Ch. 5: Syntax Coverage

Four Structures for Modeling Software

Graphs

Logic

Input Space

Syntax

Applied to

Source

Specs

DNF

Applied to

Source

Specs

Design

Use cases

Applied to

Source

Models

Integ

Input

Ch. 5: Syntax Coverage

Introduction to Software Testing (Ch 5)
Using the Syntax to Generate Tests (5.1)

- Lots of software artifacts follow strict syntax rules
- The syntax is often expressed as a grammar in a language such as BNF (Backus-Naur Form)
- **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:

\[(G\ s\ n\ |\ B\ t\ n)^*\]

- “*” 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

Make a fair example of context-free language
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

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”
Using Grammars

Stream -> action *
   -> action  action *
   -> actG action*
   -> G s n action*
   -> G digit\(^1\) digit\(^2\) . digit\(^2\) . digit\(^2\) action*
   -> G digit digit digit digit digit digit digit digit digit action*
   -> G 20 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

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
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
- 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 **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 for just one repetition!

- All TSC, PDC 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?

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>Ground Strings</td>
<td>Mutants</td>
</tr>
<tr>
<td>G 22 08.01.90</td>
<td>B 22 08.01.90</td>
</tr>
<tr>
<td>B 18 06.27.94</td>
<td>B 45 06.27.94</td>
</tr>
<tr>
<td>7 22 08.01.90</td>
<td>B 18 06.27.1</td>
</tr>
</tbody>
</table>
Questions About Mutation

- Should more than one operator be applied at the same time?
  - Or, should a mutated string contain one mutated element or several?
  - Usually not – multiple mutations can interfere with each other
  - Extensive 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 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)
2013/05/14 stopped here.
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 ::= digit{1-3}
t ::= digit{1-3}
n ::= digit2 “.” digit2 “.” digit2
digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

Ground String

G 22 08.01.90
B 18 06.27.94

Grammar

Mutation Operators

• Exchange actG and actB
• Replace digits with all other digits

Mutants using MPC

B 22 08.01.90  G 18 06.27.94
G 12 08.01.90  B 11 06.27.94
G 32 08.01.90  B 12 06.27.94
G 42 08.01.90  B 13 06.27.94
G 52 08.01.90  B 14 06.27.94
...
...

Mutants using MOC

B 22 08.01.90
B 19 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
- Compiler testing
- Valid and invalid strings
- String mutation
  - Program mutation
  - Valid strings
  - Mutants are not tests
  - Must kill mutants

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

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

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

String mutation
Grammar
## Structure of Chapter

<table>
<thead>
<tr>
<th></th>
<th>Program-based</th>
<th>Integration</th>
<th>Model-based</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grammar</strong></td>
<td>5.2.1</td>
<td>5.3.1</td>
<td>5.4.1</td>
<td>5.5.1</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>Compiler testing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Valid?</strong></td>
<td>Valid &amp; invalid</td>
<td></td>
<td></td>
<td>Valid</td>
</tr>
<tr>
<td><strong>Mutation</strong></td>
<td>5.2.2</td>
<td>5.3.2</td>
<td>5.4.2</td>
<td>5.5.2</td>
</tr>
<tr>
<td><strong>Grammar</strong></td>
<td>Programming languages</td>
<td>Programming languages</td>
<td>FSMs</td>
<td>Input languages, including XML</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>Mutates programs</td>
<td>Tests integration</td>
<td>Model checking</td>
<td>Error checking</td>
</tr>
<tr>
<td><strong>Ground?</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Valid?</strong></td>
<td>Yes, must compile</td>
<td>Yes, must compile</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Tests?</strong></td>
<td>Mutants not tests</td>
<td>Mutants not tests</td>
<td>Traces are tests</td>
<td>Mutants are tests</td>
</tr>
<tr>
<td><strong>Killing</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Notes</strong></td>
<td>Strong and weak. Subsumes other techniques</td>
<td>Includes OO testing</td>
<td>Sometimes the grammar is mutated</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5.2
Program-based Grammars

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/
Applying Syntax-based Testing to Programs

- Syntax-based criteria *originated* with programs and have been used most 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

Grammar-Based Testing

Program-based
- String mutation
- Compiler testing
- Valid and invalid strings
- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants
- Integration
- Model-Based
- Input-Based

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
BNF Testing for Compilers (5.2.1)

• 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 (5.2.2)

- 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 kill $m$ 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 goals.
- Testers can keep adding tests until all mutants have been killed.
  - **Dead 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 (equivalent to original program).
### Program-based Grammars

