Procedure



```
1 begin
2  int x, y;
3  input (x, y);
4  if(x < y)
5    output(x+y);
6  else
7    output(x*y);
8 end
```

	x	y	Expected Result
t1	0	0	0
t2	0	1	1
t3	1	0	0
t4	-1	-2	2

MS(T) = |D|/(|L|+|D|)

= 7/(1+7)

= 0.875

Line	Original	Mutant ID	Mutants
1	begin		
2	int x,y		
3	input(x,y)		
4	if(x<y)	M1	if(x+1<y)
		M2	if(x<y+1)
5	then		
6	output(x+y)	M3	output(x+1+y)
		M4	output(x+y+1)
		M5	output(x-y)
7	else		
8	output(x*y)	M6	output((x+1)*y)
		M7	output((x*(y+1))
		M8	output(x/y)
9	end		

Program	t1	t2	t3	t4	D
P(t)	0	1	0	2	{}
Mutant					
M1(t)	0	0	NE	NE	{M1}
M2(t)	0	1	0	2	{M1}
M3(t)	0	2	NE	NE	{M1,M3}
M4(t)	0	2	NE	NE	{M1,Me,M4}
M5(t)	0	-1	NE	NE	{M1,Me,M4,M5}
M6(t)	0	1	0	0	{M1,Me,M4,M5,M6}
M7(t)	0	1	1	NE	{M1,Me,M4,M5,M6,M7}
M8(t)	UD	NE	NE	NE	{M1,Me,M4,M5,M6,M7}M8

PIT

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
 </plugin>
```


```
122                 // Verify for a ".." component at next iter
123 3               if ((newcomponents.get(i)).length() > 0 &
124                 {
125                     newcomponents.remove(i);
126                     newcomponents.remove(i);
127 1                   i = i - 2;
128 1                   if (i < -1)
129                     {
130                         i = -1;
131                     }
132                 }
133             }
```

Overview

- Control Based Tests Generation
  - Prime Path Testing
- Data Flow Based Testing
  - Data Flow Graph (DFG)
  - Data Flow Testing Coverage Criteria
- Mutation Testing
  - Related Concepts
  - Mutation based test generation
  - Mutation operators design

Mutation testing Generation

- Fundamental Premise of Mutation Testing
  - It is believe that in practice, if the software contains a bug, there will usually be a set of mutants that can only be killed by a test case that also detects that bug
- Mutants are not tests, but used to find tests
- Once mutants are defined, tests must be found to cause mutants to fail when executed, that is "killing mutants"

Mutation testing Generation

- Given a mutant m ∈ 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

Questions About Mutation

- Should more than one operator be applied at the same time ?
  - Should a mutated object contain one mutated element or several?
    - Usually not – multiple mutations can interfere with each other
    - Extensive experience with program-based mutation indicates not
    - Recent research is finding exceptions
- Should every possible application of a mutation operator be considered ?
  - Necessary with program-based mutation

Questions About Mutation

- Mutation operators were defined for several languages
  - Several programming languages (Fortran, Lisp, Ada, C, C++, Java)
  - Specification languages (SMV, Z, Object-Z, algebraic specs)
  - Modeling languages (Statecharts, activity diagrams)
  - Input grammars (XML, SQL, HTML)

Mutation-Based Coverage Criteria

- Mutation Coverage (MC) : For each m ∈M, TR contains exactly one requirement, to kill m.
- The RIP model from chapter 1:
  - 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 …

Mutation-Based Coverage Criteria

- 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
- Strong Mutation Coverage (SMC)
  - For each $m \in M$, TR contains exactly one requirement, to strongly kill m.

Example

- Mutant 1 in

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

- The complete test specification to kill mutant 1:
  - Reachability : *true* // Always get to that statement
  - Infection : $A \neq B$
  - Propagation: $(B < A) = false$ // Skip the next assignment
  - Full Test Specification : $true \wedge (A \neq B) \wedge ((B < A) = false)$

    $\equiv (A \neq B) \wedge (B \geq A)$

    $\equiv (B > A)$
  - However, (A = 3, B = 5) will strongly kill mutant 1

Mutation-Based Coverage Criteria

- 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

Weak Mutation

- Weak Mutation Coverage (WMC) : For each $m \in M$, 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
  - Some mutants can be killed under weak mutation but not under strong mutation (no propagation)

Weak Mutation Example

- Mutant 1 in

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

- The complete test specification to kill mutant 1:
  - Reachability : *true* // Always get to that statement
  - Infection : $A \neq B$
  - ~~Propagation: $(B < A) = false$ // Skip the next assignment~~
  - Test Specification : $true \wedge (A \neq B) \equiv (A \neq B)$
  - (A = 5, B = 3) will weakly kill mutant 1, but not strongly

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"

Substituing, we get: "(B < A) != (B < A)" , thus no input can kill this mutant

Exercise

- Give a test input that can weakly but not strongly kill the mutant

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

Strong Versus Weak Mutation

```
1     boolean isEven (int X)
2     {
3         if (x < 0)
4             x = 0 - x;
Δ 4           x = 0;
5         if (mod(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 :
(mod(x,2) == 0) != (mod(0,2) == 0)

That is, when x is even, may strongly kill the mutant

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

Overview

- Control Based Tests Generation
  - Prime Path Testing
- Data Flow Based Testing
  - Data Flow Graph (DFG)
  - Data Flow Testing Coverage Criteria
- Mutation Testing
  - Related Concepts
  - Mutation based test generation
  - Mutation operators design

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

Designing Mutation Operators

- 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
- insert unary operator and modify unary and binary operators are effective

Mutation Operators for Java

- ABS (Absolute Value Insertion)
  - Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*
- 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*
- 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)

- 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.
- 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.
- 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)

- ASR (Assignment Operator Replacement)
  - Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.
- UOI(Unary Operator Insertion)
  - Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.
- UOD(Unary Operator Deletion)
  - Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Mutation Operators for Java (4)

- SVR (Scalar Variable Replacement)
  - Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- BSR(Bomb Statement Replacement)
  - Each statement is replaced by a special Bomb() function.

Subsumption of Other Criteria

- Mutation is widely considered the strongest and most expensive test criterion
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption actually only makes sense for weak mutation – other criteria impose local requirements, like weak mutation
  - Node coverage:
  - Edge coverage:
  - Clause coverage
  - Correlated active clause coverage
  - All-defs data flow coverage

Node Coverage

- Proof
  - Mutant Operator: Bomb()
  - Because to kill the mutant generated by Bomb(), one must find test cases to reach each statement or block. That is ,Node coverage is satisfied.

Edge Coverage

- Proof
  - Mutant operator: replace each predicate with both true and false(ROR operator). That is ,to kill true mutant, a test case must take false branch, to kill false mutant, a test case must take true branch. That is, edge coverage is satisified.

Clause Coverage

- Proof
  - Assume a predicate p and a clause a, p’(a→true), in order to kill the mutant, a test case should make value of false(Infection). The same for the false requirement

Summary

- Mutation testing is used to evaluate the defect-detect ability of test set which can be measured by applying mutation scors.
- Mutation Operators are used to mimic the bug however it is questioned that whether the mutants can represent real bugs.
- Test cases may be derived based on weak or strong mutation coverage criteria

The End