XML Schema Part 0: Primer

W3C Working Draft, 22 September 2000

This version:
http://www.w3.org/TR/2000/WD-xmlschema-0-20000922/
Latest version:
http://www.w3.org/TR/xmlschema-0/
Previous version:
http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/
Editor:
David C. Fallside (IBM) fallside@us.ibm.com

Copyright ©2000 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

XML Schema Part 0: Primer is a non-normative document intended to provide an easily readable description of the XML Schema facilities and is oriented towards quickly understanding how to create schemas using the XML Schema language. XML Schema Part 1: Structures and XML Schema Part 2: Datatypes provide the complete normative description of the XML Schema definition language, and the primer describes the language features through numerous examples which are complemented by extensive references to the normative texts.

Status of this Document

The XML Schema Part 0: Primer is a part of the W3C XML Activity.

This is a public working draft of XML Schema 1.0 for review by the public and by members of the World Wide Web Consortium. The XML Schema Working Group has agreed to its publication. Note that some sections of this draft may not be up-to-date with the XML Schema language described in Parts 1 and 2 of the XML Schema specification. Known discrepancies are noted in the text.

The Working Group does not anticipate further substantial changes to the syntax described here, although this is still a working draft, and is subject to change based on experience and on comment by the public, and other W3C working groups.

A list of current W3C working drafts can be found at http://www.w3.org/TR/. They may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress".

Table of contents

1 Introduction

2 Basic Concepts: The Purchase Order
2.1 The Purchase Order Schema
2.2 Complex Type Definitions, Element & Attribute Declarations
2.3 Simple Types
2.3.1 List Types
2.3.2 Union Types
2.4 Anonymous Type Definitions
2.5 Element Content
2.5.1 Complex Types from Simple Types
2.5.2 Mixed Content
2.5.3 Empty Content
2.6 Annotations
2.7 Building Content Models
2.8 Attribute Groups
2.9 Null Values

3. Advanced Concepts I: Namespaces, Schemas & Qualification
3.1 Target Namespaces & Unqualified Locals
3.2 Qualified Locals
3.3 Global vs. Local Declarations
3.4 Undeclared Target Namespaces

4. Advanced Concepts II: The International Purchase Order
4.1 A Schema in Multiple Documents
4.2 Deriving Types by Extension
4.3 Using Derived Types in Instance Documents
4.4 Deriving Complex Types by Restriction
4.5 Redefining Types and Groups
4.6 Substitution Groups
4.7 Abstract Elements and Types
4.8 Preventing the Creation and Use of Derived Types

5. Advanced Concepts III: The Quarterly Report
5.1 Specifying Uniqueness
5.2 Defining Keys and their References
5.3 XML Schema Constraints vs. XML 1.0 ID Attributes
5.4 Importing Types
5.4.1 Type Libraries
5.5 Any Element, Any Attribute
5.6 schemaLocation
5.7 Conformance

Appendices

A. Acknowledgements
B. Simple Types & Their Facets
C. Regular Expressions
D. Index
E. Document History

1 Introduction

This document, XML Schema Part 0: Primer, provides an easily approachable description of the XML Schema definition language, and should be used alongside the formal descriptions of the language contained in Parts 1 and 2 of the XML Schema specification. The intended audience of this document includes application developers whose programs read and write schema documents, and schema authors who need to know about the features of the language, especially features that provide functionality above and beyond what is provided by DTDs. The text assumes that you have a basic understanding of XML 1.0 and XML-Namespaces. Each major section of the primer introduces new features of the language, and describes those features in the context of concrete examples.

Section 2 covers the basic mechanisms of XML Schema. It describes how to declare the elements and attributes that appear in XML documents, the distinctions between simple and complex types, defining complex types, the use of simple types for element and attribute values, schema annotation, a simple mechanism for re-using element and attribute definitions, and null values.

Section 3, the first advanced section in the primer, explains the basics of how namespaces are used in XML and schema documents. This section is important for understanding many of the topics that appear in the other advanced sections.

Section 4, the second advanced section in the primer, describes mechanisms for deriving types from existing types, and for controlling these derivations. The section also describes mechanisms for merging together fragments of a schema from multiple sources, and for element substitution.

Section 5 covers more advanced features, including a mechanism for specifying uniqueness among attributes and elements, a mechanism for using types across namespaces, a mechanism for extending types based on namespaces, and a description of how documents are checked for conformance.

In addition to the sections just described, the primer contains a number of appendices that provide detailed reference information on simple types and a regular expression language.

The primer is a non-normative document, which means that it does not provide a definitive (from the W3C's point of view) specification of the XML Schema language. The examples and other explanatory material in this document are provided to help you understand XML Schema, but they may not always provide definitive answers. In such cases, you will need to refer to the XML Schema specification, and to help you do this, we provide many links pointing to the relevant parts of the specification. More specifically, XML Schema items mentioned in the primer text are linked to an index of element names and attributes, and a summary table of datatypes, both in the primer. The table and the index contain links to the relevant sections of XML Schema parts 1 and 2.

2 Basic Concepts: The Purchase Order

The purpose of a schema is to define a class of XML documents, and so the term "instance document" is often used to describe an XML document that conforms to a particular schema. In fact, neither instances nor schemas need to exist as documents *per se* -- they may exist as streams of bytes sent between applications, as fields in a database record, or as collections of XML Infoset "Information Items" -- but to simplify the primer, we have chosen to always refer to instances and schemas as if they are files.

Let us start by considering an instance document in a file called `po.xml`. It describes a purchase order generated by a home products ordering and billing application:

The Purchase Order, po.xml

```
<?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>
    <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>
```

The purchase order consists of a main element, `purchaseOrder`, and the subelements `shipTo`, `billTo`, and `items`. These subelements in turn contain other subelements, and so on, until a subelement such as `USPrice` contains a number rather than any subelements. Elements that contain subelements or carry attributes are said to have complex types, whereas elements that contain numbers (and strings, and dates, etc) but do not contain any subelements are said to have simple types. Some elements have attributes; attributes always have simple types.

The complex types in the instance document, and some of the simple types, are defined in the schema for purchase orders. The other simple types are defined as part of XML Schema's repertoire of built-in simple types.

Before going on to examine the purchase order schema, we digress briefly to mention the association between the instance document and the purchase order schema. As you can see by inspecting the instance document, the purchase order schema is not mentioned. An instance is not actually required to reference a schema, and although many will, we have chosen to keep this first section simple, and to assume that any processor of the instance document can obtain the purchase order schema without any information from the instance document. In later sections, we will introduce explicit mechanisms for associating instances and schemas.

2.1 The Purchase Order Schema

The purchase order schema is contained in the file `po.xsd`:

The Purchase Order Schema, po.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
```

```
<xsd:annotation>
 <xsd:documentation>
  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"/>

<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>

<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"
    use="fixed" value="US"/>
</xsd:complexType>

<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"/>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="SKU">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="\d{3}-[A-Z]{2}"/>
 </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

The purchase order schema consists of a `schema` element and a variety of subelements, most notably `element`,

complexType, and simpleType which determine the appearance of elements and their content in instance documents.

Each of the elements in the schema has a prefixxsd: which is associated with the XML Schema namespace through the declaration, xmlns:xsd="http://www.w3.org/2000/10/XMLSchema", that appears in the schema element. The prefixxsd: is used by convention to denote the XML Schema namespace, although any prefix can be used. The same prefix, and hence the same association, also appears on the names of built-in simple types, e.g. xsd:string. The purpose of the association is to identify the elements and simple types as belonging to the vocabulary of the XML Schema language rather than the vocabulary of the schema author. For the sake of clarity in the text, we just mention the names of elements and simple types (e.g.simpleType), and omit the prefix.

2.2 Complex Type Definitions, Element & Attribute Declarations

In XML Schema, there is a basic difference between complex types which allow elements in their content and may carry attributes, and simple types which cannot have element content and cannot carry attributes. There is also a major distinction between definitions which create new types (both simple and complex), and declarations which enable elements and attributes with specific names and types (both simple and complex) to appear in document instances. In this section, we focus on defining complex types and declaring the elements and attributes that appear within them.

New complex types are defined using thecomplexType element and such definitions typically contain a set of element declarations, element references, and attribute declarations. The declarations are not themselves types, but rather an association between a name and constraints which govern the appearance of that name in documents governed by the associated schema. Elements are declared using theelement element, and attributes are declared using the attribute element. For example,USAddress is defined as a complex type, and within the definition ofUSAddress we see five element declarations and one attribute declaration:

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"
      use="fixed" value="US"/>
</xsd:complexType>
```

The consequence of this definition is that any element appearing in an instance whose type is declared to be USAddress (e.g. shipTo in po.xml) must consist of five elements and one attribute. These elements must be called name, street, city, state and zip as specified by the values of the declarations'name attributes, and the elements must appear in the same sequence (order) in which they are declared. The first four of these elements will each contain a string, and the fifth will contain a decimal number. The element whose type is declared to be USAddress may appear with an attribute calledcountry which must contain the stringUS.

The USAddress definition contains only declarations involving simple typesstring, decimal and NMTOKEN. In contrast, the PurchaseOrderType definition contains element declarations involving complex types, e.g. USAddress, although note that both declarations use the same type attribute to identify the type, regardless of whether the type is simple or complex.

Defining PurchaseOrderType

```
<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>
```

In defining `PurchaseOrderType`, two of the element declarations, for `shipTo` and `billTo`, associate different element names with the same complex type, namely `USAddress`. The consequence of this definition is that any element appearing in an instance (e.g. `po.xml`) whose type is declared to be `PurchaseOrderType` must consist of elements named `shipTo` and `billTo`, each containing the five subelements (`name`, `street`, `city`, `state` and `zip`) that were declared as part of `USAddress`. The `shipTo` and `billTo` elements may also carry the `country` attribute that was declared as part of `USAddress`.

The `PurchaseOrderType` definition contains an `orderDate` attribute declaration which, like the `country` attribute declaration, identifies a simple type. In fact, all attribute declarations must reference simple types because, unlike element declarations, attributes cannot contain other elements or other attributes.

The element declarations we have described so far have each associated a name with an existing type definition. Sometimes it is preferable to use an existing element rather than declare a new element, for example:

