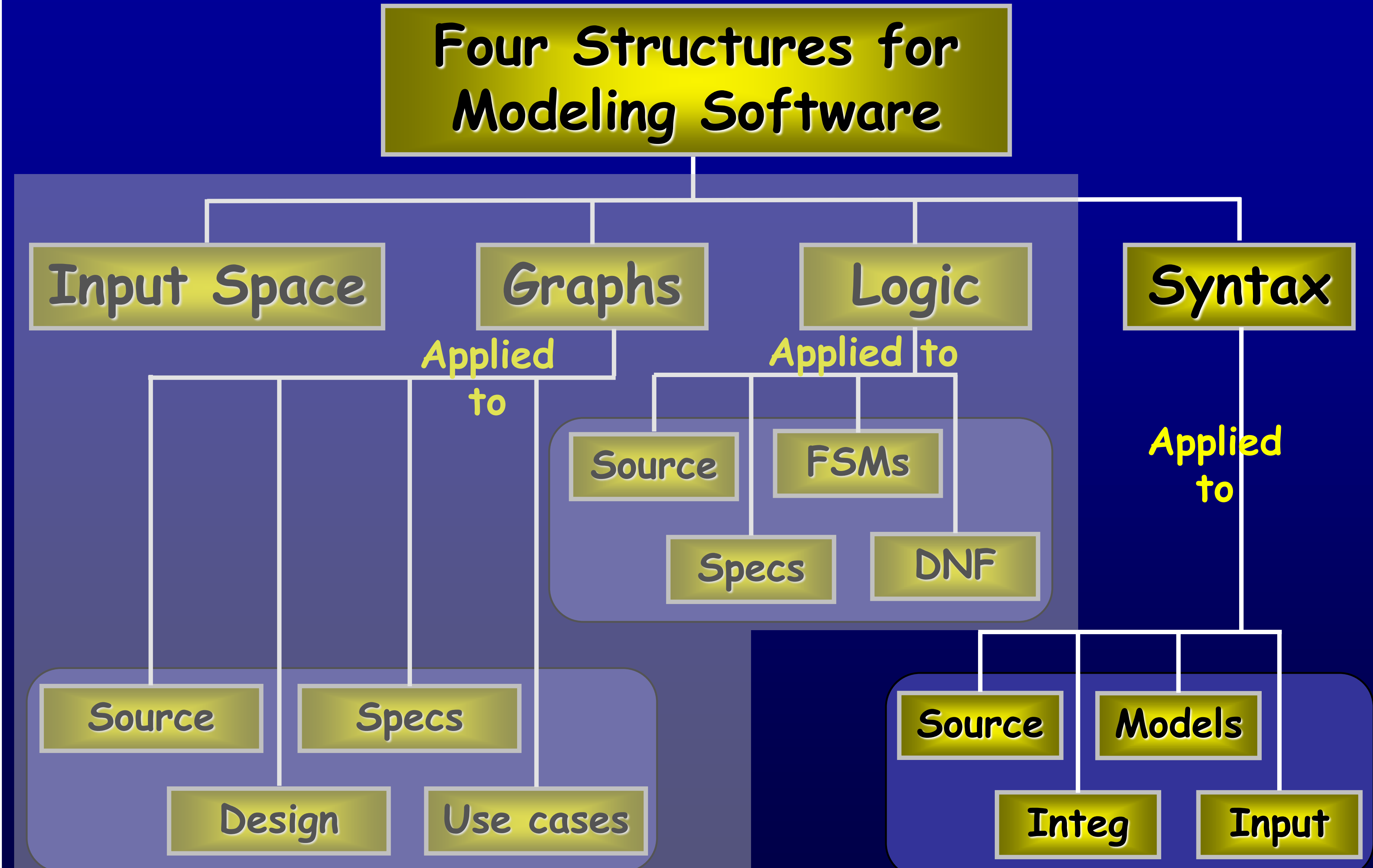
Introduction to Software Testing
Chapter 9.1
Syntax-based Testing

Paul Ammann & Jeff Offutt

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

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 <u>automata theory</u> in several ways
  - Programming languages defined in BNF
  - Program behavior described as finite state machines
  - Allowable inputs defined by grammars
- 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 25 08 01 90

B 21 06 27 94

G 21 11 21 94

B 12 01 09 03

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

BNF Grammars

Stream ::= action*

Start symbol

action ::= actG | actB

Non-terminals

actG ::= "G" s n

actB ::= "B" t n

Production rule

s ::= $digit^{1-3}$

t ::= $digit^{1-3}$

Terminals

n ::= $digit^2$ "." $digit^2$ "." $digit^2$

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

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 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 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
- 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 ?

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 <u>valid</u> or <u>invalid</u> strings
- Mutation is based on "mutation operators" and "ground strings"

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

Valid Mutants

Ground Strings	Mutants
G 26 08.01.90	*B 26 08.01.90*
B 22 06.27.94	*B 45 06.27.94*

Invalid Mutants

7 26 08.01.90

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

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

Grammar

```
Stream  ::=  action*
action  ::=  actG  |  actB
actG    ::=  "G" s  n
actB    ::=  "B"  t  n
s       ::=  digit^(1-3)
t       ::=  digit^(1-3)
n       ::=  digit^2  "."  digit^2  "."  digit^2
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

Grammar-Based Testing

Program-based

String mutation

- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants

Grammar

- Compiler testing
- Valid and invalid strings

Integration

String mutation

- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

Model-Based

String mutation

- FSMs
- Model checking
- Valid strings
- Traces are tests

Input-Based

String mutation

- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests

Grammar

- Input validation testing
- XML and others
- Valid strings

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