XML Schema 1.1, Part 3: An introduction to XML Schema 1.1

Evolve your schema with powerful wildcard support

Neil Delima
Sandy Gao
Michael Glavassevich
Khaled Noaman

November 20, 2009
(First published November 17, 2009)

A frequent goal of schema authors is to build schemas for extensibility, where wildcards play a key role in providing extensibility points. New wildcard features introduced in XML Schema 1.1 make it easier for schema authors to write extensible schemas that can tolerate changes in the future. In this third of a six part series of articles, authors Neil Delima, Sandy Gao, Michael Glavassevich, and Khaled Noaman take an in depth look at versioning features introduced by XML Schema 1.1, specifically the new powerful wildcard mechanisms and open content.

Introduction

During the W3C Workshop on XML Schema 1.0 User Experiences (see Related topics), schema versioning was one of the major concerns from schema users. When the XML data changes, the corresponding schemas also need to change. How do you ensure a level of compatibility to reduce disruptions to the applications?

Frequently used acronyms

- URI: Uniform Resource Identifier
- W3C: World Wide Web Consortium
- XML: Extensible Markup Language

People often talk about two kinds of compatibility. In the schema versioning context, backward compatibility requires that valid instances of schema version $n$ remains valid under schema version $n+1$. This is what people often have in mind when they talk about compatibility, and it’s the easier

20 Nov 2009: Updated per author request: Under heading Open content at the schema document level in paragraph 4, sentence 2, changed string “A value of mode indicates...” to “A value of none indicates...”
one to support, because the authors of schema version n+1 have access to both the schema and instances of version n.

The other kind is forward compatibility, where valid instances of schema version n+1 are also valid under schema version n. This is normally harder to achieve, because the author does not know what kind of changes might be introduced in the next version. All you can do is leave extension points in the schema to allow future extensions.

Because of the importance and difficulty in achieving forward compatibility, one of the major goals in XML Schema 1.1 is to make it easy to write forward compatible schemas. Wildcards play a key role in defining extension points in schemas, and are the focus of this article. The next article in the series will discuss other features related to schema versioning.

The W3C XML Schema working group published a Versioning Guide for XML Schema 1.1 (see Related topics). Those who seek help for versioning their schemas might also find its content interesting.

**Weakened wildcards**

Schema authors who create a complex type definition where they mix a sequence of elements and wildcards that allow the same namespace(s) as the other elements might discover that the schema they have written is invalid. The most likely reason for this error is a violation of the Unique Particle Attribution (UPA) rule defined in XML Schema 1.0 which basically states that the matching particle (for example, &lt;xs:element&gt; or &lt;xs:any&gt; in the complex type definition) can be unambiguously determined for each of the elements in the instance document. This determinism simplifies the implementation of the validator and can be useful for applications which require a mapping between elements in the instance document and particles in the schema. But it also challenges schema authors to naturally express the content they wish to allow.

The schema snippet in Listing 1 illustrates the issue that schema authors commonly face when they attempt to create extensibility points using wildcards. Consider a complex type which models the win-loss record for a sports team. In some sports like American football, ties are allowed. In others, such as basketball, a game continues until a winner is declared. A schema author might choose to make ties an optional element (with minOccurs="0"). There are potentially other statistics which can be included in a team's record aside from wins, losses, and ties, and so you might want to allow additional content with a wildcard which can be defined in a future version of the schema.

**Listing 1. Schema snippet - A win-loss record type definition**

```
&lt;xs:complexType name="record"&gt; &lt;xs:sequence&gt; &lt;xs:element name="wins" type="xs:nonNegativeInteger"/&gt; &lt;xs:element name="losses" type="xs:nonNegativeInteger"/&gt; &lt;xs:element name="ties" type="xs:nonNegativeInteger" minOccurs="0"&gt; &lt;xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any" processContents="lax"/&gt; &lt;/xs:sequence&gt; &lt;/xs:complexType&gt;
```
The issue with the above complex type definition can be illustrated with the instance document in **Listing 2**. The `wins` and `losses` elements in this instance match up with their element declarations in the schema (see **Listing 1**). When you attempt to map the `ties` element back to the complex type, you find that two choices for the particle could have matched. It could either be the `ties` element declaration (which is optional) or the wildcard which also allows `ties` to appear in the instance. Because this schema had more than one potential mapping, it violates the Unique Particle Attribution (UPA) rule in XML Schema 1.0 and thus is invalid.