#### Original Method

```c
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

#### With Embedded Mutants

```c
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    ∆1  minVal = B;
    if (B < A)
    ∆2  if (B > A)
    ∆3  if (B < minVal)
    {   ∆4          Bomb ();
        ∆5          minVal = A;
        ∆6          minVal = failOnZero (B);
    }
    return (minVal);
} // end Min
```

6 mutants
Each represents a separate program

- Replace one variable with another
- Changes operator
- Immediate runtime failure … if reached
- Immediate runtime failure if B==0 else does nothing
Syntax-Based Coverage Criteria

**Mutation Coverage (MC)**: For each \( m \in M \), TR contains exactly one requirement, to kill \( m \).

- The RIP model from chapter 1:
  - **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
- The RIP 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 \( m \) immediately on \( t \) after \( l \).

- Weakly killing satisfies *reachability* and *infection*, but not *propagation*.
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.
- Some mutants can be killed under weak mutation but not under strong mutation (no propagation).
- In practice, there is little difference.
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 ≠ B
  - **Propagation**: (B < A) = false     // Skip the next assignment
  - **Full Test Specification**: true ∧ (A ≠ B) ∧ ((B < A) = false)
    
    ≡ (A ≠ B) ∧ (B ≥ A)
    
    ≡ (B > A)

- (A = 5, B = 3) will weakly kill mutant 1, but not strongly
Equivalent Mutation Example

- Mutant 3 in the Min() example is equivalent:

```
minVal = A;
if (B < A)
\[\Delta 3 \text{ if } (B < \text{minVal})\]
```

- The infection condition is “\((B < A) \neq (B < \text{minVal})\)”

- However, the previous statement was “\(\text{minVal} = A\)”
  - Substituting, we get: “\((B < A) \neq (B < A)\)”

- **Thus no input can kill this mutant**
Strong Versus Weak Mutation

```java
1 boolean isEven (int X)
2 {
3     if (X < 0)
4         X = 0 - X;
\[4\]
5     if (double) (X/2) == ((double) X) / 2.0
6         return (true);
7     else
8         return (false);
9 }
```

Reachability: \( X < 0 \)

Infection: \( X \neq 0 \)

\((X = -6)\) will kill mutant 4 under weak mutation

Propagation:

\(((\text{double}) ((0-X)/2) == ((\text{double}) 0-X) / 2.0)\) != \(((\text{double}) (0/2) == ((\text{double}) 0) / 2.0)\)

That is, \( X \) is not even ...

Thus \( (X = -6) \) does not kill the mutant under strong mutation
Automated steps

Testing Programs with Mutation

Input test method → Create mutants → Run equivalence detector → Generate test set T → Run T on P

Define threshold → Automated steps

Threshold reached?

Run mutants:
- schema-based
- weak
- selective

Eliminate ineffective TCs

Fix P

P (T) correct?

no

no

yes

© Ammann & Offutt
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 a very 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 (e.g., incorrect variable name)
  - Encourage common test heuristics (e.g., 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 $O = \{o1, o2, \ldots\}$, they 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:**
   
   Each arithmetic expression (and subexpression) is modified by the functions `abs()`, `negAbs()`, and `failOnZero()`.

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`.

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`. 
Mutation Operators for Java (2)

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.

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.

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.
7. **ASR — Assignment Operator Replacement:**

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

8. **UOI — Unary Operator Insertion:**

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

9. **UOD — Unary Operator Deletion:**

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is deleted.
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.

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.
Summary: Subsumption of Other Criteria

- Mutation is widely considered the strongest test criterion
  - And most expensive!
  - By far the most test requirements (each mutant)
  - Not always the most tests

- Mutation subsumes other criteria by including specific mutation operators

- Subsumption actually only makes sense for weak mutation—other criteria impose local requirements, like weak mutation
  - Node coverage
  - Edge coverage
  - Clause coverage
  - General active clause coverage: Yes – Requirement on single tests
  - Correlated active clause coverage: No – Requirement on pairs of tests
  - All-defs data flow coverage
Introduction to Software Testing
Chapter 5.3
Integration and Object-Oriented Testing

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/
Integration and Object-Oriented Testing

Integration Testing

Testing connections among separate program units

- In Java, testing the way classes, packages and components are connected
  - “Component” is used as a generic term

- This tests features that are unique to object-oriented programming languages
  - inheritance, polymorphism and dynamic binding

- Integration testing is often based on couplings – the explicit and implicit relationships among software components
Instantiating Grammar-Based Testing

**Grammar-Based Testing**

**Program-based**
- String mutation
- Compiler testing
- Valid and invalid strings
- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants
- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

**Integration**
- String mutation

**Model-Based**
- String mutation
- FSMs
- Model checking
- Valid strings
- Traces are tests
- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests
- Input validation testing
- XML and others
- Valid strings

**Input-Based**
- String mutation

*Introduction to Software Testing (Ch 5)*
Grammar Integration Testing (5.3.1)

There is no known use of grammar testing at the integration level
Integration Mutation (5.3.2)

- Faults related to component integration often depend on a mismatch of assumptions
  - Callee thought a list was sorted, caller did not
  - Callee thought all fields were initialized, caller only initialized some of the fields
  - Caller sent values in kilometers, callee thought they were miles

- Integration mutation focuses on mutating the connections between components
  - Sometimes called “interface mutation”
  - Both caller and callee methods are considered
Four Types of Mutation Operators

- Change a **calling** method by **modifying values that are sent** to a called method

- Change a **calling** method by **modifying the call**

- Change a **called** method by **modifying values that enter and leave** a method
  - Includes parameters as well as variables from higher scopes (class level, package, public, etc.)

- Change a **called** method by **modifying return statements from** the method
Five Integration Mutation Operators

1. **IPVR — Integration Parameter Variable Replacement**
   
   Each parameter in a method call is replaced by each other variable in the scope of the method call that is of compatible type.

   - **This operator replaces primitive type variables as well as objects.**

2. **IUOI — Integration Unary Operator Insertion**
   
   Each expression in a method call is modified by inserting all possible unary operators in front and behind it.

   - **The unary operators vary by language and type**

3. **IPEX — Integration Parameter Exchange**
   
   Each parameter in a method call is exchanged with each parameter of compatible types in that method call.

   - **max (a, b) is mutated to max (b, a)**
Four Integration Mutation Operators (2)

4. IMCD — Integration Method Call Deletion

Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.

- Method calls that return objects are replaced with calls to “new ()”

5. IREM — Integration Return Expression Modification

Each expression in each return statement in a method is modified by applying the UOI and AOR operators.
Integration Mutation Operators—Example

1. IPVR – Integration Parameter Variable Replacement

