

Introduction to Software Testing

Chapter 5.2

Program-based Grammars

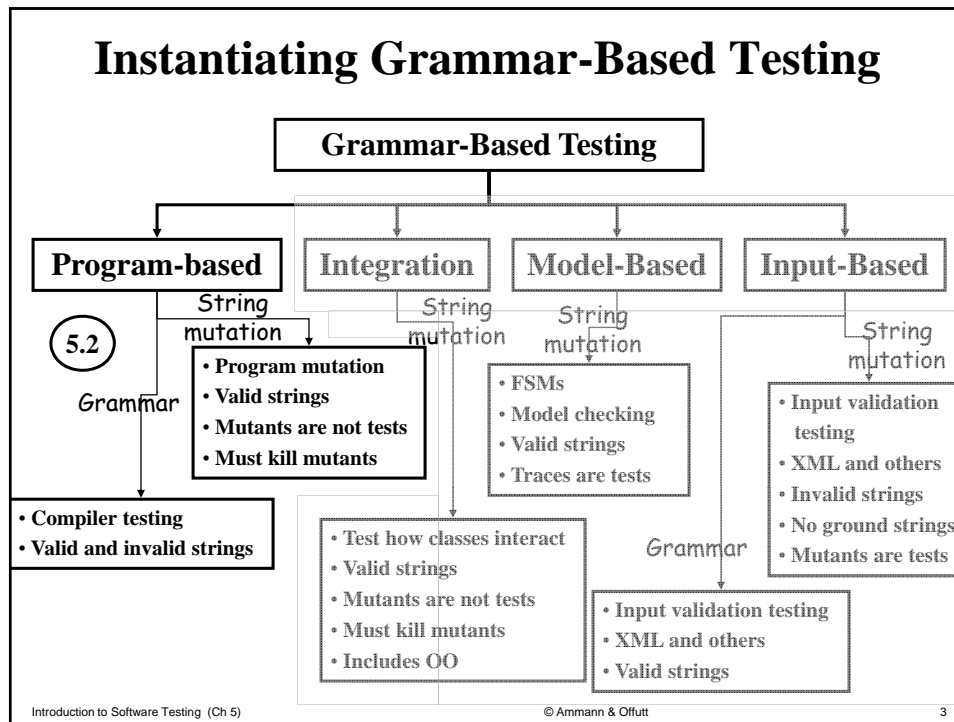
Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

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**

Instantiating Grammar-Based Testing



BNF Testing for Compilers (5.2.1)

- **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**
- **This is a very specialized application and not discussed in detail**

Program-based Grammars (5.2.2)

- 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 (valid strings)
- Mutants are not tests, but used to find 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
 - Stillborn mutant : Syntactically illegal
 - Trivial mutant : Almost every test can kill it
 - Equivalent mutant : No test can kill it (equivalent to original program)

Program-based Grammars

Original Method

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

6 mutants
Each represents a
separate program

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

Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each $m \in M$, 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 ...

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 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)
- In practice, there is little difference

Weak Mutation Example

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

```
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 : $true \wedge (A \neq B) \wedge ((B < A) = false)$**
 $\equiv (A \neq B) \wedge (B \geq A)$
 $\equiv (B > A)$
- **(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 “ $(B < A) \neq (B < \text{minVal})$ ”**
- **However, the previous statement was “ $\text{minVal} = A$ ”**
 - Substituting, we get: “ $(B < A) \neq (B < A)$ ”
- **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 (float) (X/2) == ((float) X) / 2.0
6          return (true);
7      else
8          return (false);
9  }
    
```

Reachability : $X < 0$

Infection : $X \neq 0$

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

Propagation :

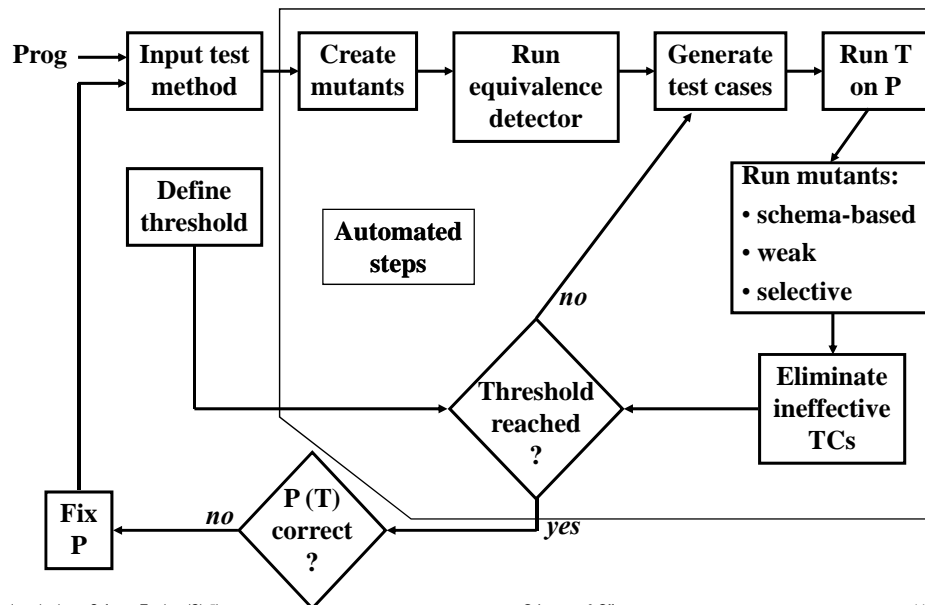
$((float) ((0-X)/2) == ((float) 0-X) / 2.0)$

$\neq ((float) (0/2) == ((float) 0) / 2.0)$

That is, X is not even ...

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

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

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o1, o2, \dots\}$ 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()*.

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.

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 actually only makes sense 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**