XML Schema 1.1, Part 1: An introduction to XML Schema 1.1

An overview of the key improvements over XML Schema 1.0 and an in-depth look at datatypes

Neil Delima
Sandy Gao
Michael Glavassevich
Khaled Noaman

With XML Schema's wide adoption and diversity of usage, schema users have requested many improvements and new capabilities. The W3C XML Schema working group has developed XML Schema 1.1 to address these most commonly requested features including several which address shortcomings of XML Schema 1.0. In this first of a multi-part series of articles, authors Neil Delima, Sandy Gao, Michael Glavassevich, and Khaled Noaman introduce XML Schema 1.1 with an overview of the features introduced in this emerging standard and take an in-depth look at the additions and changes to the datatypes portion of the specification.

View more content in this series

Introduction

Since the XML Schema 1.0 specification became a W3C Recommendation in 2001, the developer community has discussed the merits and shortcomings of the language. The W3C XML Schema Working Group has worked on the next version of the language. In 2005, with the standard gaining wide adoption in the industry and its integration into many other standards including XSLT, XQuery and WSDL, the W3C hosted a workshop to reflect on user experiences and gather feedback to help guide the evolution of the language. This workshop along with the requests of other users in the community helped the XML Schema Working Group to shape the scope of the 1.1 version of the standard.

Frequently used acronyms

• W3C: World Wide Web Consortium
In this article we start with an overview of some new features of XML Schema 1.1 and then dive deeply into the enhancements made to the Datatypes part of the specification. The standard is now formally known as XML Schema Definition Language or XSD for short. We will use this abbreviation in this article and throughout the series. Sometimes, when the intention is clear, "XML Schema" and "schema" are also used to refer this language.

As a reader, keep in mind that this article was written while XML Schema 1.1 was still under development. Some of the details may change before XML Schema 1.1 becomes a W3C Recommendation.

XML Schema 1.0 pain points

Schema authors often face certain challenges. You can work around some of them, resulting in counter-intuitive schema designs; you might handle the others with code in programming languages.

This section examines some of the most commonly encountered issues and discusses how XML Schema 1.1 can help to solve them. Detailed discussions will be available in the subsequent parts of this article.

Content model restriction

Complex types can have different kinds of content. Those that allow child elements necessarily have one of `<xs:sequence>`, `<xs:choice>`, or `<xs:all>` as their content models. When a complex type is derived by restriction from another one, both content models have to satisfy certain conditions. Such conditions are specified to ensure that what is allowed by the restriction type is also allowed by the base type.

In XML Schema 1.0, these conditions are specified using a 25-case table, and the content models have to look very similar to satisfy these conditions. This can cause problems:

- The rigid rules in the 25 cases rule out some obviously valid derivations.
- The rule allow some obviously invalid derivations (that is, restriction allows more than base).

For example, in Listing 1, the type derived removes an optional element `tns:a` from the base type. This is clearly a valid restriction, but is invalid in XML Schema 1.0.

Listing 1. A derived type removes an optional element from the base type

```xml
<complexType name="base">
  <complexContent>
    <sequence>
      <element ref="tns:a" minOccurs="0" maxOccurs="1"/>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="tns:b"/>
        <element ref="tns:c"/>
      </choice>
    </sequence>
  </complexContent>
</complexType>
```
In XML Schema 1.1, the 25-case rule is removed, and replaced with a simple concept to reflect the "what is allowed by the restriction is also allowed by the base" goal. The above example becomes valid in XML Schema 1.1.

Co-constraints

Schema authors often want to enforce rules that involve more than one element or attribute. For example, "min must be less than or equal to max", or "the number of child elements must match the size attribute". Rules like these are often called co-occurrence constraints or simply co-constraints.

XML Schema 1.0 didn’t provide any facility to support co-constraints. Users sometimes have to write Java™ or C code to check them after the XML document is loaded into memory. This hurts maintainability and makes the schemas less interoperable. Some users seek help from other XML validation languages like Schematron and Relax NG (see Related topics) for co-constraints support, which complicates their otherwise XSD-based architecture.