```
MyObject a, b;
...
callMethod (a);
Δ callMethod (b);
```

2. IUOI – Integration Unary Operator Insertion

```
callMethod (a);
Δ callMethod (a++);
```
3. **IPEX – Integration Parameter Exchange**

```
Max (a, b);
△ Max (b, a);
```

4. **IMCD – Integration Method Call Deletion**

```
X = Max (a, b);
△ X = new Integer (0);
```

5. **IREM – Integration Return Expression Modification**

```
int myMethod ()
{
    return a;
    △ return ++a;
}
```
Object-Oriented Mutation

- These five operators can be applied to non-OO languages
  - C, Pascal, Ada, Fortran, ...
- They do **not support** object oriented features
  - Inheritance, polymorphism, dynamic binding
- Two other language features that are often lumped with OO features are **information hiding** (encapsulation) and **overloading**
- Even experienced programmers often get encapsulation and access control wrong
The following five pages to be presented by 5 groups
Encapsulation, Information Hiding and Access Control

- **Encapsulation**: An abstraction mechanism to implement information hiding, which is a design technique that attempts to protect parts of the design from parts of the implementation
  - Objects can restrict access to their member variables and methods

- **Java provides four access levels** (C++ & C# are similar)
  - private
  - protected
  - public
  - default (also called package)

- **Often not used correctly** or understood, especially for programmers who are not well educated in **design**
Access Control in Java

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Same class</th>
<th>Same package</th>
<th>Different package subclass</th>
<th>Different package non-subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>package</td>
<td>Y</td>
<td>Y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>n</td>
</tr>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

- Most class variables should be **private**
- **Public** variables should seldom be used
- **Protected** variables are particularly dangerous – future programmers can accidentally override (by using the same name) or accidentally use (by mis-typing a similar name)
  - They should be called “unprotected”
Access Control in Java (2)

Class 1
- public members
- protected members
- default
- private members

Class 2

Inheritance

Class 3

Package

Class 4

Class 5

Introduction to Software Testing (Ch 5)
Introduction to Software Testing (Ch 5)

Package

Class 1
- public members
- protected members
- default
- private members

Class 2

Class 3

Class 4

Class 5

inheritance
Access Control in Java (2)

Class 1
- public members
- protected members
- default
- private members

Inheritance:
- Class 1 to Class 2
- Class 1 to Class 3
- Class 1 to Class 4
- Class 1 to Class 5

Package
Object-Oriented Language Features (Java)

- **Method overriding**
  Allows a method in a subclass to have the same name, arguments and result type as a method in its parent

- **Variable hiding**
  Achieved by defining a variable in a child class that has the same name and type of an inherited variable

- **Class constructors**
  Not inherited in the same way other methods are – must be explicitly called

- **Each object has …**
  - a *declared* type: \texttt{Parent P};
  - an *actual* type: \texttt{P = new Child();} or assignment: \texttt{P = Pold};
  - Declared and actual types allow uses of the same name to reference different variables with different types
OO Language Feature Terms

- **Polymorphic attribute**
  - An object reference that can take on various types
  - Type the object reference takes on during execution can change

- **Polymorphic method**
  - Can accept parameters of different types because it has a parameter that is declared of type Object

- **Overloading**
  - Using the same name for different constructors or methods in the same class

- **Overriding**
  - A child class declares an object or method with a name that is already declared in an ancestor class
  - Easily confused with overloading because the two mechanisms have similar names and semantics
  - Overloading is in the same class, overriding is between a class and a descendant
More OO Language Feature Terms

• Members associated with a class are called **class** or **instance** variables and methods
  – **Static methods** can operate only on static variables; not instance variables
  – **Instance variables** are declared at the class level and are available to objects

• 20 object-oriented mutation operators **defined for Java** – muJava

• Broken into 4 **general categories**
Class Mutation Operators for Java

(1) Encapsulation
AMC

(2) Inheritance
HVD, HVI, OMD, OMM, OMR, SKD, PCD

(3) Polymorphism
ATC, DTC, PTC, RTC, OMC, OMD, AOC, ANC

(4) Java-Specific
TKD, SMC, VID, DCD
1. AMC — Access Modifier Change

The access level for each instance variable and method is changed to other access levels.
1. AMC – Access Modifier Change

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int x;</td>
</tr>
<tr>
<td>Δ1 public int x;</td>
</tr>
<tr>
<td>Δ2 protected int x;</td>
</tr>
<tr>
<td>Δ3 int x;</td>
</tr>
</tbody>
</table>
OO Mutation Operators—*Inheritance*

2. **HVD** — *Hiding Variable Deletion*

Each declaration of an overriding or hiding variable is deleted.

3. **HVI** — *Hiding Variable Insertion*

A declaration is added to hide the declaration of each variable declared in an ancestor.

4. **OMD** — *Overriding Method Deletion*

Each entire declaration of an overriding method is deleted.

5. **OMM** — *Overridden Method Moving*

Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.
2. HVD – Hiding Variable Deletion

3. HVI – Hiding Variable Insertion

2. HVD – Hiding Variable Deletion

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
</tr>
<tr>
<td>int y;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>colorpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
</tr>
<tr>
<td>// int x;</td>
</tr>
<tr>
<td>int y;</td>
</tr>
<tr>
<td>// int y;</td>
</tr>
</tbody>
</table>

3. HVI – Hiding Variable Insertion

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
</tr>
<tr>
<td>int y;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>colorpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
</tr>
<tr>
<td>// int x;</td>
</tr>
<tr>
<td>int y;</td>
</tr>
<tr>
<td>// int y;</td>
</tr>
</tbody>
</table>
4. **OMD – Overriding Method Deletion**

```
void set (int x, int y)
```

```
△ // void set (int x, int y)
```

5. **OMM – Overriding Method Moving**

```
void set (int x, int y)
{ width = 5;...}
```

```
△ { width=10; super.set (x, y); }
```
**6. OMR — Overridden Method Rename**

Renames the parent’s versions of methods that are overridden in a subclass so that the overriding does not affect the parent’s method.

**7. SKD — Super Keyword Deletion**

Delete each occurrence of the `super` keyword.

**8. PCD — Parent Constructor Deletion**

Each call to a `super` constructor is deleted.
**OO Mutation Operators**

### 6. OMR – Overriding Method Rename

**point**

```java
void set (int x, int y)
```

```java
Δ void setP (int x, int y)
```

```java
void setDimension (int d)
{
    ...
    set (x, y);
    Δ setP (x, y);
    ...
}
```

**colorpoint**

```java
void set (int x, int y)
```

```java
point p;
p = new colorpoint ();
    ...
p.set (1, 2);
p.setDimension (3);
```

### 7. SKD – Super Keyword Deletion

**point**

```java
int getX()
```

```java
Δ return x;
```

**colorpoint**

```java
int getX ()
{
    return super.x;
    Δ return x;
}
```
8. PCD – Parent Constructor Deletion