**Listing 2. XML snippet - An invalid win-loss record element**
```
<record> 
  <wins>20</wins> 
  <losses>15</losses> 
  <ties>8</ties> 
  <points>48</points> 
</record>
```

As a workaround, a schema author might place a required element in between the optional one and the wildcard as in **Listing 3**. Because the `separator` element must appear in the instance there is no ambiguity between content which matches the `separator` element declaration and the wildcard which follows it.

**Listing 3. Schema snippet - Defining a required element between optional element and optional wildcard**
```
<xs:complexType name="record"> 
  <xs:sequence> 
    <xs:element name="wins" type="xs:nonNegativeInteger"/> 
    <xs:element name="losses" type="xs:nonNegativeInteger"/> 
    <xs:element name="ties" type="xs:nonNegativeInteger" minOccurs="0"/> 
    <xs:element name="separator"/> 
    <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any" processContents="lax"/> 
  </xs:sequence> 
</xs:complexType>
```

While you can often add a required element to avoid the UPA error, the content introduced into instances is often meaningless or forces an unnatural ordering of the data. Take a look at **Listing 4**. The `separator` element introduced contributes no information to the document yet must be there for the document to be valid. Ideally you do not want such an element to be part of the document.

**Listing 4. XML snippet - A valid win-loss record element**
```
<record> 
  <wins>20</wins> 
  <losses>15</losses> 
  <ties>8</ties> 
  <separator/> 
  <points>48</points> 
</record>
```

To make it easier for schema authors to create more natural content models, XML Schema 1.1 has introduced the concept of a weakened wildcard. The weakened wildcard is a relaxation of the UPA rule which resolves the contention between an element declaration and wildcard by stating that the element declaration always takes precedence over the wildcard. As a consequence, the complex type definition in **Listing 1** becomes valid in XML Schema 1.1 because the ambiguity between the element declaration and the wildcard no longer exists. The reason the wildcard was added in the first place was to allow for schema evolution. Imagine that at some point in the future we updated the definition of the `record` type to include a `points` element as in **Listing 5**. Now the `points` element in the instance in **Listing 2** is defined and because of the weakened wildcard rule it unambiguously matches its element declaration.
Listing 5. Schema snippet - An expanded win-loss record type definition

```xml
<xs:complexType name="record">
  <xs:sequence>
    <xs:element name="wins" type="xs:nonNegativeInteger"/>
    <xs:element name="losses" type="xs:nonNegativeInteger" minOccurs="0"/>
    <xs:element name="ties" type="xs:nonNegativeInteger" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="points" type="xs:nonNegativeInteger" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Negative wildcards

Sometimes it is desirable for a wildcard to not match certain names. For example, in schema 1.0, `##other` can be specified as the value of the `namespace` attribute on a wildcard (`&lt;any&gt;` or `<anyAttribute>`), indicating that this wildcard matches namespaces in namespaces other than the target namespace of the current schema document. This feature has proven very useful in leaving extension points in schemas.

But some scenarios cannot be met by `##other`. XML Schema 1.1 introduced a few mechanisms to specify exceptions for wildcards. They can collectively be called negative wildcards.

Namespace exclusion

`##other` can only be used to exclude a single namespace: the target namespace. What if you want to exclude more than one namespace? For example, if version 1 of a schema uses target namespace ".../V1", and version 2 of the schema uses ".../V2". The author might wish to leave extension points to allow names in any namespaces except for those in the namespaces of either version 1 or version 2. Listing 6 shows how you can now express this in XML Schema 1.1.

Listing 6. Schema snippet - Namespace exclusion in XML Schema 1.1

```xml
<xs:complexType>
  <xs:sequence>
    ... ...
  </xs:sequence>
</xs:complexType>
```

With this new `notNamespace` attribute, you can specify namespaces that the wildcard should not match, which has the opposite meaning of the `namespace` attribute. Obviously, only one of these two attributes is needed on a wildcard.

The `notNamespace` attribute expects a space separated list of any URI values. Similar to the `namespace` attribute, `notNamespace` also allows the special symbols `##targetNamespace` and `##local` in the list, to indicate the target namespace and the empty namespace respectively.

QName exclusion

Wildcards are often used to match names other than those explicitly specified. Listing 7 shows an example of such a case.

Listing 7. Schema snippet - Wildcards matching names other than those explicitly specified

