Scope

This document describes some of the issues associated with versioning of XML schemas. Some of the mechanisms, such as the recommended version numbering convention, are likely relevant to other data formats as well (e.g. NetCDF files).

Background

One of the key requirements of the NextGen system, and SOA-based architectures in general, is the need to gracefully evolve over time. In a large system, it is generally not possible to update all components of the system whenever a new message data type is needed, or an existing data type needs to be expanded. In addition, it is desirable to allow for on-the-fly upgrades of data producers and/or consumers in a loosely-coupled manner, requiring only minimal coordination between data producers and consumers to effect any desired changes. These system agility requirements dictate the adoption of a coherent versioning strategy from the outset.

Two types of compatibility are generally referenced when discussing versioning, backwards-compatibility and forwards-compatibility. Backwards compatibility is defined as the ability of a version 2 data consumer to consume data from a version 1 data producer. Forwards compatibility is the inverse - the ability of a version 2 data producer to produce data that will still be consumable by a version 1 data consumer (slightly harder problem). For a loosely-coupled system, both types of compatibility are highly desirable if they can be implemented in a practical way.

The eXtensible Markup Language (XML) is extensible in the sense that the set of tags for any given language defined using XML is specified by the designers of that language. This is very different than extensibility over time for a given language. Version 1.0 of XML Schema does not provide much in the way of support for the latter, and this is generally recognized as being a significant weakness of the version 1.0 specification. Version 1.1 of the specification, due to be release in early 2008, addresses many of the versioning weaknesses, but widespread adoption of this newer version will take some time.

XML Schema Versioning Mechanisms

Existing Versioning Mechanisms

As mentioned, version 1.0 of XML schema does not provide robust support for versioning. There are several mechanisms that allow for some level of extensibility, but they are generally flawed in some way. For example, there is an XML schema 'any' type that can be used as a placeholder in a schema type definition to represent one or more future data elements of arbitrary type. However, due to constraints in the way the 'any' element is defined, there are issues with using it when defining an extensible XML type hierarchy. This constraint significantly limits the usefulness of the 'any' element. A second common extension practice is to add a separate '<ext>' container element to those data types where extension is desirable, and add all element extensions within that block. This approach suffers from the fact that, over time, data structures all become cluttered with '<ext>' elements and data types are partitioned into 'first-class' citizens (the original defined elements) and 'second-class'
citizens (the new elements contained within the ‘<ext>’ element). This results in semantic confusion over time, as some of the new elements in a structure may be just as important as the original elements.

**Versioning Mechanisms incorporated into XML Schema 1.1**

The W3C is aware of the above limitations of XML Schema 1.0, and has added improved support for versioning in the proposed XML Schema 1.1. One part of their effort has consisted of gathering a comprehensive set of versioning use cases. The use cases are available at:

http://www.w3.org/XML/2005/xsd-versioning-use-cases

Most (if not all) of the use cases are supported by mechanisms incorporated in XML Schema 1.1. A document describing the ways in which the new schema features can be used to implement versioning solutions is available at:

http://www.w3.org/TR/xmlschema-guide2versioning

As mentioned in the document, perhaps the cleanest and easiest to use new feature with respect to versioning is to use the 'open content model', which provides the ability to implicitly allow elements of any type to be added (interleaved) at any position in the schema. This is implemented by simply including the 'openContent' schema fragment within a schema, as shown below:

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.example.org/ns/personName/1"
           elementFormDefault="qualified"
           xmlns:namens="http://www.example.org/ns/personName/1">
  <xs:openContent mode="interleave">
    <xs:any namespace="##any" processContents="lax"
             minOccurs="0" maxOccurs="unbounded"/>
  </xs:openContent>

  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element name="given" type="xs:string"/>
      <xs:element name="family" type="xs:string"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>