```java
class point {
    point (int x, int y)
    ...
}
```

```java
class colorpoint {
    colorpoint (int x, int y, int color)
    {
        super (x, y);
        // super (x, y);
        ...
    }
}
```
The declared type of each parameter object is changed in the declaration.

9. ATC — Actual Type Change
The actual type of a new object is changed in the `new()` statement.

10. DTC — Declared Type Change
The declared type of each new object is changed in the declaration.

11. PTC — Parameter Type Change
The declared type of each parameter object is changed in the declaration.

12. RTC — Reference Type Change
The right side objects of assignment statements are changed to refer to objects of a compatible type.
9. ATC – Actual Type Change

point

colorpoint

point p;
p = new point ();
\(\Delta\) p = new colorpoint ();

10. DTC – Declared Type Change

point

colorpoint

point p;
\(\Delta\) colorpoint p;
p = new colorpoint ();
11. PTC – Parameter Type Change

```
boolean equals (point p)
{ . . . }
\Delta boolean equals (colorpoint p)
{ . . . }
```

12. RTC – Reference Type Change

```
point p;
colorpoint cp = new colorpoint (0, 0);
point3D p3d = new point3D (0, 0, 0);
p = cp;
\Delta p = p3d;
```
13. OMC — *Overloading Method Change*

For each pair of methods that have the same name, the bodies are interchanged.

14. OMD — *Overloading Method Deletion*

Each overloaded method declaration is deleted, one at a time.

15. AOC — *Argument Order Change*

The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.

16. ANC — *Argument Number Change*

The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
**OO Mutation Operators—Example**

**13. OMC – Overloading Method Change**

```c
void set (int x, int y) { S1 }
void set (int x, int y, int z) { S2 }
```

**14. OMD – Overloading Method Deletion**

```c
// void set (int x, int y) { ... }  
void set (int x, int y, int z) { S1 }
```
**OO Mutation Operators—Example**

15. **AOC – Argument Order Change**

```java
point3D p;
p.set(1, 2, 't');
```

16. **ANC – Argument Number Change**

```java
point3D p;
p.set(1, 2, 3);
```

```java
point3D p;
p.set(1, 2, 't');
\[ \Delta p.set('t', 1, 2); \]
```

```java
point3D p;
p.set(1, 2, 3);
\[ \Delta p.set(2, 3);
\Delta p.set(3); \]
```
17. TKD — *this* Keyword Deletion

Each occurrence of the keyword *this* is deleted.

18. SMC — Static Modifier Change

Each instance of the *static* modifier is removed, and the *static* modifier is added to instance variables.

19. VID — Variable Initialization Deletion

Remove initialization of each member variable.

20. DCD — Default Constructor Delete

Delete each declaration of default constructor (with no parameters).
17. TKD – This Keyword Deletion

```
void set (int x, int y)
{
    this.x = x;
    △1 x = x;
    this.y = y;
    △2 y = y;
}
```

18. SMC – Static Modifier Change

```
public static int x = 0;
△1 public int x = 0;
    public int y = 0;
△2 public static int y = 0;
    ...
```
**OO Mutation Operators—Example**

19. **VID—Variable Initialization Deletion**

```
int x = 5;
Δ int x;
...
```

20. **DCD—Default Constructor Delete**

```
// point() {
...

Δ // point() {
...
```

Introduction to Software Testing (Ch 5)
Integration Mutation Summary

- Integration testing often looks at **couplings**
- We have not used **grammar testing** at the integration level
- Mutation testing modifies **callers** and **callee**s
- **OO mutation** focuses on inheritance, polymorphism, dynamic binding, information hiding and overloading
  - The access levels make it easy to make mistakes in OO software
- **muJava** is an educational / research tool for mutation testing of Java programs
  - [http://cs.gmu.edu/~offutt/mujava/](http://cs.gmu.edu/~offutt/mujava/)
Model-based Grammars

Model-based
Languages that describe software in abstract terms

- **Formal** specification languages
  - Z, SMV, OCL, ...

- **Informal** specification languages

- **Design** notations
  - Statecharts, FSMs, UML notations

- **Model-based** languages are becoming more widely used
Instantiating Grammar-Based Testing

Grammar-Based Testing

Program-based
- 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

Model-Based
- String mutation
- FSMs
- Model checking
- Valid strings
- Traces are tests
- Includes OO

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

5.4

Introduction to Software Testing (Ch 5)
BNF Grammar Testing (5.4.1)

- Terminal symbol coverage and production coverage have only been applied to algebraic specifications
- Algebraic specifications are not widely used
- This is essentially research-only, so not covered in this book
Specification-based Mutation (5.4.2)

- A **finite state machine** is essentially a graph $G$
  - Nodes are states
  - Edges are transitions

- A **formalization** of an FSM is:
  - *States* are **implicitly defined** by declaring variables with limited range
  - The *state space* is then the **Cartesian product** of the ranges of the variables
  - *Initial states* are defined by **limiting the ranges** of some or all of the variables
  - *Transitions* are defined by **rules** that characterize the source and target of each transition
Example SMV Machine

MODULE main
#define false 0
#define true 1
VAR
  x, y : boolean;
ASSIGN
  init (x) := false;
  init (y) := false;
  next (x) := case
    !x & y : true;
    !y     : true;
    x      : false;
    true   : x;
  esac;
  next (y) := case
    x & !y : false;
    x & y   : y;
    !x & y  : false;
    true    : true;
  esac;

• Initial state : (F, F)
• Value for x in next state:
  - if x=F and y=T, next state has x=T
  - if y=F, next state has x=T
  - if x=T, next state has x=F
  - otherwise, next state x does not change
• Value for y in next state:
  - if (T, F), next state has y=F
  - if (T, T), next state y does not change
  - if (F,T), next state has y=F
  - otherwise, next state has y=T
• Any ambiguity in SMV is resolved by the order of the cases
• “true : x” corresponds to “default” in programming
Example SMV Machine

- Converting from SMV to FSM is mechanical and easy to automate
- SMV notation is smaller than graphs for large finite state machines