```
<xsd:element ref="comment" minOccurs="0"/>
```

This declaration references an existing element, `comment`, that was declared elsewhere in the purchase order schema. In general, the value of the `ref` attribute must reference a global element, i.e. one that has been declared under `schema` rather than as part of a complex type definition. The consequence of this declaration is that an element called `comment` may appear in an instance document, and its content must be consistent with that element's type, in this case, `string`.

Both elements and attributes may be declared globally. `comment` is one example of a global element which we reference from an element declaration contained in the `PurchaseOrderType` definition. We could similarly declare attributes under `schema`, and reference them using the `ref` attribute from attribute declarations contained in type definitions. Note that global declarations cannot contain references, global declarations must identify simple and complex types directly.

The `comment` element is optional within `PurchaseOrderType` because the value of the `minOccurs` attribute in its declaration is 0. In general, an element is required to appear when the value of `minOccurs` is 1 or more. The maximum number of times an element may appear is determined by the value of a `maxOccurs` attribute in its declaration. This may be a positive integer value such as 41, or the term `unbounded` to indicate there is no maximum number of occurrences. The default value for both the `minOccurs` and the `maxOccurs` attributes is 1. Thus, when an element such as `comment` is declared without a `maxOccurs` attribute, the element may not occur more than once. Be sure that if you specify a value for only the `minOccurs` attribute, it is less than or equal to the default value of `maxOccurs`, i.e. it is 0 or 1. Similarly, if you specify a value for only the `maxOccurs` attribute, it must be greater than or equal to the default value of `minOccurs`, i.e. 1 or more. If both attributes are omitted, the element must appear exactly once.

Attributes may appear once or not at all (the default), and so the syntax for specifying occurrences of attributes is different than the syntax for elements. In particular, a `use` attribute is used in an attribute declaration to indicate whether the attribute is `required` or `optional`, and if `optional` whether the attribute's value is `fixed`

or whether there is a `default`. A second attribute, `value`, provides any value that is called for. To illustrate, `po.xsd` contains a declaration for the `country` attribute, which is declared with `use` and `value` values of `fixed` and `US` respectively. This declaration means that the appearance of a `country` attribute is optional, although its value must be `US` if it does appear, and if it does not appear, a schema processor will create a `country` attribute with this value.

The values of the attributes used in element and attribute declarations to constrain the occurrences of elements and attributes are summarised in Table 1.

Table 1. Occurrence Constraints for Elements and Attributes

Elements (minOccurs, maxOccurs) fixed, default	Attributes use, value	Notes
(1, 1) -, -	required, -	element/attribute must appear once, it may have any value
(1, 1) 37, -	required, 37	element/attribute must appear once, its value must be 37
(2, unbounded) 37, -	n/a	element must appear twice or more, its value must be 37; in general, minOccurs and maxOccurs' values may be positive integers, and maxOccurs' value may also be "unbounded"
(0, 1) -, -	optional, -	element/attribute may appear once, it may have any value
(0, 1) 37, -	fixed, 37	element/attribute may appear once, if it does appear its value must be 37, if it does not appear its value is 37
(0, 1) -, 37	default, 37	element/attribute may appear once; if it does not appear its value is 37, otherwise its value is that given
(0, 2) -, 37	n/a	element may appear once, twice, or not at all; if it does not appear its value is 37, otherwise its value is that given; in general, minOccurs and maxOccurs' values may be positive integers, and maxOccurs' value may also be "unbounded"
(0, 0) -, -	prohibited, -	element/attribute must not appear
Note that neither minOccurs, maxOccurs, nor use may appear in the declarations of global elements and attributes.		

So far, we have described how to define new complex types (e.g. `PurchaseOrderType`), and declare elements (e.g. `purchaseOrder`) and attributes (e.g. `orderDate`). These activities generally involve naming, and the question naturally arises: What happens if two things are given the same name? The answer depends upon the two things in question, although in general the more similar are the two things, the more likely there will be a conflict.

Here are some examples to illustrate when same names cause problems. If the two things are both types, say I define a complex type called USStates and a simple type called USStates, there is a conflict. If the two things are a type and an element or attribute, say I define a complex type called USAddress and I declare an element called USAddress, there is no conflict. If the two things are elements within different types (i.e. not global elements), say I declare one element called name as part of the USAddress type and a second element called name as part o the Item type, there is no conflict. (Such elements are sometimes called local element declarations). Finally, if the two things are both types and you define one and XML Schema has defined the other, say you define a simple type called decimal, there is no conflict. The reason for the apparent contradiction in the last example is that the two types belong to different namespaces. We'll explore the use of schema and namespaces in a later section.

2.3 Simple Types

The purchase order schema declares several elements and attributes that have simple types. Some of these simple types, such as `string` and `decimal`, are built-in to XML Schema, while others are derived from the built-in's. For example, the `partNum` attribute has a type called `SKU` (Stock Keeping Unit) that is derived from `string`. Both built-in simple types and their derivations can be used in all element and attribute declarationsTable 2 lists all the simple types built-in to XML Schema, along with an example of each type.

Table 2. Simple Types Built-In to XML Schema

Simple Type	Examples (delimited by commas)	Notes
string	Confirm this is electric	
byte	-1, 126	see (3)
unsignedByte	0, 126	see (3)
binary	62696E617279	see (1)
integer	-126789, -1, 0, 1, 126789	see (3)
positiveInteger	1, 126789	see (3)
negativeInteger	-126789, -1	see (3)
nonNegativeInteger	0, 1, 126789	see (3)
nonPositiveInteger	-126789, -1, 0	see (3)
int	-1, 126789675	see (3)
unsignedInt	0, 1267896754	see (3)
long	-1, 12678967543233	see (3)
unsignedLong	0, 12678967543233	see (3)
short	-1, 12678	see (3)
unsignedShort	0, 12678	see (3)
decimal	-1.23, 0, 123.4, 1000.00	see (3)
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to single-precision 32-bit floating point, NaN is "not a number", see (3)
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to double-precision 64-bit floating point, see (3)
boolean	true, false	
time	13:20:00.000, 13:20:00.000-05:00	see (3)
timeInstant	1999-05-31T13:20:00.000-05:00	May 31st 1999 at 1.20pm Eastern Standard Time which is 5 hours behind Co-Ordinated Universal Time, see (3)
timePeriod	1999-05-31T13:20	see (3)
timeDuration	P1Y2M3DT10H30M12.3S	1 year, 2 months, 3 days, 10 hours, 30 minutes, 12.3 seconds
date	1999-05-31	see (3)
month	1999-05	May 1999, see (3)
year	1999	1999, see (3)
century	19	the 1900's, see (3)

recurringDay	----31	every 31st day, see (3)
recurringDate	--05-31	every May 31st, see (3)
recurringDuration	--05-31T13:20:00	May 31st every year at 1.20pm Co-Ordinated Universal Time, format similar to timeInstant, see (1) & (3)
Name	shipTo	XML 1.0 Name type
QName	po:USAddress	XML Namespace QName
NCName	USAddress	XML Namespace NCName, i.e. a QName without the prefix and colon
uriReference	http://www.example.com/, http://www.example.com/doc.html#ID5	
language	en-GB, en-US, fr	valid values for xml:lang as defined in XML 1.0
ID		XML 1.0 ID attribute type, see (2)
IDREF		XML 1.0 IDREF attribute type, see (2)
IDREFS		XML 1.0 IDREFS attribute type, see (2)
ENTITY		XML 1.0 ENTITY attribute type, see (2)
ENTITIES		XML 1.0 ENTITIES attribute type, see (2)
NOTATION		XML 1.0 NOTATION attribute type, see (2)
NMTOKEN	US, Brésil	XML 1.0 NMTOKEN attribute type, see (2)
NMTOKENS	US UK, Brésil Canada Mexique	XML 1.0 NMTOKENS attribute type, i.e. a whitespace separated list of NMTOKEN's, see (2)
Notes: (1) Authors must apply facets to the simple types binary and recurringDuration types in order to use them. (2) To retain compatibility between XML Schema and XML 1.0 DTDs, the simple types ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS should only be used in attributes. (3) A value of this type can be represented by more than one lexical format, e.g. 100 and 1.0E2 are both valid float formats representing "one hundred". However, rules have been established for this type that define a canonical lexical format, see XML Schema Part 2.		

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particula we can derive a new simple type by restricting an existing simple type, in other words, the legal range of values for the new type are a subset of the existing type's range of values. We use the `simpleType` element to define and name the new simple type. We use the `restriction` element to indicate the existing (base) type, and to identify the "facets" that constrain the range of values. A complete list of facets is provided in Appendix B.

Suppose we wish to create a new type of integer called `myInteger` whose range of values is between 10000 and 99999 (inclusive). We base our definition on the built-in simple type `integer`, whose range of values also

includes integers less than 10000 and greater than 99999. To define `myInteger`, we restrict the range of the `integer` base type by employing two facets, `minInclusive` and `maxInclusive`:

Defining myInteger, Range 10000-99999

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

The example shows one particular combination of a base type and two facets used to define `myInteger`, but a look at the list of built-in simple types and their facets (Appendix B) should suggest other viable combinations.

The purchase order schema contains another, more elaborate, example of a simple type definition. A new simple type called `SKU` is derived (by restriction) from the simple type `string`. Furthermore, we constrain the values of `SKU` using a facet called `pattern` in conjunction with the regular expression "`\d{3}-[A-Z]{2}`" that is read "three digits followed by a hyphen followed by two upper-case ASCII letters":

Defining the Simple Type "SKU"

```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

This regular expression language is described more fully in Appendix C.

XML Schema defines fourteen facets which are listed in Appendix B. Among these, the `enumeration` facet is one the most useful and it can be used to constrain the values of almost every simple type, except the `boolean` type. The `enumeration` facet limits a simple type to a set of distinct values. For example, we can use the `enumeration` facet to define a new simple type called `USState`, derived from `string`, whose value must be one of the standard US state abbreviations:

Using the 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>
```

`USState` would be a good replacement for the `string` type currently used in the `state` element declaration. By making this replacement, the legal values of a `state` element, i.e. the `state` subelements of `billTo` and `shipTo`, would be limited to one of `AK`, `AL`, `AR`, etc. Note that the enumeration values specified for a particular type must be unique.

2.3.1 List Types

XML Schema has the concept of a list type, in addition to the so-called atomic types that constitute most of the types listed in Table 2. The value of an atomic type is indivisible from XML Schema's perspective; For example, the `NMTOKEN` value `US` is indivisible in the sense that no part of `US`, such as the character "S", has any meaning by itself. In contrast, list types are comprised of sequences of atomic types and consequently the parts of a sequence (the "atoms") themselves are meaningful. For example, `NMTOKENS` is a list type, and an element of this type would be a white-space delimited list of `NMTOKEN`'s, such as "US UK FR". XML Schema has three built-in list types, they are `NMTOKENS`, `IDREFS`, and `ENTITIES`.

In addition to using the built-in list types, you can create new list types by derivation from existing atomic types. (You cannot create list types from existing list types, nor from complex types). For example, to create a list of `myInteger`'s:

Creating a List of myInteger's

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

And an element in an instance document whose content conforms to `listOfMyIntType` is:

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

Several facets can be applied to list types: `length`, `minLength`, `maxLength`, and `enumeration`. For example, to define a list of exactly six US states (`SixUSStates`), we first define a new list type called `USStateList` from `USState`, and then we derive `SixUSStates` by restricting `USStateList` to only six items:

List Type for Six US States

```
<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>
```

Elements whose type is `SixUSStates` must have six items, and each of the six items must be one of the (atomic) values of the enumerated type `USState`, for example:

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

Note that it is possible to derive a list type from the atomic type `string`. However, a `string` may contain white space, and white space delimits the items in a list type, so you should be careful using fixed length list types whose base type is `string`. For example, suppose a list type is defined with a `length` facet equal to 3, and base type `string`, then the following 3 item list is legal:

```
Asie Europe Afrique
```

But the following 3 "item" list is illegal:

```
Asie Europe Amérique Latine
```

Even though "Amérique Latine" may exist as a single string outside of the list, when it is included in the list, the whitespace between Amérique and Latine effectively creates a fourth item, and so the latter example will not conform to the 3-item list type.

2.3.2 Union Types

Atomic types and list types enable an element or an attribute value to be one or more instances of one atomic type. In contrast, a union type enables an element or attribute value to be one or more instances of one type drawn from the union of multiple atomic and list types. To illustrate, we create a union type for representing American states as singleton letter abbreviations or lists of numeric codes. The `zipList` union type is built from one atomic type and one list type:

Union Type for Zipcodes

```
<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>
```

When we define a union type, the `memberTypes` attribute value is a list of all the types in the union.

Now, assuming we have declared an element called `zips` of type `zipUnion`, valid instances of the element are:

```
<zips>CA</zips>
<zips>95630 95977 95945</zips>
<zips>AK</zips>
```

Two facets, `pattern` and `enumeration`, can be applied to a union type.

2.4 Anonymous Type Definitions

Schemas can be constructed by defining sets of named types such as `PurchaseOrderType` and then declaring elements such as `purchaseOrder` that reference the types using the `type`= construction. This style of schema construction is straightforward but it can be unwieldy, especially if you define many types that are referenced only once and contain very few constraints. In these cases, a type can be more succinctly defined as an anonymous type which saves the overhead of having to be named and explicitly referenced.

The definition of the type `Items` in `po.xsd` contains two element declarations that use anonymous types (`item` and `quantity`). In general, you can identify anonymous types by the lack of a `type`= in an element (or attribute) declaration, and by the presence of an un-named (simple or complex) type definition:

Two Anonymous Type Definitions

```
<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"/>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

In the case of the `item` element, it has an anonymous complex type consisting of the elements `productName`, `quantity`, `USPrice`, `comment`, and `shipDate`, and an attribute called `partNum`. In the case of the `quantity` element, it has an anonymous simple type derived from `integer` whose value ranges between 1 and 99.

2.5 Element Content

The purchase order schema has many examples of elements containing other elements (e.g. `items`), elements having attributes and containing other elements (e.g. `shipTo`), and elements containing only a simple type of value (e.g. `USPrice`). However, we have not seen an element having attributes but containing only a simple type of value, nor have we seen an element that contains other elements mixed with character content, nor have we seen an element that has no content at all. In this section we'll examine these variations in the content models of elements.

2.5.1 Complex Types from Simple Types

Let us first consider how to declare an element that has an attribute and contains a simple value. In an instance document, such an element might appear as:

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

The purchase order schema declares a `USPrice` element that is a starting point:

```
<xsd:element name="USPrice" type="decimal"/>
```

Now, how do we add an attribute to this element? As we have said before, simple types cannot have attributes and `decimal` is a simple type. Therefore, we must define a complex type to carry the attribute declaration. We also want the content to be simple type `decimal`. So our original question becomes: How do we define a complex type that is based on the simple type `decimal`? The answer is to *derive* a new complex type from the simple type `decimal`:

Deriving a Complex Type from a Simple Type

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

We use the `complexType` element to start the definition of a new (anonymous) type. To indicate that the content model of the new type contains only character data and no elements, we use a `simpleContent` element. Finally, we derive the new type by extending the simple `decimal` type. The extension consists of adding a `currency` attribute using a standard attribute declaration. (We cover type derivation in detail in Section 4). The `internationalPrice` element declared in this way will appear in an instance as shown in the example above.

2.5.2 Mixed Content

The construction of the purchase order schema may be characterized as elements containing subelements, and the deepest subelements contain character data. XML Schema also provides for the construction of schemas where character data can appear alongside subelements, and character data is not confined to the deepest subelements.