```xml
<xs:complexType name="referenceType">
  <xs:sequence>
    <xs:element ref="tns:uri"/>
    <xs:element ref="tns:description" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```
Each reference type requires a uri child element and an optional description child element followed by any number of child elements for extensions. This seems to work fine; unfortunately, it also allows the following instance (Listing 8):

**Listing 8. XML snippet - A reference element with multiple uri children**

```xml
&lt;reference&gt; &lt;uri&gt;...&lt;/uri&gt; &lt;uri&gt;...&lt;/uri&gt; &lt;/reference&gt;
```

Now the application processing the reference element will have trouble deciding which uri child element to use. This is caused by the wildcard matching more names than intended. To fix this, you can use the new disallowed names concept introduced in XML Schema 1.1, as in Listing 9.

**Listing 9. Schema snippet - Using disallowed names**

```xml
&lt;xsd:complexType name="referenceType"&gt; &lt;xsd:sequence&gt; &lt;xsd:element ref="tns:uri"/&gt; &lt;xsd:element ref="tns:description minOccurs="0"/&gt; &lt;xsd:any processContents="lax" notQName="tns:uri" minOccurs="0" maxOccurs="unbounded"/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;
```

With the notQName attribute, the schema author can provide a list of QNames that the wildcard should not match. This updated type definition forbids the above instance with two uri child elements.

**Exclusion of known siblings**

Sometimes the schema author might wish to exclude a long list of names, which makes it difficult to use the notQName attribute specifying all those names. XML Schema 1.1 identified two cases that can happen very often, and provided mechanisms to simplify them.

If you define a complex type describing a person, there will be many elements in the type, for the name, date of birth, address, occupation, and so on. If you also want to use a wildcard (or an open content) to allow additional information to be added, then you want to limit the wildcard to not match elements already declared in the type.

To do this, use the notQName attribute and list all the known element names. Not only would the exclusion list be very long, it would also be difficult to maintain. If a new element is added to the type, you have to remember to add its name to notQName. In XML Schema 1.1, such an exclusion can be easily described using ##definedSibling (Listing 10):

**Listing 10. Schema snippet - QName exclusion using ##definedSibling**

```xml
&lt;xsd:complexType name="personType"&gt; &lt;xsd:sequence&gt; &lt;xsd:element ref="tns:name"/&gt; &lt;xsd:element ref="tns:dateOfBirth"/&gt; &lt;xsd:element ref="tns:address"/&gt; &lt;xsd:element ref="tns:occupation"/&gt; &lt;xsd:any processContents="lax" notQName="##definedSibling" minOccurs="0" maxOccurs="unbounded"/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;
```

You can use the keyword ##definedSibling as a value in the notQName attribute to indicate that the wildcard does not match any element name that is already explicitly declared in the containing complex type. This includes those elements inherited (through extension) from the base type.

Note that ##definedSibling does not apply to attribute wildcards (&lt;anyAttribute&gt;), because XML does not allow same named attributes to appear on one element.
Exclusion of known globals

If future versions of a schema are expected to introduce new concepts (hence new elements or attributes) in the current target namespace, then it is important to have wildcards or open contents in complex types that allow the new names. At the same time, the wildcards should not allow concepts that are already known to the current version of the schema. Otherwise, they are already included in the complex type definitions.

Take the personType in Listing 10 above. If there is a global element declaration for person, because of the wildcard, the following xml snippet (Listing 11) is valid with respect to personType:

Listing 11. XML snippet - A person element

```xml
<person>  
  <name>...</name>  
  <dateOfBirth>...</dateOfBirth>  
  <address>...</address>  
  <occupation>...</occupation>  
  <person>...</person>  
</person>
```

To avoid this, XML Schema 1.1 provides another special keyword for use in the notQName attribute, ##defined indicates that this wildcard does not match any name for which there is a global declaration. You can update the wildcard in the personType complex type as follows (Listing 12):

Listing 12. Schema snippet - personType definition

```xml
<xs:complexType name="personType">  
  <xs:sequence> ...  
    <xs:any processContents="lax" notQName="#definedSibling #defined" minOccurs="0" maxOccurs="unbounded"/>  
  </xs:sequence>  
</xs:complexType>
```

Now it will not match either the explicitly declared elements in personType, or any globally declared elements. As a result, the instance where a person element appears in another person element is disallowed.