XML Schema 1.1 supports co-constraints natively. The newly introduced <xs:assert> element can include conditions specified in XPath 2.0 (see Related topics) expressions. Listing 2 shows an example:

Listing 2. Co-constraints in XML Schema 1.1

```xml
<xs:complexType name="intRange">
  <xs:attribute name="min" type="xs:int"/>
  <xs:attribute name="max" type="xs:int"/>
  <xs:assert test="@min <= @max"/>
</xs:complexType>
```

Schema evolution

People often find the need to evolve their schemas, to add extensions for new information. The wildcard is a powerful tool designed for this purpose. It can be used in the earlier versions of the schema to leave extension points, and in later versions, concrete elements can be introduced in place of the wildcard. But wildcards have some unfortunate shortcomings:

- The very controversial Unique Particle Attribution (UPA) rule makes it difficult to use optional wildcards.
Wildcards are not expressive enough to describe "everything except the following."
Repetition of the same wildcard for every complex type to make the entire schema extensible is tedious.

XML Schema 1.1 makes schema evolution much easier. Among other things, wildcards are improved tremendously. They no longer violate UPA when conflicting with explicitly specified elements, they can exclude a list of namespaces or a list of names, and they can even be defaulted. It is easier than ever to write extensible schemas.

For example, to express a content model for "one and only one element called userName and any number of any other elements, before or after userName", you define the content model as in Listing 3.

Listing 3. Content model in XML Schema 1.0

```xml
<xs:sequence>
  <xs:element ref="tns:userName"/>
  <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
  <xs:element ref="tns:userName"/>
  <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
</xs:sequence>
```

But this is invalid in XML Schema 1.0. When an element named userName is encountered, it is ambiguous whether it matches the wildcard `<xs:any>` or the element declaration `<xs:element>`. To work around this problem, some people insert separator elements between the wildcard and the element. This works but makes both the schema and the XML documents quite ugly. Yet another problem is that the wildcard also allows userName, so the "one and only one" rule cannot be enforced.

In XML Schema 1.1, the schema snippet in Listing 3 becomes valid because wildcards are weakened, meaning that when an element can match either an element declaration or a wildcard, the element declaration always takes precedence. This avoids the UPA problem. With the help of the negative wildcard, you can now express the "one and only one userName and anything else" rule as in Listing 4.

Listing 4. Content model in XML Schema 1.1

```xml
<xs:sequence>
  <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip" notQName="tns:userName"/>
  <xs:element ref="tns:userName"/>
  <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip" notQName="tns:userName"/>
</xs:sequence>
```

XML Schema datatypes

The XML Schema specification consists of two parts: Structures and Datatypes (see Related topics). In this section, we will cover some of the changes in the Datatypes portion of the specification that part of XML Schema 1.1 introduced. In future articles, we will go into more details about the changes to the Structures part.
Alignment with XQuery 1.0 and XPath 2.0 data model types

The type system used by the W3C XQuery 1.0, XPath 2.0, XSLT 2.0 and XQuery 1.0 and XPath 2.0 Data Model Recommendations (see Related topics) is an extension of the W3C XML Schema 1.0 Recommendation. In addition to the XML Schema 1.0 built-in primitive data types, these specifications defined five additional data types in the XML Schema 1.0 namespace, namely: anyAtomicType, untyped, untypedAtomic, dayTimeDuration, and yearMonthDuration. To align the type systems of XML Schema and these specifications, the XML Schema 1.1 data types specification introduced three of these data types, namely: anyAtomicType, dayTimeDuration, and yearMonthDuration.

anyAtomicType

The anyAtomicType is a special XML Schema 1.1 built-in data type derived by restriction from anySimpleType. Since anyAtomicType is the base for all primitive data types, the value and lexical space of anyAtomicTypes is the union of the value and lexical spaces of all primitive data types. To explain this better, see the XML Schema (Listing 5) and the valid XML document (Listing 6) below. In this example, an element of type anyAtomicType can contain a string or integer as a valid value. It can also be cast to a more specific type derived from anyAtomicType using xsi:type. It should be pointed out that anyAtomicType does not define any constraining facets and thus you cannot use it as the base type of a user-defined simple type.

