The basics of using XML Schema to define elements
Get started using XML Schema instead of DTDs for defining the structure of XML documents

Ashvin Radiya, President and CTO
Vibha Dixit, CEO
AvantSoft, Inc.
August 2000

Contents:

Using namespaces
Defining elements
Expressing constraints
Going beyond the basics
Resources
About the authors

The new XML Schema system, now nearing acceptance as a W3C recommendation, aims to provide a rich grammatical structure for XML documents that overcomes the limitations of the DTD (see the sidebar, Limitations of DTDs). This article demonstrates the flexibility of schemas and shows how to define the most fundamental building block of XML documents -- the element -- in the XML Schema system.

XML Schema is more powerful than DTD. To illustrate the power of the XML Schema mechanism, the first three listings briefly compare the different ways of representing elements. Listing 1 shows an excerpt of an XML document. Listing 2 shows these two elements declared in DTD syntax, and Listing 3 consists of the corresponding XML Schema syntax. Note that the syntax in Listing 3 is the same as XML syntax. Through the schema, a validating parser can verify that the element `InvoiceNo` is a positive integer and the element `ProductID` consists of one letter between A and Z followed by six digits. By contrast, a validating parser referring to the DTD can only verify that these elements are represented as strings.

Listing 1: An XML document fragment

```
<InvoiceNo>123456789</InvoiceNo>

<ProductID>J123456</ProductID>
```

Listing 2: DTD fragment describing elements in Listing 1

```
<!ELEMENT InvoiceNo (#PCDATA)>

<!ELEMENT ProductID (#PCDATA)>
```

Listing 3: XML Schema fragment describing elements in Listing 1

```
<element name='InvoiceNo' type='positive-integer'/>

<element name='ProductID' type='ProductCode'/>

<simpleType name='ProductCode' base='string'>
  <pattern value='[A-Z]{1}d{6}'/>
</simpleType>
```

Using namespaces in XML Schema

In the collaborative world, one person may be processing documents from many other parties and the different parties may want to represent their data elements differently. Moreover, in a single document, they may need to separately refer to elements with the same name that are created by different parties. How can you distinguish between such different definitions with the same name? XML Schema allows the concept of namespaces to distinguish the definitions.

A given XML Schema defines a set of new names such as the names of elements, types, attributes, attribute groups, whose definitions and declarations are written in the schema. Listing 3 defines the names as `InvoiceNo`, `ProductID`, and `ProductCode`.

The names defined in a schema are said to belong to its *target namespace*. A namespace itself has a fixed but arbitrary name that

must follow the URL syntax. For example, you can set the name of the namespace for the schema excerpted in Listing 3 to be: `http://www.SampleStore.com/Account`.

The syntax of namespace names can be confusing. Even though the namespace name starts with `http://`, it does not refer to a file at that URL that contains the schema definition. In fact, the URL `http://www.SampleStore.com/Account` does not refer to any file at all, only to an assigned name.

Definitions and declarations in a schema can refer to names that may belong to other namespaces. In this article, we refer to those namespaces as *source namespaces*. Each schema has one target namespace and possibly many source namespaces. In fact, every name in a given schema belongs to some namespace. The names for the namespaces can be fairly long, but they can be abbreviated with the syntax of `xmlns` declaration in the XML Schema document. We can add more to the example schema as shown in Listing 4 to illustrate these concepts.

Limitations of DTDs
Although DTDs have served SGML and HTML developers well for 20 years as a mechanism of describing structured information, DTDs have severe restrictions compared to XML Schema.

DTDs call for elements to consist of one of three things:

- A text string
- A text string with other child elements mixed together
- A set of child elements