Provided that a global element is not declared for telephone, the updated personType allows a person element as in Listing 13.

Listing 13. Schema snippet - A person element definition using known globals exclusion

```xml
<person>  
  <name>...</name>  
  <dateOfBirth>...</dateOfBirth>  
  <address>...</address>  
  <occupation>...</occupation>  
  <telephone>...</telephone>  
</person>
```

In the next version of the schema, if a telephone element is added, then this instance becomes invalid. This is working by design, to signal that personType in the new schema really should have been updated to include telephone, if it is expected to appear in person.

Open contents

In XML Schema 1.0, the sequence of sub-elements allowed by a complex type is completely determined by its content model—element declarations and wildcards organized in &lt;sequence&gt;, &lt;choice&gt;, and &lt;all&gt; model groups. XML Schema 1.1 extended this further by providing a mechanism to accept sub-elements other than those explicitly defined
in the content model. This mechanism is commonly referred to as an open content. To understand open contents, let us consider the XML snippet from Listing 14, which is an illustration of a sample single CD entry from a CD catalog.

**Listing 14. XML snippet - CD entry from a CD catalog**

```xml
<cd id="0001">  
  <artist>Foo Faa</artist>  
  <album>Blah Blah</album>  
  <genre>Alternative</genre>  
  <price>11.99</price>  
  <currency>USD</currency>  
  <release_date>01-01-2009</release_date>  
  <song>    
    <track>XML XML</track>  
    <duration>1.45</duration>  
  </song> </cd>
```

Now look at a schema snippet (Listing 15) that describes the `cd` element in a flexible manner, and allows a schema author to augment the content of the `cd` element without the need to change the schema.

**Listing 15. Schema snippet - CD entry definition**

```xml
<xsd:complexType name="CatalogEntry">  
  <xsd:sequence>    
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>    
    <xsd:element name="artist" type="xs:string"/>    
    <xsd:element name="album" type="xs:string"/>    
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>    
    <xsd:element name="price" type="xs:decimal"/>    
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>    
    <xsd:element name="release_date" type="xs:dateTime"/>    
  </xsd:sequence>  
  <xsd:attribute name="id" type="xs:string"/> </xsd:complexType>  
<xsd:element name="cd" type="tns:CatalogEntry"/>
```

As you can see from the schema in Listing 15, optional elements appearing in the xml snippet (Listing 14) (namely `genre`, `currency`, and `song`) are specified in the schema through the many element wildcard definitions, `<xsd:any>`, scattered through the complex type definition, `CatalogEntry`. This can make the schema hard to read and results in extra work, sometimes duplication by requiring the schema author to insert wildcard declarations through the schema.

Open content addresses this issue by providing default wildcards, which extend the content model to accept elements anywhere or only at the end of the content model. Open contents can be specified at the level of the schema or the complex type. Note that the open content wildcard is even weaker than the explicitly specified wildcards. That is, if an element in the sub-element sequence can match either an explicit wildcard or the open content wildcard, the explicit wildcard takes precedence.

**Open content in complex type definitions**

To specify open content on a complex type, include an `<xsd:openContent>` child element in the complex type definition or in the `<xsd:restriction>` and `<xsd:extension>` children of the complex type definition. The `<xsd:openContent>` element can contain optional `id` and `mode` attributes.

The value of the `mode` attribute determines how the content model is extended. The value `interleave` indicates that elements matching the open content wildcard can be accepted anywhere in the sub-element sequence, whereas the value `suffix` indicates that elements can be accepted only at the end of the sequence. The `mode` attribute can also take a value `none`, which we will discuss in more detail in the next subsection.
The child of the `<xs:openContent>` element is an element wildcard.

In **Listing 16**, we illustrate how to define the `cd` element using the new open content feature in XML Schema 1.1. It shows how you can replace element wildcards from the schema snippet in **Listing 15** with an open content.

