Introduction to Software Testing

Chapter 9

Syntax-based Testing

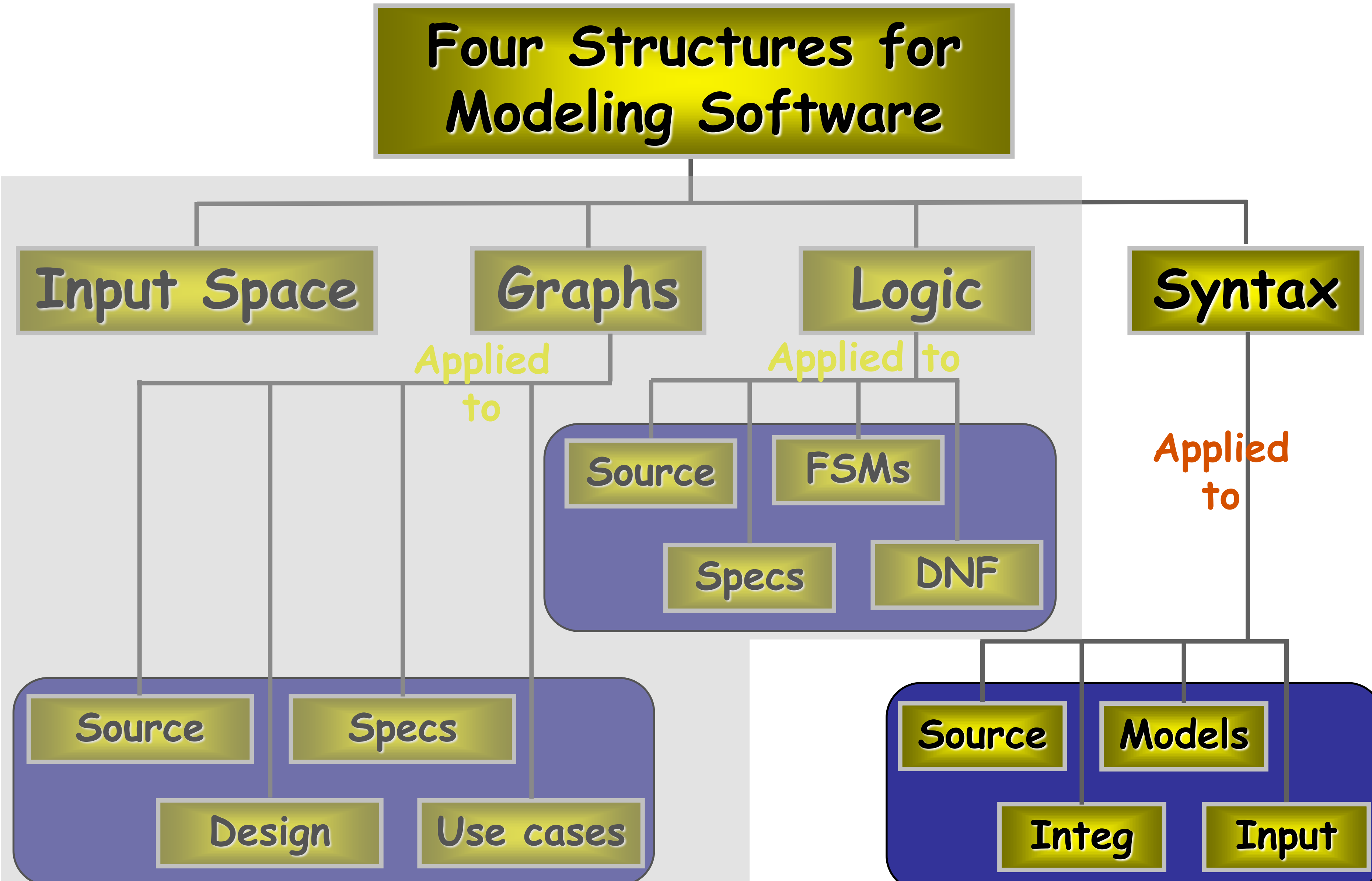
Instructor: Morteza Zakeri

Slides by: **Paul Ammann & Jeff Offutt**

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

Modified by: **Morteza Zakeri**

Ch. 9: Syntax Coverage



Using the Syntax to Generate Tests

- Lots of software artifacts follow strict syntax rules
- The **syntax** is often expressed as a grammar in a language such as BNF
- Syntactic descriptions can come from many sources
  - Programs
  - Integration elements
  - Design documents
  - Input descriptions
- Tests are created with two general goals
  - Cover the syntax in some way
  - Violate the syntax (invalid tests)