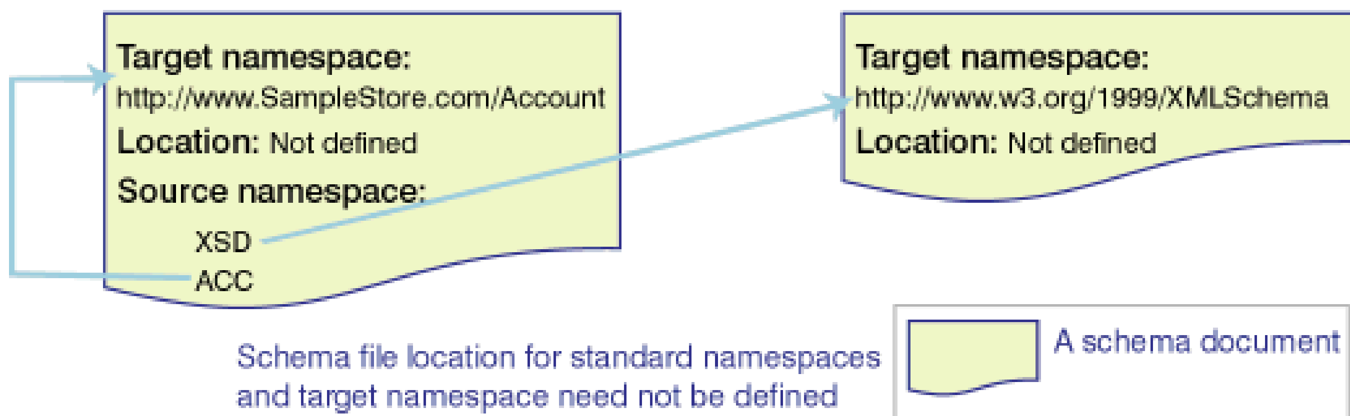
DTD does not have XML syntax and offers only limited support for types or namespaces.

Listing 4: Target and source namespaces

```
<!--XML Schema fragment in file schema1.xsd-->



<xsd:schema targetNamespace='http://www.SampleStore.com/Account'
      xmlns:xsd='http://www.w3.org/1999/XMLSchema'
      xmlns:ACC= 'http://www.SampleStore.com/Account'>

<xsd:element name='InvoiceNo' type='xsd:positive-integer'/>

<xsd:element name='ProductID' type='ACC:ProductCode'/>

<xsd:simpleType name='ProductCode' base='xsd:string'>
  <xsd:pattern value='[A-Z]{1}d{6}'/>
</xsd:simpleType>
```

In the XML Schema in Listing 4, the `targetNamespace` name is `http://www.SampleStore.com/Account`, which contains the names `InvoiceNo`, `ProductID`, and `ProductCode`. The names `schema`, `element`, `simpleType`, `pattern`, `string`, and `positive-integer` belong to source namespace `http://www.w3.org/1999/XMLSchema`, which is abbreviated as `xsd` through the `xmlns` declaration. There is nothing special about the alias name `xsd`; we could have chosen any name. For convenience and simplicity in the rest of this article, we use `xsd` to refer to namespace `http://www.w3.org/1999/XMLSchema` and we omit the qualification `xsd` in some code snippets. In this example, the `targetNamespace` also happens to be one of the source namespaces because the name `ProductCode` is used in defining other names.

Figure 1: Namespaces for Listing 4



The schema fragment in Listing 4 does not need to specify locations of source schema files. For the overall "schema of schemas," `http://www.w3.org/1999/XMLSchema`, you need not specify a location because it is well known. For the source namespace `http://www.SampleStore.com/Account`, you do not need to specify a location since it also happens to be the name of the target namespace that is being defined in this file. To understand better how to specify the schema location and use the default namespace, consider the extension to the example in Listing 5.

Listing 5: Multiple source namespaces, importing a namespace

```
<!--XML Schema fragment in file schema1.xsd-->

<schema targetNamespace='http://www.SampleStore.com/Account'
      xmlns='http://www.w3.org/1999/XMLSchema'
      xmlns:ACC= 'http://www.SampleStore.com/Account'
      xmlns:PART= 'http://www.PartnerStore.com/PartsCatalog'>

<import namespace='http://www.PartnerStore.com/PartsCatalog'
            schemaLocation='http://www.ProductStandards.org/repository/alpha.xsd'/>

<element name='InvoiceNo' type='positive-integer'/>

<element name='ProductID' type='ACC:ProductCode'/>

<simpleType name='ProductCode' base='string'>
  <pattern value='[A-Z]{1}d{6}'/>
</simpleType>

<element name='stickyGlue' type='PART:SuperGlueType'/>
```

