Introduction to Software Testing
Chapter 5.5
Input Space Grammars

Paul Ammann & Jeff Offutt

www.introsoftwaretesting.com

Input Space Grammars

Input Space

The set of allowable inputs to software

- **There are many ways to describe the input space**
  - **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**

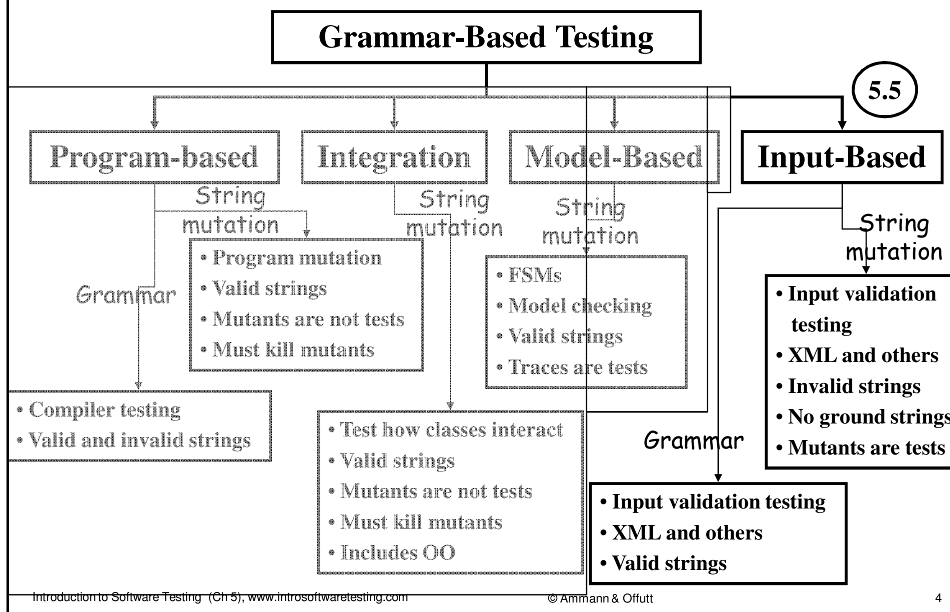
Validating Inputs

<u>Input Validation</u>

Deciding if input values can be processed by the software

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

Instantiating Grammar-Based Testing



Input Space BNF Grammars (5.5.1)

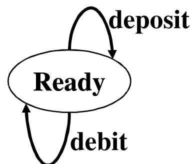
- Consider a program that processes a sequence of deposits and debits to a bank

Inputs

deposit 5306 $4.30
debit 0343 $4.14
deposit 5306 $7.29

Initial Grammar

(deposit account amount | debit account amount) *



FSM to represent the grammar

© Ammann & Offutt

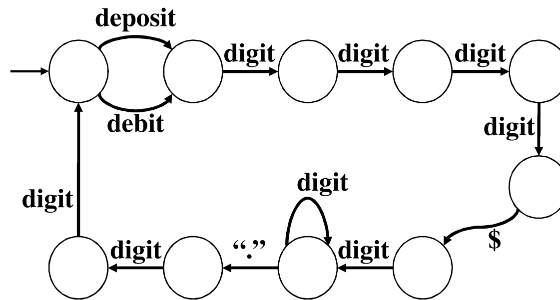
Grammar for Bank Example

- Grammars are more expressive than regular expressions – they can capture more details

```
bank      ::=  action*
action    ::=  dep | deb
dep       ::=  "deposit" account amount
deb       ::=  "debit"  account amount
account   ::=  digit4
amount    ::=  "$" digit+ "." digit2
digit     ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
               "7" | "8" | "9"
```


© Ammann & Offutt

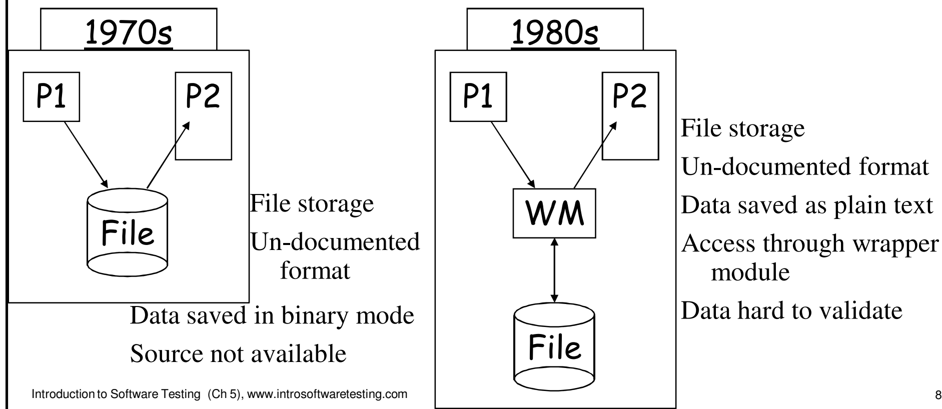
FSM for Bank Grammar



- Derive tests by systematically replacing each non-terminal with a production
- If the tester designs the grammar from informal input descriptions, do it early
  - In time to improve the design
  - Mistakes and omissions will almost always be found