```c
MODULE main
#define false 0
#define true 1
VAR
    x, y : boolean;
ASSIGN
    init (x) := false;
    init (y) := false;
    next (x) := case
        !x & y : true;
        !y        : true;
        x         : false;
        true    : x;
    esac;
    next (y) := case
        x & !y : false;
        x & y  : y;
        !x & y : false;
        true    : true;
    esac;
```
Using SMV Descriptions

• Finite state descriptions can capture **system behavior** at a very high level – suitable for communicating with end users
• The verification community has built **powerful analysis tools** for finite state machines expressed in SMV
• These tools produce **explicit evidence** for properties that are not true
• This "evidence" is presented as sequences of states, called "counterexamples"
• Counterexamples are **paths** through the FSM that can be used as **test cases**
Mutations and Test Cases

• Mutating FSMs requires **mutation operators**
• Most FSM mutation operators are **similar** to program language operators

**Constant Replacement operator:**
• changes a constant to each other constant
• in the `next(y)` case: `!x & y : false` is mutated to `!x & y : true`
• To kill this mutant, we need a **sequence of states** (a path) that the original machine allows but the mutated machine does not
• This is what **model checkers** do
  – Model checkers find **counterexamples** – paths in the machine that violate some **property**
  – Properties are written in “**temporal logic**” – logical statements that are true for some period of time
  – `!x & y: false` has different result from `!x & y: true`
Counter-Example for FSM

assume x is input.
next (y) := case
  x & !y : false;
  x & y  : y;
  !x & y : false;
  true    : true;
esac;

Δ 1

SPEC AG (!x & y) → AX (y=true)

written in
SMV as

assume x is input.
next (y) := case
  x & !y : false;
  x & y  : y;
  !x & y : false;
  true    : true;
esac;

mutated
FSM

Introduction to Software Testing (Ch 5)
Counter-Example for FSM

• The model checker should produce:

```c
/* state 1 */ { x = 0, y = 0 }
/* state 2 */ { x = 1, y = 1 }
/* state 3 */ { x = 0, y = 1 }
/* state 4 */ { x = 1, y = 0 }
```

• This represents a test case that goes from nodes FF to TT to FT to TF in the original FSM
  – The last step in the mutated FSM will be to TT, killing the mutant

• If no sequence is produced, the mutant is equivalent
  – Equivalence is undecidable for programs, but decidable for FSMs
Model-Based Grammars Summary

• **Model-checking** is slowly growing in use

• **Finite state machines** can be encoded into model checkers

• Properties can be defined on FSMs and model checking used to find **paths that violate** the properties

• **No equivalent** mutants

• Everything is **finite**
Input Space Grammars

Input Space
The set of allowable inputs to software

- The input space can be described in many ways
  - User manuals
  - Unix man pages
  - Method signature / Collection of method preconditions
  - A language

- Most input spaces can be described as grammars
- Grammars are usually not provided, but creating them is a valuable service by the tester
  - Errors will often be found simply by creating the grammar
Using Input Grammars

- Software should **reject** or **handle** invalid data
- Programs often do this **incorrectly**
- Some programs (rashly) **assume** all input data is correct
- Even if it works **today** …
  - What about after the program goes through some **maintenance changes**?
  - What about if the component is **reused** in a new program?
- Consequences can be **severe** …
  - The **database** can be corrupted
  - **Users** are not satisfied
  - Many **security vulnerabilities** are due to unhandled exceptions … from invalid data
Validating Inputs

Input Validation

Deciding if input values can be processed by the software

• Before starting to process inputs, wisely written programs check that the inputs are valid
• How should a program recognize invalid inputs?
• What should a program do with invalid inputs?
• If the input space is described as a grammar, a parser can check for validity automatically
  – This is very rare
  – It is easy to write input checkers – but also easy to make mistakes
Instantiating Grammar-Based Testing

Grammar-Based Testing

Program-based
- 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

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

Grammar

5.5

Introduction to Software Testing (Ch 5)
Input Space BNF Grammars (5.5.1)

- Input spaces can be expressed in many forms
- A common way is to use some form of grammar
- We will look at three grammar-based ways to describe input spaces
  1. Regular expressions
  2. BNF grammars
  3. XML and Schema
- All are similar and can be used in different contexts
Consider a program that processes a sequence of deposits and debits to a bank

**Inputs**
- deposit 5306 $4.30
- debit 0343 $4.14
- deposit 5306 $7.29

**Initial Regular Expression**

(deposit account amount | debit account amount) *

**FSM to represent the grammar**
Grammars are more expressive than regular expressions—they can capture more details

