XML Technologies and Applications

Rajshekhar Sunderraman

Department of Computer Science
Georgia State University
Atlanta, GA 30302
raj@cs.gsu.edu

II. XML Structural Constraint Specification (DTDs and XML Schema)

December 2005

Outline

- Introduction
- XML Basics
- XML Structural Constraint Specification
  - Document Type Definitions (DTDs)
  - XML Schema
- XML/Database Mappings
- XML Parsing APIs
  - Simple API for XML (SAX)
  - Document Object Model (DOM)
- XML Querying and Transformation
  - XPath
  - XQuery
  - XSLT
- XML Applications

Document Type Definitions (DTDs)

- DTD: Document Type Definition; A way to specify the structure of XML documents.
- A DTD adds syntactical requirements in addition to the well-formed requirement.
- DTDs help in
  - Eliminating errors when creating or editing XML documents.
  - Clarifying the intended semantics.
  - Simplifying the processing of XML documents.
- Uses “regular expression” like syntax to specify a grammar for the XML document.
- Has limitations such as weak data types, inability to specify constraints, no support for schema evolution, etc.

Example: An Address Book

```
<person>
    <name> Homer Simpson </name>
    <greet> Dr. H. Simpson </greet>
    <addr>1234 Springwater Road </addr>
    <addr> Springfield USA, 98765 </addr>
    <tel> (321) 786 2543 </tel>
    <fax> (321) 786 2544 </fax>
    <tel> (321) 786 2544 </tel>
    <email> homer@math.springfield.edu </email>
</person>
```

Exactly one name

At most one greeting

As many address lines as needed (in order)

Mixed telephones and faxes

As many as needed

Specifying the Structure

- `name` — a name element
- `greet?` — an optional (0 or 1) greet elements
- `name, greet?` — a name followed by an optional greet
- `addr*` — to specify 0 or more address lines
- `tel | fax` — a tel or a fax element
- `(tel | fax)*` — 0 or more repeats of tel or fax
- `email*` — 0 or more email elements

Specifying the Structure (continued)

- So the whole structure of a person entry is specified by

```
name, greet?, addr*, (tel | fax)*, email*
```

- Regular expression syntax (inspired from UNIX regular expressions)
- Each element type of the XML document is described by an expression (the leaf level element types are described by the data type (PCDATA)
- Each attribute of an element type is also described in the DTD by enumerating some of its properties (OPTIONAL, etc.)

Element Type Definition

For each element type E, a declaration of the form:

```
<!ELEMENT   E       content-model>
```

where the content-model is an expression:

```
Content-model ::=
  EMPTY  | ANY | #PCDATA | E' |
  P1, P2 | P1 | P2 |  P1?  | P1+  | P1* | (P)
```

- `E'` element type
- `P1 , P2` concatenation
- `P1 | P2` disjunction
- `P?` optional
- `P+` one or more occurrences
- `P*` the Kleene closure
- `(P)` grouping

Element Type Definition

The definition of an element consists of exactly one of the following:

- *A regular expression* (as defined earlier)
- EMPTY: element has no content
- ANY: content can be any mixture of PCDATA and elements defined in the DTD
- *Mixed content* which is defined as described on the next slide
- (#PCDATA)

The Definition of Mixed Content

Mixed content is described by a repeatable OR group

```
(#PCDATA | element-name | …)*
```

- Inside the group, no regular expressions – just element names
- #PCDATA must be first followed by 0 or more element names, separated by |
- The group can be repeated 0 or more times

Address-Book Document with an Internal DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE addressbook [
   <!ELEMENT addressbook (person*)>
   <!ELEMENT person (name, greet?, address*,
                     (fax | tel)*, email*)>
   <!ELEMENT name   (#PCDATA)>
   <!ELEMENT greet  (#PCDATA)>
   <!ELEMENT address(#PCDATA)>
   <!ELEMENT tel    (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT email  (#PCDATA)>
]>
```

The Rest of the Address-Book Document

```
<addressbook>
  <person>
          <name>Jeff Cohen</name>
          <greet> Dr. Cohen</greet>
        <email>jc@penny.com</email>
  </person>
</addressbook>
```

Some Difficult Structures

Each employee element should contain name, age and ssn elements in some order

```
<!ELEMENT employee
    ((name, age, ssn) | (age, ssn, name) |
     (ssn, name, age) | ...
    )>
```

Too many permutations!

Attribute Specification in DTDs