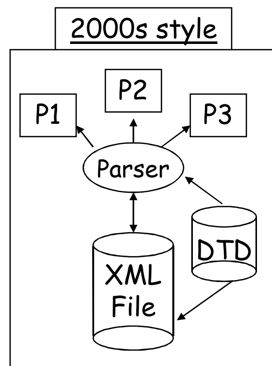
XML

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



XML in Web Services

- 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>
    <ISBN>0471043281</ISBN>
    <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

Book Grammar – Schema

```
<xs:element name = “books”>
  <xs:complexType>
  <xs:sequence>
    <xs:element name = “book” maxOccurs = “unbounded”>
      <xs:complexType>
      <xs:sequence>
        <xs:element name = “ISBN” type = “isbnType” minOccurs = “0”/>
        <xs:element name = “author” type = “xs:string”/>
        <xs:element name = “publisher” type = “xs:string”/>
        <xs:element name = “price”  type = “priceType”/>
        <xs:element name = “year”   type = “yearType”/>
      </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
```

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


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

© Ammann & Offutt

Mutating Input Grammars

- Mutants are tests
- Create valid and invalid strings
- No ground strings – no killing
- Mutation operators listed here are general and should be refined for specific grammars

© Ammann & Offutt

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

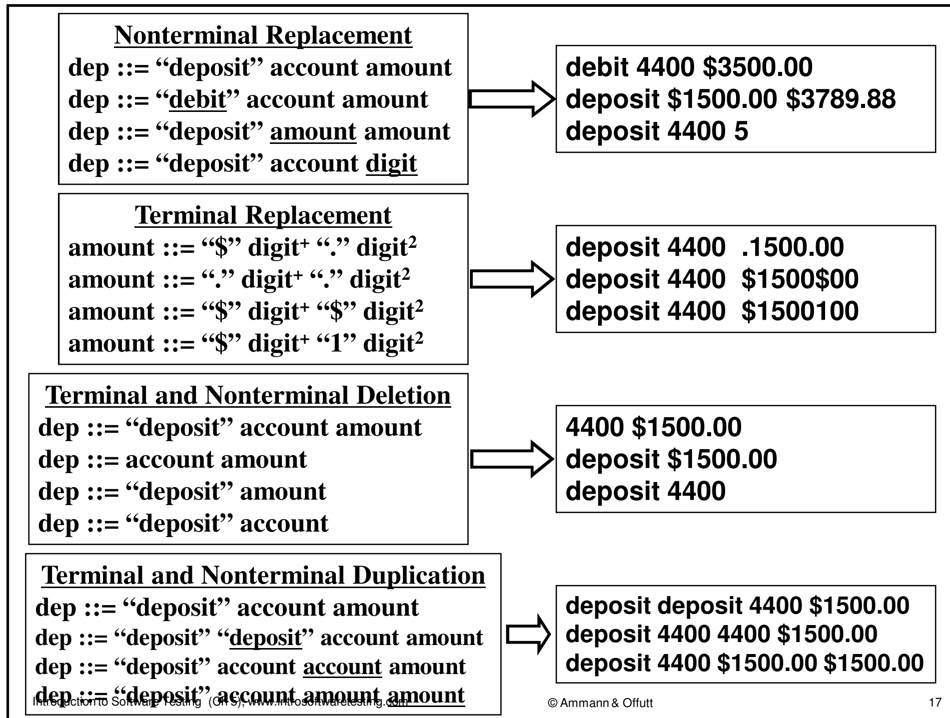
Every terminal and nonterminal symbol in a production is deleted.

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 …**



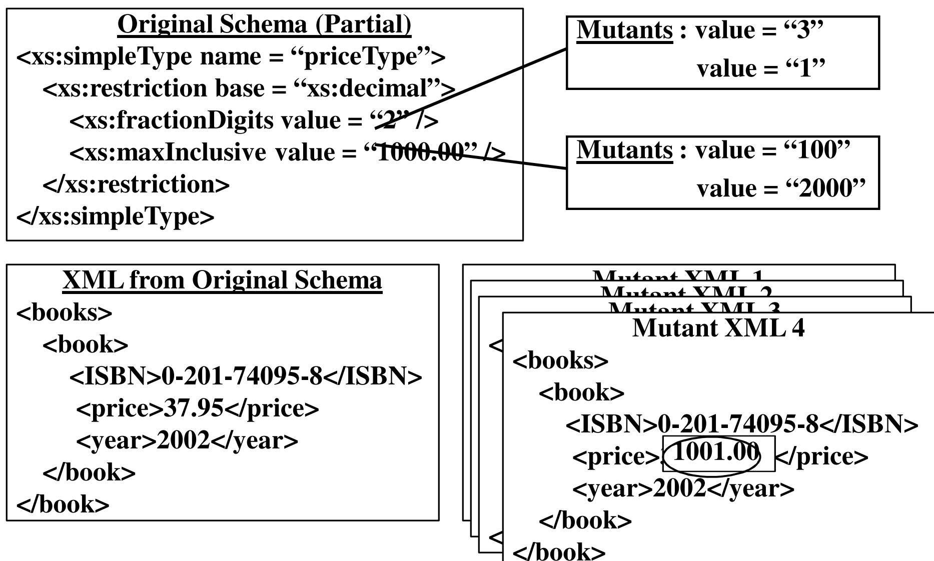
Notes and Applications

- **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



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**