Listing 5. Sample XML Schema for anyAtomicType

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="test" xmlns:pfx="test">
  <element name="root">
    <complexType>
      <sequence>
        <element name="elanyAtomicType" type="anyAtomicType"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Listing 6. Sample XML document for anyAtomicType

```xml
<pfx:root xmlns:pfx="test" xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <elanyAtomicType>Test</elanyAtomicType>
  <elanyAtomicType>12345</elanyAtomicType>
  <elanyAtomicType xsi:type="xs:string">Test</elanyAtomicType>
  <elanyAtomicType xsi:type="xs:integer">12345</elanyAtomicType>
</pfx:root>
```

yearMonthDuration

The duration datatype specified in the XML Schema 1.0 Datatypes Recommendation (see Related topics) is a partially ordered type that represents a period of time. For example, the duration values P30D and P1M are incomparable since a month can contain anywhere from 28 to 31 days. To allow durations to be comparable, XML Schema 1.1 introduced two new totally
ordered datatypes, namely: `yearMonthDuration` and `dayTimeDuration`, derived by restriction from `duration`.

In XML Schema 1.1, the `yearMonthDuration` datatype is derived from `duration` by restricting its lexical representation to contain only the year and month components. You can express this with the regular expression: `'-?P\[0-9]+(Y(\[0-9\]+M)?|M)`. The value of the year and month components allow an unsigned integer. The optional minus sign indicates a negative `yearMonthDuration`. The value space of the `duration` datatype consists of an integral number of months and a decimal number of seconds. The value space of the `yearMonthDuration` datatype is a restriction of the value space of the `duration` datatype whose seconds property is zero (0).

A positive `yearMonthDuration` of one year and six months can be represented lexically as `P1Y6M` or `P18M`. The value of this `yearMonthDuration` is 18 months. Examples of valid `yearMonthDuration` values include `P1Y2M`, `P12Y`, `-P20M`, while the following representations are invalid: `P-1Y`, `P1Y-1M`, and `P1YM`. The `yearMonthDuration` datatype is fully ordered. For any 2 `yearMonthDuration` values `D1` and `D2`, the ordering relationship between `D1` and `D2` can be established. That is, either `D1 > D2`, or `D1 < D2`.

User-defined datatypes can be derived by restriction from `yearMonthDuration`, by specifying constraining facets allowed by `duration`. Since `yearMonthDuration` is derived by restriction from `duration`, its fundamental facet, ordered, is partial which remains unchanged by derivation. However `yearMonthDuration` is in fact totally ordered.

`dayTimeDuration`

Similar to the `yearMonthDuration`, the `dayTimeDuration` datatype is derived from `duration` by restricting its lexical representation to only contain the day and time (hour, minute, and seconds) components from the `duration` datatype. This can be expressed by durations that match the regular expression `[^YM]*[^DT]*`. The values of the days, hours, and minutes components are not restricted, but allow an arbitrary unsigned `xs:integer`. Similarly, the values of the seconds component allows an arbitrary unsigned `xs:decimal`. The optional minus sign indicates a negative `dayTimeDuration`. The value space of the `dayTimeDuration` datatype is a restriction of the value space of the `duration` datatype with a months property value of zero and a fractional seconds value.

