

# Introduction to Software Testing

## Chapter 5.1

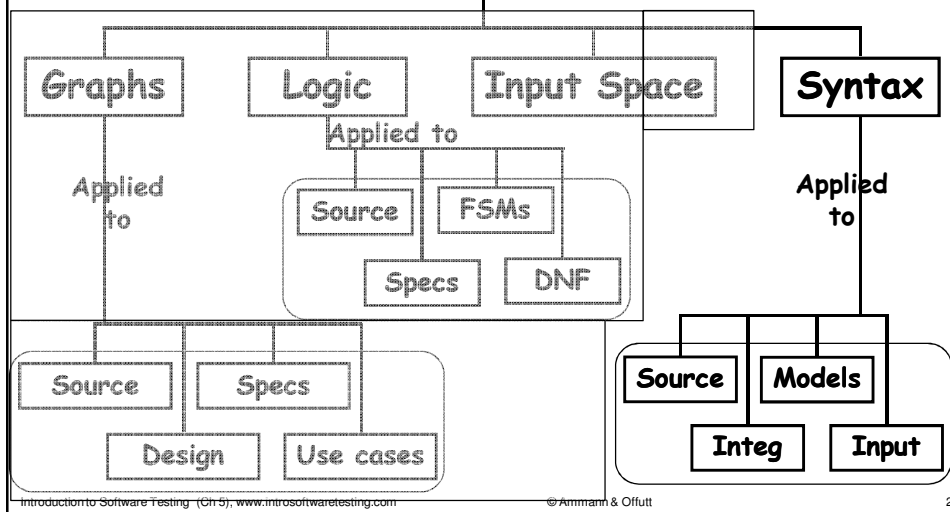
### Syntax-based Testing

Paul Ammann & Jeff Offutt

[www.introsoftwaretesting.com](http://www.introsoftwaretesting.com)

## Ch. 5 : Syntax Coverage

### Four Structures for Modeling Software

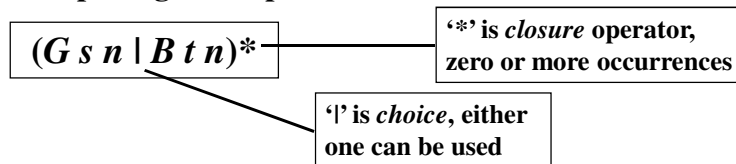


## Using the Syntax to Generate Tests (5.1)

- Lots of software artifacts follow strict syntax rules
- The syntax is often expressed as some sort of grammar 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
- A simple regular expression:



- 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 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 17 08.01.90***

