Introduction to Software Testing Chapter 9.5 Input Space Grammars

Paul Ammann & Jeff Offutt

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

Input Space Grammars

<u>Input Space</u>

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 (i.e., syntax of inputs to a program)
- Grammars are usually not provided, but creating them is a valuable service by the tester
  - Errors will often be found simply by <u>creating the grammar</u>

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

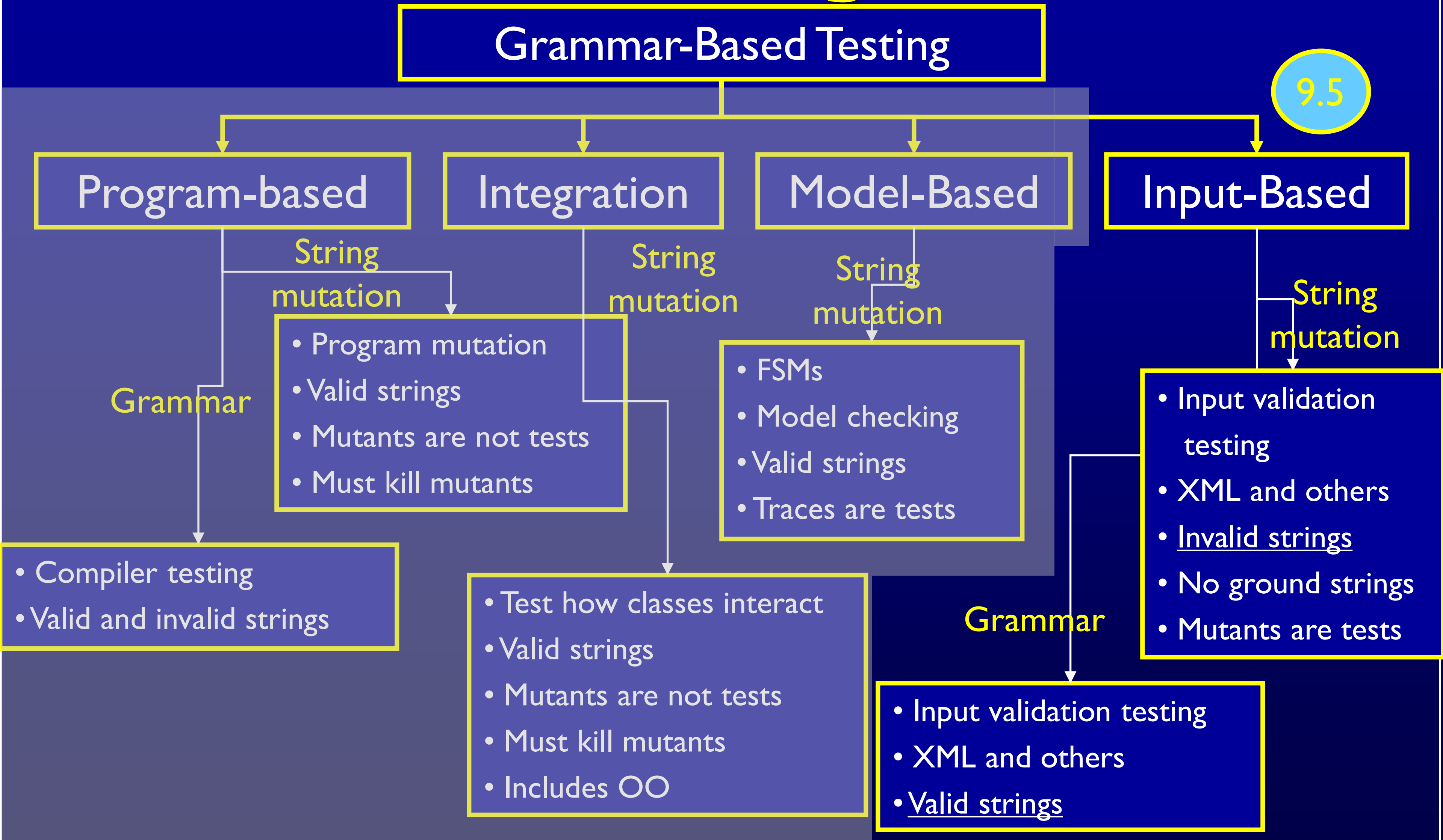
Validating Inputs

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



Input Space BNF Grammars (9.5.1)

- 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

Regular Expressions

Consider a program that processes a sequence of <u>deposits</u> and <u>debits</u> to a bank