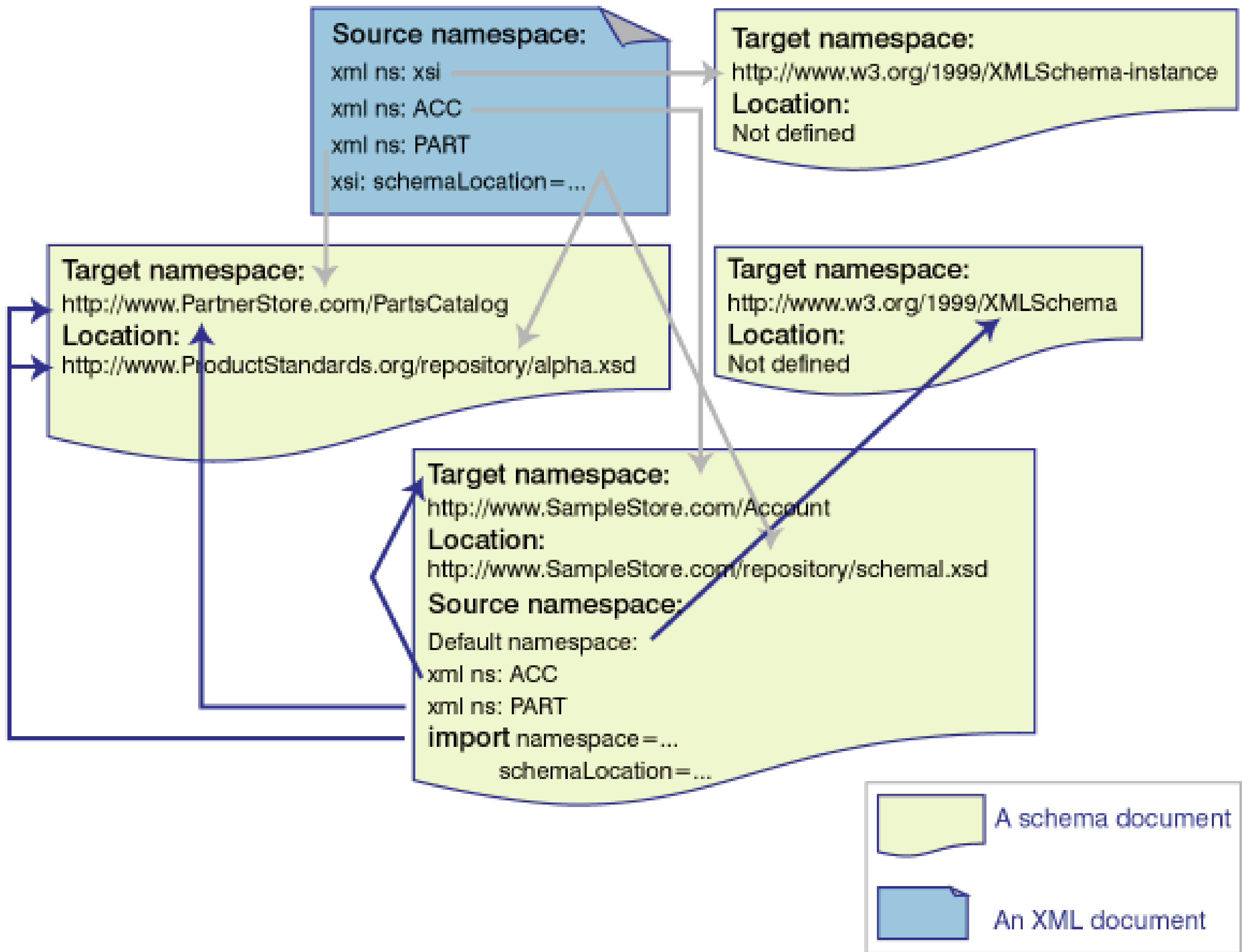
Listing 5 includes one more namespace reference: `http://www.PartnerStore.com/PartsCatalog`. This namespace is different from `targetNamespace` and standard namespaces. As a result, it must be imported using the `import` declaration element whose `schemaLocation` attribute specifies the location of the file that contains the schema. The default namespace is `http://www.w3.org/1999/XMLSchema`, whose `xmlns` declaration does not have a name. Every unqualified name such as `schema` and `element` belongs to default namespace `http://www.w3.org/1999/XMLSchema`. If your schema refers to several names from one namespace, it is more convenient to designate that as the default namespace.

An XML instance document may refer to names of elements from multiple namespaces that are defined in multiple schemas. To refer to and abbreviate the name of a namespace, again use `xmlns` declarations. We use the `schemaLocation` attribute from the XML Schema instance namespace to specify the file locations. Note that this attribute differs from the same named attribute `schemaLocation` of `xsd` namespace in the previous examples.