```
<!ELEMENT height (#PCDATA)>
<!ATTLIST height
      dimension CDATA #REQUIRED
      accuracy CDATA  #IMPLIED >
```

- The `dimension` attribute is required
- The `accuracy` attribute is optional
- CDATA is the “type” of the attribute – character data

The Format of an Attribute Definition

```
<!ATTLIST element-name attr-name attr-type
          attr-default>
```

- The default value is given inside quotes
- Attribute types:
  - `CDATA`
  - `ID, IDREF, IDREFS`
- `ID, IDREF, IDREFS` are used for references
- Attribute Default
  - #REQUIRED: the attribute must be explicitly provided
  - #IMPLIED: attribute is optional, no default provided
  - "*value*": if not explicitly provided, this value inserted by default
  - #FIXED "*value*": as above, but only this value is allowed

Recursive DTDs

```
<DOCTYPE genealogy [
   <!ELEMENT    genealogy (person*)>
   <!ELEMENT    person (
                name,
                dateOfBirth,
                person,          -- mother
                person   )>      -- father
   ...
]>
```

Problem with this DTD: Parser does not see the recursive structure and looks for “person” sub-element indefinitely!

Recursive DTDs (cont'd)

```
<DOCTYPE genealogy [
   <!ELEMENT   genealogy (person*)>
   <!ELEMENT   person (
               name,
               dateOfBirth,
               person?,          -- mother
               person?  )>     -- father
   ...
]>
```

The problem with this DTD is if only one "person" sub-element is present, we would not know if that person is the father or the mother.

Using ID and IDREF Attributes

```
<!DOCTYPE family [
  <!ELEMENT family   (person)*>
  <!ELEMENT person  (name)>
  <!ELEMENT name    (#PCDATA)>
  <!ATTLIST  person
       id ID #REQUIRED
       mother IDREF #IMPLIED
       father IDREF #IMPLIED
       children IDREFS #IMPLIED>
]>
```

IDs and IDREFs

- `ID` attribute: unique within the entire document.
  - An element can have at most one `ID` attribute.
  - No default (fixed default) value is allowed.
    - #required: a value must be provided
    - #implied: a value is optional

- IDREF attribute: its value must be some other element's ID value in the document.

- IDREFS attribute: its value is a set, each element of the set is the ID value of some other element in the document.

```
<person id="898" father="332" mother="336"
      children="982 984 986">
```

Some Conforming Data

```
<family>
    <person  id="lisa"  mother="marge" father="homer">
       <name> Lisa Simpson </name>
    </person>
    <person  id="bart"  mother="marge" father="homer">
       <name> Bart Simpson </name>
    </person>
    <person id="marge" children="bart lisa">
        <name> Marge Simpson </name>
    </person>
    <person id="homer" children="bart lisa">
        <name> Homer Simpson </name>
    </person>
</family>
```

Limitations of ID References

- The attributes mother and father are references to IDs of other elements.
- However, those are not necessarily person elements!
- The mother attribute is not necessarily a reference to a female person.

An Alternative Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE family [
   <!ELEMENT family (person)*>
   <!ELEMENT person (name, mother?, father?,
                     children?)>
      <!ATTLIST person id ID #REQUIRED>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT mother EMPTY>
      <!ATTLIST mother idref IDREF #REQUIRED>
   <!ELEMENT father EMPTY>
      <!ATTLIST father idref IDREF #REQUIRED>
   <!ELEMENT children EMPTY>
      <!ATTLIST children idrefs IDREFS #REQUIRED>
]>
```

Empty sub-elements instead of attributes

The Revised Data

```
<family>

<person id="marge">
 <name>Marge Simpson</name>
 <children
     idrefs="bart lisa"/>
 </person>

 <person id="homer">
 <name>Homer Simpson</name>
 <children
   idrefs="bart lisa"/>
</person>

<person id="bart">
 <name>Bart Simpson</name>
 <mother idref="marge"/>
 <father idref="homer"/>
</person>

<person id="lisa">
 <name>Lisa Simpson</name>
 <mother idref="marge"/>
 <father idref="homer"/>
</person>

</family>
```