To illustrate, consider the following snippet from a customer letter that uses some of the same elements as the purchase order:

Snippet of Customer Letter

```
<letterBody>
<salutation>Dear Mr.<name>Robert Smith</name>.</salutation>
Your order of <quantity>1</quantity> <productName>Baby
Monitor</productName> shipped from our warehouse on
<shipDate>1999-05-21</shipDate>. ....
</letterBody>
```

Notice the text appearing between elements and their child elements. Specifically, text appears between th elements `salutation`, `quantity`, `productName` and `shipDate` which are all children of `letterBody`, and text appears around the element name which is the child of a child of `letterBody`. The following snippet of a schema declares `letterBody`:

Snippet of Schema for Customer Letter

```
<xsd:element name="letterBody">
 <xsd:complexType mixed="true">
  <xsd:sequence>
   <xsd:element name="salutation">
    <xsd:complexType mixed="true">
     <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="quantity" type="xsd:positiveInteger"/>
   <xsd:element name="productName" type="xsd:string"/>
   <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
   <!-- etc -->
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

The elements appearing in the customer letter are declared, and their types are defined using the `element` and `complexType` element constructions we have seen before. To enable character data to appear between the child-elements of `letterBody`, the `mixed` attribute on the type definition is set to true.

Note that the `mixed` model in XML Schema differs fundamentally from the `mixed` model in XML 1.0. Under the XML Schema mixed model, the order and number of child elements appearing in an instance must agree with the order and number of child elements specified in the model. In contrast, under the XML 1.0 mixed model, the order and number of child elements appearing in an instance cannot be constrained. In sum, XML Schema provides full schema validation of mixed models in contrast to the partial schema validation provided by XML 1.0.

2.5.3 Empty Content

Now suppose that we want the `internationalPrice` element to convey both the unit of currency and the price as attribute values rather than as separate attribute and content values. For example:

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

Such an element has no content at all, and we say that its content model is empty. To define a type whose content is empty, we essentially define a type that allows only elements in its content, but we do not actually declare any elements and so the type's content model is empty:

An Empty Complex Type

```
<xsd:element name="internationalPrice">
 <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>
```

In this example, we define an (anonymous) type having `complexContent`, i.e. only elements. The `complexContent` element signals that we intend to restrict or extend the content model of a complex type; The

`restriction` of `anyType` declares two attributes but does not introduce any element content (see Section 4.4 for more details on restriction). The `internationalPrice` element declared in this way will appear in an instance as shown in the example above.

The `anyType` represents an abstraction called the `ur-type` which is the base type from which all simple and complex types are derived. An `anyType` type does not constrain its content in any way. It is possible to use `anyType` like other types, for example:

```
<xsd:element name="anything" type="xsd:anyType" />
```

The content of the element declared in this way is unconstrained, so the element value may be 423.46, but it may be any other sequence of characters as well. In general, it is probably better to avoid unconstrained types in favour of constrained types such as `decimal`, `string`, etc.

2.6 Annotations

XML Schema provides three elements for annotating schemas for the benefit of both human readers and applications. In the purchase order schema, we put a basic schema description and copyright information inside the `documentation` element, which is the recommended location for human readable material.

The `appInfo` element, which we did not use in the purchase order schema, can be used to provide information for tools, stylesheets and other applications. An interesting example using `appInfo` is one of the schemas that describes some of the simple types in XML Schema Part 2: Datatypes. Information describing this schema, e.g. which facets are applicable to particular simple types, is represented inside `appInfo` elements, which was used by an application to automatically generate text for the XML Schema Part 2 document.

Both `documentation` and `appInfo` appear as subelements of `annotation`, which may itself appear at the beginning of most schema constructions. To illustrate, the following example shows `annotation` elements appearing at the beginning of an element declaration and a complex type definition:

Annotations in Element Declaration & Complex Type Definition

```
<xsd:element name="internationalPrice">
 <xsd:annotation>
  <xsd:documentation>element declared with anonymous type</xsd:documentation>
 </xsd:annotation>
 <xsd:complexType>
  <xsd:annotation>
   <xsd:documentation>empty anonymous type with 2 attributes</xsd:documentation>
  </xsd:annotation>
  <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>
```

The `annotation` element may also appear at the beginning of other schema constructions such as those indicated by the elements `schema`, `simpleType`, and `attribute`.

2.7 Building Content Models

The definitions of complex types in the purchase order schema all declare sequences of elements that must

appear in the instance document. The occurrence of individual elements declared in the so-called content models of these types may be optional, as indicated by a 0 value for the attribute `minOccurs` (e.g. in `comment`), or otherwise constrained depending upon the values of `minOccurs` and `maxOccurs`. XML Schema also provides constraints that apply to groups of elements appearing in a content model. These constraints mirror those available in XML 1.0 plus some additional constraints. Note that the constraints do not apply to attributes.

XML Schema enables a group of elements to be defined and named, so that the elements can be used to build up the content models of complex types (thus mimicking common usage of parameter entities in XML 1.0). Un-named groups of elements can also be defined, and along with elements in named groups, they can be constrained to appear in the same order (sequence) as they are declared. Alternatively, they can be constrained so that only one of the elements may appear in an instance.

To illustrate, we modify the `PurchaseOrderType` definition from the purchase order schema using two groups so that purchase orders may contain either separate shipping and billing addresses, or a single address for those cases in which the shippee and billee are co-located:

Nested Choice and Sequence Groups

```
<xsd:complexType name="PurchaseOrderType">
 <xsd:sequence>
  <xsd:choice>
   <xsd:group   ref="shipAndBill" />
   <xsd:element name="singleUSAddress" type="USAddress" />
  </xsd:choice>
  <xsd:element   ref="comment"    minOccurs="0"/>
  <xsd:element   name="items"     type="Items" />
 </xsd:sequence>
 <xsd:attribute name="orderDate" type="xsd:date" />
</xsd:complexType>

<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress" />
    <xsd:element name="billTo" type="USAddress" />
  </xsd:sequence>
</xsd:group>
```

The `choice` group element allows only one of its children to appear in an instance. One child is an inner `group` element that references the named group `shipAndBill` consisting of the element sequence `shipTo`, `billTo`, and the second child is a `singleUSAddress`. Hence, in an instance document, the `purchaseOrder` element must contain either a `shipTo` element followed by a `billTo` element or a `singleUSAddress` element. The `choice` group is followed by the `comment` and `items` element declarations, and both the `choice` group and the element declarations are children of a `sequence` group. The effect of these various groups is that the address element(s) must be followed by `comment` and `items` elements in that order.

There exists a third option for constraining elements in a group: All the elements in the group may appear once or not at all, and they may appear in any order. The `all` group (which provides a simplified version of the SGML &-Connector) is limited to the top-level of any content model. Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once, i.e. the permissible values of `minOccurs` and `maxOccurs` are 0 and 1. For example, to allow the child elements of `purchaseOrder` to appear in any order, we could redefine `PurchaseOrderType` as:

An 'All' Group

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <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:all>
  <xsd:attribute name="orderDate" type="xsd:date" />
</xsd:complexType>
```

By this definition, `acomment` element may optionally appear within `purchaseOrder`, and it may appear before or after any `shipTo`, `billTo` and `items` elements, but it can appear only once. Moreover, the stipulations of an `all` group do not allow us to declare an element such as `comment` outside the group as a means of enabling it to appear more than once. XML Schema stipulates that an `all` group must appear as the sole child at the top of a content model. In other words, the following is illegal:

Illegal Example with an 'All' Group

```
<xsd:complexType name="PurchaseOrderType">
 <xsd:sequence>
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element name="items"  type="Items" />
  </xsd:all>
  <xsd:sequence>
   <xsd:element ref="comment" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:sequence>
 <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

Finally, named and un-named groups that appear in content models (represented by `group` and `choice`, `sequence`, `all` respectively) may carry `minOccurs` and `maxOccurs` attributes. By combining and nesting the various groups provided by XML Schema, and by setting the values of `minOccurs` and `maxOccurs`, it is possible to represent any content model expressible with an XML 1.0 DTD. Furthermore, the `all` group provides additional expressive power.

2.8 Attribute Groups

Suppose we want to provide more information about each item in a purchase order, for example, each item's weight and preferred shipping method. We can accomplish this by adding `weightKg` and `shipBy` attribute declarations to the `item` element's (anonymous) type definition:

Adding Attributes to the Inline Type Definition

```
<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"/>
    <!-- add weightKg and shipBy attributes -->
   <xsd:attribute name="weightKg"   type="xsd:decimal"/>
   <xsd:attribute name="shipBy">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="air"/>
      <xsd:enumeration value="land"/>
      <xsd:enumeration value="any"/>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

Alternatively, we can create a named attribute group containing all the desired attributes of an `item` element, and reference this group by name in the `item` element declaration:

Adding Attributes Using an Attribute Group

```
<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>

  <!-- attributeGroup replaces individual declarations -->
  <xsd:attributeGroup ref="ItemDelivery"/>
 </xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="ItemDelivery">
  <xsd:attribute name="partNum" type="SKU"/>
  <xsd:attribute name="weightKg"  type="xsd:decimal"/>
  <xsd:attribute name="shipBy">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="air"/>
      <xsd:enumeration value="land"/>
      <xsd:enumeration value="any"/>
     </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:attributeGroup>
```

Using an attribute group in this way can improve the readability of schema, and facilitates updating schema because an attribute group can be defined and edited in one place and referenced in multiple definitions and declarations. These characteristics of attribute groups make them similar to parameter entities in XML 1.0. Note that an attribute group may contain other attribute groups. Note also that both attribute declarations and attribute group references must appear at the end of complex type definitions.

2.9 Null Values

One of the purchase order items listed in `po.xml`, the `Lawnmower`, does not have a `shipDate` element. Within the context of our scenario, the schema author may have intended such absences to indicate `items` not yet shipped. But in general, the absence of an element does not have any particular meaning: It may indicate that the information is unknown, or not applicable, or the element may be absent for some other reason. Sometimes it is desirable to represent an unshipped `item`, unknown information, or inapplicable information *explicitly* with an element, rather than by an absent element. For example, it may be desirable to represent a "null" value being sent to or from a relational database with an element that is present. Such cases can be represented using XML Schema's null mechanism which enables an element to appear with or without a non-null value.

XML Schema's null mechanism involves an "out of band" null signal. In other words, there is no actual null value that appears as element content, instead there is an attribute to indicate that the element content is null. To illustrate, we can modify the `shipDate` element declaration so that nulls can be signalled:

```
<xsd:element name="shipDate" type="xsd:date" nullable="true"/>
```

And to explictly represent that `shipDate` has a null value in the instance document, we set the null attribute (from the XML Schema namespace for instances) to true:

```
<shipDate xsi:null="true"></shipDate>
```

The `null` attribute is defined as part of the XML Schema namespace for instances (`http://www.w3.org/2000/10/XMLSchema-instance`), and so it must appear in the instance document with a prefix (`xsi:`) associated with that namespace. (As with the `xsd:` prefix, the `xsi:` prefix is used by convention only). Note that the null mechanism applies only to element values, and not to attribute values. An element with `xsi:null="true"` may not have any element content but it may still carry attributes.

3. Advanced Concepts I: Namespaces, Schemas & Qualification

A schema can be viewed as a collection (vocabulary) of type definitions and element declarations whose names belong to a particular namespace called a target namespace. The target namespace enables us to distinguish between definitions and declarations from different vocabularies. For example, target namespaces would enable us to distinguish between the declaration for `element` in the XML Schema language vocabulary, and a declaration for `element` in a hypothetical chemistry language vocabulary. The former is part of the `http://www.w3.org/2000/10/XMLSchema` target namespace, and the latter is part of another target namespace.

When we want to check that an instance document conforms to one or more schemas (through a process called schema validation), we need to identify which element and attribute declarations and type definitions in the schemas should be used to check which elements and attributes in the instance document. The target namespace plays an important role in the identification process. We examine the role of the target namespace in the next section.

The schema author also has several options that affect how the identities of elements and attributes are represented in instance documents. More specifically, the author can decide whether or not the appearance of locally declared elements and attributes in an instance must be qualified by a namespace, using either an explicit prefix or implicitly by default. The schema author's choice regarding qualification of local elements and attributes has a number of implications regarding the structures of schemas and instance documents, and we examine some of these implications in the following sections.

3.1 Target Namespaces & Unqualified Locals

In a new version of the purchase order schema (`po1.xsd`), we explicitly declare a target namespace, and specify that both locally defined elements and locally defined attributes must be unqualified. The target namespace in `po1.xsd` is `http://www.example.com/PO1`, as indicated by the value of the `targetNamespace` attribute.

Qualification of local elements and attributes can be globally specified by a pair of attributes, `elementFormDefault` and `attributeFormDefault`, on the `schema` element, or can be specified separately for each local declaration using the `form` attribute. All such attributes' values may each be set to `unqualified` or `qualified`, to indicate whether or not locally declared elements and attributes must be unqualified.

In `po1.xsd` we globally specify the qualification of elements and attributes by setting the values of both `elementFormDefault` and `attributeFormDefault` to `unqualified`. Strictly speaking, these settings are unnecessary because the values are the defaults for the two attributes; We make them to highlight the contrast between this case and other cases to be described later.

Purchase Order Schema with Target Namespace, po1.xsd

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">

 <element name="purchaseOrder" type="po:PurchaseOrderType"/>
 <element name="comment"       type="string"/>

 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="shipTo" type="po:USAddress"/>
   <element name="billTo" type="po:USAddress"/>
   <element ref="po:comment" minOccurs="0"/>
   <!-- etc -->
  </sequence>
 </complexType>

 <complexType name="USAddress">
  <sequence>
   <element name="name"   type="string"/>
   <element name="street" type="string"/>
   <!-- etc -->
  </sequence>
 </complexType>

 <!-- etc -->

</schema>
```

To see how the target namespace of this schema is populated, we'll examine in turn each of the type definitions and element declarations. Starting from the end of the schema, we first define a type called `USAddress` that consists of the elements `name`, `street`, etc. One consequence of this type definition is that the `USAddress` type is included in the schema's target namespace. We next define a type called `PurchaseOrderType` that consists of the elements `shipTo`, `billTo`, `comment`, etc. `PurchaseOrderType` is also included in the schema's target namespace. Notice that the type references in the three element declarations are prefixed, i.e. `po:USAddress`, `po:USAddress` and `po:comment`, and the prefix is associated with the namespace `http://www.example.com/PO1`. This is the same namespace as the schema's target namespace, and so a processor of this schema will know to look within this schema for the definition of the type `USAddress` and the declaration of the element `comment`. It is also possible to refer to types in another schema with a different target namespace, hence enabling re-use of definitions and declarations between schemas.

At the beginning of the schema `po1.xsd`, we declare the elements `purchaseOrder` and `comment`. They are included in the schema's target namespace. The `purchaseOrder` element's type is prefixed, for the same reason that `USAddress` is prefixed. In contrast, the `comment` element's type, `string`, is not prefixed. The `po1.xsd` schema contains a default namespace declaration and so unprefixed types such as `string`, and unprefixed elements such as `element` and `complexType`, are associated with the default namespace, `http://www.w3.org/2000/10/XMLSchema`. In fact, this is the target namespace of XML Schema itself, and so a processor of `po1.xsd` will know to look within the schema of XML Schema (otherwise known as the "schema for schemas") for the definition of the type `string` and the declaration of the element called `element`.

Let us now examine how the target namespace of the schema affects a conforming instance document:

A Purchase Order with Unqualified Locals, po1.xml