**Listing 16. Schema snippet - CD entry using open content**

```xml
<xs:complexType name="CatalogEntry"> 
  <xs:openContent mode="interleave"> 
    <xs:any namespace="##any" processContents="skip"/>
  </xs:openContent> 
  <xs:sequence> 
    <xs:element name="artist" type="xs:string"/>
    <xs:element name="album" type="xs:string"/>
    <xs:element name="price" type="xs:decimal"/>
    <xs:element name="release_date" type="xs:dateTime"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string"/>
</xs:complexType>
<xs:element name="cd" type="tns:CatalogEntry"/>
```

In **Listing 16**, the complex type definition contains a sequence of four child elements that are explicitly defined. In addition, the `<xs:openContent>` element allows elements from any namespace to appear anywhere within these child elements.

**Open content at the schema document level**

Schema authors often need to add the same kind of wildcard to a large number of complex types to allow future extension. This begs for the ability to specify a default open content that is applied to all the complex types. This reduces the effort to write and maintain the schema, as well as ensures that no complex type is accidentally left inextensible.

To specify default open content, include an `<xs:defaultOpenContent>` child element under the `<xs:schema>` element. Like the `<xs:openContent>` element, the `<xs:defaultOpenContent>` element contains an element wildcard and similar optional `id` and `mode` attributes, where `mode` takes either `interleave` or `suffix` as its value.

In addition, the default open content element can contain an optional `appliesToEmpty` attribute. When the value of the `appliesToEmpty` attribute is `true`, the default open content is applied to all complex types in the current schema document. The value `false` indicates that the default open content does not apply if a complex type might otherwise have an empty content model.

Another way to override the default behavior is to specify `none` as the value of `mode` on a complex type's `<xs:openContent>` element. A value of `none` indicates that this complex type does not make use of the default open content.

In **Listing 17**, we modify the schema snippet from **Listing 16** to use a default open content instead of an open content at the complex type level.

**Listing 17. Schema snippet - CD entry using default open content**

```xml
<xs:schema ...>
  <xs:defaultOpenContent mode="interleave"> 
    <xs:any namespace="##any" processContents="skip"/>
  </xs:defaultOpenContent> 
  <xs:complexType name="CatalogEntry"> 
    <xs:sequence> 
      <xs:element name="artist" type="xs:string"/>
      <xs:element name="album" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
      <xs:element name="release_date" type="xs:dateTime"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
  <xs:element name="cd" type="tns:CatalogEntry"/>
</xs:schema>
```
The content model of the complex type definition, `CatalogEntry`, contains a sequence of four explicitly defined child elements as well as an open content courtesy of the `&lt;xs:defaultOpenContent&gt;` element defined at the schema level.

**Default schema-document wide attributes**

In XML Schema 1.0, schema authors have the ability to define a common set of attributes for a given complex type by using `&lt;xs:attributeGroup&gt;`. **Listing 18** shows an example of an attribute group that defines two commonly used attributes: `width` and `height`.

**Listing 18. Schema snippet - Common attributes defined using an attribute group**

```
&lt;xs:attributeGroup name="dimensionGroup"> 
  &lt;xs:attribute name="width" type="xs:int"/&gt; 
  &lt;xs:attribute name="height" type="xs:int"/&gt; 
&lt;/xs:attributeGroup&gt; 
&lt;xs:complexType name="dimensionType"> 
  &lt;xs:attributeGroup ref="tns:dimensionGroup"/&gt; 
&lt;/xs:complexType&gt; 
```

If the set of attributes happened to be common to many complex type definitions, there was no easy way to indicate that fact in XML Schema 1.0, other than to include the attribute group reference in all complex type definitions. **Listing 19** illustrates how, in XML Schema 1.0, many complex type definitions can define the same set of attributes by referring to the same attribute group.