Grammar Coverage Criteria

- Software engineering makes practical use of automata theory in several ways
  - Programming languages defined in BNF
  - Program behavior described as finite state machines (FSMs)
  - Allowable inputs defined by grammars (e.g., PDF, HTML, …)
- A simple regular expression:

(*G s n* | *B t n*)*

'*' is *closure* operator, zero or more occurrences

'|' is *choice*, either one can be used

- Any sequence of "*G s n*" and "*B t n*"
- '*G*' and '*B*' could represent commands, methods, or events
- '*s*', '*t*', and '*n*' can represent arguments, parameters, or values
- '*s*', '*t*', and '*n*' could represent literals or a set of values

Test Cases from Grammar

- A string that satisfies the derivation rules is said to be "*in the grammar*"
- A test case is a sequence of strings that satisfy the regular expression
- Suppose 's', 't' and 'n' are numbers

G	26	08	01	90
B	22	06	27	94
G	22	11	21	94
B	13	01	09	03

Could be one test with four parts or four separate tests, etc.

BNF Grammars

```
Stream  ::=  action*
action  ::=  actG | actB
actG    ::=  "G" s n
actB    ::=  "B" t n
s       ::=  digit1-3
t       ::=  digit1-3
n       ::=  digit2 "." digit2 "." digit2
digit   ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

Start symbol

Non-terminals

Production rule

Terminals

Using Grammars

Stream ::= action action *
::= actG action*
::= G s n action*
::= G digit$^{1-3}$ digit^2 . digit^2 . digit^2 action*
::= G digitdigit digitdigit.digitdigit.digitdigit action*
::= G 25 08.01.90 action*
...

- Recognizer: Is a string (or test) in the grammar?
  - This is called parsing
  - Tools exist to support parsing
  - Programs can use them for input validation
- Generator: Given a grammar, derive strings in the grammar.

Mutation as Grammar-Based Testing

Grammar-based Testing

- **UnMutated Derivations** (*valid strings*)
- **Mutated Derivations** (*invalid strings?*)
  - **Grammar Mutation** (*invalid strings*)
  - **Ground String Mutation**
    - **Invalid Strings**
    - **Valid Strings**

Now we can define generic coverage criteria

Grammar-based Coverage Criteria

(9.1.1)

- The most common and straightforward criteria use every terminal and every production at least once.

<u>Terminal Symbol Coverage (TSC)</u>: TR contains each terminal symbol *t* in the grammar *G*.

<u>Production Coverage (PDC)</u>: TR contains each production *p* in the grammar *G*.

- PDC subsumes TSC
- Grammars and graphs are interchangeable
  - PDC is equivalent to EC, TSC is equivalent to NC
- Other graph-based coverage criteria could be defined on grammar
  - But have not

Grammar-based Coverage Criteria

- A related criterion is the impractical one of deriving all possible strings

Derivation Coverage (DC): TR contains every possible string that can be derived from the grammar *G*.

- The number of TSC tests is bound by the number of terminal symbols
  - 13 in the stream grammar
- The number of PDC tests is bound by the number of productions
  - 18 in the stream grammar
- The number of DC tests depends on the details of the grammar
  - 2,000,000,000 in the stream grammar!
- All TSC, PDC and DC tests are in the grammar … how about tests that are **NOT** in the grammar? Negative testing

Mutation Testing

(9.1.2)

- Grammars describe both valid and invalid strings
- Both types can be produced as mutants
- A mutant is a variation of a valid string
  - Mutants may be **valid** or **invalid** strings
- Mutation is based on "mutation operators" and "ground strings".

What is Mutation?

General View

mutation operators

We are performing mutation analysis whenever we

- use well defined rules
- defined on syntactic descriptions
- to make systematic changes