```
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
                   orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <!-- etc -->
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <!-- etc -->
    </billTo>
    <apo:comment>Hurry, my lawn is going wild!</apo:comment>
    <!-- etc -->
</apo:purchaseOrder>
```

The instance document declares one namespace, `http://www.example.com/PO1`, and associates it with the prefix `apo:`. This prefix is used to qualify two elements in the document, namely `purchaseOrder` and `comment`. The namespace is the same as the target namespace of the schema in `po1.xsd`, and so a processor of the instance document will know to look in that schema for the declarations of `purchaseOrder` and `comment`. In fact, target namespaces are so named because of the sense in which there exists a target namespace for the elements `purchaseOrder` and `comment`. Target namespaces in the schema therefore control the validation of corresponding namespaces in the instance.

The prefix `apo:` is applied to the global elements `purchaseOrder` and `comment` elements. Furthermore, `elementFormDefault` and `attributeFormDefault` require that the prefix is *not* applied to any of the the locally declared elements such as `shipTo`, `billTo`, `name` and `street`, and it is *not* applied to any of the attributes (which were all declared locally). The `purchaseOrder` and `comment` are global elements because they are declared in the context of the schema as a whole rather than within the context of a particular type. For example, the declaration of `purchaseOrder` appears as a child of the `schema` element in `po1.xsd`, whereas the declaration of `shipTo` appears as a child of the `complexType` element that defines `PurchaseOrderType`.

When local elements and attributes are not required to be qualified, an instance author may require more or less knowledge about the details of the schema to create schema valid instance documents. More specifically, if the author can be sure that only the root element (such as `purchaseOrder`) is global, then it is a simple matter to qualify only the root element. Alternatively, the author may know that all the elements are declared globally, and so all the elements in the instance document can be prefixed, perhaps taking advantage of a default namespace declaration. (We examine this approach in Section 3.3). On the other hand, if there is no uniform pattern of global and local declarations, the author will need detailed knowledge of the schema to correctly prefix global elements (and attributes).

3.2 Qualified Locals

Elements and attributes can be independently required to be qualified, although we'll start by describing qualification of local elements. To specify that all locally declared elements in a schema must be qualified, we set the value of `elementFormDefault` to `qualified`:

Modifications to po1.xsd for Qualified Locals

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

 <element name="purchaseOrder" type="po:PurchaseOrderType"/>
 <element name="comment"       type="string"/>

 <complexType name="PurchaseOrderType">
  <!-- etc -->
 </complexType>

 <!-- etc -->

</schema>
```

And in this conforming instance document, we qualify all the elements explicitly:

A Purchase Order with Explicitly Qualified Locals

```
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
                   orderDate="1999-10-20">
    <apo:shipTo country="US">
        <apo:name>Alice Smith</apo:name>
        <apo:street>123 Maple Street</apo:street>
        <!-- etc -->
    </apo:shipTo>
    <apo:billTo country="US">
        <apo:name>Robert Smith</apo:name>
        <apo:street>8 Oak Avenue</apo:street>
        <!-- etc -->
    </apo:billTo>
    <apo:comment>Hurry, my lawn is going wild!</apo:comment>
    <!-- etc -->
</apo:purchaseOrder>
```

Alternatively, we can replace the explicit qualification of every element with implicit qualification provided by a default namespace, as shown here inpo2.xml:

A Purchase Order with Default Qualified Locals, po2.xml

```
<?xml version="1.0"?>
<purchaseOrder xmlns="http://www.example.com/PO1"
                   orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <!-- etc -->
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <!-- etc -->
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <!-- etc -->
</purchaseOrder>
```

In po2.xml, all the elements in the instance belong to the same namespace, and the namespace statement declares a default namespace that applies to all the elements in the instance. Hence, it is unnecessary to explicitly prefix any of the elements. As another illustration of using qualified elements, the schemas in Section 5 all require qualified elements.

Qualification of attributes is very similar to the qualification of elements. Attributes that must be qualified, either because they are declared globally or because the `attributeFormDefault` attribute is set to `qualified`, appear prefixed in instance documents. One example of a qualified attribute is the `xsi:null` attribute that was introduced in Section 2.9. In fact, attributes that are required to be qualified must be explicitly prefixed because the XML-Namespaces specification does not provide a mechanism for defaulting the namespaces of attributes. Attributes that are not required to be qualified appear in instance documents without prefixes, which is the typical case.

The qualification mechanism we have described so far has controlled all local element and attribute declarations within a particular target namespace. It is also possible to control qualification on a declaration by declaration basis using the `form` attribute. For example, to require that the locally declared attribute `publicKey` is qualified in instances, we declare it in the following way:

Requiring Qualification of Single Attribute

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
 <!-- etc -->
 <element name="secure">
  <complexType>
   <sequence>
    <!-- element declarations -->
   </sequence>
   <attribute name="publicKey" form="qualified">
    <simpleType>
     <restriction base="binary">
      <encoding value="base64"/>
     </restriction>
    </simpleType>
   </attribute>
  </complexType>
 </element>
</schema>
```

Notice that the value of the `form` attribute overides the value of the `attributeFormDefault` attribute for the `publicKey` attribute only. Also, the `form` attribute can be applied to an element declaration in the same manner. An instance document that conforms to the schema is:

Instance with a Qualified Attribute

```
<?xml version="1.0"?>
<purchaseOrder xmlns="http://www.example.com/PO1"
                   xmlns:po="http://www.example.com/PO1"
                   orderDate="1999-10-20">
    <!-- etc -->
    <secure po:publicKey="GpM7">
        <!-- etc -->
    </secure>
</purchaseOrder>
```

3.3 Global vs. Local Declarations

Another authoring style, when all the element names are unique within a namespace, is to create a schema in which all elements are global. This is similar in effect to the use of <!ELEMENT> in a DTD. In the example below, we have modified the originalpo1.xsd such that all the elements are declared globally. Notice that we have omitted the `elementFormDefault` and `attributeFormDefault` attributes in this example to emphasise that their values are irrelevant when there are only global element and attribute declarations.

Modified version ofpo1.xsd using only global element declarations

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1">

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

 <element name="shipTo"  type="po:USAddress"/>
 <element name="billTo"  type="po:USAddress"/>
 <element name="comment" type="string"/>

 <element name="name" type="string"/>
 <element name="street" type="string"/>

 <complexType name="PurchaseOrderType">
  <sequence>
   <element ref="po:shipTo"/>
   <element ref="po:billTo"/>
   <element ref="po:comment" minOccurs="0"/>
   <!-- etc -->
  </sequence>
 </complexType>

 <complexType name="USAddress">
  <sequence>
   <element ref="po:name"/>
   <element ref="po:street"/>
   <!-- etc -->
  </sequence>
 </complexType>

 <!-- etc -->

</schema>
```

This "global" version ofpo1.xsd will validate the instance documentpo2.xml which, as we described previously, is also schema valid against the "qualified" version ofpo1.xsd. In other words, both schema approaches can validate the same, namespace defaulted, document. Thus, in one respect the two schema approaches are similar, although in another important respect the two schema approaches are very different. Specifically, when all

elements are declared globally, it is not possible to take advantage of local names. For example, you can only declare one global element called "title". However, you can locally declare one element called "title" that has a string type, and is a subelement of "book"; And within the same schema (target namespace) you can declare a second element also called "title" that is an enumeration of the values "Mr Mrs Ms".

3.4 Undeclared Target Namespaces

In Section 2 we explained the basics of XML Schema using a schema that did not declare a target namespace and an instance document that did not declare a namespace. So the question naturally arises: What is the target namespace in these examples and how is it referenced?

In the purchase order schema, `po.xsd`, we did not declare a target namespace for the schema, nor did we declare a prefix (like `po:` above) associated with the schema's target namespace with which we could refer to types and elements defined and declared within the schema. The consequence of not declaring a target namespace in a schema is that the definitions and declarations from that schema, such as `USAddress` and `purchaseOrder`, are referenced without namespace qualification. In other words there is no explicit namespace prefix applied to the references nor is there any implicit namespace applied to the reference by default. So for example, the `purchaseOrder` element is declared using the type reference `PurchaseOrderType`. In contrast, all the XML Schema elements and types used in `po.xsd` are explicitly qualified with the prefix `xsd:` that is associated with the XML Schema namespace.

Element declarations from a schema with no target namespace validate unqualified elements in the instance document. That is, they validate elements for which no namespace qualification is provided by either an explicit prefix or by default (`xmlns:`). So, to validate a traditional XML 1.0 document which does not use namespaces at all, you must provide a schema with no target namespace. Of course, there are many XML 1.0 documents that do not use namespaces, so there will be many schema documents written without target namespaces; you must be sure to give to your processor a schema document that corresponds to the vocabulary you wish to validate.

4. Advanced Concepts II: The International Purchase Order

The purchase order schema described in Chapter 2 was contained in a single document, and most of the schema constructions-- such as element declarations and type definitions-- were constructed from scratch. In reality, schema authors will want to compose schemas from constructions located in multiple documents, and create new types based on existing types. In this section, we examine mechanisms that enable such compositions and creations.

4.1 A Schema in Multiple Documents

As schemas become larger, it is often desirable to divide their content among several schema documents for purposes such as ease of maintenance, access control, and readability. For these reasons, we have taken the schema constructs concerning addresses out of `po.xsd`, and put them in a new file called `address.xsd`. The modified purchase order schema file is called `ipo.xsd`:

The International Purchase Order Schema, ipo.xsd

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO">

 <annotation>
  <documentation>
   International Purchase order schema for Example.com
   Copyright 2000 Example.com. All rights reserved.
  </documentation>
 </annotation>

 <!-- include address constructs -->
 <include
  schemaLocation="http://www.example.com/schemas/address.xsd"/>

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

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

 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="shipTo"     type="ipo:Address"/>
   <element name="billTo"     type="ipo:Address"/>
   <element ref="ipo:comment" minOccurs="0"/>
   <element name="Items"      type="ipo:Items"/>
  </sequence>
  <attribute name="orderDate" type="date"/>
 </complexType>

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

 <simpleType name="SKU">
  <restriction base="string">
   <pattern value="\d{3}-[A-Z]{2}"/>
  </restriction>
 </simpleType>

</schema>
```

The file containing the address constructs is:

Addresses for International Purchase Order schema, address.xsd

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO">

 <annotation>
  <documentation>
   Addresses for International Purchase order schema
   Copyright 2000 Example.com. All rights reserved.
  </documentation>
 </annotation>

 <complexType name="Address">
  <sequence>
   <element name="name"   type="string"/>
   <element name="street" type="string"/>
   <element name="city"   type="string"/>
  </sequence>
 </complexType>

 <complexType name="USAddress">
  <complexContent>
   <extension base="ipo:Address">
    <sequence>
     <element name="state" type="ipo:USState"/>
     <element name="zip"   type="positiveInteger"/>
    </sequence>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="UKAddress">
  <complexContent>
   <extension base="ipo:Address">
    <sequence>
     <element   name="postcode" type="ipo:UKPostcode"/>
    </sequence>
    <attribute name="exportCode" type="positiveInteger"
                   use="fixed" value="1"/>
   </extension>
  </complexContent>
 </complexType>

 <!-- other Address derivations for more countries -->

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

 <!-- simple type definition for UKPostcode -->

</schema>
```

The various purchase order and address constructions are now contained in two schema files, `ipo.xsd` and `address.xsd`. To include these constructions as part of the international purchase order schema, in other words to include them in the international purchase order's namespace, `ipo.xsd` contains the `include` element:

```
<include schemaLocation="http://www.example.com/schemas/address.xsd"/>
```

The effect of this `include` element is to bring in the definitions and declarations contained in `address.xsd`, and make them available as part of the international purchase order schema target namespace. The one important caveat to using `include` is that the target namespace of the included components must be the same as the target namespace of the including schema, in this case `http://www.example.com/IPO`. Bringing in definitions and declarations using the `include` mechanism effectively adds these components to the existing target namespace. In Section 4.5, we will describe a similar mechanism that enables you to modify certain components when they are brought in.

In our example, we have shown only one including document and one included document. In practice it is possible to include more than one document using multiple `include` elements, and documents can include documents that themselves include other documents. However, nesting is legal only if all the included parts of the schema are declared with the same target namespace.

Instance documents that conform to schema whose definitions span multiple schema documents need only reference the 'topmost' document, and the common namespace, and it is the responsibility of the processor to gather together all the definitions specified in the various included documents. In our example above, the instance document `ipo.xml` (see Section 4.3) references only the common target namespace, `http://www.example.com/IPO`, and the one schema file `http://www.example.com/schemas/ipo.xsd`. The processor is responsible for obtaining the schema file `address.xsd`.

In Section 5.4 we describe how schemas can be used to validate content from more than one namespace.

4.2 Deriving Types by Extension

To create our address constructs, we start by creating a complex type called `Address` in the usual way (see `address.xsd`). The `Address` type contains the basic elements of an address: a name, a street and a city. (Such a definition will not work for all countries, but it will serve the purposes of our example.) From this starting point we derive two new complex types that contain all the elements of the original type plus additional elements that are specific to addresses in the US and the UK. The technique we use here to derive new (complex) address types by extending an existing type is the same technique we used in in Section 2.5.1, except that our base type here is a complex type whereas our base type in the previous section was a simple type.

We define the two new complex types, `USAddress` and `UKAddress`, using the `complexType` element. In addition, we indicate that the content models of the new types are complex, i.e. contain elements, by using the `complexContent` element, and we indicate that we are extending the base type `Address` by the value of the `base` attribute on the `extension` element.