Listing 6: Using multiple namespace names from multiple schemas

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

<ACC:rootElement xmlns:ACC='http://www.SampleStore.com/Account'
      xmlns:PART='http://www.PartnerStore.com/PartsCatalog'
      xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
      xsi:schemaLocation='http://www.PartnerStore.com/PartsCatalog
                          http://www.ProductStandards.org/repository/alpha.xsd
                          http://www.SampleStore.com/Account
                          http://www.SampleStore.com/repository/schema1.xsd'>

<ACC:InvoiceNo>123456789</ACC:InvoiceNo>
```

Figure 2: Namespaces for Listings 5 and 6



Defining elements
To define an element is to define its name and content model. In XML Schema, the content model of an element is defined by its type. Then, the instance elements in an XML document can have only values that fit the types defined in its schema.

A type can be simple or complex. A simple type cannot contain elements or attributes in its value. A complex type can create the effect of embedding elements in other elements or it can associate attributes with an element. (The examples in this article to this point have been user-defined simple types (see `ProductCode`)). The XML Schema spec also includes predefined simple types (see the sidebar Simple types).
A *derived simple type* constrains the values of its base type. For example, the values of derived simple type `ProductCode` is a subset of the values of base type `string`.

Simple types
XML Schema specification defines a number of simple types for values, as shown in Table 2: Predefined simple types of values.

Simple, non-nested elements have a simple type
An element that does not contain attributes or other elements can be defined to be of a simple type, predefined or user-defined, such as `string`, `integer`, `decimal`, `time`, `ProductCode`, etc.

Listing 7: Some simple types for elements

```
<element name='age' type='integer'/>

<element name='price' type='decimal'/>
```

Elements with attributes must have a complex type
Now, try adding the attribute `currency` to the simple element `price` from Listing 7. You can't. An element of a simple type cannot

have an attribute. If you want to add an attribute, you must define `price` as a complex type. In the example in Listing 8, we have defined what is called an *anonymous type*, where no explicit name is given to the complex type. In other words, the `name` attribute of the `complexType` element is not defined.

Listing 8: A complex element type

```
<element name='price'>

  <complexType base='decimal' derivedBy='extension'>

    <attribute name='currency' type='string'/>

  </complexType>
</element>

<!-- In XML instance document, we can write: <price currency='US'>45.50</price> -->
```

Elements that embed other elements must have a complex type

In an XML document, an element may embed other elements. This requirement is expressed directly in the DTD. XML Schema instead defines an element, which has a type, and that type can have declarations of other elements and attributes. See Table 1 for a simple example.

Table 1: A comparison of complex data types in DTD and XML Schema

XML document

```
<Book>
   <Title>Cool XML<Title>
   <Author>Cool Guy</Author>
</Book>
```

DTD

```
<!ELEMENT Book (Title, Author)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
```

XML Schema

```
<element name='Book' type='BookType'/>
<complexType name='BookType'>
    <element name='Title' type='string'/>
    <element name='Author' type='string'/>
</complexType>
```

Although the XML code in Table 1 conforms to both DTD and XML Schema fragments, there is a big difference between them. In a DTD, all elements are global, whereas the XML Schema in the table allows `Title` and `Author` to be defined locally -- to occur only within the element `Book`. To exactly duplicate the effect of the DTD declarations in XML Schema, the elements `Title` and `Author` must have a global scope, as in Listing 9. The `ref` attribute of element `element` allows you to refer to previously declared elements.

Listing 9: A complex type defined with global simple types

```
<element name='Title' type='string'/>

<element name='Author' type='string'/>

<element name='Book' type='BookType'/>

<complexType name='BookType'>

   <element ref='Title'/>

   <element ref='Author'/>

</complexType>
```

In the examples in Table 1 and Listing 9, `BookType` is global and can be used to declare other elements. By contrast, Listing 10 makes the type local to the definition of element `Book` and also makes it anonymous. Note that the XML document fragment in Table 1 matches all three schema fragments in Table 1, Listing 9, and Listing 10.

Listing 10: Hiding `BookType` as a local type

```
<element name='Title' type='string'/>

<element name='Author' type='string'/>

<element name='Book'>

   <complexType>

      <element ref='Title'/>

      <element ref='Author'/>

   </complexType>

</element>
```

Expressing sophisticated constraints on elements
XML Schema offers greater flexibility than DTD for expressing constraints on the content model of elements. At the simplest level, as in DTD, you can associate attributes with an element declaration and indicate that a sequence of one only (1), zero or more (*), or one or more (+) elements from a given set of elements can occur in it. You can express additional constraints in XML Schema using, for example, `minOccurs` and `maxOccurs` attributes of element `element` and using `choice`, `group`, and `all` elements.

Listing 11: Expressing constraints on element types

