COM506 Professional Web Services Development

Practical A2: XML Validation using Schema

Aims

- To explore the limits of DTD validation of XML data
- To introduce XML Schema as a type-aware validation technique
- To present an XML Schema as an alternative to DTD validation
- To demonstrate XSD techniques for restricting the scope of simple (valued) elements
- To demostrate XSD techniques for specifying the composition of complex elements

Contents

A2.1 Limitations of DTD Validation

In the last practical we saw how we can use Document Type Definitions (DTDs) to impose structure on XML documents. Using the XML plugin in Notepad++, we were able to ensure that our XML documents conformed to the intended architecture.

However, there is a serious limitation to DTD validation that can be illustrated by the following example.

Consider the simple XML file *stock.xml* that records stock levels for a range of products.

```
<?xml version="1.0"?>
<!DOCTYPE stock SYSTEM "stock.dtd">

<stock>

<item id="101">
   <name>Baked Beans</name>
   <numInStock>20</numInStock>
</item>

<item id="102">
   <name>Canned Soup</name>
   <numInStock>35</numInStock>
</item>

</stock>
```

The file is validated by a DTD *stock.dtd*.

```
<!ELEMENT stock (item*)>

<!ELEMENT item (name, numInStock)>
<!ATTLIST item id CDATA #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT numInStock (#PCDATA)>
```

Do it now!
Load the files *stock.xml* and *stock.dtd* into Notepad++. Now validate the XML document by selecting *Plugins* | *XML Tools* | *Validate now*.

As the XML structure matches that in the DTD definition, the validator responds with a positive result.

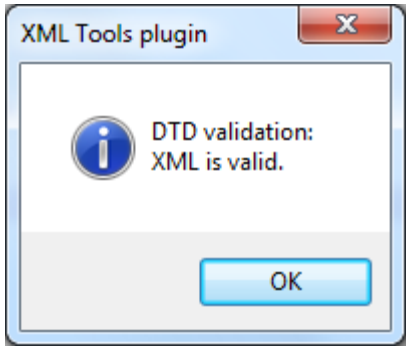

Figure A2.1. Valid DTD

Do it now!
Now edit the `<numInStock>` value for one of the items to a nonsensical value – for example `<numInStock>cheese</numInStock>`. Run the XML validator again.

You should find that the validator continues to produce a positive result – even though the value for the `<numInStock>` element is clearly invalid. This lack of data type checking is the biggest single drawback of the DTD approach to XML validation.

A2.2 Introducing XML Schema

An XML Schema defines the structure of an XML document. As for DTDs, Schema define the elements and attributes that can appear in a document, as well as the order, number and relationship of child elements.

Unlike DTDs, however, XML Schema also define the **data types** of element values – using both a range of standard types and also permitting the definition of specialised complex types.

XML Schema documents are themselves written in XML, so they can be parsed and validated like any regular XML document.

An XML Schema for the *stock.xml* example is presented below.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

<xs:element name="stock">
  <xs:complexType>
     <xs:sequence>

       <xs:element name="item" maxOccurs="unbounded">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="name" type="xs:string"/>
             <xs:element name="numInStock" type="xs:integer"/>
           </xs:sequence>

           <xs:attribute name="id" type="xs:integer"
                         use="required"/>