- to the syntax or to objects developed from the syntax

What is Mutation?

General View

mutation operators

We are performing mutation analysis whenever we

grammars

- use well defined rules
- defined on syntactic descriptions
- to make systematic changes
- to the syntax or to objects developed from the syntax

What is Mutation?

General View

We are performing mutation analysis whenever we

- use well defined rules
- defined on syntactic descriptions
- to make systematic changes
- to the syntax or to objects developed from the syntax

mutation operators

grammars

Applied universally or according to empirically verified distributions

What is Mutation?

General View

We are performing mutation analysis whenever we

- use well defined rules
- defined on syntactic descriptions
- to make systematic changes
- to the syntax or to objects developed from the syntax

mutation operators

grammars

Applied universally or according to empirically verified distributions

Grammar

Ground strings

(tests or programs)

Mutation Testing

- **Ground string:** A valid string in the grammar
  - The term "ground" is used as an analogy to algebraic ground terms
- **Mutation Operator:** A rule that specifies syntactic variations of strings generated from a grammar
- **Mutant:** The result of one application of a mutation operator
  - A mutant is a string either in the grammar or very close to being in the grammar.

Mutants and Ground Strings

- The key to mutation testing is the design of the mutation operators
  - Well designed operators lead to powerful testing
- Sometimes mutant strings are based on ground strings
- Sometimes they are derived directly from the grammar
  - Ground strings are used for valid tests
  - Invalid tests do not need ground strings

Mutants

Ground Strings	Valid Mutants	Invalid Mutants
G 26 08.01.90	B 26 08.01.90	7 26 08.01.90
B 22 06.27.94	B 45 06.27.94	B 22 06.27.1

Questions About Mutation

- Should more than one operator be applied at the same time?
  - Should a mutated string contain more than one mutated element?
  - Usually not – multiple mutations can interfere with each other
  - 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
- Mutation operators have been defined for many languages
  - 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*)

Killing Mutants

- When ground strings are mutated to create valid strings, the hope is to exhibit different behavior from the ground string.
  - This is normally used when the grammars are programming languages, the strings are programs, and the ground strings are pre-existing programs
- Killing Mutants: Given a mutant $m \in M$ for a derivation D and a test t, t is said to kill m if and only if the output of t on D is different from the output of t on m
- The derivation D may be represented by the list of productions or by the final string

Syntax-based Coverage Criteria

- Coverage is defined in terms of killing mutants

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

- Coverage in mutation equates to number of mutants killed.

- The amount of mutants killed is called the mutation score.

Syntax-based Coverage Criteria

- When creating invalid strings, we just apply the operators
- This results in two simple criteria
- It makes sense to either use every operator once or every production once

<u>Mutation Operator Coverage (MOC)</u>: For each mutation operator, TR contains exactly one requirement, to create a mutated string *m* that is derived using the mutation operator.

<u>Mutation Production Coverage (MPC)</u>: For each mutation operator, TR contains several requirements, to create one mutated string *m* that includes every production that can be mutated by that operator.

Example

Grammar

