Introduction to Software Testing

Chapter 5.2
Program-based Grammars

Paul Ammann & Jeff Offutt

Updated by Sunae Shin

Applying Syntax-based Testing to Programs

- Syntax-based criteria originated with programs and have been used most 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

Program-based Grammars

- The original and most widely known 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
  - Mutants are not tests, but <u>used to find tests</u>
  - Once mutants are defined, tests must be found to cause mutants to fail when executed
  - This is called "killing mutants"

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

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

<u>Mutation Coverage (MC)</u> : For each $m \in M$, TR contains exactly one requirement, to kill m.

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**

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**
- **A few mutants can be killed under weak mutation but not under strong mutation (no propagation)**

Weak Mutation Example

```
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
```

- **Mutant 1 in the Min( ) example is:**

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

- **The complete test specification to kill mutant 1:**
  - **Reachability:** ***true*** **// Always get to that statement (mutant is on the first statement)**
  - **Infection :** $A \neq B$ **// In order to infect, the value B must be different from A**
  - **Propagation:** $(B < A) = \mathit{false}$ **// Skip the next assignment**
  - **Full Test Specification:** $\mathit{true} \wedge (A \neq B) \wedge ((B < A) = \mathit{false})$

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

$$\equiv (B > A)$$

- **The test case (A = 5, B = 7) should cause mutant 1 to result in a failure**

Equivalent Mutation Example

```
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
```

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

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

- **Reachability: true**
- **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**

Mutation example

```
...
found := ~~FALSE;~~ TRUE
i := 1;
while(not(found)) and (i <= x) do begin // x is the length
   if a[i] = c then
     found := TRUE
   else
     i := i + 1
end
if (found)
   print("Character %c appears at position %i");
else
   print("Character is not present in the string");
end
...
```

- **Replace Found := FALSE; with Found := TRUE;**
- **Note: It is better in Mutation Testing to make only one small change at a time to avoid the danger of introduced faults with interfering effects**

Mutation example

```
…
found := ~~FALSE;~~ TRUE
i := 1;
while(not(found)) and (i <= x) do begin // x is the length
   if a[i] = c then
     found := TRUE
   else
     i := i + 1
end
if (found)
   print("Character %c appears at position %i");
else
   print("Character is not present in the string");
end
…
```

- **Failure: "character a appears at position 1" instead of saying "character is not present in the string"**
- **Mutant is killed**

Why Mutation Works

<u>Fundamental Premise of Mutation Testing</u>

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

- **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:

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 / (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 != Y)
```

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 ^ a > 0)
```

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 >>> 2;
```

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;
```

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);
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**

- **Subsumption can only be defined for weak mutation – other criteria impose local requirements, like weak mutation**
  - **Node coverage**
  - **Edge coverage**
  - **Clause coverage**
  - **General active clause coverage**
  - **Correlated active clause coverage**
  - **All-defs data flow coverage**

Example 1

```
//Effects: If numbers null throw NullPointerException
//   else return LAST occurrence of val in numbers[]
//   If val not in numbers[] return -1
public static int findVal (int numbers[], int val)
{
    int findVal = -1;
    for (int i=0; i<numbers.length; i++)
// for (int i=(0+1); i<numbers.length; i++)
        if (numbers [i] == val)
            findVal = i;
    return (findVal);
}
```

- **Find a test input that kills mutant *m*.**

- **Sol: Any input with val only in numbers[0] works. An example is: (numbers, val) = ([1, 0], 1)**

Example 2

```
//Effects: If x null throw NullPointerException
//   else return the sum of the values in x

public static int sum (int[] x)
{
    int s = 0;
    for (int i=0; i < x.length; i++) }
    {
        s = s + x[i];
    // s = s - x[i]; //AOR
    }
    return s;
 }
```

- **Find a test input that kills mutant *m*.**

- **Sol: Any input with a nonzero sum works.**

  An example is: x = [1, 2, 3]