<u>Inputs</u>

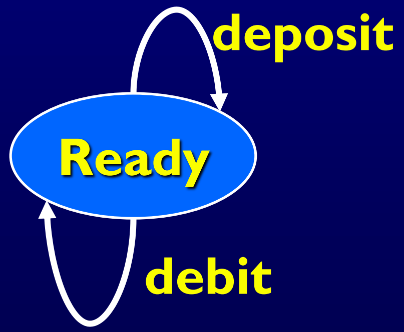
deposit 5306 $4.30

debit 0343 $4.14

deposit 5306 $7.29

<u>Initial Regular Expression</u>

(deposit account amount | debit account amount) *



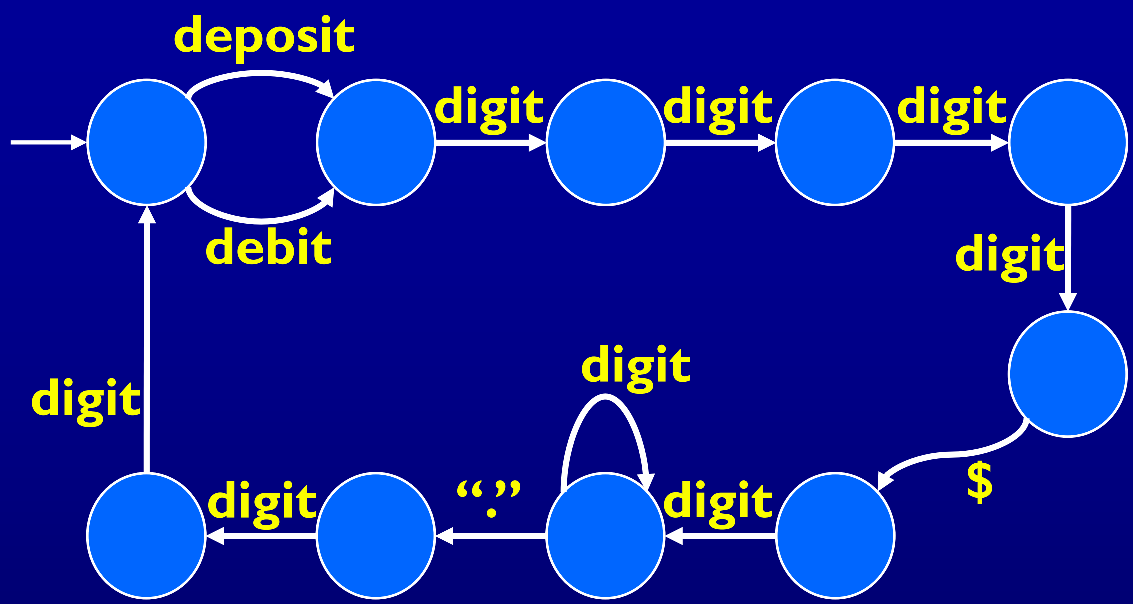
FSM to represent the grammar

BNF Grammar for Bank Example

Grammars are <u>more expressive</u> than regular expressions–they can capture more details

```
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"
```

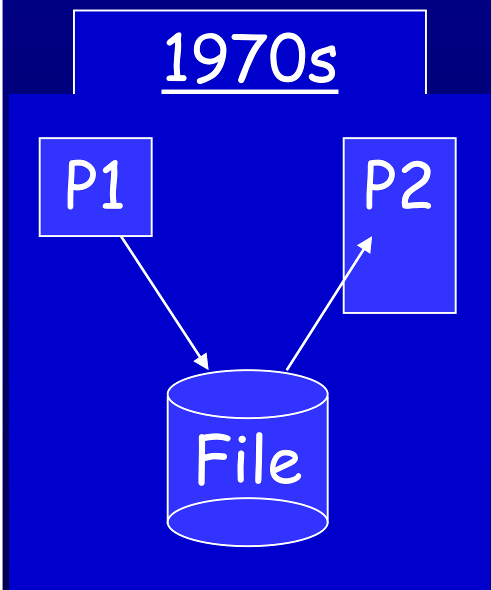
FSM for Bank Grammar



- 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 Can 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**