```
Stream  ::=  action*
action  ::=  actG | actB
actG    ::=  "G" s n
actB    ::=  "B" t n
s       ::=  digit1-3
t       ::=  digit1-3
n       ::=  digit2 "." digit2 "." digit2
digit   ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Ground String

G 25 08.01.90

B 21 06.27.94

Mutation Operators

- *Exchange actG and actB*
- *Replace digits with all other digits*

Mutants using MOC

B 25 08.01.90

B 23 06.27.94

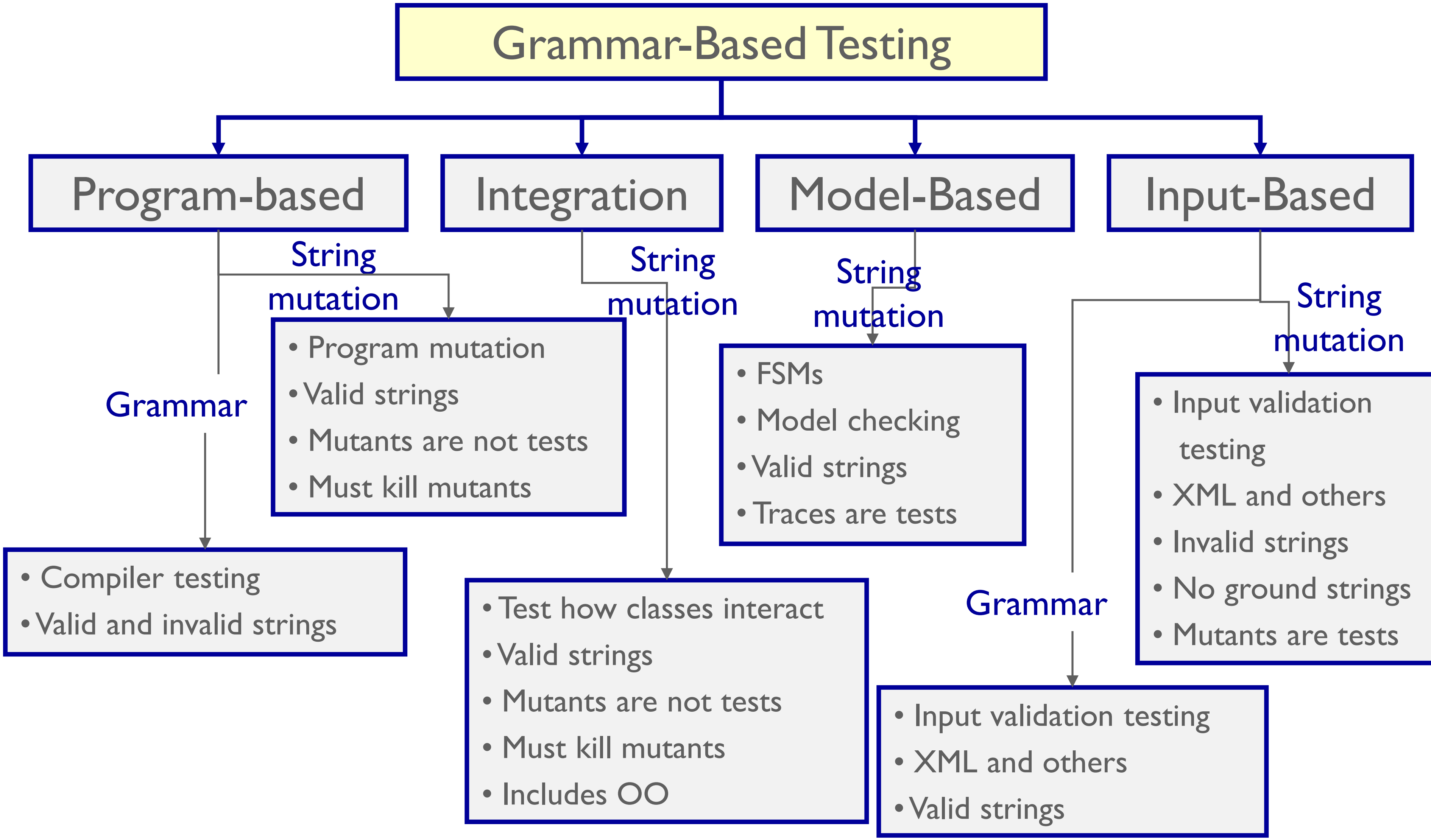
Mutants using MPC

B 25 08.01.90	*G 21 06.27.94*
G 15 08.01.90	*B 22 06.27.94*
G 35 08.01.90	*B 23 06.27.94*
G 45 08.01.90	*B 24 06.27.94*
…	…

Mutation Testing

- The number of test requirements for mutation depends on two things
  - The syntax of the artifact being mutated
  - The mutation operators
- Mutation testing is very difficult to apply by hand
- Mutation testing is very effective – considered the "gold standard" of testing
- Mutation testing is often used to **evaluate** other criteria

Instantiating Grammar-Based Testing



Structure of Chapter

	Program-based	Integration	Model-based	Input space
Grammar	9.2.1	9.3.1	9.4.1	9.5.1
Grammar	Programming languages	No known applications	Algebraic specifications	Input languages, including XML
Summary	Compiler testing			Input space testing
Valid?	Valid & invalid			Valid
Mutation	9.2.2	9.3.2	9.4.2	9.5.2
Grammar	Programming languages	Programming languages	FSMs	Input languages, including XML
Summary	Mutates programs	Tests integration	Model checking	Error checking
Ground?	Yes	Yes	Yes	No
Valid?	Yes, must compile	Yes, must compile	Yes	No
Tests?	Mutants not tests	Mutants not tests	Traces are tests	Mutants are tests
Killing	Yes	Yes	Yes	No
Notes	Strong and weak. Subsumes other techniques	Includes OO testing		Sometimes the grammar is mutated