A positive `dayTimeDuration` of one day, two hours, three minutes, and 4.5 seconds can be represented lexically as `P1DT2H3M4.5S`. The value of this `yearMonthDuration` is 93784.5 
(1*24*60*60+2*60*60+3*60+4.5) fraction seconds. Note that if the number of days, hours, minutes, seconds is zero, you can omit it from the lexical representation provided that at least one of these is present. If the `dayTimeDuration` consists of only days, then the designator `T` must be absent. Some more examples of valid `dayTimeDuration` include `P1D`, `PT25H`, `P22DT2H`, `PT1H9M5.5S`, `-PT20M`, `-PT60.60S` and examples of invalid `dayTimeDuration` are `P-5D`, `P1D1M1H1S`, `PDT1M`, `P5H`, and `P1DT`. Like `yearMonthDuration` the `dayTimeDuration` datatype is fully ordered.
Datatypes derived by restriction from `dayTimeDuration` can specify the same constraining facets as those of the `duration` datatype. Note that the value of the `whitespace` facets for the `yearMonthDuration` and `dayTimeDuration` is fixed to `collapse` and cannot be changed.

Listing 7 illustrates a valid XML Schema 1.1 fragment that uses the `yearMonthDuration` and `dayTimeDuration` datatypes.

**Listing 7. Sample XML Schema for `yearMonthDuration` and `dayTimeDuration`**

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="test" xmlns:pxfx="test">
  <simpleType name="ymdBase">
    <restriction base="yearMonthDuration">
      <minInclusive value="P1Y6M"/>
    </restriction>
  </simpleType>
  <simpleType name="ymdDerived">
    <restriction base="ymdBase">
      <minInclusive value="P19M"/>
    </restriction>
  </simpleType>
  <simpleType name="dtdBase">
    <restriction base="dayTimeDuration">
      <maxInclusive value="-P2DT2H"/>
    </restriction>
  </simpleType>
  <simpleType name="dtdDerived">
    <restriction base="dtdBase">
      <maxInclusive value="-P51H"/>
    </restriction>
  </simpleType>
  <element name="root">
    <complexType>
      <sequence>
        <element name="elYearMonthDuration" type="ymdDerived"/>
        <element name="elDayTimeDuration" type="dtdDerived"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Listing 7 illustrates a valid XML Schema 1.1 fragment that uses the `yearMonthDuration` and `dayTimeDuration` data types. The simple type `ymdDerived` restricts the base type `ymdBase`, which in turn restricts the XML Schema `yearMonthDuration` built-in data type using the `minInclusive` facet. Since `yearMonthDuration` is totally ordered, the value P19M of the derived type `ymdDerived` is greater than the value P1Y6M of the base type `ymdBase` which makes it a valid restriction. Similarly, the simple type `dtdDerived` restricts the base type `dtdBase` which in turn restricts the XML Schema `dayTimeDuration` built-in data type using the `maxInclusive` facet. In this case, the negative duration of -P51H of the derived type, `dtdDerived`, is less than that of the base type -P2DT2H. The element, `root`, contains child elements `elYearMonthDuration` and `elDayTimeDuration` of types `ymdDerived` and `dtdDerived` respectively.

**precisionDecimal**

The `precisionDecimal` is a new type introduced in XML Schema 1.1 to support the new IEEE-754 floating-point decimal type. It varies from `decimal` in that precision decimal numbers carry not only
a numeric value but also an arithmetic precision that is retained. `precisionDecimal` also includes values for positive infinity (`+INF`) and negative infinity (`-INF`), and for not a number (`NaN`). It also differentiates between positive zero (+0) and negative zero (-0).

The lexical space of `precisionDecimal` is the set of all decimal numerals (with or without a decimal point), numerals in scientific (exponential) notation, and the character strings `INF`, `+INF`, `-INF`, and `NaN`.

User-defined datatypes derived by restriction from `precisionDecimal` can specify the same constraining facets as those of `decimal`. In addition, two new constraining facets, `maxScale` and `minScale`, are introduced to allow derived types to narrow down the value space of `precisionDecimal`. A `maxScale` puts an upper limit while `minScale` puts a lower limit on the arithmetic precision of `precisionDecimal` values.

In Listing 8, we define a new `price` type that accepts values between -999,999.99 and 999,999.99.