Consistency of ID and IDREF Attribute Values

- If an attribute is declared as ID
  - The associated value must be distinct, i.e., different elements (in the given document) must have different values for the ID attribute.
  - Even if the two elements have different element names
- If an attribute is declared as IDREF
  - The associated value must exist as the value of some ID attribute (no dangling “pointers”)
- Similarly for all the values of an IDREFS attribute
- ID, IDREF and IDREFS attributes are *not* typed

Adding a DTD to the Document

A DTD can be

- *internal*
  - The DTD is part of the document file
- *external*
  - The DTD and the document are on separate files
  - An external DTD may reside
    - In the local file system (where the document is)
    - In a remote file system

Connecting a Document with its DTD

- An internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE db [<!ELEMENT ...> … ]>
<db> ... </db>
```

- A DTD from the local file system:

```
<!DOCTYPE db SYSTEM "schema.dtd">
```

- A DTD from a remote file system:

```
<!DOCTYPE db SYSTEM
    "http://www.schemaauthority.com/schema.dtd">
```

Well-Formed XML Documents

- An XML document (with or without a DTD) is *well-formed* if
  - Tags are syntactically correct
  - Every tag has an end tag
  - Tags are properly nested
  - There is a root tag
  - A start tag does not have two occurrences of the same attribute

Valid Documents

- A well-formed XML document is *valid* if it conforms to its DTD, that is,
  - The document conforms to the regular-expression grammar
  - The attributes types are correct, and
  - The constraints on references are satisfied

XML Schema

XML Schema

An XML Schema:

- defines elements that can appear in a document
- defines attributes that can appear within elements
- defines which elements are child elements
- defines the sequence in which the child elements can appear
- defines the number of child elements
- defines whether an element is empty or can include text
- defines default values for attributes

The purpose of a Schema is to define the legal building blocks of an XML document, just like a DTD.

XML Schema – Better than DTDs

XML Schemas

- are easier to learn than DTD
- are extensible to future additions
- are richer and more useful than DTDs
- are written in XML

- support data types

Example: Shipping Order

```
<?xml version="1.0"?>
<shipOrder>

<shipTo>
<name>Svendson</name>
<street>Oslo St</street>
<address>400 Main</address>
<country>Norway</country>
</shipTo>

<items>
<item>
<title>Wheel</title>
<quantity>1</quantity>
<price>10.90</price>
</item>

<item>
<title>Cam</title>
<quantity>1</quantity>
<price>9.90</price>
</item>
</items>

</shipOrder>
```

XML Schema for Shipping Order

```
<xsd:schema xmlns:xsd=http://www.w3.org/1999/XMLSchema>

<xsd:element name="shipOrder" type="order"/>

<xsd:complexType name="order">
  <xsd:element name="shipTo" type="shipAddress"/>
  <xsd:element name="items" type="cdItems"/>
</xsd:complexType>

<xsd:complexType name="shipAddress">
  <xsd:element name="name“ type="xsd:string"/>
  <xsd:element name="street" type="xsd:string"/>
  <xsd:element name="address" type="xsd:string"/>
  <xsd:element name="country" type="xsd:string"/>
</xsd:complexType>
```

XML Schema - Shipping Order (continued)

```
<xsd:complexType name="cdItems">
  <xsd:element name="item" minOccurs="0"
               maxOccurs="unbounded" type="cdItem"/>
</xsd:complexType>

<xsd:complexType name="cdItem">
  <xsd:element name="title" type="xsd:string"/>
  <xsd:element name="quantity“
               type="xsd:positiveInteger"/>
  <xsd:element name="price" type="xsd:decimal"/>
</xsd:complexType>

</xsd:schema>
```

Purchase Order – A more detailed example

- **Instance document**: An XML document that conforms to an XML Schema
- Elements that contain sub-elements or carry attributes are said to have **complex types**
- Elements that contain numbers (and strings, and dates, etc.) but do not contain any sub-elements are said to have **simple types.**
- **Attributes** always have simple types.

Purchase Order – A more detailed example

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">

  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>

  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
```

Purchase Order – Continued

```
<comment>Hurry, my lawn is going wild!</comment>
<items>

  <item partNum="872-AA">
    <productName>Lawnmower</productName>
    <quantity>1</quantity>
    <USPrice>148.95</USPrice>
    <comment>Confirm this is electric</comment>
  </item>

  <item partNum="926-AA">
    <productName>Baby Monitor</productName>
    <quantity>1</quantity>
    <USPrice>39.98</USPrice>
    <shipDate>1999-05-21</shipDate>
    </item>
  </items>

</purchaseOrder>
```