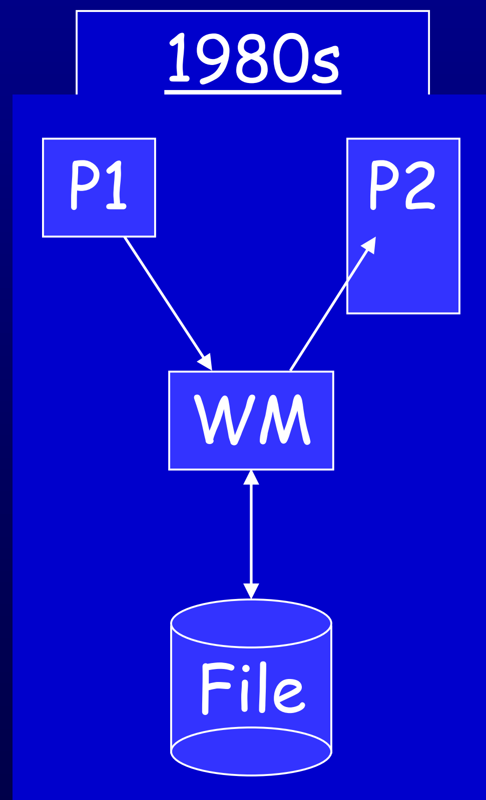


File storage

Un-documented format

Data saved in binary mode

Source not available



File storage

Un-documented format

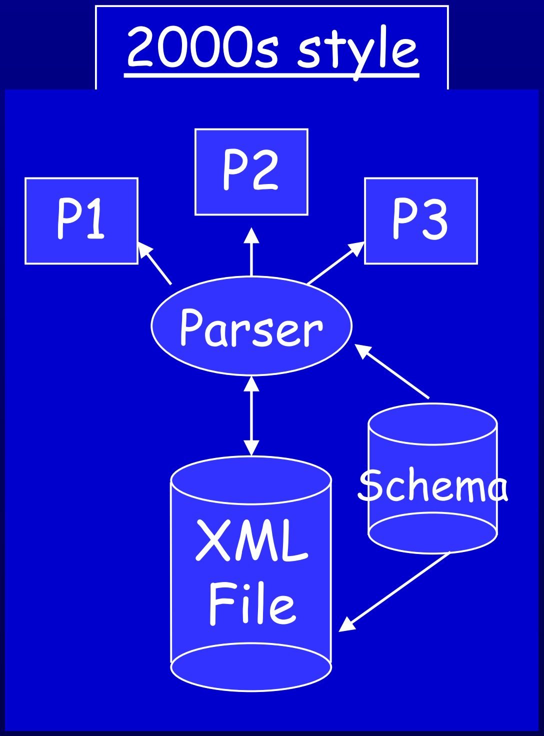
Data saved as plain text

Access through wrapper module

Data hard to validate

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

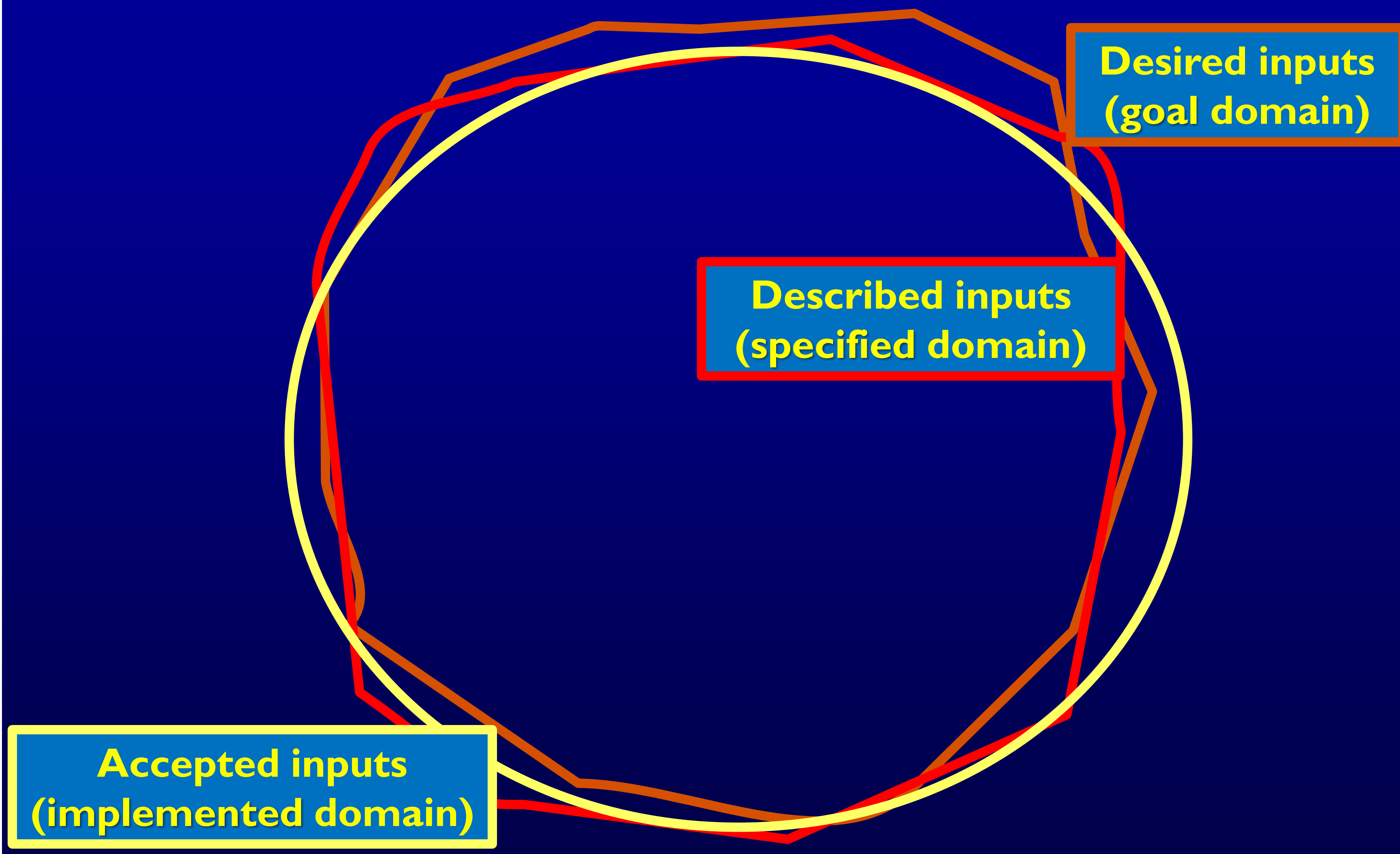
XML for Book Example

```xml
<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