         </xs:complexType>
       </xs:element>
     </xs:sequence>

   </xs:complexType>
</xs:element>

</xs:schema>
```

There is a lot of new terminology in this document, so it is worth examining it carefully.

The **<xs:schema>** element is the root element of every XML Schema and serves 2 purposes. First, it defines the **xs** namespace, using the URL of the W3C Schema definition as the identifier. (Remember that this also requires that we prefix every element from this namespace with **xs**). Secondly, the **elementFormDefault="qualified"** attribute indicates that any elements declared in this Schema used by an XML document that references this Schema must include a namespace prefix. This is rather technical, but it is sufficient for our purposes that you simply use this **<xs:schema>** syntax for every Schema you produce.

Next, we define the **<stock>** element and specify that it is of **complexType** (i.e. contains further elements rather than a literal value) and that it's components represent a `sequence`. (Note that in this, we have a sequence of exactly one element (**<item>**), but we still have to define it as a sequence. We will see alternatives to this later in this practical.

Now, we define format of the **<item>** element, which is repeated a number of times within **<stock>**. This is also of **complexType**, comprising further elements that are ordered as a sequence. Note the **maxOccurs="unbounded"** attribute in the **<xs:element>** definition, specifying that we can have any number of **<item>** elements within the **<stock>**. There is also a corresponding **minOccurs** attribute that can be specified.

Note: **minOccurs** and **maxOccurrs** normally have numeric values. For example, to specify that that a number should be between 1 and 10 (inclusive), we would set **minOccurs** to "1" and **maxOccurs** to "10".

Within the **<item>** definition, we come to the two values that define the element. Now we can see how to specify whether an element's data should contain an integer value, a string value, or some other type. There is a wide range of types available, but the most common are shown in the table below.

xs:boolean	e.g. true, false
xs:date	e.g. 2017-09-29 (yyyy-mm-dd)
xs:decimal	e.g. 1.92, 123.456, etc.
xs:integer	e.g. 100, 200, 300, etc.
xs:string	e.g. "Any character data"
xs:time	e.g. 09:15:00 (hh:mm:ss)

Finally, we define the `id` attribute for the **<item>** element. Note that the definition of the attribute comes after the definition of the object, and inside the **complexType** qualifier. The **use="required"** assignment denotes that the attribute is compulsory. If this is omitted, then the attribute is deemed to be optional.

Do it now!
Carefully study the Schema definition and ensure that you understand the purpose of each line of code.

In order to use this Schema to validate the XML document, we need to connect it through the XML root note.

```
<?xml version="1.0"?>
<stock
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="stock.xsd">

<item id="101">
   <name>Baked Beans</name>
   <numInStock>50</numInStock>
</item>

<item id="102">
   <name>Canned Soup</name>
   <numInStock>35</numInStock>
</item>

</stock>
```

Do it now!
Load the files *stockWithSchema.xml* and *stock.xsd* into Notepad++. Now validate the XML document by selecting *Plugins* | *XML Tools* | *Validate now*.

If all is well, you should get the message in Figure A2.2 reporting that the XML has been successfully validated

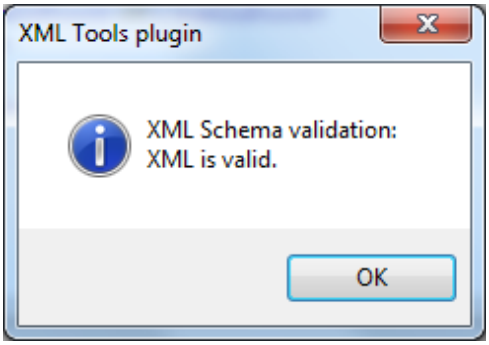

Figure A2.2. Successful Schema Validation

The purpose of using a Schema was to enable the kind of type checking that the DTD was unable to trap. Let's now repeat the previous experiment.

Do it now!
Once again, edit the **`<numInStock>`** value for one of the items to a nonsensical value – for example **`<numInStock>cheese</numInStock>`**. Run the XML validator again.

This time we should have a different result. The new string value in the **`<numInStock>`** element violates the schema requirement for an **`<xs:integer>`** value, and the error message shown in Figure A2.3 is produced.

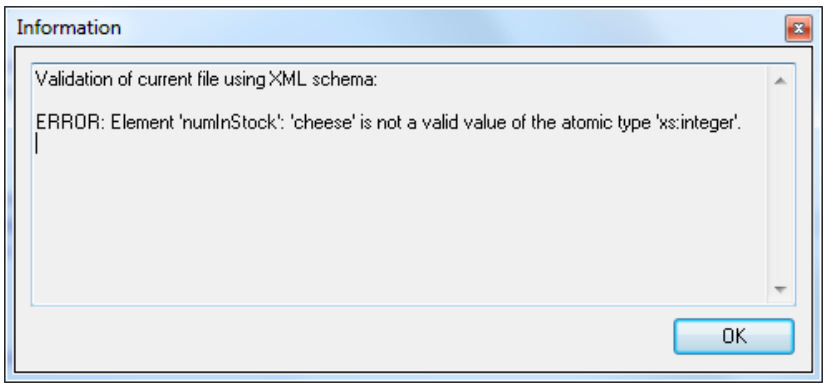

Figure A2.3. Unsuccessful Schema Validation

Try it now!
We wish to add a new element called **<price>** to the definition of an **<item>**. Hence, a new **<item>** might be

