Reusing types in XML Schema
Reusing Schemas

- Many benefits
  - sharing existing definitions
  - faster development
- Traditional techniques for schema reuse:
  - some notion of import and the ability to resolve name conflicts
  - inheritance, based on sub-typing
Reusing XML Schemas: Extending Types

- Extension allows to add new fields in a complex type: inheritance

```xml
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="country" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```
Reusing XML Schemas: Extending Types

Extension allows to add new fields in a complex type: inheritance

```xml
<xsd:complexType name="USAddress" >
  <xsd:complexContent>
    <xsd:extension base="Address">
      <xsd:sequence>
        <xsd:element name="state" type="USState"/>
        <xsd:element name="zip" type="USZipCode"/>
      </xsd:sequence>
      <xsd:attribute name="phone" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Conceptually same as:
Reusing XML Schemas: Extending Types

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="country" type="xsd:string"/>
    <xsd:element name="state" type="USState"/>
    <xsd:element name="zip" type="USZipCode"/>
  </xsd:sequence>
  <xsd:attribute name="phone" type="xsd:string"/>
</xsd:complexType>
Reusing XML Schemas: Extending Types

- Writing XML documents in the presence of derived types:
- using the xsi:type attribute at the instance level

```xml
<nyAddress xsi:type="po:USAddress">
  <street>115W 82ND</street>
  <city>New York City</city>
  <country>USA</country>
  <state>NY</state>
  <zip>10024</zip>
  <phone>212 579 4946</phone>
</nyAddress>

<address xsi:type="po:Address">
  <street>115W 82ND</street>
  <city>New York City</city>
  <country>USA</country>
</address>
```
Reusing XML Schemas: Restricting Types

- It is possible to derive new types by restricting the content models of existing types
- similar to restricting simple types
  - but must repeat the definition of the base type.
Reusing XML Schemas: Restricting Types

- It is possible to derive new types by restricting the content models of existing types.

```xml
<xsd:complexType name="USAddress">
  <xsd:complexContent>
    <xsd:restriction base="po:Address">
      <xsd:sequence>
        <xsd:element name="country" type="xsd:string" fixed="USA"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

*include the definition of Address as is, except the elements/attributes that have been restricted*
Reusing XML Schemas: Restricting Types

- Idea: represent *set inclusion*
- Spirit is to allow:
  - *restricted datatypes*
    - e.g., the code for US states is not any string, but a two-letter word.
  - narrowed range for sequences
    - e.g., changing the min and max occurs for an element
- ...

*can be reduced to restricting facets of the XML Schema data types*
Reusing XML Schemas: Restricting Derivations

- disallow all derivations of a type
- disallow extensions of a type
- disallow restrictions of a type

```
<xsd:complexType name="Address" final="#all" ...>
This type cannot be extended nor restricted
```

```
<xsd:complexType name="Address" final="extended" ...>
This type cannot be extended
```

```
<xsd:complexType name="Address" final="restriction" ...>
This type cannot be restricted
```
Reusing XML Schemas: Restricting Derivations

- We can disallow a base type from being replaced by the types obtained from it by restriction or extension

```xml
<xsd:complexType name="Address" block="restriction">
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city" type="xsd:string"/>
   <xsd:element name="country" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="PurchaseOrder">
   <xsd:element name="shipTo" type="Address"/>
   ...
</xsd:complexType>
```
We can disallow a base type from being replaced by the types obtained from it by restriction or extension.

```xml
<xsd:complexType name="PurchaseOrder">
    <xsd:element name="shipTo" type="Address"/>
    ...
</xsd:complexType>
<xsd:element name="order" type="PurchaseOrder"/>