```plaintext
bank ::= action*
action ::= dep | deb
dep ::= "deposit" account amount
deb ::= "debit" account amount
account ::= digit^4
amount ::= "$" digit+ "." digit^2
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
        "7" | "8" | "9"
```
• Derive tests by \textit{systematically replacing} each non-terminal with a production

• If the tester designs the grammar from informal input descriptions, \textbf{do it early}
  
  – In time to \textit{improve} the design
  
  – \textbf{Mistakes} and \textbf{omissions} will almost always be found
Using XML to Describe Input Spaces

- Software components that pass data must agree on **format**, **types**, and **organization**
- Web applications have **unique requirements**:  
  - Very **loose coupling** and **dynamic integration**

1970s

- **P1**
- **P2**

File

- File storage
- Un-documented format
- Data saved in binary mode
- Source not available

1980s

- **P1**
- **P2**

WM

File

- File storage
- Un-documented format
- Data saved as plain text
- Access through wrapper module
- Data hard to validate

Introduction to Software Testing (Ch 5)
XML in Very Loosely Coupled Software

- Data is passed directly between components
- XML allows data to be self-documenting

- P1, P2, and P3 can see the format, contents, and structure of the data
- Data sharing is independent of type
- Format is easy to understand
- Grammars are defined in DTDs or Schemas
XML for Book Example