**Listing 8. Sample XML Schema fragment that uses precisionDecimal**

```xml
<xs:simpleType name='price'>
  <xs:restriction base='xs:precisionDecimal'>
    <xs:totalDigits value='8'/>
    <xs:minScale value='2'/>
    <xs:maxScale value='2'/>
  </xs:restriction>
</xs:simpleType>
```

One thing to remember when using `NaN` is that it is incomparable with any other value including itself. So if you use `NaN` for any of the bounding facets (`minInclusive`, `maxInclusive`, `minExclusive` or `maxExclusive`), you will end up with a datatype that has an empty value space.

Similarly, including `NaN` in an enumeration does not make it accept `NaN` values. If you would like to have `NaN` as part of the value space, define a union type that includes a `NaN` only datatype (by specifying a pattern facet with a value of "`NaN`").

**Timezone versus timezone offset**

The `date`, `time`, and `dateTime` related datatypes specified in the XML Schema 1.0 specification included an optional timezone in the form of `((+ | -) hh : mm) | 'Z'`. When a timezone value is added to a Universal Coordinated Time (UTC) `dateTime`, it results in a date and time in that timezone.

Although the XML Schema 1.0 specification meant a timezone offset, it used the term timezone to describe it, which caused some confusion since timezone and timezone offset represented two different concepts. A timezone identifies a specific location or region (for example, Pacific Time) while a timezone offset is the difference in hours and minutes between UTC and a particular timezone (for example, 11:00-05:00). The XML Schema 1.1 specification has rectified this problem and now differentiates between timezone and timezone offset.
Leap seconds

A leap second is an extra second added to the last day of the month of March, June, October or December which means that the last minute in the day for that month has more than 60 seconds. A leap second is added in order to keep UTC within 0.9 seconds of observed astronomical time.

Because the date- and time-related types, defined in the XML Schema 1.1 specification, do not support leap seconds, they cannot be used to represent the final second, in UTC, of any of the days that have leap seconds added to them. An example of such date is 1972-06-30. Users need to make appropriate changes at the application level to handle such dates if it is important to keep track of leap seconds.

Implementation-defined simple types and facets

The XML Schema specification defines a number of primitive types, such as string, boolean, and double, that a processor understands and provides an implementation for. Many systems need more types than those defined as built-ins in the specification. You can meet some of these needs can be met by deriving types from existing ones, but not others.

Implementation-defined primitive types

XML Schema 1.1 now allows implementors of XML Schema processors to define their own primitive simple types. It is up to each XML Schema processor to decide whether to recognize such types or not.

Implementors need to follow these rules:

- Use anyAtomicType as the base type.
- Decide which of the constraining facets apply and what they mean when applied (NOTE: you have to include a whiteSpace facet).
- Define the mechanism to reference the new type with a target namespace different from http://www.w3.org/2001/XMLSchema (which is controlled by W3C).
- Define the lexical space, value space, and lexical mapping of the new type.
- Define the equality relationship.
- Define the values of fundamental facets.

As an implementor of an XML processor, we might define a special date datatype that conforms to the format of day-month-year, but uses various separators, not just a hyphen (-). In keeping with the rules defined above, we use anyAtomicType as the base type, and we define a new namespace which we might call "http://www.example.com/XMLSchema-primitiveTypes". We want our date to be represented in the format: day, separator, month, separator, year. In the lexical space for the date datatype, representations for day, month, and year, will have the same representation as the ones defined in XML Schema 1.1 and the same rules. We want separator to be one of three values: period (.), hyphen (-), or slash (/).

We also define the facets we will support in our implementation. The fundamental facets can include the following facets and values:
• ordered: partial
• bounded: false
• cardinality: countably infinite
• numeric: false

Per the rules, we need to include a `whiteSpace` facet, and we will define it with a value of "collapsed", which applies to `date` and all derived datatypes. Per the XML Schema 1.1 specification, we can also define other constraining facets and values as we choose, such as:

• `pattern`
• `enumeration`
• `maxInclusive`
• `maxExclusive`
• `minInclusive`
• `minExclusive`
• `assertions`
• `dateSeparator` (implementation-defined)

Using this definition, "2008-11-01", "2008.11.01", and "2008/11/01" are all valid lexical representations of date, and they all denote the same day "November 1, 2008".

