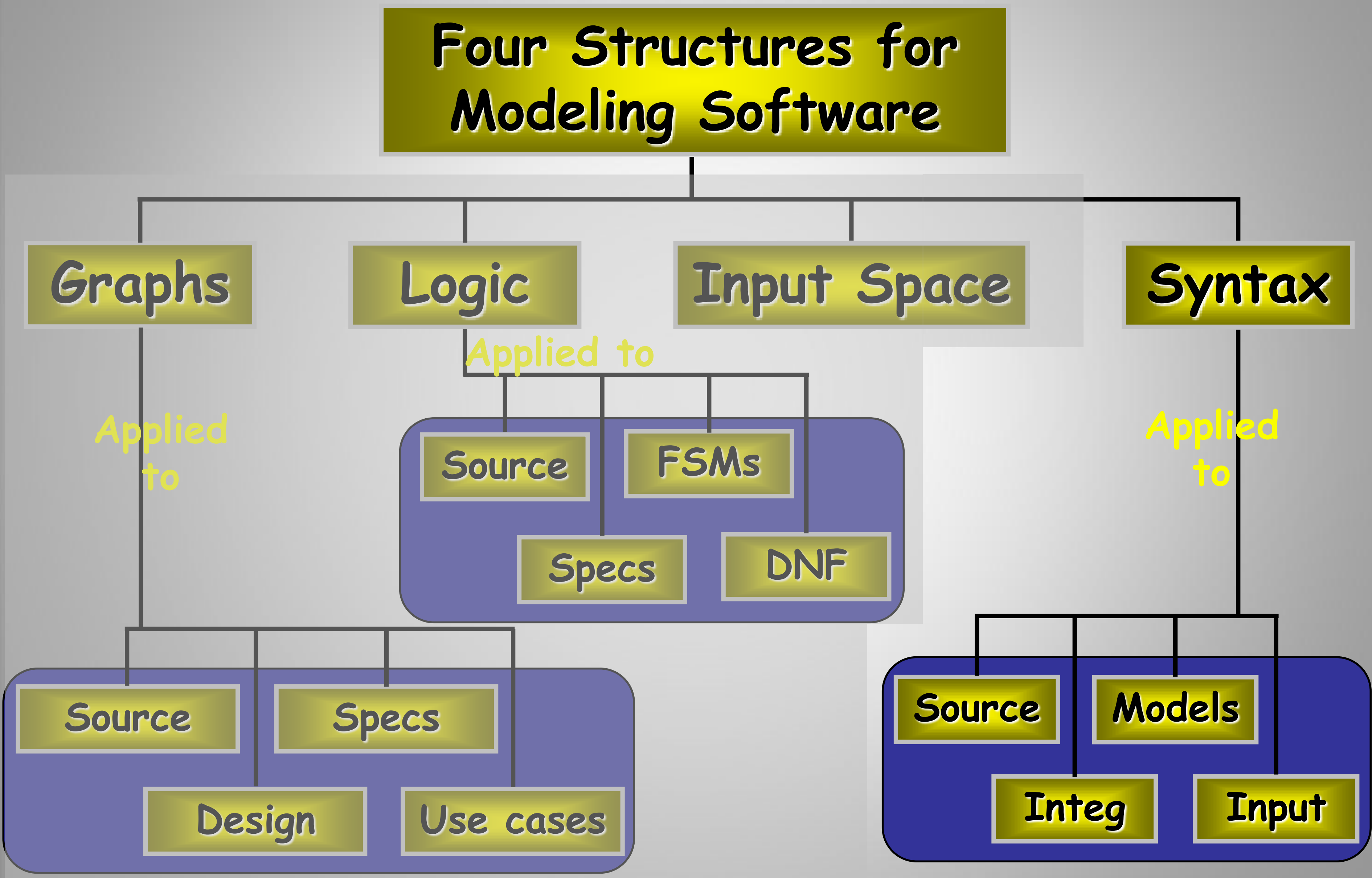
Introduction to Software Testing

Chapter 5.1
Syntax-based Testing

Paul Ammann & Jeff Offutt

Updated by Sunae Shin

Ch. 5 : 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**
  - **Allowable inputs defined by grammars**

- **Backus normal form (BNF) is a notation technique for context-free grammars**
  - **Often used to describe the syntax of languages used in computing**
    - **such as computer programming languages, document formats, instruction sets and communication protocols.**