```
<books>
  <book>
    <title>The Art of Software Testing</title>
    <author>Glen Myers</author>
    <publisher>Wiley</publisher>
    <price>50.00</price>
    <year>1979</year>
  </book>
</books>
```

- XML messages are defined by **grammars**
  - Schemas and DTDs
- Schemas can define many kinds of **types**
- Schemas include “**facets**,” which refine the grammar

*schemas define input spaces for software components*
Representing Input Domains

- **Desired inputs** (goal domain)
- **Described inputs** (specified domain)
- **Accepted inputs** (implemented domain)
Example Input Domains

• Goal domains are often **irregular**

• **Goal domain for credit cards**†
  – First digit is the Major Industry Identifier
  – First 6 digits and length specify the issuer
  – Final digit is a “check digit”
  – Other digits identify a specific account

• **Common specified** domain
  – First digit is in { 3, 4, 5, 6 } (travel and banking)
  – Length is between 13 and 16

• **Common implemented** domain
  – All digits are numeric

† More details are on: http://www.merriampark.com/anatomycc.htm
This region is a rich source of software errors ...
Using Grammars to Design Tests

- This form of testing allows us to focus on interactions among the components
  - Originally applied to Web services, which depend on XML
- A formal model of the XML grammar is used
- The grammar is used to create valid as well as invalid tests
- The grammar is mutated
- The mutated grammar is used to generate new XML messages
- The XML messages are used as test cases
Introduction to Software Testing (Ch 5)
XML Constraints – “Facets”

<table>
<thead>
<tr>
<th>Boundary Constraints</th>
<th>Non-boundary Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxOccurs</td>
<td>enumeration</td>
</tr>
<tr>
<td>minOccurs</td>
<td>use</td>
</tr>
<tr>
<td>length</td>
<td>fractionDigits</td>
</tr>
<tr>
<td>maxExclusive</td>
<td>pattern</td>
</tr>
<tr>
<td>maxInclusive</td>
<td>nillable</td>
</tr>
<tr>
<td>maxLength</td>
<td>whiteSpace</td>
</tr>
<tr>
<td>minExclusive</td>
<td>unique</td>
</tr>
<tr>
<td>minInclusive</td>
<td></td>
</tr>
<tr>
<td>minLength</td>
<td></td>
</tr>
<tr>
<td>totalDigits</td>
<td></td>
</tr>
</tbody>
</table>
Generating Tests

• **Valid** tests
  – Generate tests as **XML messages** by deriving strings from grammar
  – Take **every production** at least once
  – Take **choices** … “maxOccurs = “unbounded” means use 0, 1 and more than 1

• **Invalid** tests
  – **Mutate** the grammar in structured ways
  – Create **XML messages** that are “almost” valid
  – This explores the **gray space** on the previous slide
Generating Tests

- The criteria in section 5.1.1 can be used to generate tests
  - Production and terminal symbol coverage

- The only choice in books is based on “minOccurs”

- PC requires two tests
  - ISBN is present
  - ISBN is not present

- The facets are used to generate values that are valid
  - We also want values that are not valid …
Mutation for Input Grammars (5.5.2)

- Software should reject or handle invalid data
- A very common mistake is for programs to do this incorrectly
- Some programs (rashly) assume that all input data is correct
- Even if it works today …
  - What about after the program goes through some maintenance changes?
  - What about if the component is reused in a new program?
- Consequences can be severe …
  - Most security vulnerabilities are due to unhandled exceptions … from invalid data
- To test for invalid data (including security testing), mutate the grammar