**Implementation-defined facets**

The XML Schema specification defines a set of constraining facets (such as `minInclusive` or `maxLength`) that you can apply to simple types. A constraining facet is a construct that you can use to control the value space of simple type during derivation. A schema aware processor understands and supports constraining facets.

Similarly to implementation-defined primitive types, XML Schema 1.1 allows implementors to define their own constraining facets and it is up to the XML Schema processor to support such facets or not.

Here are some rules to follow:

• Define the properties of the facet.
• Define the behavior of the facet.
• Define the mechanism to reference the new facet with namespace other than `http://www.w3.org/2001/XMLSchema` (as the W3C controls that namespace).
• Define the primitive datatypes the new constraining facet applies to.

In **Listing 9**, you see how an XML processor implementor might define the `dateSeparator` facet that restricts the separator in the value space of the implementation-defined `date` and all datatypes derived from it.
Listing 9. An example of implementation-defined facet

```xml
<dateSeparator
    fixed = boolean : false
    id = ID
    value = '-' | '.' | '/'

    ... >
    (optional element content here)
    ...
</dateSeparator>
```

The facet definition might define other attributes with a non-schema namespace in addition to fixed, id, and value. Any derived datatype can then restrict the value space of the implementation-defined date by applying the dateSeparator facet.

Now look at how a user might use this implementation-defined data type and its implementation-defined facet. In Listing 10, we define a new type, specialDate, that uses the new facet to restrict the representation of date to accept only values that have slash (/) as a separator.

Listing 10. An example of derived type based on an implementation-defined type

```xml
<xs:simpleType name="specialDate">
    <xs:restriction base="xyz:date">
        <xyz:dateSeparator value="/" />
    </xs:restriction>
</xs:simpleType>
```

Now only "2008/11/01" is allowed by specialDate, and "2008-11-01" and "2008.11.01" are not.

Conclusion

Introduction

Other articles in this series

- XML Schema 1.1, Part 2: Co-occurrence constraints using XPath 2.0
- XML Schema 1.1, Part 3: Evolve your schema with powerful wildcard support

In this article we gave an overview of XML Schema 1.1, highlighting the pain points of XML Schema 1.0 and briefly how XML Schema 1.1 addresses several of these with examples of content model restriction, co-constraints and schema evolution through the use of wildcards. We then took an in-depth look at the enhancements made to the Datatypes portion of the specification, including the new data types and allowance for implementation-defined primitive types and facets. In Part 2 of the series, we will further explore the new co-constraint features, specifically assertions and the conditional type assignment mechanism.
Related topics

- **XML 1.0 specification**: Read about XML and how it enables generic SGML to be served, received, and processed on the Web.
- **XML Schema Part 1: Structures Second Edition**: Learn more about the W3C XML Schema language and how it describes the structure and constrains the contents of XML 1.0 documents, including those which exploit the XML Namespace facility. This specification depends on XML Schema Part 2: Datatypes.
- **W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures**: Check out the latest specification of the W3C XML Schema language.
- **W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes**: Find more information on the new datatypes added to the W3C XML Schema language.
- **XQuery 1.0**: Learn more about XML Query language, which uses the structure of XML to express queries across all kinds of data.
- **XML Path Language 2.0**: Learn more about the XPath language.
- **XSL Transformations (XSLT) Version 2.0**: Review this specification that defines the syntax and semantics of the XSLT 2.0 language.
- **XQuery 1.0 and XPath 2.0 Data Model (XDM)**: Read about this W3C specification which is the data model of XPath 2.0, XSLT 2.0, and XQuery languages.
- **Schematron**: Check out this language for making assertions about the presence or absence of patterns in XML documents.
- **RELAX NG**: Explore a schema language for XML.
- **The XML Parser for Java (Xerces2-J)**: Try this parser distributed by Apache.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **IBM technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM trial software for product evaluation**: Build your next project with trial software available for download directly from developerWorks, including application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- **developerWorks podcasts**: Listen to interesting interviews and discussions for software developers.