```
<item id="103">
   <name>Meatballs</name>
   <numInStock>23</numInStock>
   <price>1.37</price>
</item>
```

Make the changes to the XML and Schema files that support this modification. Test your modifications by validating the XML against the Schema.

Try it now!
New company policy requires that we must have between 5 and 10 products (**<item>**s) in stock at all times. Make the change to the Schema that enforces this rule and provide XML files that (a) violate the minimum constraint, (b) violate the maximum constraint, and (c) satisfy this new policy.

Try it now!
Revisit the *moduleCatalogue.xml* example from Practical A1. Create the XML file *moduleCatalogueSchema.xml* that is validated by the Schema file *moduleCatalogueSchema.xsd*.

A2.3 XML Schema Elements

The stock example in the previous section gives a flavor of XML Schema definition, but there are many more options that enable us to very precisely specify the structure and types of the elements in our XML data.

A formal classification of XML Schema elements is illustrated in Figure A2.4.

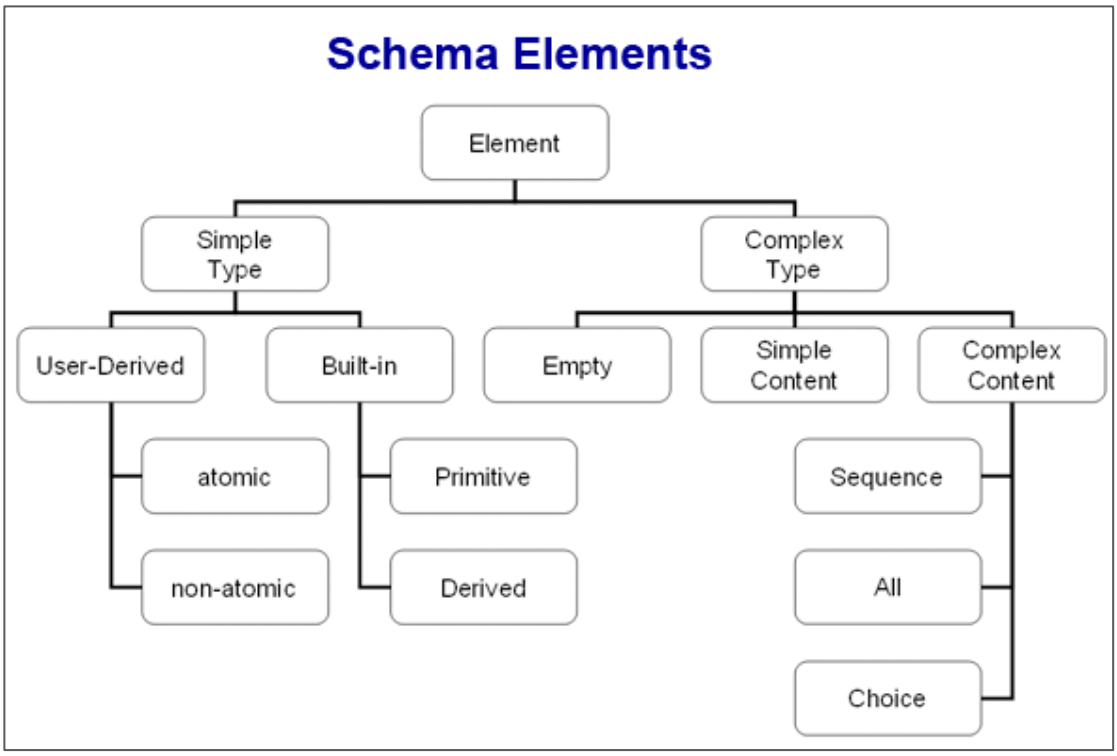

Figure A2.4. XML Schema Elements

The top-level division is between simple and complex elements, where **simple elements** are those that contain typed values (e.g. integer, Boolean, date, string, etc.) and **complex elements** are those that comprise further XML structures (e.g. **`<item>`** in our stock example.

A2.3.1 Simple Elements

We have already seen the main data types available (**`boolean`**, **`date`**, **`integer`**, **`string`**, etc.), but there is also a wide range of qualifiers that limit the allowable range of values within those types. Some of the most useful are summarised below.

String Length

When restricting the scope of a simple element type, we have to specify the element within a **`<xs:simpleType>`** definition This provides a container in which we can state that we wish to restrict (**`<xs:restriction>`**) the data type (e.g. **`base="xs:string"`**). Finally, we can use **`<xs:length>`** to specify the nature of the restriction.

```
<xs:element name="password">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:length value="8" />
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

We can also use **<xs:minLength>** and **<xs:maxLength>** to specify lower and upper limits on string length. For example, to specify that the password element must be between 6 and 12 characters we could use

```
<xs:element name="password">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:minLength value="6" />
         <xs:maxLength value="12" />
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

Limiting Numeric Values

There are a range of numeric limiters that place bounds on the size of the number and the number of digits used. For example, to specify an integer percentage value (maybe a student mark) between 0 and 100, we could use

```
<xs:element name="examResult">
   <xs:simpleType>
      <xs:restriction base="xs:integer">
         <xs:minInclusive value="0" />
         <xs:maxInclusive value="100" />
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

There are also corresponding qualifiers **minExclusive** and **maxExclusive**, which would specify a range greater than the **minExclusive** value and less than the **maxExclusive** value (i.e. 1-99 in the example above).