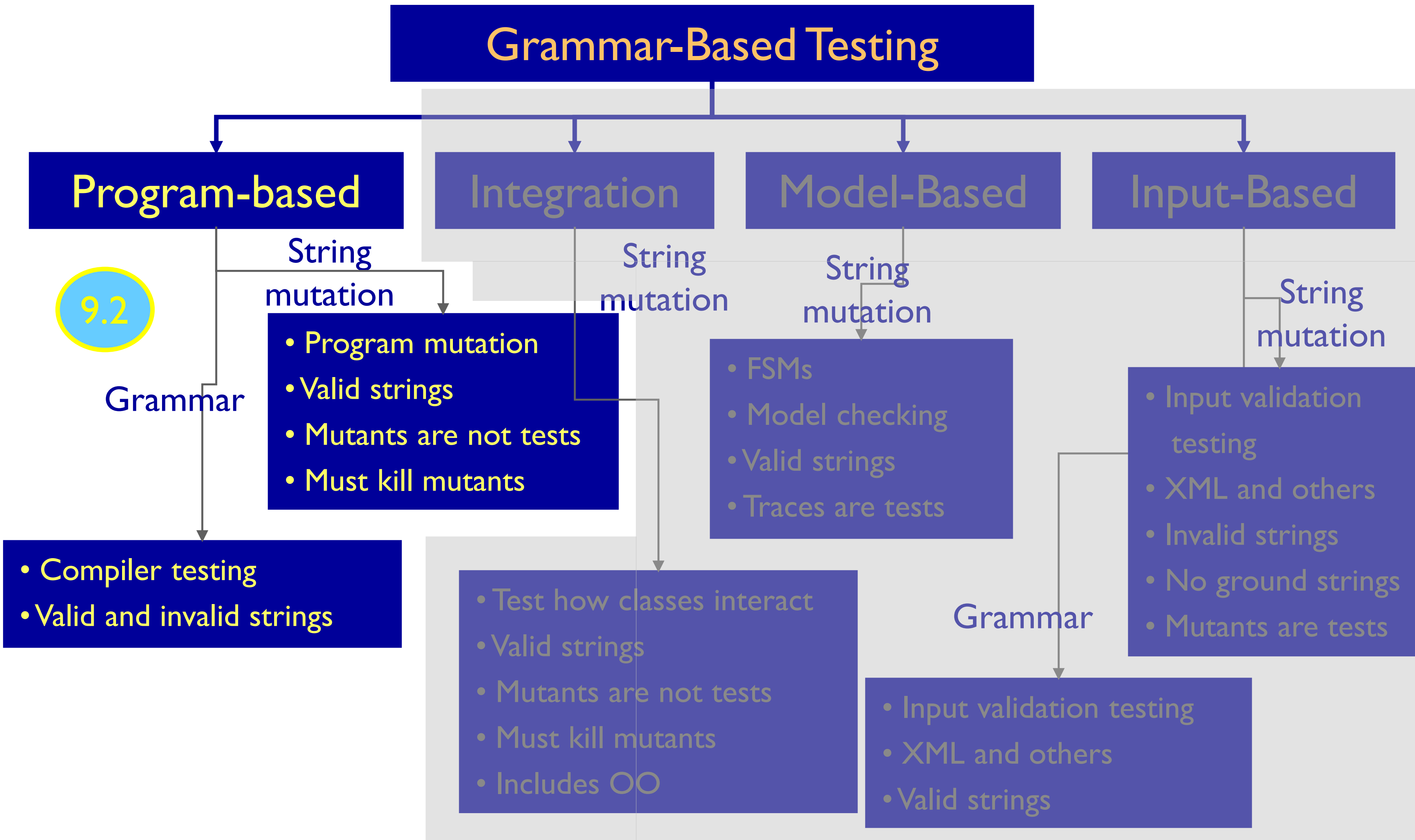
Introduction to Software Testing
Chapter 9

Section 9.2
Program-based Grammars

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 (9.2.1)

- 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 (9.2.2)

- 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 <u>kill</u> 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 have been killed
  - *Dead (killed) 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 $m \in M$, 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)
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants

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 : $true \wedge (A \neq B) \wedge ((B < A) = false)$
  $\equiv (A \neq B) \wedge (B \geq A)$
  $\equiv (B > A)$

- Weakly kill mutant 1, but not strongly? A = 5, B = 3

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)"
  - This is a logical contradiction !