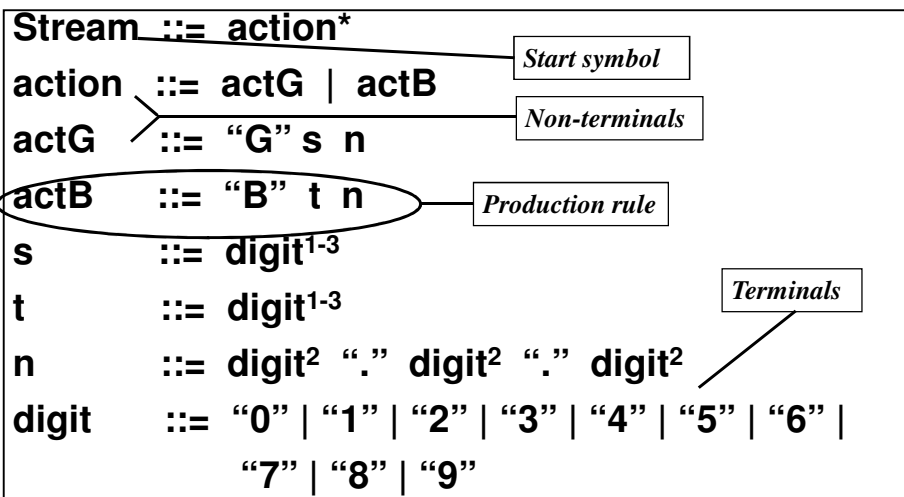
***B 13 06.27.94***

***G 12 11.21.94***

***B 04 01.09.03***

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

## BNF Grammars



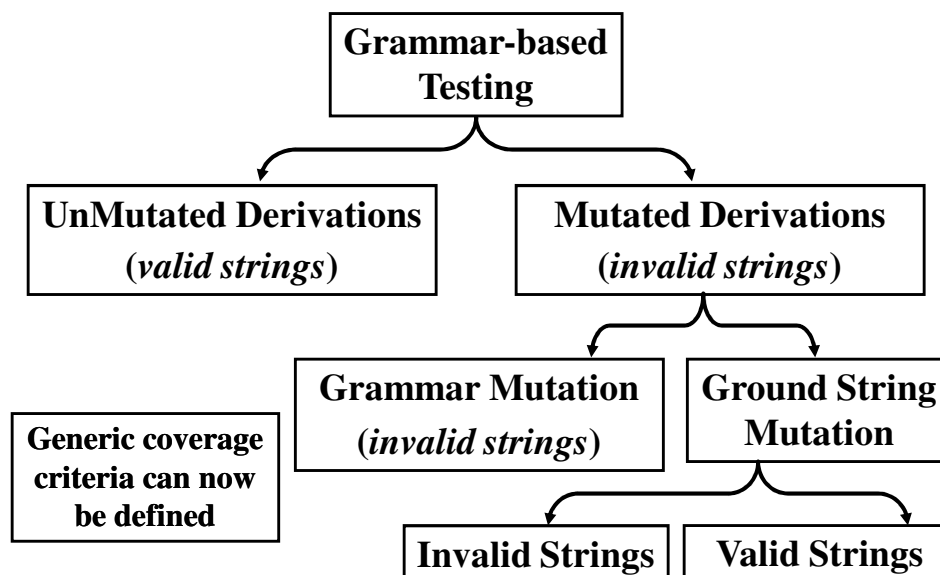
## Using Grammars

```

Stream ::= action action *
       ::= actG action*
       ::= G s n action*
       ::= G digit1-3 digit2 . digit2 . digit2 action*
       ::= G digitdigit digitdigit.digitdigit.digitdigit action*
       ::= G 16 08.01.90 action*
       ...
    
```

- **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

## Mutation as Grammar-Based Testing



## 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 (PC) : TR contains each production  $p$  in the grammar  $G$ .**

- PC subsumes TSC
- Grammars and graphs are interchangeable
- Other graph-based coverage criteria could be defined on grammar
  - But have not

## Syntax-based Coverage Criteria

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

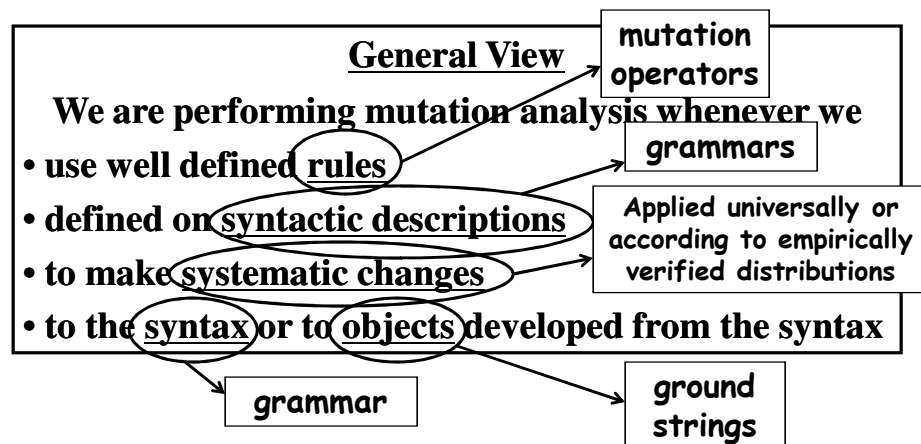
**Derivation Coverage (DC) : TR contains every possible string that can be derived from the grammar  $G$ .**

- The number of TSC tests is bound by the number of terminal symbols
  - 13 in the stream grammar
- The number of PC 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, PC and DC tests are in the grammar ... how about tests that are NOT in the grammar ?