We can also specify the precision of numeric data by using **<xs:totalDigits>** and **<xs:fractionDigits>**, which limit the total number of digits to be used in an **<xs:decimal>** value and the number of digits to be used after the decimal point. For example, to specify that a price value can have no more that 5 digits in total and that 2 of them must come after the decimal point, we could use

```
<xs:element name="price">
   <xs:simpleType>
      <xs:restriction base="xs:decimal">
         <xs:totalDigits value="5" />
         <xs:fractionDigits value="2" />
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

Enumerated Values

Sometimes we have a pre-defined set of values, where an element can take one of the set but no other. For example, a footballer might be described as one of "goalkeeper", "defender", "midfielder" or "striker". In XML Schema we can define this as

```
<xs:element name="position">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:enumeration value="goalkeeper" />
         <xs:enumeration value="defender" />
         <xs:enumeration value="midfielder" />
         <xs:enumeration value="striker" />
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

Try it now!
Modify the stock example to explore the simple element value restrictions presented above. For example, you might:

a) limit the name of an item to a maximum of 10 characters so that it fits on a shelf edge label
b) make sure that no price is greater than 4 digits, with no more than 2 digits after the decimal point
c) only permit a limited range of products such as Beans, Soup or Meatballs

A2.3.2 Complex Elements

Complex elements are those that comprise other elements (i.e. anything that is not a simple string/integer/date/etc. value). The **<xs:complexType>** element provides for 3 types of *model groups* which specify the structure and order in which child elements can appear within their parent element..

<xs:sequence>

As we have already seen, the **<xs:sequence>** element defines a complex element as a sequence, where the child elements must appear in the order in which they are defined. For example, if we are defining a menu consisting of a starter, followed by a main course, followed by a dessert, we might have

```
<xs:element name="menu">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="starter" type="xs:string" />
         <xs:element name="main" type="xs:string" />
         <xs:element name="dessert" type="xs:string" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

<xs:all>

Sometimes we have a set of elements that must be included in the description of an item, but we don't care about the order in which they are presented. The **<xs:all>** element allows us to specify a set of elements that can appear in any order. For example, the ingredients of beans on toast might be specified as

```
<xs:element name="beansOnToast">
   <xs:complexType>
      <xs:all>
         <xs:element name="bread" type="xs:string" />
         <xs:element name="butter" type="xs:string" />
         <xs:element name="beans" type="xs:string" />
      </xs:all>
   </xs:complexType>
</xs:element>
```

<xs:choice>

Where only one of a selection of elements is required, the **<xs:choice>** tag enables us to list a selection of alternatives. For example, in a student records database, the year of study might be recorded as

```
<xs:element name="yearOfStudy">
   <xs:complexType>
      <xs:choice>
         <xs:element name="Year 1" type="xs:string" />
         <xs:element name="Year 2" type="xs:string" />
         <xs:element name="Year 3" type="xs:string" />
         <xs:element name="Year 4" type="xs:string" />
      </xs:choice>
   </xs:complexType>
</xs:element>
```

Try it now!
Experiment with the stock example to test the **<xs:all>** and **<xs:choice>** complex element constructs presented above (**<xs:sequence>** has already been demonstrated). For example you might:

a) allow the **<name>**, **<numInStock>** and **<price>** elements to be presented in any order

b) only allow any one of the elements to be presented

Try it now!
Revisit the *letter* example from Practical A1 (the initial version without namespaces) and replace your DTD validation with the Schema equivalent. Make sure that you take advantage of the enhanced type checking capabilities of Schema validation.

A2.4 Further Information

- http://www.learn-xml-schema-tutorial.com
  XML Schema Tutorial

- http://www.w3schools.com/xml/schema_intro.asp
  W3Schools Schema Tutorial

- http://en.wikipedia.org/wiki/XML_schema
  Schema definition from Wikipedia

- http://www.differencebetween.net/technology/difference-between-xml-schema-and-dtd/
  Difference between DTD and Schema

- https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes
  XML Schema Tutorial – Defining Elements and Attributes

- http://www.xml.com/pub/a/2001/08/22/easyschema.html
  XML Schema complex types