Grammar Coverage Criteria

- **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 be commands, methods, or events**
  - **'*s*', '*t*', and '*n*' could represent arguments, parameters, or values**
  - **'*s*', '*t*', and '*n*' could be literals or a set of values**

Test Cases from Grammar

- **A test case is a sequence of strings that satisfy the regular expression**
  - **Suppose 's', 't' and 'n' are numbers**

G 23 08 01 90

B 19 06 27 94

G 18 11 21 94

B 10 01 09 03

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

BNF Grammars

- A more expressive grammar is often used
- The prior example can be refined into a grammar form as follows:

```
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

Terminals

The symbols on the left of the ::= sign are all nonterminals

Rewriting of a given nonterminal is called production

Numeric range (a-b) means there has to be at least a repetitions, and no more than b

Using Grammars

Stream ::= action action *

::= actG action*

::= G s n action*

::= G $\text{digit}^{1\text{-}3}$ digit^2 . digit^2 . digit^2 action*

::= G digitdigit digitdigit.digitdigit.digitdigit action*

::= G 20 08.01.90 action*

…

- **Grammers can be used in two ways: recognizer and generator**
  - **Recognizer : Given a string (or test), is the string 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**

Syntax-based Coverage Criteria

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

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

Production Coverage (PDC) : 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**

Syntax-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*.

Syntax-based Coverage Criteria

```
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"
```

- The number of TSC tests is bound by the number of terminal symbols
  - 13 in the stream grammar

    (G, B, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- The number of PDC tests is bound by the number of productions
  - 18 in the stream grammar

    (note the '|' symbol adds productions)

- The number of DC tests depends on the details of the grammar
  - 2,000,000,000 in the stream grammar (generally can be infinite)
- All TSC, PDC and DC tests are in the grammar … how about tests that are NOT in the grammar ?

Syntax-based Coverage Criteria