When a complex type is derived by extension, its effective content model is the content model of the base type plus the content model specified in the type derivation. Furthermore, the two content models are treated as two children of a sequential group. In the case of `UKAddress`, the content model of `UKAddress` is the content model of `Address` plus the declarations for a `postcode` element and an `exportCode` attribute. This is like defining the `UKAddress` from scratch as follows:

Example

```
<complexType name="UKAddress">
 <sequence>
  <!-- content model of Address -->
  <element name="name" type="string"/>
  <element name="street" type="string"/>
  <element name="city" type="string"/>

  <!-- appended element declaration -->
  <element name="postcode" type="ipo:UKPostcode"/>
 </sequence>

 <!-- appended attribute declaration -->
 <attribute name="exportCode" type="positiveInteger"
            use="fixed" value="1"/>
</complexType>
```

4.3 Using Derived Types in Instance Documents

In our example scenario, purchase orders are generated in response to customer orders which may involve shipping and billing addresses in different countries. The international purchase order `ipo.xml` below, illustrates one such case where goods are shipped to the UK and the bill is sent to a US address. Clearly it is very useful if the schema for international purchase orders does not have to spell out every possible combination of international addresses for billing and shipping, and even more so if we can add new complex types of international address simply by creating new derivations of `Address`.

XML Schema allows us to define the `billTo` and `shipTo` elements as `Address` types (see `ipo.xsd`) but to use instances of international addresses in place of instances of `Address`. In other words, an instance document whose content conforms to the `UKAddress` type will be valid if that content appears within the document at a location where an `Address` is expected (assuming the `UKAddress` content itself is valid). To make this feature of XML Schema work, and to identify exactly which derived type is intended, the derived type must be identified in the instance document. The type is identified using the `xsi:type` attribute which is part of the XML Schema instance namespace. In the example, `ipo.xml`, use of the `UKAddress` and `USAddress` derived types is identified through the values assigned to the `xsi:type` attributes.

An International Purchase order, ipo.xml

```
<?xml version="1.0"?>
<ipo:purchaseOrder
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
  xmlns:ipo="http://www.example.com/IPO"
  orderDate="1999-12-01">

    <shipTo exportCode="1" xsi:type="ipo:UKAddress">
        <name>Helen Zoe</name>
        <street>47 Eden Street</street>
        <city>Cambridge</city>
        <postcode>CB1 1JR</postcode>
    </shipTo>

    <billTo xsi:type="ipo:USAddress">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>

    <items>
        <item partNum="833-AA">
            <productName>Lapis necklace</productName>
            <quantity>1</quantity>
            <USPrice>99.95</USPrice>
            <ipo:comment>Want this for the holidays!</ipo:comment>
            <shipDate>1999-12-05</shipDate>
        </item>
    </items>
</ipo:purchaseOrder>
```

In Section 4.8 we'll see how to prevent derived types from being used in this sort of substitution.

4.4 Deriving Complex Types by Restriction

In addition to deriving new complex types by extending content models, it is also possible to derive new types by restricting the content models of existing types. Restriction of complex types is conceptually the same as restriction of simple types, except that the restriction of complex types involves a type's declarations rather than the acceptable range of a simple type's values. A complex type derived by restriction is very similar to its base type, except that its declarations are more limited than the corresponding declarations in the base type. In fact, the values represented by the new type are a subset of the values represented by the base type (as is also the case with restriction of simple types). In other words, an application prepared for the values of the base type would not be surprised by the values of the restricted type.

For example, suppose we want to update our definition of the list of `items` in an international purchase order so that it must contain *at least* one `item` on order; The schema shown in `ipo.xsd` allows an `items` element to appear without any child `item` elements. To create our new `ConfirmedItems` type, we define the new type in the usual way, indicate that it is derived by restriction from the base type `Items`, indicate that we are deriving the new type by restriction, and provide a new (more restrictive) value for the minimum number of `item` element occurrences. Notice that types derived by restriction must repeat all the components of the base type definition:

Deriving ConfirmedItems by Restriction from Items

```
<complexType name="ConfirmedItems">
 <complexContent>
  <restriction base="ipo:Items">
   <sequence>

    <!-- item element is different than in Items -->
    <element name="item" minOccurs="1" maxOccurs="unbounded">

     <!-- remainder of definition is same as Items -->
     <complexType>
      <sequence>
       <element name="productName" type="string"/>
       <element name="quantity">
        <simpleType>
         <restriction base="positiveInteger">
          <maxExclusive value="100"/>
         </restriction>
        </simpleType>
       </element>
       <element name="USPrice" type="decimal"/>
       <element ref="ipo:comment" minOccurs="0"/>
       <element name="shipDate" type="date" minOccurs="0"/>
      </sequence>
      <attribute name="partNum" type="ipo:SKU"/>
     </complexType>
    </element>

   </sequence>
  </restriction>
 </complexContent>
</complexType>
```

This change, requiring at least one child element rather than allowing zero or more child elements, narrows the allowable number of child elements from a minimum of 0 to a minimum of 1. Note that all `ConfirmedItems` type elements will also be acceptable as `Item` type elements.

To further illustrate restriction, Table 3 shows a number of examples of how element and attribute declarations within type definitions may be restricted (the table shows element syntax although the first three examples are equally valid attribute restrictions).

Table 3. Restriction Examples

Base	Restriction	Notes
	default="1"	setting a default value where none was previously given
	fixed="100"	setting a fixed value where none was previously given
	type="string"	specifying a type where none was previously given
(minOccurs, maxOccurs)	**(minOccurs, maxOccurs)**	
(0, 1)	(0, 0)	deletion of optional component
(0, unbounded)	(0, 0) (0, 37)	
(1, 9)	(1, 8) (2, 9) (4, 7) (3, 3)	
(1, unbounded)	(1, 12) (3, unbounded) (6, 6)	
(1, 1)	-	cannot restrict minOccurs or maxOccurs

4.5 Redefining Types and Groups

In Section 4.1 we described how to include definitions and declarations obtained from external schema files having the same target namespace. The `include` mechanism enables you to use externally created schema components "as-is", that is, without any modification. We have just described how to derive new types by extension and by restriction, and the `redefine` mechanism we will now describe enables you to redefine simple and complex types, groups, and attribute groups that are obtained from external schema files. Like the `include` mechanism, `redefine` requires the external components to be in the same target namespace as the redefining schema, although external components from schemas that have no namespace can also be redefined. In the latter cases, the redefined components become part of the redefining schema's target namespace.

To illustrate the `redefine` mechanism, we will use it instead of the `include` mechanism in the International Purchase Order schema, `ipo.xsd`, and we will use it to modify the definition of the complex type `Address` contained in `address.xsd`:

Using redefine in the International Purchase Order

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO>

 <!-- bring in address constructs -->
 <redefine
  schemaLocation="http://www.example.com/schemas/address.xsd">

  <!-- redefinition of Address -->
  <complexType name="Address">
   <complexContent>
     <extension base="Address">
      <sequence>
       <element name="country" type="string"/>
      </sequence>
     </extension>
   </complexContent>
  </complexType>

 </redefine>

 <!-- etc -->

</schema>
```

The `redefine` element acts very much like the `include` element, it includes all the declarations and definitions from the `address.xsd` file. The complex type definition of `Address` uses the familiar extension syntax to add a `country` element to the definition of `Address`. However, note that the base type is also `Address`. Outside of the `redefine` element, any such attempt to define a complex type with the same name (and in the same namespace) as the base from which it is being derived would cause an error. But in this case, there is no error, and the extended definition of `Address` becomes the only definition of `Address`.

Now that `Address` has been redefined, the extension applies to all schema components that make use of `Address`. For example, address.xsd contains definitions of international address types that are derived from `Address`. These derivations will reflect the redefined `Address` type, as shown in the following snippet:

Snippet of ipo.xml using Redefined Address

```
....
<shipTo exportCode="1" xsi:type="ipo:UKAddress">
 <name>Helen Zoe</name>
 <street>47 Eden Street</street>
 <city>Cambridge</city>
 <!-- country was added to Address which is base type of UKAddress -->
 <country>United Kingdom</country>
 <!-- postcode was added as part of UKAddress -->
 <postcode>CB1 1JR</postcode>
</shipTo>
....
```

Our example has been carefully constructed so that the redefined `Address` type does not conflict in any way with the types that are derived from the original `Address` definition. But note that it would be very easy to create a conflict. For example, if the international address type derivations had extended `Address` by adding a `country` element, then the redefinition of `Address` would be adding an element of the same name to the content model of `Address`. It is illegal to have two elements of the same name (and in the same target namespace) but different types in a content model, and so the attempt to redefine `Address` would cause an error. In general, `redefine` does not protect you from such errors, and it should be used cautiously.

4.6 Substitution Groups

XML Schema provides a mechanism, called substitution groups, that allows elements to be substituted for other elements. More specifically, elements can be assigned to a special class of elements that are said to be substitutable for a particular named element which is called the exemplar. Note that the exemplar must be a global element. For example, we can declare two elements called `customerComment` and `shipComment` and assign them to a substitution group whose exemplar is `comment`, and so `customerComment` and `shipComment` can be used anyplace that we are able to use `comment`. Elements in a substitution group must have the same type as the examplar, or they can have a type that has been derived from the exemplar's type. To declare these two new elements, and to make them substitutable for the `comment` element, we use the following syntax:

Declaring Elements Substitutable for comment

```
<element name="shipComment" type="string"
    substitutionGroup="ipo:comment" />
<element name="customerComment" type="string"
    substitutionGroup="ipo:comment" />
```

When these declarations are added to the international purchase order schema, `comment` can be substituted for in the instance document, for example:

Snippet of ipo.xml with Substituted Elements

```
....
 <items>
   <item partNum="833-AA">
     <productName>Lapis necklace</productName>
     <quantity>1</quantity>
     <USPrice>99.95</USPrice>
     <ipo:shipComment>Use gold wrap if possible</ipo:shipComment>
     <ipo:customerComment>
       Want this for the holidays!
     </ipo:customerComment>
     <shipDate>1999-12-05</shipDate>
   </item>
 </items>
....
```

Note that when an instance document contains element substitutions whose types are derived from those of thei exemplars, it is *not* necessary to identify the derived types using the `xsi:type` construction that we described in Section 4.3.

The existence of a substitution group does not require any of the elements in that class to be used, nor does it preclude use of the exemplar. It simply provides a mechanism for allowing elements to be used interchangeably.

4.7 Abstract Elements and Types

XML Schema provides a mechanism to force substitution for a particular element or type. When an element or type is declared to be "abstract", it cannot be used in an instance document. When an element is declared to be abstract, a member of that element's substitution group must appear in the instance document. When an element's corresponding type definition is declared as abstract, all instances of that element must use `xsi:type` to indicate a derived type that is not abstract.

In the substitution group example we described in Section 4.6, it would be useful to specifically disallow use of the `comment` element so that instances must make use of the `customerComment` and `shipComment` elements. To declare the `comment` element abstract, we modify its original declaration in the international purchase order schema, `ipo.xsd`, as follows:

```
<element name="comment" type="string" abstract="true"/>
```

With `comment` declared as abstract, instances of international purchase orders are now only valid if they contain `customerComment` and `shipComment` elements.

Declaring an element as abstract requires the use of a substitution group. Declaring a type as abstract simply requires the use of a type derived from it (and identified by the `xsi:type` attribute) in the instance document. Consider the following schema definition:

Schema for Vehicles

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
         targetNamespace="http://cars.example.com/schema"
         xmlns:target="http://cars.example.com/schema">

 <complexType name="Vehicle" abstract="true"/>

 <complexType name="Car">
  <complexContent>
   <extension base="target:Vehicle"/>
  </complexContent>
 </complexType>

 <complexType name="Plane">
  <complexContent>
   <extension base="target:Vehicle"/>
  </complexContent>
 </complexType>

 <element name="transport" type="target:Vehicle"/>
</schema>
```

The `transport` element is not abstract, therefore it can appear in instance documents. However, because its type definition is abstract, it may never appear in an instance document without an `xsi:type` attribute that refers to a derived type. That means the following is not schema-valid:

```
<transport xmlns="http://cars.example.com/schema" />
```

because the `transport` element's type is abstract. However, the following is schema-valid:

```
<transport xmlns="http://cars.example.com/schema"
           xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
           xsi:type="Car"/>
```

because it uses a non-abstract type that is substitutable for `Vehicle`.

4.8 Controlling the Creation and Use of Derived Types

So far, we have been able to derive new types and use them in instance documents without any restraints. In reality, schema authors will sometimes want to control derivations of particular types, and the use of derived types in instances.

XML Schema provides a couple of mechanisms that control the derivation of types. One of these mechanisms allows the schema author to specify that for a particular (simple or complex) type, new types may not be derived from it, either (a) by restriction, (b) by extension, or (c) at all. To illustrate, suppose we want to prevent any derivation of the `Address` type by restriction because we intend for it only to be used as the base for extended types such as `USAddress` and `UKAddress`. To prevent any such derivations, we slightly modify the original definition of `Address` as follows:

Preventing Derivations by Restriction of Address

```
<complexType name="Address" final="restriction">
 <sequence>
  <element name="name"   type="string"/>
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
 </sequence>
</complexType>
```

The `restriction` value of the `final` attribute prevents derivations by restriction. Preventing derivations at all, or by extension, are indicated by the values `#all` and `extension` respectively. There exists also an optional `finalDefault` attribute on the `schema` element whose value can be one of the values allowed for the `final` attribute. The effect of specifying the `finalDefault` attribute is equivalent to specifying a `final` attribute on every type definition and element declaration in the schema.