**Listing 19. Schema snippet - Common attributes defined in multiple complex type definitions**

```
&lt;xs:attributeGroup name="dimensionGroup"> 
  &lt;xs:attribute name="width" type="xs:int"/&gt; 
  &lt;xs:attribute name="height" type="xs:int"/&gt; 
&lt;/xs:attributeGroup&gt; 
&lt;xs:complexType name="dimensionType"> 
  &lt;xs:attributeGroup ref="tns:dimensionGroup"/&gt; 
&lt;/xs:complexType&gt; 
&lt;xs:complexType name="sofa"> 
  &lt;xs:attributeGroup ref="tns:dimensionGroup"/&gt; 
&lt;/xs:complexType&gt; 
```

XML Schema 1.1 has introduced the notion of default attribute groups. On the `&lt;xs:schema&gt;` element, you can designate an attribute group definition as the default (using the `defaultAttributes` attribute). This attribute group definition will automatically be included in each complex type defined in the schema document. In **Listing 20** below, both `dimensionType` and `sofa` will include the attributes defined in the attribute group `dimensionGroup`. There is no need to explicitly reference the attribute group in either complex type definition.

**Listing 20. Schema snippet - Common attributes defined using default attributes**

```
&lt;xs:schema .... defaultAttributes="tns:dimensionGroup"/&gt; 
&lt;xs:attributeGroup name="dimensionGroup"> 
  &lt;xs:attribute name="width" type="xs:int"/&gt; 
  &lt;xs:attribute name="height" type="xs:int"/&gt; 
&lt;/xs:attributeGroup&gt; 
&lt;xs:complexType name="dimensionType"> 
  &lt;xs:attributeGroup ref="tns:dimensionGroup"/&gt; 
&lt;/xs:complexType&gt; 
&lt;xs:complexType name="sofa"> 
  &lt;xs:attributeGroup ref="tns:dimensionGroup"/&gt; 
&lt;/xs:complexType&gt; 
```

If a complex type definition wants to override the default behavior (that is, you do not want to include the attribute group), you can set the `defaultAttributesApply` attribute on the
&lt;xs:complexType&gt; element to false. In Listing 21, the &lt;xs:complexType&gt; named person overrides the default behavior of default attributes (by indicating that you do not want to include the list of default attributes).

**Listing 21. Schema snippet - Overriding the behavior of default attributes**

```xml
&lt;xs:schema ... defaultAttributes="tns:dimensionGroup"/&gt; &lt;xs:attributeGroup name="dimensionGroup"/&gt; &lt;xs:attribute name="width" type="xs:int"/&gt; &lt;xs:attribute name="height" type="xs:int"/&gt; &lt;/xs:attributeGroup&gt; &lt;xs:complexType name="dimensionType"&gt; ... &lt;/xs:complexType&gt; &lt;xs:complexType name="person" defaultAttributesApply="false"/&gt; ... &lt;/xs:complexType&gt; ... &lt;/xs:schema&gt;
```

Default attribute groups make it easier to specify attributes which every complex type in a schema should accept (for example, `xml:id` and `xml:lang`, or an attribute wildcard).

**Conclusion**

**Other articles in this series**

- XML Schema 1.1, Part 1: An overview of the key improvements over XML Schema 1.0 and an in-depth look at datatypes
- XML Schema 1.1, Part 2: Co-occurrence constraints using XPath 2.0

In this article, we discussed some of the versioning features in XML Schema 1.1, highlighting the changes to wildcard support and the addition of open content to allow XML Schema authors to write schemas that can be compatible with future versions. In Part 4 of the series, we will explore more versioning features such as conditional inclusion and component override.
Related topics

- XML 1.0 specification (Fourth Edition): Read about XML and how it enables generic SGML to be served, received, and processed on the Web.
- XML Schema Part 1: Structures Second Edition: Learn more about the W3C XML Schema language and how it describes the structure and constrains the contents of XML 1.0 documents, including those which exploit the XML Namespace facility. This specification depends on XML Schema Part 2: Datatypes.
- W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures: Check out the latest specification of the W3C XML Schema language.
- W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes: Find more information on the new datatypes added to the W3C XML Schema language.
- Guide to Versioning XML Languages using XML Schema 1.1: Explore a guide for schema authors seeking help for versioning their schemas.
- W3C Workshop on XML Schema 1.0 User Experiences: Dig into a workshop where participants shared their user experience with XML Schema 1.0.
- The XML Parser for Java (Xerces2-J): Try this parser distributed by Apache.
- IBM certification: Find out how you can become an IBM-Certified Developer.
- XML technical library: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- IBM product evaluation versions: Get your hands on application development tools and middleware products.

© Copyright IBM Corporation 2009

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)