  <xs:element name="personName" type="namens:nameType"/>
</xs:schema>
```

This approach removes the need for schema developers to insert their own 'any' elements wherever flexibility in the content model is needed, minimizing the effort involved in developing a schema as well as reducing clutter in the final schema.

**Schema Version Numbering**

The above versioning guide does a good job addressing how one will be able in the future to specify data types within schemas in extensible ways, but is incomplete in one respect - it does not yet specify a overall strategy for management of the schema versions themselves (section 10.4, 'Version numbers' is currently blank). A separate document discussing this issue is available at:

http://www.w3.org/2001/tag/doc/versioning-xml
This document is not specific to XML Schema version 1.1, and does not provide clear guidance on a single best strategy for versioning of schemas, but rather describes the tradeoffs associated with a number of the commonly-used strategies in the present environment. Another document describing versioning best practices is available on Roger Costello's XML site at:


Both of these documents discuss the option of using schema namespace attributes as one of the primary versioning mechanisms. Support for the The changes between versions break down into three basic types:

1. Schema changes that do not affect the encoded XML documents that conform to the schema (patches). This includes things such as changes to the schema comments and/or formatting.
2. Additions or changes to the schema that can be implemented in such a way that backwards and forwards compatibility can be preserved. This includes new elements within existing data types, and new data types.
3. Changes that break backwards and/or forwards compatibility. This includes changes such as restriction of element content, removal of schema types, or other structural changes.

Support for all three cases can be supported using a compound version number of the form:

$major.$minor.$patch

When used as a namespace suffix, this would typically result in a schema namespace like:

targetNamespace="http://www.opengis.net/gml/3.2.1"

Since namespace changes in the context of today's tooling are still generally disruptive, and changes that fall into the 'patch' category do not, by definition, affect the XML content model, an alternative is to use only a two-part version number suffix. This would result in a typical namespace like:

targetNamespace="http://www.opengis.net/gml/3.2"

This approach eliminates any need to change code and/or related schemas when patches are released. Note that for backwards-compatible changes that do add to the content model, the minor number is always required, in order to indicate to data consumers that the additional content is available in a schema instance. This requirement does incur a certain amount of schema maintenance overhead when using many of the current XML processing tools. This is discussed in the following section, along with some suggested tooling improvements.

**Tooling Support for Schema Versioning**

If using low-level XML parsers (DOM/SAX/XPATH...) support for versioning using version numbers embedded in schema namespaces is straightforward to implement. The low-level readers simply access the namespace information and set up parser options as appropriate. The situation becomes more complicated when tools are used to automatically generate code that binds the XML data types to class objects (most commonly accomplished using the Java language). The tools tend to generate fixed package names based on the namespaces, and have limited support for managing the backwards/forwards compatibility issues. There is no standardized way, for example, to specify that version 3.2 is compatible with version 3.1, and a data consumer that was built using the 3.1 schema should map incoming XML data types in the 3.2 namespace to the objects generated for the 3.1.namespace.

This issue is not a problem with XML Schema, per se, but is related to the lack of version management in
version 1.0 of XML Schema. If XML Schema version 1.0 had specified a versioning policy, the tools certainly
would have supported it. Given that support for versioning is on the way, but isn't here yet, one approach is to
attempt to find a practical approach to improving the tooling in advance of the formal XML Schema 1.1
specification release and downstream tooling improvements. One possible approach is to do the following:

1. **Generate class package names that are unique only down to the major version number level.** This
   prevents coders from having to maintain and manage separate versions of packages for all the minor
   (forwards/backwards compatible) releases. The XMLBeans XML-Java binding package provides a
certain level of support for this today using a configuration file that allows the Java package name to be
customized on a namespace-by-namespace basis. However, one must manually specify each namespace
   mapping, requiring a new ‘rule’ for each new minor version. The addition of a wildcard capability would
   allow for all namespaces of the form 'http://opengis.net/gml/3.X' to generate code in the package
   'net.opengis.gml._3', without having to manually specify each individual mapping.

   For other binding technologies that, a post-processing step after code generation can be used to change
   the package names (a simple Perl script could be used to accomplish this)

2. **Modify the code generation tools to include support for a 'namespace compatibility mapping'
   feature.** This would allow for data consumers to evaluate incoming namespaces for compatibility based
   on some simple pattern matching rules. For example, a consumer that is compatible with version 3.X of
   a namespace would specify a mapping rule like:

   http://www.opengis.net/gml/3.* --> http://www.opengis.net/gml/3

   This would allow a consumer based on version 3.1 of a schema to successfully parse instances based on
   3.2 of the schema, since both would be mapped (bound) to the same set of Java objects. Note that the
default behavior of two of the more popular binding technologies, XMLBeans and JAXB, is to ignore
unknown content, so extra elements added in new minor releases of a schema should not cause problems
for consumers based on earlier versions of the schema - the new elements are simply discarded.

   This mechanism could be implemented with relatively little effort in a number of the key XML/Java
binding libraries (e.g. XMLBeans, JAXB). The basic mechanism is flexible enough that alternate
namespace versioning conventions (such as the W3C year-based convention scheme could be supported
as well.)