schemas define input spaces for software components

Representing Input Domains

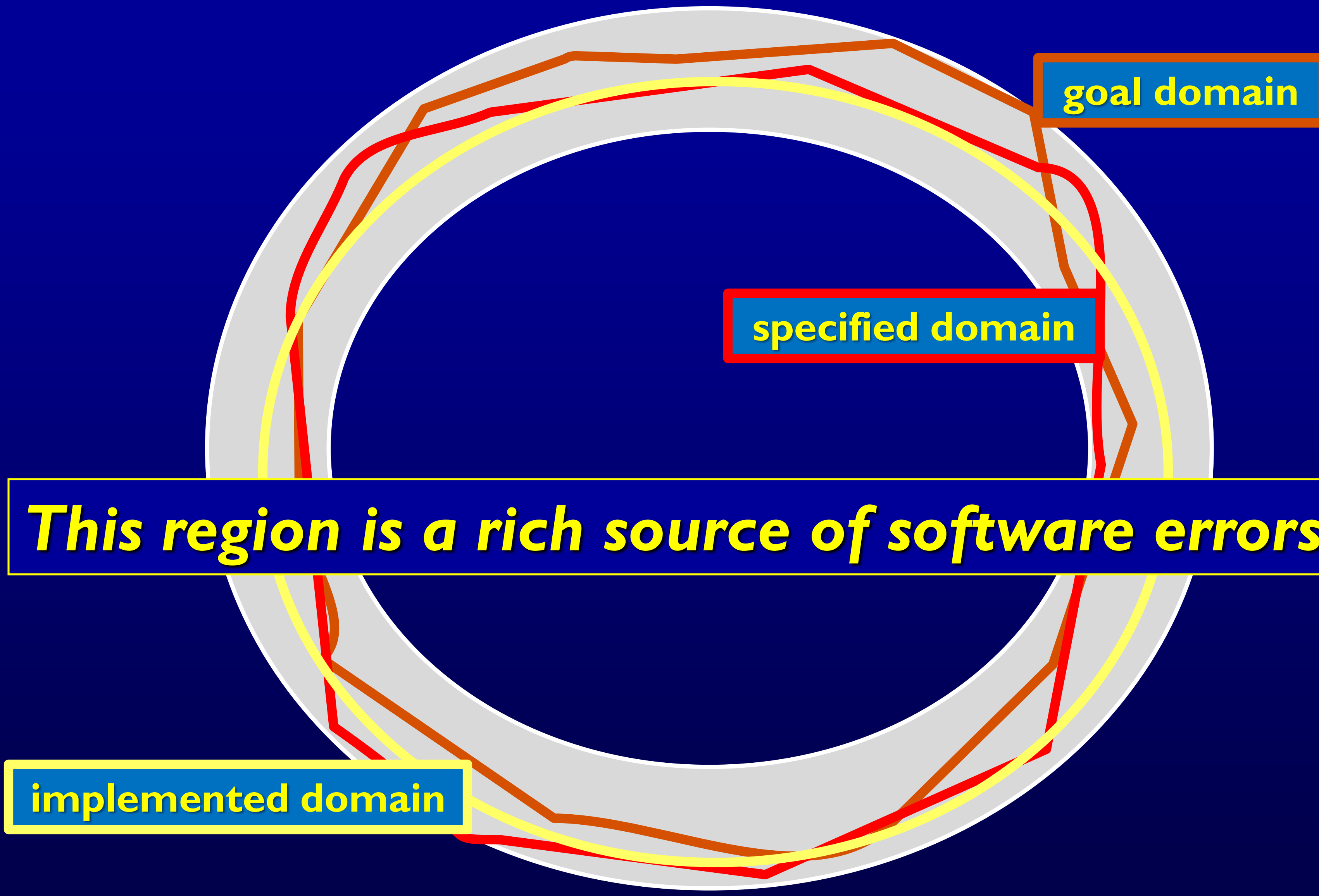


Example Input Domains

- 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

Representing Input Domains



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

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 = “title” 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>
```