- 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 (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__ :

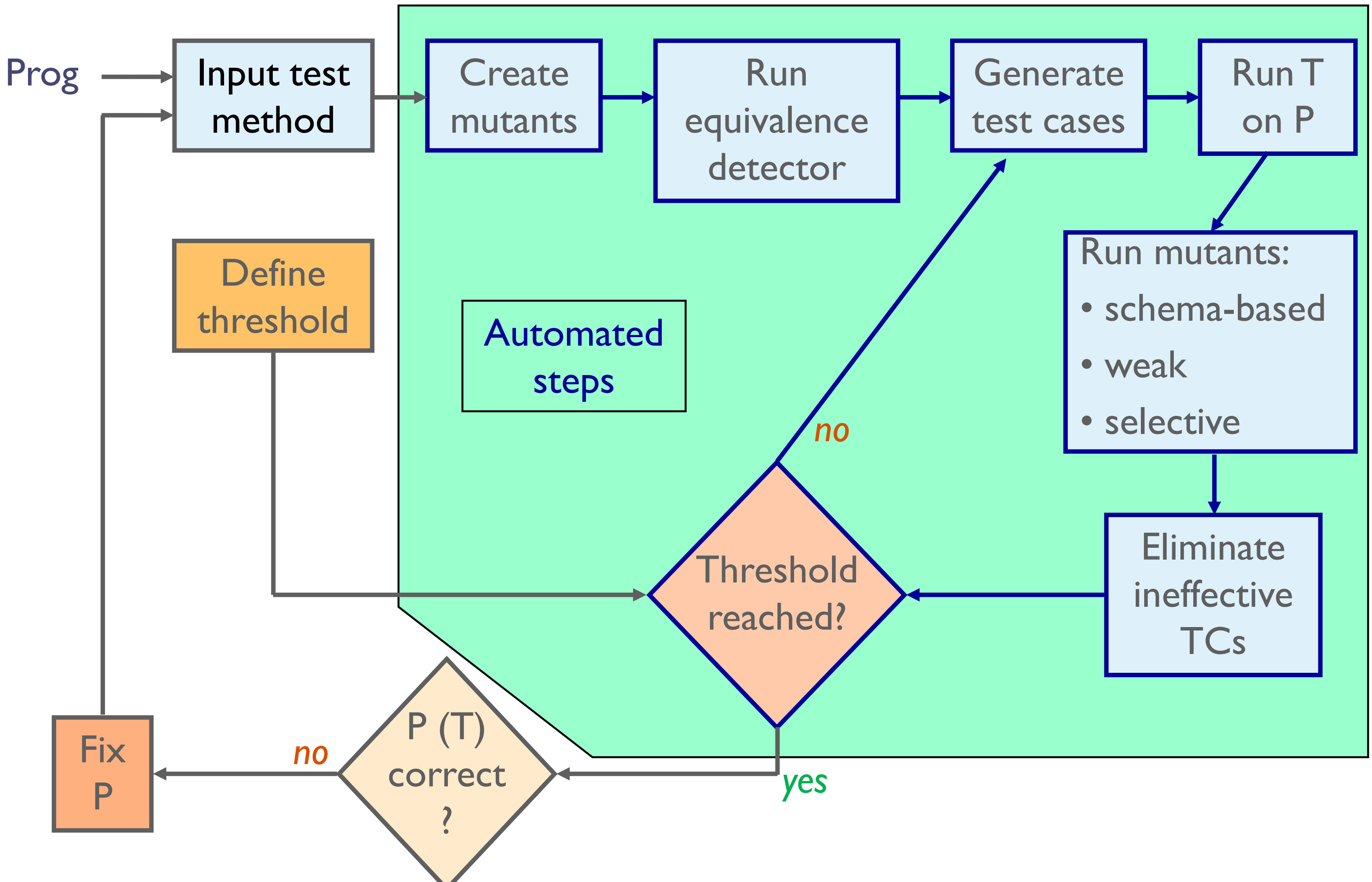
((double) ((0-X)/2) == ((double) 0-X) / 2.0)

!= ((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



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);
Δ1    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