Mutating Input Grammars

• Mutants are tests

• Create valid and invalid strings

• No ground strings – no killing
  – mutation to production rules, instead of real programs, interfaces, …

• Mutation operators listed here are general and should be refined for specific grammars
Input Grammar Mutation Operators

1. Nonterminal Replacement
Every nonterminal symbol in a production is replaced by other nonterminal symbols.

2. Terminal Replacement
Every terminal symbol in a production is replaced by other terminal symbols.

3. Terminal and Nonterminal Deletion
Every terminal and nonterminal symbol in a production is deleted.

4. Terminal and Nonterminal Duplication
Every terminal and nonterminal symbol in a production is duplicated.
Mutation Operators

• Many strings may not be useful

• Use additional type information, if possible

• Use judgment to throw tests out

• Only apply replacements if “they make sense”

• Examples …
Nonterminal Replacement

dep ::= “deposit” account amount

dep ::= “deposit” amount amount

dep ::= “deposit” account digit

Terminal Replacement

amount ::= “$” digit+ “.” digit^2

amount ::= “.” digit+ “.” digit^2

amount ::= “$” digit+ “$” digit^2

amount ::= “$” digit+ “1” digit^2

Terminal and Nonterminal Deletion

dep ::= “deposit” account amount

dep ::= account amount

dep ::= “deposit” amount

dep ::= “deposit” account

Terminal and Nonterminal Duplication

dep ::= “deposit” account amount

dep ::= “deposit” “deposit” account amount

dep ::= “deposit” account account amount

dep ::= “deposit” account amount amount
Notes and Applications

• We have more experience with program-based mutation than input grammar based mutation
  – Operators are less “definitive”

• Applying mutation operators
  – Mutate grammar, then derive strings
  – Derive strings, mutate a derivation “in-process”

• Some mutants give strings in the original grammar (equivalent)
  – These strings can easily be recognized to be equivalent
Mutating XML

- XML schemas can be mutated

- If a schema does not exist, testers should derive one
  - As usual, this will help find problems immediately

- Many programs validate messages against a grammar
  - Software may still behave correctly, but testers must verify

- Programs are less likely to check all schema facets
  - Mutating facets can lead to very effective tests
Test Case Generation – Example

Original Schema (Partial)

```xml
<xs:simpleType name = “priceType”>
  <xs:restriction base = “xs:decimal”>
    <xs:fractionDigits value = “2” />
    <xs:maxInclusive value = “1000.00” />
  </xs:restriction>
</xs:simpleType>
```

Mutants:
- value = “3”
- value = “1”
- value = “100”
- value = “2000”

XML from Original Schema

```xml
<books>
  <book>
    <price>37.95</price>
    <year>2002</year>
  </book>
</books>
```

Mutant XML 1

```xml
<books>
  <book>
    <price>37.95</price>
    <year>2002</year>
  </book>
</books>
```

Mutant XML 2

```xml
<books>
  <book>
    <price>37.95</price>
    <year>2002</year>
  </book>
</books>
```

Mutant XML 3

```xml
<books>
  <book>
    <price>37.95</price>
    <year>2002</year>
  </book>
</books>
```

Mutant XML 4

```xml
<books>
  <book>
    <price>1500.00</price>
    <year>2002</year>
  </book>
</books>
```
Input Space Grammars Summary

• This application of mutation is **fairly new**

• Automated **tools** do not exist

• Can be used **by hand** in an “ad-hoc” manner to get effective tests

• Applications to **special-purpose grammars** very promising
  – XML
  – SQL
  – HTML