Built-in types

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

XML Constraints – "Facets"

Boundary Constraints	Non-boundary Constraints
maxOccurs	enumeration
minOccurs	use
length	fractionDigits
maxExclusive	pattern
maxInclusive	nillable
maxLength	whiteSpace
minExclusive	unique
minInclusive	
minLength	
totalDigits	

Generating Tests

- Valid tests
  - Generate tests as XML messages by <u>deriving strings from grammar</u>
  - Take <u>every production</u> at least once
  - Take <u>choices</u> … "maxOccurs = "unbounded" means use 0, 1 and more than 1
- Invalid tests
  - Mutate the grammar in structured ways
  - Create XML messages that are <u>"almost" valid</u>
  - This explores the gray space on the previous slide
    - Invalid inputs often cause the software to <u>behave in surprising ways</u>, which malicious parties can use to their advantage
      - E.g., buffer overflow attack, SQL injection, …

Generating Tests

- The criteria in section 9.1.1 can be used to generate tests
  - Production and terminal symbol coverage
- The only choice in the books grammar is based on "minOccurs"
- PC (production coverage) 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 …

Mutating Input Grammars (9.5.2)

- 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 operators listed here are general and <u>should be refined for specific grammars</u>

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

deposit $1500.00 $3789.88
deposit 4400 5

Terminal Replacement

amount ::= "$" digit$^+$ "." digit^2
amount ::= "." digit$^+$ "." digit^2
amount ::= "$" digit$^+$ "$" digit^2
amount ::= "$" digit$^+$ "1" digit^2

deposit 4400 .1500.00
deposit 4400 $1500$00
deposit 4400 $1500100

Terminal and Nonterminal Deletion

dep ::= "deposit" account amount
dep ::= account amount
dep ::= "deposit" amount
dep ::= "deposit" account

4400 $1500.00
deposit $1500.00
deposit 4400

Terminal and Nonterminal Duplication

dep ::= "deposit" account amount
dep ::= "deposit" "deposit" account amount
dep ::= "deposit" account account amount
dep ::= "deposit" account amount amount

deposit deposit 4400 $1500.00
deposit 4400 4400 $1500.00
deposit 4400 $1500.00 $1500.00

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)

```
<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"

Mutants : value = "100"
value = "2000"

XML from Original Schema

```
<books>
   <book>
      <ISBN>0-201-74095-8</ISBN>
      <price>37.95</price>
      <year>2002</year>
   </book>
</book>
```

Mutant XML 1

Mutant XML 2

Mutant XML 3

Mutant XML 4

```
<books>
   <book>
      <ISBN>0-201-74095-8</ISBN>
      <price>1500.00   </price>
      <year>2002</year>
   </book>
</book>
```

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