Ch. 5 : Syntax Coverage

Four Structures for Modeling Software

Graphs  Logic  Input Space  Syntax

Applied to

Source  FSMs  Specs  bNF

Design  Use cases

Source  Models

Integ  Input
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:
  \[(G \text{ s n} | B \text{ t n})^*\]
  ‘\*' 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 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

```
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”
```

Start symbol

Non-terminals

Production rule

Terminals
Using Grammars

- **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

\[
\text{Stream ::= action action *}
\]
\[
::= \text{actG action*}
\]
\[
::= \text{G s n action*}
\]
\[
::= \text{G digit}^{1-3} \text{ digit}^2 \cdot \text{ digit}^2 \cdot \text{ digit}^2 \text{ action*}
\]
\[
::= \text{G digitdigit digitdigit.digitdigit.digitdigit.digitdigit.digitdigit.action*}
\]
\[
::= \text{G 16 08.01.90 action*}
\]

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**

Generic coverage criteria can now be defined
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?

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
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

<table>
<thead>
<tr>
<th>Valid Mutants</th>
<th>Invalid Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ground Strings</strong></td>
<td><strong>Mutants</strong></td>
</tr>
<tr>
<td>G 17 08.01.90</td>
<td>B 17 08.01.90</td>
</tr>
<tr>
<td>B 13 06.27.94</td>
<td>B 45 06.27.94</td>
</tr>
<tr>
<td>B 13 06.27</td>
<td>B 13 06.27</td>
</tr>
<tr>
<td>G 17 08.01.90</td>
<td>B 17 08.01.90</td>
</tr>
</tbody>
</table>
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

**Grammar**

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

**Ground String**

```
G 17 08.01.90
B 13 06.27.94
```

**Mutant 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**

**Grammar-Based Testing**

- **Program-based**
  - String mutation
  - • Program mutation
  - • Valid strings
  - • Mutants are not tests
  - • Must kill mutants
  - • 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
  - • Algebraic specifications
  - • Input languages, including XML

- **Input-Based**
  - String mutation
  - • Input validation testing
  - • XML and others
  - • Invalid strings
  - • No ground strings
  - • Mutants are tests

---

**Structure of Chapter**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Program-based</th>
<th>Integration</th>
<th>Model-based</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>5.2.1</td>
<td>5.3.1</td>
<td>5.4.1</td>
<td>5.5.1</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Program-based</th>
<th>Integration</th>
<th>Model-based</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar</td>
<td>Programming languages</td>
<td>No known applications</td>
<td>Algebraic specifications</td>
<td>Input languages, including XML</td>
</tr>
<tr>
<td>Summary</td>
<td>Compiler testing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid?</td>
<td>Valid &amp; invalid</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Program-based</th>
<th>Integration</th>
<th>Model-based</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>5.2.2</td>
<td>5.3.2</td>
<td>5.4.2</td>
<td>5.5.2</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Program-based</th>
<th>Integration</th>
<th>Model-based</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar</td>
<td>Programming languages</td>
<td>Programming languages</td>
<td>FSMs</td>
<td>Input languages, including XML</td>
</tr>
<tr>
<td>Summary</td>
<td>Mutates programs</td>
<td>Tests integration</td>
<td>Model checking</td>
<td>Error checking</td>
</tr>
<tr>
<td>Ground?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Valid?</td>
<td>Yes, must compile</td>
<td>Yes, must compile</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Tests?</td>
<td>Mutants not tests</td>
<td>Mutants not tests</td>
<td>Traces are tests</td>
<td>Mutants are tests</td>
</tr>
<tr>
<td>Killing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Notes</td>
<td>Strong and weak. Subsumes other testing techniques</td>
<td>Includes OO testing</td>
<td></td>
<td>Sometimes the grammar is mutated</td>
</tr>
</tbody>
</table>