Another type-derivation mechanism controls which facets can be applied in the derivation of a new simple type. When a simple type is defined, the `fixed` attribute may be applied to any of its facets to prevent a derivation of that type from modifying the value of the fixed facets. For example, we can define a `Postcode` simple type as:

Preventing Changes to Simple Type Facets

```
<simpleType name="Postcode">
  <restriction base="string">
    <length value="7" fixed="true"/>
  </restriction>
</simpleType>
```

Once this simple type has been defined, we can derive a new postal code type in which we apply a facet not fixed in the base definition, for example:

Legal Derivation from Postcode

```
<simpleType name="UKPostcode">
  <restriction base="ipo:Postcode">
    <pattern value="[A-Z]{2}\d\s\d[A-Z]{2}"/>
  </restriction>
</simpleType>
```

However, we cannot derive a new postal code in which we re-apply any facet that was fixed in the base definition:

Illegal Derivation from Postcode

```
<simpleType name="UKPostcode">
 <restriction base="ipo:Postcode">
  <pattern value="[A-Z]{2}\d\d[A-Z]{2}"/>
  <!-- illegal attempt to modify facet fixed in base type -->
  <length value="6" fixed="true"/>
 </restriction>
</simpleType>
```

In addition to the mechanisms that control type derivations, XML Schema provides a mechanism that controls which derivations and substitution groups may be used in instance documents. In Section 4.3, we described how

the derived types, `USAddress` and `UKAddress`, could be used by the `shipTo` and `billTo` elements in instance documents. These derived types can replace the content model provided by the `Address` type because they are derived from the `Address` type. However, replacement by derived types can be controlled using the `block` attribute in a type definition. For example, if we want to block any derivation-by-restriction from being used in place of `Address` (perhaps for the same reason we defined `Address` with `final="restriction"`), we can modify the original definition of `Address` as follows:

Preventing Derivations by Restriction of Address in the Instance

```
<complexType name="Address" block="restriction">
 <sequence>
  <element name="name"   type="string"/>
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
 </sequence>
</complexType>
```

The `restriction` value on the `block` attribute prevents derivations-by-restriction from replacing `Address` in an instance. However, it would not prevent `UKAddress` and `USAddress` from replacing `Address` because they were derived by extension. Preventing replacement by derivations at all, or by derivations-by-extension, are indicated by the values `#all` and `extension` respectively. As with `final`, there exists also an optional `blockDefault` attribute on the `schema` element whose value can be one of the values allowed for the `block` attribute. The effect of specifying the `blockDefault` attribute is equivalent to specifying a `block` attribute on every type definition and element declaration in the schema.

5. Advanced Concepts III: The Quarterly Report

The home-products ordering and billing application can generate ad-hoc reports that summarise how many of which types of products have been billed on a per region basis. An example of such a report, one that covers the fourth quarter of 1999, is shown in `4Q99.xml`.

Notice that in this section we use qualified elements in the schema, and default namespaces where possible in tl instances.

Quarterly Report, 4Q99.xml

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  period="P3M" periodEnding="1999-12-31">

 <regions>
  <zip code="95819">
   <part number="872-AA" quantity="1"/>
   <part number="926-AA" quantity="1"/>
   <part number="833-AA" quantity="1"/>
   <part number="455-BX" quantity="1"/>
  </zip>
  <zip code="63143">
   <part number="455-BX" quantity="4"/>
  </zip>
 </regions>

 <parts>
  <part number="872-AA">Lawnmower</part>
  <part number="926-AA">Baby Monitor</part>
  <part number="833-AA">Lapis Necklace</part>
  <part number="455-BX">Sturdy Shelves</part>
 </parts>

</purchaseReport>
```

The report lists, by number and quantity, the parts billed to various zip codes, and it provides a description of each part mentioned. In summarising the billing data, the intention of the report is clear and the data is unambiguous because a number of constraints are in effect. For example, each zip code appears only once (uniqueness constraint). Similarly, the description of every billed part appears only once although parts may be billed to several zip codes (referential constraint), see for example part number 455-BX. In the following sections, we'll see how to specify these constraints using XML Schema.

The Report Schema, report.xsd

```
<schema targetNamespace="http://www.example.com/Report"
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:r="http://www.example.com/Report"
        xmlns:xipo="http://www.example.com/IPO"
        elementFormDefault="qualified">

 <!-- for SKU -->
 <import namespace="http://www.example.com/IPO"/>

 <annotation>
  <documentation>
   Report schema for Example.com
   Copyright 2000 Example.com. All rights reserved.
  </documentation>
 </annotation>

 <element name="purchaseReport">
  <complexType>
   <sequence>
    <element name="regions" type="r:RegionsType"/>
    <element name="parts" type="r:PartsType"/>
   </sequence>
   <attribute name="period" type="timeDuration"/>
   <attribute name="periodEnding" type="date"/>
  </complexType>

  <unique name="dummy1">
   <selector>regions/zip</selector>
```

```
    <field>@code</field>
   </unique>

   <key name="pNumKey">
    <selector>parts/part</selector>
    <field>@number</field>
   </key>

   <keyref name="dummy2" refer="pNumKey">
    <selector>regions/zip/part</selector>
    <field>@number</field>
   </keyref>
  </element>

  <complexType name="RegionsType">
   <sequence>
    <element name="zip" maxOccurs="unbounded">
     <complexType>
      <sequence>
       <element name="part">
        <complexType>
         <complexContent>
          <restriction base="anyType">
           <attribute name="number" type="xipo:SKU"/>
           <attribute name="quantity" type="positiveInteger"/>
          </restriction>
         </complexContent>
        </complexType>
       </element>
      </sequence>
      <attribute name="code" type="positiveInteger"/>
     </complexType>
    </element>
   </sequence>
  </complexType>

  <complexType name="PartsType">
   <sequence>
    <element name="part" maxOccurs="unbounded">
     <complexType>
      <simpleContent>
       <extension base="string">
        <attribute name="number" type="xipo:SKU"/>
       </extension>
      </simpleContent>
     </complexType>
    </element>
   </sequence>
  </complexType>

</schema>
```

5.1 Specifying Uniqueness

XML Schema enables us to indicate that any attribute or element value must be unique within a certain scope. To indicate that one particular attribute or element value is unique, we use the `unique` element first to "select" a set of elements, and then to identify the attribute or element "field" relative to each selected element that has to be unique within the scope of the set of selected elements. In the case of our report schema, `report.xsd`, the `selector` element contains an XPath expression (see XML Path Language 1.0), `regions/zip`, that selects a list of all the `zip` elements in a report instance, and the `field` element contains a second XPath expression, `@code`, that specifies that the `code` attribute values of those elements must be unique. Note that the XPath expressions limit the scope of what must be unique. The report might contain another `code` attribute, but its value does not

have to be unique because it lies outside the scope defined by the XPath expressions.

Moreover, we can also indicate combinations of fields that must be unique. To illustrate, suppose we can relax the constraint that zip codes may only be listed once, although we still want to enforce the constraint that any product is listed only once within a given zip code. We could achieve such a constraint by specifying that the combination of zip code and product number must be unique. From the report document, `4Q99.xml`, the combined values of `zipcode` and `number` would be: {95819 872-AA}, {95819 926-AA}, {95819 833-AA}, {95819 455-BX}, and {63143 455-BX}. Clearly, these combinations do not distinguish between `zipcode` and `number` combinations derived from single or multiple listings of any particular zip, but the combinations would unambiguously represent a product listed more than once within a single zip. In other words, a schema processor could detect violations of the uniqueness constraint.

To define combinations of values, we simply add `field` elements to identify all the values involved. So, to add the part number value to our existing definition, we add a new `field` element whose XPath expression, `part/@number`, identifies the `number` attribute of `part` elements that are children of the `zip` elements identified by `regions/zip`:

A Unique Composed Value

```
<unique name="dummy1">
 <selector>regions/zip</selector>
 <field>@code</field>
 <field>part/@number</field>
</unique>
```

5.2 Defining Keys and their References

In the 1999 quarterly report, the description of every billed part appears only once. We could enforce this constraint using `unique`, however, we also want to ensure that every part-quantity element listed under a zipcode has a corresponding part description. We enforce the constraint using the `key` and `keyRef` elements. the report schema, `report.xsd`, shows that the `key` and `keyRef` constructions are applied using almost the same syntax as `unique`. The key element applies to the `number` attribute value of `part` elements that are children of the `parts` element. This declaration of `number` as a key means that its value must be unique and not nullable, and the name that is associated with the key, `pNumKey`, makes the key referenceable from elsewhere.

To ensure that the part-quantity elements have corresponding part descriptions, we say that the `number` attribute (`<field>@number</field>`) of those elements (`<selector>regions/zip/part</selector>`) must reference the `pNumKey` key. This declaration of `number` as a keyRef does not mean that its value must be unique, but it does mean there must exist a `pNumKey` with the same value.

As you may have figured out by analogy with `unique`, it is possible to define combinations of `key` and `keyRef` values. Using this mechanism, we could go beyond simply requiring the product numbers to be equal, and define a combination of values that must be equal. Such values may involve combinations of multiple value types (`string`, `integer`, `date`, etc), provided that the order and type of the `field` element references is the same in both the `key` and `keyRef` definitions.

5.3 XML Schema Constraints vs. XML 1.0 ID Attributes

XML 1.0 provides a mechanism for ensuring uniqueness using the ID attribute and its associated attributes IDREF and IDREFS. This mechanism is also provided in XML Schema through the `ID`, `IDREF`, and `IDREFS` simple types which can be used for declaring XML 1.0-style attributes. XML Schema also introduces new mechanisms that are more flexible and powerful. For example, XML Schema's mechanisms can be applied to any

element and attribute content, regardless of its type. In contrast, ID is a type of *attribute* and so it cannot be applied to attributes, elements or their content. Furthermore, Schema enables you to specify the scope within which uniqueness applies whereas the scope of an ID is fixed to be the whole document. Finally, Schema enables you to create `key`s or a `keyRef` from combinations of element and attribute content whereas ID has no such facility.

5.4 Importing Types

The report schema, `report.xsd`, makes use of the simple type `xipo:SKU` that is defined in another schema, and more specifically, in another target namespace. Recall that we used `include` so that the schema in `ipo.xsd` could make use of definitions and declarations from `address.xsd`. We cannot use `include` here because it can only pull in definitions and declarations from a schema whose target namespace is the same as the including schema's target namespace. Hence, the `include` element does not identify a namespace (although it does require a `schemaLocation`). The import mechanism that we describe in this section is an important mechanism that enables schema components from different target namespaces to be used together, and hence enables the schema validation of instance content defined across multiple namespaces.

To import the type `SKU` and use it in the report schema, we identify the namespace in which `SKU` is defined, and associate that namespace with a prefix for use in the report schema. Concretely, we use the `import` element to identify `SKU`'s target namespace (`http://www.example.com/IPO`), and we associate the namespace with the prefix `xipo` using a standard namespace declaration. The simple type `SKU`, defined in the namespace `http://www.example.com/IPO`, may then be referenced as `xipo:SKU` in any of the report schema's definitions and declarations.

In our example, we imported one simple type from one external namespace, and used it for declaring attributes. XML Schema in fact permits multiple schema components to be imported, from multiple namespaces, and they can be referred to in both definitions and declarations. For example in `report.xsd` we could additionally reuse the `comment` element declared in `ipo.xsd` by referencing that element in a declaration:

```
<element ref="xipo:comment"/>
```

Note however, that we cannot reuse the `shipTo` element from `po.xsd`, and the following is not legal because only global schema components can be imported:

```
<element ref="xipo:shipTo"/>
```

In `ipo.xsd`, `comment` is declared as a global element, in other words it is declared as an element of the `schema`. In contrast, `shipTo` is declared locally, in other words it is an element declared inside a complex type definition, specifically the `PurchaseOrderType` type.

Complex types can also be imported, and they can be used as the base types for deriving new types. Only named complex types can be imported; Local, anonymously defined types cannot. Suppose we want to include in our reports the name of an analyst, along with contact information. We can reuse the (globally defined) complex type `USAddress` from `address.xsd`, and extend it to define a new type called `Analyst` by adding the new elements `phone` and `email`:

Defining Analyst by Extending USAddress

```
<complexType name="Analyst">
 <complexContent>
  <extension base="xipo:USAddress">
   <sequence>
    <element name="phone" type="string"/>
    <element name="email" type="string"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Using this new type we declare an element called `analyst` as part of the `purchaseReport` element declaration (declarations not shown) in the report schema. Then, the following instance document would conform to the modified report schema:

Instance Document Conforming to Report Schema with Analyst Type

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  period="P3M" periodEnding="1999-12-31">
  <!-- regions and parts elements omitted -->
   <analyst>
        <name>Wendy Uhro</name>
        <street>10 Corporate Towers</street>
        <city>San Jose</city>
        <state>CA</state>
        <zip>95113</zip>
        <phone>408-271-3366</phone>
        <email>uhro@example.com</email>
   </analyst>
</purchaseReport>
```

When schema components are imported from multiple namespaces, each namespace must be identified with a separate `import` element. The `import` elements themselves must appear as the first children of the `schema` element. Furthermore, each namespace must be associated with a prefix, using a standard namespace declaration, and that prefix used to qualify references to any schema components belonging to that namespace. Finally, `import` elements optionally contain a `schemaLocation` attribute to help locate resources associated with the namespaces. We discuss the `schemaLocation` attribute in more detail in a later section.

5.4.1 Type Libraries

As XML schemas become more widespread, schema authors will want to create simple and complex types that can be shared and used as the basic building blocks for building new schemas. XML Schemas already provides types that play this role: the simple types summarised in Appendix B. However, schema authors will want to go beyond this basic collection of simple data types, and create libraries of types to represent currency, units of measurement, business addresses, and so on. Each library might consist of a schema containing one or more definitions, for example, a schema containing a currency type:

Example Currency Type in Type Library

```
<schema targetNamespace="http://www.example.com/Currency"
        xmlns:c="http://www.example.com/Currency"
        xmlns="http://www.w3.org/2000/10/XMLSchema">

 <annotation>
  <documentation>
   Definition of Currency type based on ISO 4217
  </documentation>
 </annotation>