<order>
    <shipTo xsi:type="USAddress">
        <street>115W 82nd</street>
        <city>New York</city>
        <country>USA</country>
        ...
    </shipTo>
</order>
```

**XML Schema**

**Invalid Instance**
Reusing XML Schema Types: Short Comings

- **Restriction and Extension are not possible together**
  - cannot add elements/attributes to a type
  - and
  - restricting the definition of an *existing sub-element* of the type
Reusing XML Schema Types: Short Comings

- Restriction is purely syntactic, and is not taken into consideration directly in querying XML documents

  - E.g., if asking for all addresses, one would expect to get the addresses whose type is US
    - write it explicitly in the query!
## DTD vs. XML Schema

<table>
<thead>
<tr>
<th>Features</th>
<th>DTD</th>
<th>XML Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax in XML</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supporting Namespace</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>include &amp; import</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No. Built-in types</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>User defined types</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Type domain constraints</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Explicit Null value</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Type extension</td>
<td>No</td>
<td>Yes Except Simple Type</td>
</tr>
<tr>
<td>Attribute default value</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Choice among attributes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Optional/required Attr.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Attribute domain constraints</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Element default value</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>Element content model</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Choice among elements</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Min &amp; Max Occurrence</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Unordered List</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## DTD vs. XML Schema

<table>
<thead>
<tr>
<th>Features</th>
<th>DTD</th>
<th>XML Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constraints</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniqueness for attributes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Uniqueness for non-attributes</td>
<td>No</td>
<td>YES</td>
</tr>
<tr>
<td>Key for attributes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Key for non-attributes</td>
<td>No</td>
<td>YES</td>
</tr>
<tr>
<td>Foreign key for attributes</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Foreign key for non-attributes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Misc.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open model</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Documentation</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Embedded HTML</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Self-describability</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Context-dependent XML Typing

- **DTDs** do not allow one to use the same element but with different structure in different contexts.
- Cannot distinguish between *used car* and *new car* postings:

```
dealer
   /\    /
  UsedCars  NewCars
     \    /  \       /  
    posting posting
       \         /  
       model year model
```
Context-dependent XML Typing

- DTDs do not allow one to use the same element but with different structure in different contexts

- Specialized DTDs allow one to decouple the element names from types!
Specialized DTDs

- A specialized DTD over an alphabet $\Sigma$ is a pair $(d, \mu)$ with
  - $d$ a DTD defined over the types in $\Sigma' \rightarrow \Sigma$
  - $\mu: \Sigma' \rightarrow \Sigma$ is a function that maps types to element names

Dealer $\rightarrow$ UsedCars NewCars
UsedCars $\rightarrow$ postingUsed$^*$
NewCars $\rightarrow$ postingNew$^*$
postingUsed $\rightarrow$ model year
postingNew $\rightarrow$ model

$\mu(\text{Dealer}) = \text{Dealer}$
$\mu(\text{UsedCars}) = \text{UsedCars}$
$\mu(\text{NewCars}) = \text{NewCars}$
$\mu(\text{postingUsed}) = \text{posting}$
$\mu(\text{postingNew}) = \text{posting}$
Specialized DTDs

- UsedCars
  - posting\textsuperscript{used}
    - model
    - year
  - posting\textsuperscript{new}
    - model

- NewCars
  - posting\textsuperscript{new}
    - model
  - posting\textsuperscript{used}
    - model
Formal Foundations of XML Schemas

- Basic Questions on XML Schemas (~ Specialized DTDs)
  - Validation: how hard?
  - Expressive power: what can be defined?
  - Closure properties: union, intersection, difference
  - Complexity of manipulations
- Tool: powerful connection to tree automata!
Formal Foundations of XML Schemas

- **Theorem**: Specialized DTDs define precisely the regular tree languages (over unranked trees) and so are equivalent to top-down and bottom-up non deterministic tree automata

- **Closure properties**: union, intersection, complement

- **Algorithms** for:
  - validation wrt. DTD,
  - decidable inclusion testing of DTDs,
  - computing DTDs for union or intersection, etc.

- **Static analysis**
XML DDL: Summary

- In XML:
  1. from grammar to object-oriented
  2. schema wired with the data
  3. emphasis on semantics for exchange

- Formal Expressive Power:
  1. XML Schema can express precisely the regular tree languages (over unranked trees)
XML DDL: Summary

- XML Schemas are a tremendous advancement over DTDs:
  - *Enhanced datatypes*
    - 37+ versus 10
    - Can create your own datatypes
    - Can define the lexical representation
  - *Written in XML*
    - Enables use of XML tools
  - *Object-oriented flavor*
    - Can extend or restrict a type
  - Can *express sets*: the child elements may occur in any order
  - Can *specify element content as being unique* (integrity constraints and uniqueness constraints)
  - Can *define elements with the same name but different type*
  - Can *define elements with null content*
XML DDL: Summary

- Complete but complex XML Schema specification...
  - Many research work with interesting and complementary properties
  - Yet no approach that reconciles all of the above
- And still some difficult problems to solve:
  - Concrete integrity constraint language that is tractable
  - syntactic vs. semantics notion of sub-typing?
- Use of types for language typing
- Use of types for query processing
- Use of types for storage