# Mutation Testing

- 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 ?



## Mutation Testing

- **Ground string**: A string in the grammar
  - The term “ground” is used as a reference 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

## 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

### Valid Mutants

<u>Ground Strings</u>	<u>Mutants</u>
<i>G 17 08.01.90</i>	<i>B 17 08.01.90</i>
<i>B 13 06.27.94</i>	<i>B 45 06.27.94</i>

### Invalid Mutants

<i>13 17 08.01.90</i>
<i>B 13 06.27</i>

## Questions About Mutation

- **Should more than one operator be applied at the same time ?**
  - Should a mutated string contain one mutated element or several?
  - Almost certainly not – multiple mutations can interfere with each other
  - Extensive experience with program-based mutation indicates not
- **Should every possible application of a mutation operator be considered ?**
  - Necessary with program-based mutation
- **Mutation operators exist 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
- 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.

**Mutation Coverage (MC) : 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

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

**Mutation Production Coverage (MPC) : 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

```
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"
```

Grammar

### Ground String

*G 17 08.01.90*

*B 13 06.27.94*

### Mutation Operators

• *Exchange actG and actB*

• *Replace digits with other digits*

### Mutants using MOC

*B 17 08.01.90*

*B 19 06.27.94*

### Mutants using MPC

*B 17 08.01.90 G 13 06.27.94*

*G 27 08.01.90 B 11 06.27.94*

*G 37 08.01.90 B 14 06.27.94*

*G 47 08.01.90 B 15 06.27.94*

*G 57 08.01.90 B 16 06.27.94*

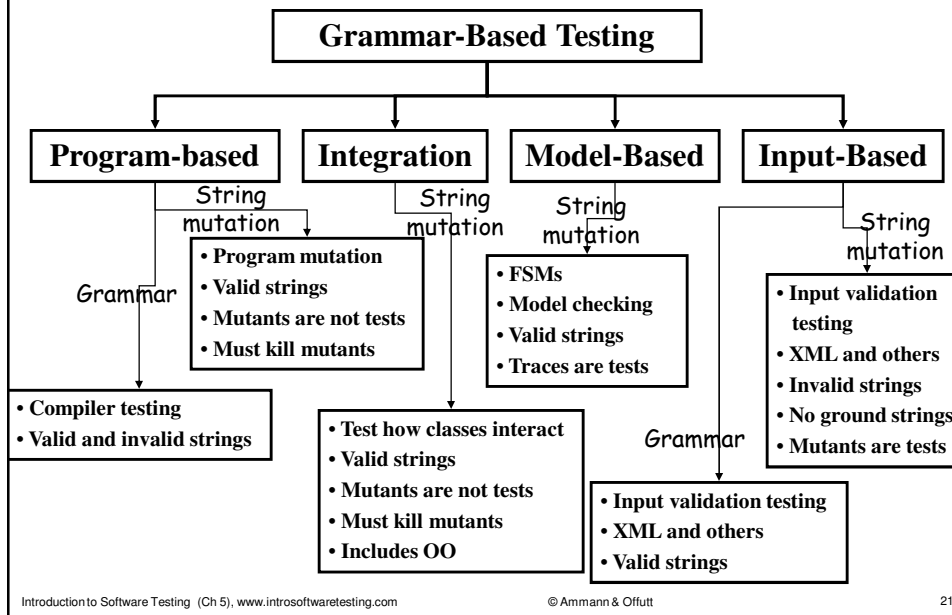
...

...

## 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
<b>Grammar</b>	5.2.1	5.3.1	5.4.1	5.5.1
Grammar	Programming languages	No known applications	Algebraic specifications	Input languages, including XML
Summary	Compiler testing			Input space testing
Valid?	Valid & invalid			Valid
<b>Mutation</b>	5.2.2	5.3.2	5.4.2	5.5.2
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 (Ch 5), www.introsoftwaretesting.com © Ammann & Offutt 22