 <complexType name="Currency">
  <simpleContent>
   <extension base="decimal">
    <attribute name="name">
     <simpleType>
      <restriction base="string">

        <enumeration value="AED">
         <annotation>
          <documentation>
           United Arab Emirates: Dirham (1 Dirham = 100 Fils)
          </documentation>
         </annotation>
        </enumeration>

        <enumeration value="AFA">
         <annotation>
          <documentation>
           Afghanistan: Afghani (1 Afghani = 100 Puls)
          </documentation>
         </annotation>
        </enumeration>

        <enumeration value="ALL">
         <annotation>
          <documentation>
           Albania, Lek (1 Lek = 100 Qindarka)
          </documentation>
         </annotation>
        </enumeration>

        <!-- and other currencies -->

      </restriction>
     </simpleType>
    </attribute>
   </extension>
  </simpleContent>
 </complexType>

</schema>
```

An example of an element appearing in an instance and having this type:

```
<convertFrom name="AFA"/>199.37</convertFrom>
```

Once we have defined the currency type, we can make it available for re-use in other schemas through the `import` mechanism just described.

5.5 Any Element, Any Attribute

In previous sections we have seen several mechanisms for extending the content models of complex types. For example, a mixed content model can contain arbitrary character data in addition to elements, and for example, a content model can contain particular elements whose types are imported from external namespaces. However, these mechanisms provide very broad and very narrow controls respectively. The purpose of this section is to describe a flexible mechanism that enables content models to be extended by any elements and attributes belonging to specified namespaces.

To illustrate, consider a version of the quarterly report, `4Q99html.xml`, in which we have embedded an HTML representation of the XML parts data. The HTML content appears as the content of the element `htmlExample`, and the default namespace is changed on the outermost HTML element (`table`) so that all the HTML elements belong to the HTML namespace, `http://www.w3.org/1999/xhtml`:

Quarterly Report with HTML, 4Q99html.xml

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  period="P3M" periodEnding="1999-12-31">

 <regions>
   <!-- part sales listed by zipcode, data from 4Q99.xml -->
 </regions>

 <parts>
   <!-- part descriptions from 4Q99.xml -->
 </parts>

 <htmlExample>
  <table xmlns="http://www.w3.org/1999/HTML"
         border="0" width="100%">
   <tr>
     <th align="left">Zip Code</th>
     <th align="left">Part Number</th>
     <th align="left">Quantity</th>
   </tr>
   <tr><td>95819</td><td> </td><td> </td></tr>
   <tr><td> </td><td>872-AA</td><td>1</td></tr>
   <tr><td> </td><td>926-AA</td><td>1</td></tr>
   <tr><td> </td><td>833-AA</td><td>1</td></tr>
   <tr><td> </td><td>455-BX</td><td>1</td></tr>
   <tr><td>63143</td><td> </td><td> </td></tr>
   <tr><td> </td><td>455-BX</td><td>4</td></tr>
  </table>
 </htmlExample>

</purchaseReport>
```

To permit the appearance of HTML in the instance document we modify the report schema by declaring a new element `htmlExample` whose content is defined by the `any` element. In general, an `any` element specifies that any well-formed XML is permissible in a type's content model. In the example, we require the XML to belong to the namespace `http://www.w3.org/1999/xhtml`, in other words, it should be HTML. The example also requires there to be at least one element present from this namespace, as indicated by the values of `minOccurs` and `maxOccurs`:

Modification to purchaseReport Declaration to Allow HTML in Instance

```
<element name="purchaseReport">
 <complexType>
  <sequence>
   <element name="regions" type="r:RegionsType"/>
   <element name="parts" type="r:PartsType"/>
   <element name="htmlExample">
    <complexType>
     <sequence>
      <any namespace="http://www.w3.org/1999/xhtml"
           minOccurs="1" maxOccurs="unbounded"
           processContents="skip"/>
     </sequence>
    </complexType>
   </element>
  </sequence>
  <attribute name="period" type="timeDuration"/>
  <attribute name="periodEnding" type="date"/>
 </complexType>
</element>
```

The modification permits some well-formed XML belonging to the namespace `http://www.w3.org/1999/xhtml` to appear inside the `htmlExample` element. Therefore `4Q99html.xml` is permissible because there is one element which (with its children) is well formed, the element appears inside the appropriate element (`htmlExample`), and the instance document asserts that the element and its content belongs to the required namespace. However, the HTML may not actually be valid because nothing in `4Q99html.xml` by itself can provide that guarantee. If such a guarantee is required, the value of the `processContents` attribute should be set to `strict` (which is in fact the default value). In this case, an XML processor is obliged to obtain the schema associated with the required namespace, and validate the HTML appearing within the `htmlExample` element. Alternatively, the value of the `processContents` attribute can be set to `lax`, in which case the processor will validate the HTML on a can-do basis: It will validate elements and attributes for which it can obtain schema information, but it will not signal errors for those it cannot obtain schema information.

Namespaces may be used to permit and forbid element content in various ways depending upon the value of th `nameSpace` attribute, as shown in Table 4:

Table 4. Namespace Attribute in Any

Value of Namespace Attribute	Allowable Element Content
##any	Any well-formed XML from any namespace (default)
##local	Any well-formed XML that is not qualified, i.e. not declared to be in a namespace
##other	Any well-formed XML in a namespace different from the target namespace of the type being defined
"http://www.w3.org/1999/xhtml ##targetNamespace"	Any well-formed XML belonging to any namespace in the (whitespace separated) list; ##targetNamespace is shorthand for the target namespace of the type being defined

In addition to the `any` element which enables element content according to namespaces, there is a corresponding `anyAttribute` element which enables attributes to appear in elements. For example, we can permit any HTML attribute to appear as part of the `htmlExample` element by adding `anyAttribute` to its declaration:

Modification to htmlExample Declaration to Allow HTML Attributes

```
<element name="htmlExample">
 <complexType>
  <sequence>
   <any namespace="http://www.w3.org/1999/xhtml"
        minOccurs="1" maxOccurs="unbounded"
        processContents="skip"/>
  </sequence>
  <anyAttribute namespace="http://www.w3.org/1999/xhtml"/>
 </complexType>
</element>
```

This declaration permits an HTML attribute, say `href`, to appear in the `htmlExample` element. For example:

An HTML attribute in the htmlExample Element

```
....
  <htmlExample xmlns:h="http://www.w3.org/1999/xhtml"
               h:href="http://www.example.com/reports/4Q99.html">
     <!-- HTML markup here -->
  </htmlExample>
....
```

The `nameSpace` attribute in an `anyAttribute` element can be set to any of the values listed in Table 4 for the `any` element, and `anyAttribute` can be specified with a `processContents` attribute. In contrast to an `any` element, `anyAttribute` cannot constrain the number of attributes that may appear in an element.

5.6 schemaLocation

XML Schema uses the `schemaLocation` and `xsi:schemaLocation` attributes in three circumstances.

1. In an instance document, the attribute `xsi:schemaLocation` provides hints from the author to a processor regarding the location of schema documents. The author warrants that these schema documents are relevant to checking the validity of the document content, on a namespace by namespace basis. For example, we can indicate the location of the Report schema to a processor of the Quarterly Report:

Using schemaLocation in the Quarterly Report, 4Q99html.xml

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/Report
  http://www.example.com/Report.xsd"
  period="P3M" periodEnding="1999-12-31">