Purchase Order – Continued

Defining the USAddress Type

```
<xsd:complexType name="USAddress" >
<xsd:sequence>
  <xsd:element name="name"   type="xsd:string"/>
  <xsd:element name="street" type="xsd:string"/>
  <xsd:element name="city"   type="xsd:string"/>
  <xsd:element name="state"  type="xsd:string"/>
  <xsd:element name="zip"    type="xsd:decimal"/>
</xsd:sequence>
<xsd:attribute name="country"
               type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

Purchase Order – Continued

In contrast, the `PurchaseOrderType` definition contains element declarations involving complex types.

```
<xsd:element name="comment" type="xsd:string"/>
```

The `comment` element is globally defined under the schema element.

```
<xsd:complexType name="PurchaseOrderType">
<xsd:sequence>
  <xsd:element name="shipTo" type="USAddress"/>
  <xsd:element name="billTo" type="USAddress"/>
  <xsd:element ref="comment" minOccurs="0"/>
  <xsd:element name="items"  type="Items"/>
</xsd:sequence>

<xsd:attribute name="orderDate" type="xsd:date"/>

</xsd:complexType>
```

Purchase Order – Continued

```
<xsd:complexType name="Items">
<xsd:sequence>
  <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="productName" type="xsd:string"/>
        <xsd:element name="quantity">
          <xsd:simpleType>
            <xsd:restriction base="xsd:positiveInteger">
              <xsd:maxExclusive value="100"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="USPrice"  type="xsd:decimal"/>
        <xsd:element ref="comment"   minOccurs="0"/>
        <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
```

Purchase Order – Continued

```
<!-- Stock Keeping Unit, a code for identifying
   products -->
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

- The above type restricts the SKU code to start with 3 digits followed by a “-” followed by 2 upper-case letters.
- The earlier example of restricting a simple type was “quantity” wit a sub-type of 1 to 99.
- Restriction of a simple type starts with a “base” simple type and using “pattern” elements are restricted to a subset.

Purchase Order – Continued

Complete XML Schema Specification:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:annotation>
  <xsd:documentation xml:lang="en">
    Purchase order schema for Example.com.    Copyright 2000
    Example.com. All rights reserved.
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

<xsd:element name="comment" type="xsd:string"/>

  Complex Type PurchaseOrderType
  Complex Type USAddress
  Complex Type Items
  Simple Type SKU

</xsd:schema>
```

Deriving New Simple Types

A large collection of built-in types are available in XML Schema

`xsd:string, xsd:integer, xsd:positiveInteger,`
`xsd:decimal, xsd:boolean, xsd:date, xsd:NMTOKENS`, etc.

Deriving New Simple Types: We have seen two examples: SKU and Quantity. The following example defines `myInteger` (value between 10000 and 99999) using two `facets`

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Deriving new Simple types - Continued

Enumeration facet:

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Deriving new Simple types - Continued

XML Schema has 3 built-in list types: NMTOKENS, IDREFS, ENTITIES

Creating new list types from simple types:

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

The following XML fragment conforms to the above SimpleType:

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

Deriving new Simple types - Continued

Several facets can be applied to list types: `length, minLength, maxLength, enumeration`

For example, to define a list of exactly six US states (`SixUSStates`)

- First define a new list type called `USStateList` from `USState`
- Then derive `SixUSStates` by restricting `USStateList` to only six items

```
<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>

<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>

<sixStates>PA NY CA NY LA AK</sixStates>
```

Deriving Complex Types from Simple Types

So far we have seen how to introduce “attributes” in elements of Complex Types. How to declare an element that has simple content and an attribute as well such as:

```
<intPrice currency="EUR">423.46</intPrice>
```

This is done as follows:

```
<xsd:element name="intPrice">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Deriving Complex Types from Simple Types

How to declare an empty element with one or more attributes:

```
<intPrice currency="EUR" value="423.46"/>
```

```
<xsd:element name="intPrice">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:attribute name="currency" type="xsd:string"/>
        <xsd:attribute name="value"    type="xsd:decimal"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

XML Schema - Summary

- A flexible and powerful schema language
- Syntax is XML itself
- Variety of data types and ability to extend type system
- Variety of data “facets” and “patterns” to impose domain constraints
- Can define advanced constraints such as “primary key” and “referential integrity”