```
bank    ::= action*
action  ::= dep | deb
dep     ::= "deposit" account amount
deb     ::= "debit" account amount
account ::= digit^3
amount  ::= "$" digit^5 "." digit^2
digit   ::= "0" | "1" | "2" | "3" | "4" |
            "5" | "6" | "7" | "8" | "9"
```

- **BNF syntax example for bank transactions:**
  - **Terminal Symbol Coverage (TSC): The set of test requirements, TR, contains each terminal symbol t in the grammar G - 14 in bank example**
  - **Production Coverage (PC): TR contains each production p in the grammar G - 17 in bank example**

Syntax-based Coverage Criteria

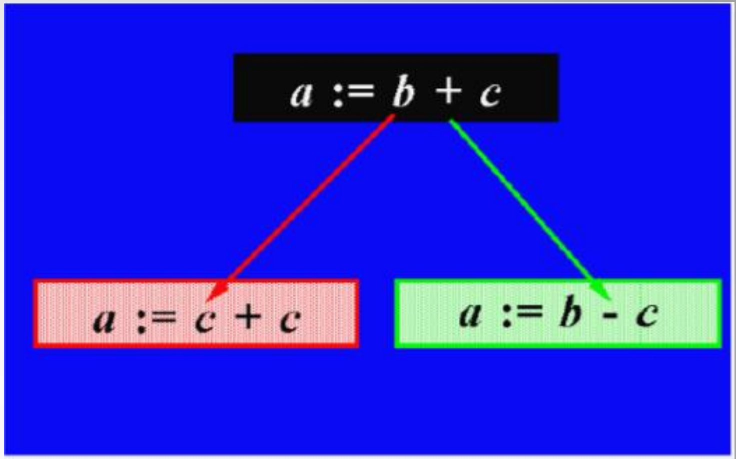
bank -> action*
-> **action action***
-> **dep** action*
-> **deposit account amount** action*
-> deposit **digit**^{3} amount action*
-> deposit **digit digit**^{2} amount action*
-> deposit **7** digit^2 amount action*
-> deposit 7 **digit digit** amount action*
-> deposit 7**3** digit amount action*
-> deposit 73**9** amount action*
-> deposit 739 **$ digit**^{5} **. digit**^{2} action*
-> deposit 739 $**digit**^{2} .digit^2 action*
-> deposit 739 $**digit digit**.digit^2 action*
-> deposit 739 $**1** digit.digit^2 action*
-> deposit 739 $1**2**. digit^2 action*
-> deposit 739 $12. **digit digit** action*
-> deposit 739 $12.**3 digit** action*
-> deposit 739 $12.3**5** action*

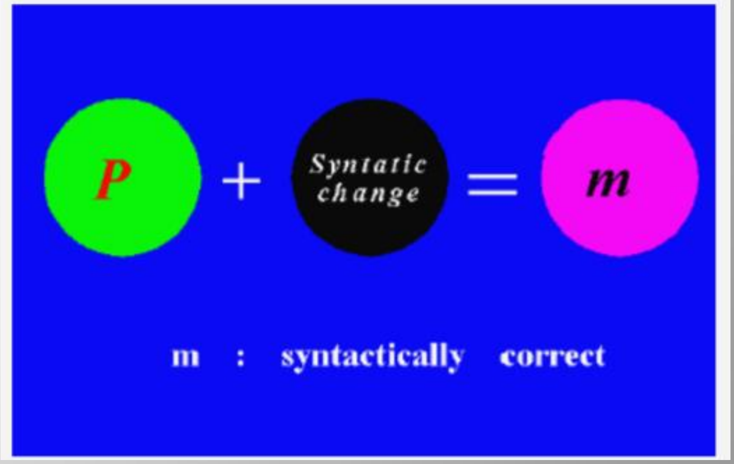
Mutation Testing

- **We say that an input is *valid* if it is in the language specified by the grammar, and *invalid* otherwise.**
  - **It is common to require a program to reject malformed inputs – should clearly be tested**
  - **Grammars describe both valid and invalid strings (useful to produce invalid strings from grammar)**

Mutation Testing

- **Basic idea:**
  - **Take a program and test data generated for that program**
  - **Create a number of similar programs (mutants), each differing from the original in one small way**
    - **e.g., replace addition operator by multiplication operator**
  - **The original test data are then run through the mutants**





Mutation Testing

- **A mutant is a variation of a valid string**
  - **Mutants may be valid or invalid strings**

- **Mutation is based on a set of "mutation operators" and "ground strings"**

1. **Ground string: A string in the grammar**
   - **The term "ground" is used as a reference to algebraic ground terms**
2. **Mutation Operator : A rule that specifies syntactic variations of strings generated from a grammar**
3. **Mutant : The result of one application of a mutation operator**
   - **A mutant is a string**

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 <u>based on ground strings</u>**
- **Sometimes they are derived <u>directly from the grammar</u>**
  - **Ground strings are used for valid tests**

<u>Valid Mutants</u>

<u>Ground Strings</u>	<u>Mutants</u>
G 23 08.01.90	*B 23 08.01.90*
B 19 06.27.94	*B 45 06.27.94*

<u>Invalid Mutants</u>

7 23 08.01.90
B 19 06.27

Mutation Operators - example

- **Conditionals Boundary Mutator**
  - **The conditionals boundary mutator replaces the relational operators <, <=, >, >=**
  - **with their boundary counterpart as per the table below.**

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

Mutation Operators - example

- **Negate Conditionals Mutator**
  - **The negate conditionals mutator will mutate all conditionals found according to the replacement table below.**

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

Mutation Operators - example

Original program

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

A mutant

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index>=10)
        break;
}
```

Questions About Mutation

- **Should more than one operator be applied at the same time ?**
  - **Should a mutated string contain one mutated element or several?**
  - **Usually not – multiple mutations can interfere with each other**
  - **Mutation only for one element at a time**

- **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*)**

Killing Mutants

- **When ground strings are mutated to create valid strings, the hope is to exhibit different behavior from the ground string**

- **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**

Killing Mutants

- Faults (or mutations) are seeded into your code, then your tests are run.
- If your tests fail then the mutation is killed, if your tests pass then the mutation lived.
- The quality of your tests can be gauged from the percentage of mutations killed.

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 23 08.01.90

B 19 06.27.94

Mutation Operators

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

Mutants using MOC

B 23 08.01.90

B 15 06.27.94

Mutants using MPC

B 22 08.01.90	*G 19 06.27.94*
G 13 08.01.90	*B 11 06.27.94*
G 33 08.01.90	*B 12 06.27.94*
G 43 08.01.90	*B 13 06.27.94*
G 53 08.01.90	*B 14 06.27.94*
…	…

Mutation Testing - Summary

- **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**