 <!-- etc -->

</purchaseReport>
```

The `schemaLocation` attribute contains pairs of values: The first member of each pair is the namespace for which the second member is the hint describing where to find to an appropriate schema document. The presence of these hints does not require the processor to obtain or use the cited schema documents, and the processor is free to use other schemas obtained by any suitable means, or to use no schema at all.

A schema is not required to have a namespace (see Section 3.4) and so there is a `noNamespaceSchemaLocation`

attribute which is used to provide hints for the locations of schema documents that do not have target namespaces.

2. In a schema, the `include` element has a required `schemaLocation` attribute, and it contains a URI reference which must identify a schema document. The effect is to compose a final effective schema by merging the declarations and definitions of the including and the included schemas. For example, in Section 4, the type definitions of `Address`, `USAddress`, `UKAddress`, `USState` (along with their attribute and local element declarations) from `address.xsd` were added to the element declarations of `purchaseOrder` and `comment`, and the type definitions of `PurchaseOrderType`, `Items` and `SKU` (along with their attribute and local element declarations) from `ipo.xsd` to create a single schema.

3. Also in a schema, the `import` element has optional `nameSpace` and `schemaLocation` attributes. If present, the `schemaLocation` attribute is understood in a way which parallels the interpretation of `xsi:schemaLocation` in (1). Specifically, it provides a hint from the author to a processor regarding the location of a schema document that the author warrants supplies the required components for the namespace identified by the `nameSpace` attribute. To import components that are not in any target namespace, the `import` element is used without a `nameSpace` attribute (and with or without a `schemaLocation` attribute). References to components imported in this manner are unqualified.

Note that the `schemaLocation` is only a hint and some processors and applications will have reasons to not use it; For example, an HTML editor may have a built-in HTML schema.

5.7 Conformance

An instance document may be processed against a schema to verify whether the rules specified in the schema are honored in the instance. Typically, such processing actually does two things, (1) it checks for conformance to the rules, a process called schema validation, and (2) it adds supplementary information that is not immediately present in the instance, such as types and default values, called infoset contributions.

The author of an instance document, such as a particular purchase order, may claim, in the instance itself, that it conforms to the rules in a particular schema. The author does this using the `schemaLocation` attribute discussed above. But regardless of whether a `schemaLocation` attribute is present, an application is free to process the document against any schema. For example, a purchasing application may have the policy of always using a certain purchase order schema, regardless of any `schemaLocation` values.

Conformance checking can be thought of as proceeding in steps, first checking that the root element of the document instance has the right contents, then checking that each subelement conforms to its description in a schema, and so on until the entire document is verified. Processors are required to report what checking has been carried out.

To check an element for conformance, the processor first locates the declaration for the element in a schema, and then checks that the `targetNamespace` attribute in the schema matches the actual namespace URI of the element (or, alternatively, that the schema does not have a `targetNamespace` attribute and the instance element is not namespace-qualified).

Supposing the namespaces match, the processor then examines the type of the element, either as given by the declaration in the schema, or by an `xsi:type` attribute in the instance. If the latter, the instance type must be an allowed substitution for the type given in the schema; what is allowed is controlled by the `block` attribute in the element declaration. At this same time, default values and other infoset contributions are applied.

Next the processor checks the immediate attributes and contents of the element, comparing these against the attributes and contents permitted by the element's type. For example, considering a `shipTo` element such as the

one in Section 2.1, the processor checks what is permitted for an `Address`, because that is the `shipTo` element's type.

If the element has a simple type, the processor verifies that the element has no attributes or contained elements, and that its character content matches the rules for the simple type. This sometimes involves checking the character sequence against regular expressions or enumerations, and sometimes it involves checking that the character sequence represents a value in a permitted range.

If the element has a complex type, then the processor checks that any required attributes are present and that their values conform to the requirements of their simple types. It also checks that all required subelements are present, and that the sequence of subelements (and any mixed text) matches the content model declared for the complex type. Regarding subelements, schemas can either require exact name matching, permit substitution by an equivalent element or permit substitution by any element allowed by an 'any' particle.

Unless a schema indicates otherwise (as it can for 'any' particles) conformance checking then proceeds one level more deeply by looking at each subelement in turn, repeating the process described above.

Appendices

A. Acknowledgements

Many people have contributed ideas, material and feedback that has improved this document. In particular, the editor would like to acknowledge contributions from David Beech, Paul Biron, Don Box, Allen Brown, David Cleary, Dan Connolly, Roger Costello, Martin Gudgin, Dave Hollander, Joe Kesselman, John McCarthy, Andrew Layman, Eve Maler, Ashok Malhotra, Noah Mendelsohn, Henry Thompson, and Priscilla Walmsley for validating the examples.

B. Simple Types & their Facets

The legal values for each simple type can be constrained through the application of one or more facets. Tables B1.a, B1.b and B1.c list all of XML Schemas built-in simple types and the facets applicable to each type. The names of the simple types and the facets are linked from the tables to the corresponding descriptions inXML Schema Part 2: Datatypes

Table B1.a. Simple Types & Applicable Facets

Simple Types	Facets				
	length	**minLength**	**maxLength**	**pattern**	**enumeration**
string	y	y	y	y	y
byte				y	y
unsignedByte				y	y
binary	y	y	y		y
integer				y	y
positiveInteger				y	y
negativeInteger				y	y
nonNegativeInteger				y	y

nonPositiveInteger				y	y
int				y	y
unsignedInt				y	y
long				y	y
unsignedLong				y	y
short				y	y
unsignedShort				y	y
decimal				y	y
float				y	y
double				y	y
boolean				y	
time				y	y
timeInstant				y	y
timePeriod				y	y
timeDuration				y	y
date				y	y
month				y	y
year				y	y
century				y	y
recurringDay				y	y
recurringDate				y	y
recurringDuration				y	y
Name	y	y	y	y	y
QName	y	y	y	y	y
NCName	y	y	y	y	y
uriReference	y	y	y	y	y
language	y	y	y	y	y
ID	y	y	y	y	y
IDREF	y	y	y	y	y
IDREFS	y	y	y		y
ENTITY	y	y	y	y	y
ENTITIES	y	y	y		y
NOTATION	y	y	y	y	y
NMTOKEN	y	y	y	y	y
NMTOKENS	y	y	y		y

The facets listed in Table B1.b apply only to simple types which are ordered. Not all simple types are ordered and so B1.b does not list all of the simple types.

Table B1.b. Simple Types & Applicable Facets							
Simple Types	**Facets**						
	max Inclusive	**max Exclusive**	**min Inclusive**	**min Exclusive**	**precision**	**scale**	**encoding**
byte	y	y	y	y	y	y	
unsignedByte	y	y	y	y	y	y	
binary							y
integer	y	y	y	y	y	y	
positiveInteger	y	y	y	y	y	y	
negativeInteger	y	y	y	y	y	y	
nonNegativeInteger	y	y	y	y	y	y	
nonPositiveInteger	y	y	y	y	y	y	
int	y	y	y	y	y	y	
unsignedInt	y	y	y	y	y	y	
long	y	y	y	y	y	y	
unsignedLong	y	y	y	y	y	y	
short	y	y	y	y	y	y	
unsignedShort	y	y	y	y	y	y	
decimal	y	y	y	y	y	y	
float	y	y	y	y			
double	y	y	y	y			
time	y	y	y	y			
timeInstant	y	y	y	y			
timePeriod	y	y	y	y			
timeDuration	y	y	y	y			
date	y	y	y	y			
month	y	y	y	y			
year	y	y	y	y			
century	y	y	y	y			
recurringDay	y	y	y	y			
recurringDate	y	y	y	y			
recurringDuration	y	y	y	y			

As shown in Table B1.c, the period and duration facets apply only to temporal simple types.

Table B1.c. Simple Types & Applicable Facets		
Simple Types	**Facets**	
	period	**duration**
time	y	y
timeInstant	y	y
timePeriod	y	y
timeDuration		
date	y	y
month	y	y
year	y	y
century	y	y
recurringDay	y	y
recurringDate	y	y
recurringDuration	y	y

C. Regular Expressions

XML Schema's `pattern` facet uses a regular expression language that supports Unicode. The language is similar to the regular expression language used in the Perl Programming language, although expressions are matched against entire lexical representations rather than user-scoped lexical represention such as line and paragraph. For this reason, the expression language does not contain the metacharacters ^ and $, although ^ is used to express exception, e.g. [^0-9]x.

Table C1. Examples of Regular Expressions	
Expression	Match(s)
Chapter \d	Chapter 0, Chapter 1, Chapter 2 ....
Chapter\s\d	Chapter followed by a single whitespace character (space, tab, newline, etc), followed by a single digit
Chapter\s\w	Chapter followed by a single whitespace character (space, tab, newline, etc), followed by a word character (XML 1.0 Letter or Digit)
Española	Española
\p{Lu}	any uppercase character, the value of \p{} (e.g. "Lu") is defined byUnicode
\p{IsGreek}	any Greek character, the 'Is' construction may be applied to any block name (e.g. "Greek") as defined by Unicode
\P{IsGreek}	any non-Greek character, the 'Is' construction may be applied to any block name (e.g. "Greek") as defined by Unicode
a*x	x, ax, aax, aaax ....
a?x	ax, x
a+x	ax, aax, aaax ....
(a|b)+x	ax, bx, aax, abx, bax, bbx, aaax, aabx, abax, abbx, baax, babx, bbax, bbbx, aaaax ....
[abcde]x	ax, bx, cx, dx, ex
[a-e]x	ax, bx, cx, dx, ex
[-ae]x	-x, ax, ex
[ae-]x	ax, ex, -x
[a-e-[bd]]x	ax, cx, ex
[^0-9]x	any non-digit character followed by the character x
\Dx	any non-digit character followed by the character x
.x	any character followed by the character x
.*abc.*	1x2abc, abc1x2, z3456abchooray ....
ab{2}x	abbx
ab{2,4}x	abbx, abbbx, abbbbx
ab{2,}x	abbx, abbbx, abbbbx ....
(ab){2}x	ababx

D. Index

XML Schema Elements. Each element name is linked to a formal XML description in either the Structures or Datatypes parts of the XML Schema specification. Element names are followed by one or more links to examples (identified by section number) in the Primer.

- `all`: 2.7
- `annotation`: 2.6
- `any`: 5.5
- `anyAttribute`: 5.5
- `appInfo`: 2.6
- `attribute`: 2.2

- attributeGroup: 2.8
- choice: 2.7
- complexContent: 2.5.3
- complexType: 2.2
- documentation: 2.6
- element: 2.2
- enumeration: 2.3
- extension:
- field: 5.1
- group: 2.7
- import: 5.4
- include: 4.1
- key: 5.2
- keyRef: 5.2
- length: 2.3.1
- list: 2.3.1
- maxInclusive: 2.3
- maxLength: 2.3.1
- minInclusive: 2.3
- minLength: 2.3.1
- pattern: 2.3
- redefine: 4.5
- restriction: 2.3, 4.4
- schema: 2.1
- selector: 5.1
- sequence: 2.7
- simpleContent: 2.5.1
- simpleType: 2.3
- union: 2.3.2
- unique: 5.1

XML Schema Attributes. Each attribute name is followed by one or more pairs of references. Each pair of references consists of a link to an example in the Primer, plus a link to a formal XML description in either the Structures or Datatypes parts of the XML Schema specification.

- abstract: element declaration [Structures], complex type definition[Structures]
- attributeFormDefault: schema element [Structures]
- base: simple type definition[Datatypes], complex type definition[Structures]
- block: complex type definition[Structures],
- blockDefault: schema element [Structures],
- elementFormDefault: schema element [Structures]
- final: complex type definition[Structures]
- finalDefault: schema element [Structures]
- fixed: simple type definition[Datatypes]
- form: element declaration [Structures], attribute declaration [Structures]
- itemType: list type definition[Datatypes]
- memberTypes: union type definition[Datatypes]
- maxOccurs: element declaration [Structures]
- minOccurs: element declaration [Structures]
- mixed: complex type definition [Structures]
- name: element declaration [Structures], attribute declaration [Structures], complex type definition [Structures], simple type definition[Datatypes]

- `nameSpace`: any element [Structures], `include` element [Structures]
- `noNamespaceSchemaLocation`: instance element [Structures]
- `xsi:null`: instance element [Structures]
- `nullable`: element declaration [Structures]
- `processContents`: any element [Structures], `anyAttribute` element [Structures]
- `ref`: element declaration [Structures]
- `schemaLocation`: include specification [Structures], redefine specification [Structures], import specification [Structures]
- `xsi:schemaLocation`: instance attribute [Structures]
- `substitutionGroup`: element declaration [Structures]
- `targetNamespace`: `schema` element [Structures]
- `type`: element declaration [Structures], attribute declaration [Structures]
- `xsi:type`: instance element [Structures]
- `use`: attribute declaration [Structures]
- `value`: attribute declaration [Structures], facet specification

XML Schema's simple types are described inTable 2.

E. Document History

> September 22 draft submitted as public draft.

- Corrections in examples: po.xsd in sec 2.1, "List type ..." in sec 2.3.1, "Deriving ..." in sec 2.5.1, "Illegal example ..." in sec 2.7, "Adding attributes to the inline" in sec 2.8, ipo.xsd in sec 4.1, "Preventing changes ..." in sec 4.8, "Legal ..." in sec 4.8, "Illegal ..." in sec 4.8, report.xsd in sec 5, "A unique ..." in sec 5.1, "Defining analyst ..." in sec 5.4, "Instance document ..." in sec 5.4, "Modification to purchaseReport ..." in sec 5.5. These corrections should mean that the examples now validate.
- Transposed order of secs 2.5.2 and 2.5.3
- Edited sec 2.5.3 (empty content) to use new syntax
- Corrected errors in sematics and description of union type, sec 2.3.2
- Removed extraenous sequence tag in first example in sec 2.7
- Deleted sec 2.5.4 on "Default Content", and merged some of its description of ur-type into sec 2.5.3.

September 13 draft published to WG.

- Added xsd: to anyType in examples.
- Reworded parts of secs. 2.3 (after Table 2), 2.3.1, 2.3.2 to clarify definition and restriction in derivation of simple types, and corrected some syntax errors. Added itemType and memberTypes attributes to index.
- Added new section 4.5 describing redefine with text and examples. Shifted previous subsections accordingly, 4.6 -> 4.7 etc. Added appropriate index entries. Foreshadowed redefine in section 4.1.
- Added additional text to "fixed" example in Table 1. Added sentence saying that ref= not applicable in global declarations (sec 2.2, para 9, last sentence).

September 10 draft published to WG.

Removed string entry from Table B1.b because it is not an ordered type. Changed pattern facet entry for binary in Table B1.a. Corrected example of boolean in Table 2. Improved binary example in Table 2. Annotated Table 2 to indicate conceptual nature of binary and recurringDuration types. Added text and examples to sec. 4.7 describing fixed attribute applied to facets, added this attribute to index. Added subsec. 2.3.1 using existing list type text, added a new subsec. 2.3.2 describing union types. Changed syntax in examples and index entries to reflect new syntax for restricting simple types ('normally' and by list). Annotated Table 2 to indicate simple types having canonical formats.

Changed syntax and text to reflect renaming of equivalence class to substitution group. Corrected namespace URI for XML Schema (to http://www.w3.org/2000/10/XMLSchema), and corrected several

XML Schema instance namespace URI (to http://www.w3.org/2000/10/XMLSchema-instance). Changed all examples to reflect new syntax of complexType content models, and started changing text and index to reflect these changes. Fixed some typos and cleaned up some sentences.

August 7 draft published to WG.

Added a new "type library" section 5.4.1.

July 16 draft published to WG.

Fixed examples of minOccurs and maxOccurs per WG decision to make defaults 1 and 1. Converted all ' in markup to ". Re-ordered datatypes in Table 2 and Tables B1.*. Added mention of ur-type in sec 2.5. Clarified use of min/maxOccurs and use attributes in global declarations in Table 1. Modified sec 5.6 to correct statement about optionality of namespace attribute, included example of schemaLocation, and mention of noNamespaceSchemaLocation. In sec 4.5 added clarification of xsi:type usage with equivalence classes.

In response to logged last call comments (identified by [LC-number]), and general comments:

- Added missing maxOccurs attributes to element declarations having minOccurs=0 [LC-38]
- Added clarifying text to Table 2 regarding delimitation of multiple example and NMTOKENS
- Added missing xsd:'s to examples in sec 2.6
- Fixed error in date example in Table 2
- Corrected namespace URI for XHTML
- Added text saying attribute groups can contain attribute groups [LC-169]
- Fixed up blockDefault wording in sec 4.7, added final- and blockDefault's to index [LC-39]
- Regarding LC-205: use EUR [8], use UK consistently [MW5], fix per [MW1], fix Unicode links per [MW7], explain SKU acronym [MW2], complex type Address with country attb fixed to US renamed to USAddress [5], use USPrice and weightKg [4], added international example with accented character in sec 2.3 [6], added sentence to clarify scope of address type in sec 4.2 [10], clarified regex expression in sec 2.3 [7], fixed opening paragraph of sec 2.8 [MW4]
- Fixed typos and other minor items: text involving myInteger and listOfMyIntType, an href to sec 5, zipcode to zip [LC-35], lists cannot be derived from lists [LC-88], extraneous links on "name" in sec 2.2, qualified changed to unqualified in sec 3.1 [LC-42] and minor fixes to surrounding text, shipper/biller changed to shippee/billee [LC-53], corrected part/@number in sec 5.1, publicKey attribute declaration syntax in sec 3.2

April 7th draft submitted for final call.

Changes to wording of Sections 3, 3.1, 3.2, 3.3

April 6th draft published to WG.

Added links to index pointing into Parts 1 and 2, and into body of Primer. Added links into body of Primer text pointing to Index. Substantially rewrote Section 3 to describe qualification of local elements & attributes. Changed schemas in Section 5 to use qualified elements.

April 2nd draft published to WG.

Updated Index. Modified descriptions of attribute occurrence constraints per issue 222 decision. Added examples for annotation and clarified text. Clarified introduction to nulls. New introduction to Section 4. Clarified Schema/XML 1.0 mixed models. Corrected minOccurs error for all-group elements. Corrected derivation-by-restriction example. Numerous small clarifications, and typo fixes.

March 20th draft published to WG for "last lap" review.

Added new section 3 describing namespaces and schema, and removed consequently redundant text from document. Renumbered sections after new sec 3. Added simple list type description. Rewrote section on Building Content Models (formerly "Groups"), to reflect new <choice> etc syntax. Added default/fixed distinction in sec 2. Rewrote prevention of restriction explanation in sec 4.7. Added description of abstract types. Added new example of defining a simple type in sec 2. Changed datatype names to reflect new camelCase convention. Added new built-in datatypes. Added mention of global attributes. Numerous text edits and typo corrections.

February 25th draft submitted as public draft.

Rewrote "Section X covers .." in Introduction. Added xsd: to simple types in sec. 2 and textual explanation. In sec. 2.3 added URLs to text and Table 1 linking it to Datatypes spec. Corrected URI for

xsi:. Added sec. 3.0 explanation for namespace convention change schema. Substantially rewrote se. 3.5 to fix error, and created a new section 3.6 to cover Abstract Elements. Changed "exact" to "block" in sec 3.6. Fixed error in location of unique/key defns in sec. 4.0, and XPath expressions. Fixed ##local and ##TargetNS, and clarified anyAttb in sec 4.5. Appendix B added URLs to Table B1 linking to Datatypes spec. Fixed general typos.

February 23rd draft published to WG.

Added sections on import, schemaLocation, conformance, wildcard, and type content. Replaced "source" with "base". Modified HTML for W3C compliance. Moved Types of Content section from section 3 to section 2. Updated use of namespaces in instance and schema in all sections, reworked section 2 text to account for these changes. Fixed typos, added/deleted text at suggestion of WG members.

February 16th draft published to WG.

Added regular expression description as an appendix. Substantial re-ordering and rewrite of section 2. Added index as an appendix. Fixed large number of typos and adopted TypeName and elementName naming convention.

February 9th, first draft published to WG.