```
<element name='Title' type='string'/>

<element name='Author' type='string'/>

<element name='Book'>

  <complexType>

    <element ref='Title' minOccurs='0'/>

    <element ref='Author' maxOccurs='2'/>

  </complexType>

</element>
```

In Listing 11, the occurrence of `Title` is optional in `Book` (similar to the DTD '?'). However, Listing 11 also says that there must be at least one, but no more than two, authors in the element `Book`. The default value of `minOccurs` and `maxOccurs` is 1 for `element`. The element `choice` allows only one of its children to appear in an instance. Another element, `all`, expresses the constraint that all child elements in the group may appear once or not at all, and they may appear in any order. Listing 12 expresses the constraint that both `Title` and `Author` must occur in `Book` in any order, or neither will. Such constraints are difficult to express in a DTD.

Listing 12: Indicating that all types must be defined for an element

```
<xsd:element name='Title' type='string'/>

<xsd:element name='Author' type='string'/>

<xsd:element name='Book'>

  <xsd:complexType>

    <xsd:all>

      <xsd:element ref='Tile'/>

      <xsd:element ref='Author'/>

    </xsd:all>

  </xsd:complexType>

</xsd:element>
```

Going beyond the basics

We have covered the most fundamental concepts needed to define elements in XML Schema, giving you a flavor of its power through simple examples. Many more powerful mechanisms are available:

- XML Schema includes extensive support for type inheritance, enabling the reuse of previously defined structures. Using what are called *facets*, you can derive new types that represent a smaller subset of values of some other types, for example, to define a subset by enumeration, range, or pattern matching. In the example for this article, `ProductCode` type was defined using `pattern` facet. A subtype can also add more element and attribute declarations to the base type.
- Several mechanisms can control whether a subtype can be defined at all or whether a subtype can be substituted in a specific document. For example, it is possible to express that `InvoiceType` (type of Invoice number) cannot be subtyped, that is, no one can define a new version of `InvoiceType`. You can also express that, in a particular context, no subtype of `ProductCode` type can be substituted.
- Besides subtyping, it is possible to define equivalence types such that the value of one type can be replaced by another type.
- By declaring an element or type to be abstract, XML Schema provides a mechanism to force substitution for it.
- For convenience, groups of attributes and elements can be defined and named. That makes reuse possible by subsequently referring to the groups.
- XML Schema provides three elements -- `appInfo`, `documentation`, and `annotation` -- for annotating schemas for both human readers (`documentation`) and applications (`appInfo`).
- You can express uniqueness constraints based on certain attributes of child elements.

Explore XML Schema further through the documentation on the W3C site (see Resources) and by watching the dW XML zone for more coverage. Now that the XML Schema specification has been approved to move forward to Candidate for Recommendation, no doubt more and more of you will begin to use it.

Resources

- Free Web-form access to XSV, an XML Schema Validator from University of Edinburgh/W3C (alpha).
- Free downloadable XML Schema validator from the Apache project.
- Evaluation version of commercial XML Schema aware editor/validator from Extensibility.
- Free downloadable XML Schema validator, based on the February 2000 version of XML Schema, from Oracle.
- Free downloadable tools from http://www.ibm.com/alphaworks.
- A detailed backgrounder on the XML Schema specification to date: *XML Schema Part 0: Primer.*
- The current W3C specification documents for the two parts of XML Schema: *XML Schema Part 1: Structures* and *XML Schema Part 2: Datatypes*.

About the authors

Ashvin Radiya is the founder and president of AvantSoft, Inc. As the CTO, he leads the development and delivery of AvantSoft training courses on state-of-the-art Java programming and related technologies. He also builds and manages strategic

partnerships with Fortune 100 companies. Ashvin has extensive industry, academic, and professional experience. He worked at IBM, Austin, on advanced CORBA-based distributed object-oriented products. He has extensive knowledge and expertise in the areas of mobile commerce, XML, Java, JavaBeans, Enterprise JavaBeans components, InfoBus, Security, CORBA, and distributed object-oriented programming. Ashvin received his Ph.D. in Computer Science from Syracuse University. You can contact Ashvin Radiya at ashvin@avantsoft.com.

Vibha Dixit plays a key role of Business Development Manager and Technologist at AvantSoft, Inc. She is responsible for business planning, managing strategic partnerships, acquiring new customers, sales and marketing. She also actively participates in defining AvantSoft's technology goals and directions in the areas of mobile commerce, XML, and Java technology. Vibha has a unique experience in business management as well as strong industry experience in computer technology. Before joining AvantSoft, she worked at the IBM Santa Teresa lab on distributed transactional object middleware. At Ohio Supercomputer Center, she was involved in the design and development of an operating system for multicomputers. Vibha received her Ph.D. in Computer Science from Ohio State University. She has also completed an executive MBA course from Southern Methodist University. You can contact Vibha Dixit at vibha@avantsoft.com.

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Privacy | Legal | Contact