# Notes

Compiled by SMR in review for NGDS version policy prep. 1/18/2011

**Web Service Contract Versioning Fundamentals Part II: Version Identifiers and Versioning Strategies**   
by [Thomas Erl](http://www.thomaserl.com/presskit/), [David Orchard](http://www.soamag.com/contributors/bio-dorchard.php) and [James Pasley](http://www.soamag.com/contributors/bio-jpasley.php) Published: Jaunary 19, 2009 (SOA Magazine Issue XXV: January 2009) <http://www.soamag.com/I25/0109-4.php>

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here: version="2.0". Additional period + decimal pairs that lead to more detailed version numbers are sometimes used, e.g.: version="2.0.1.1". Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes. The following convention has emerged in the industry to assign additional meaning to these numbers:

• A minor version is expected to be backwards compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards compatible with versions 5.0 and 5.1.

• A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards compatible with version 4.0.

it has become a common convention to use date values in namespaces when versioning XML schemas, as follows: xmlns=<http://actioncon.com/schema/po/2010/09>

The expression of version information must be standardized across all service contracts within the boundary of a service inventory. Some authority must guarantee the linearity, consistency, and description quality of version information.

Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

A backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service's existing consumers…, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value

Loose strategy: make parts of the contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements. XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions. For example, see ATOM. Wildcards allow undefined content to be passed in intechange documents, and provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards places a greated burden of validation on the underlying service logic.

<http://docs.oasis-open.org/ubl/prd-UBL-NDR-2.0.pdf>

3.7 modularity

A namespace is a collection of semantically related elements, types and attributes. For larger namespaces, schema modules – internal schema modules – may be defined. UBL document schemas may have zero or more internal modules that they include. The document schema for a namespace then includes those internal modules.

If two namespaces are mutually dependent then clearly, importing one will cause the other to be imported as well. For this reason there must not exist circular dependencies between UBL schema modules. By extension, there must not exist circular dependencies between namespaces. A namespace “A” dependent upon type definitions or element declaration defined in another namespace “B” must import “B’s” document schema.

[SSM2] A document schema in one UBL namespace that is dependent upon type definitions or element declarations defined in another namespace MUST only import the document schema from that namespace.

To ensure there is no ambiguity in understanding this rule, an additional rule is necessary to address potentially circular dependencies as well – schema A must not import internal schema modules of schema B.

[SSM3] A document schema in one UBL namespace that is dependant upon type definitions or element declarations defined in another namespace MUST NOT import internal schema modules from that namespace.

3.6 Versioning Scheme

Major version information is captured within the namespace name of each UBL schema module while combined major and minor version information is captured within the xsd:version attribute of the xsd:schema element.

In UBL, the major-version field must be changed in a release that breaks compatibility with the previous release of that namespace. If a change does not break compatibility then only the minor version need change. Subsequent minor releases begin with minor-version 1.

If either of the common namespaces requires a major version change then its namespace URI must change. If its namespace URI changes then any schema that imports the new version of the namespace must also change (to update the namespace declaration). And since this would require a major version change to the importing schema, its namespace URI in turn must change. The outcome is twofold:

* There should never be ambiguity at the point of reference in a namespace declaration or version identification. A dependent schema imports precisely the version of the namespace that is needed. The dependent schema never needs to account for the possibility that the imported namespace can change.
* When a dependent schema is upgraded to import a new version of a schema, the dependent schema’s version must change.

Minor version changes, however, would not require changes to the namespace URI of any schemas. Because of this, semantic compatibility across minor versions (as well as major versions) is essential. Semantic compatibility in this sense pertains to preserving the business function.

Version numbers are based on a logical progression. All major and minor version numbers will be based on positive integers. Version numbers always increment positively by one [sometimes increments >1 are used to indicate significant changes]

Schema Versioning Strategy and Policy

http://www.mncourts.gov/?page=3390

Last updated - 03/28/2011

This topic describes the schema versioning strategy and policies.

## Objectives

Enable schema to evolve.

Provide support for prior versions of schemas.

Minimize the impact of schema changes to integration partners.

Keep the number of supported schema versions down to a manageable number.

Eliminate the need for integration partners to make unnecessary changes to their applications due to new schema versions.

Give our partners as much time as possible to react to schema changes.

Define a mechanism to gracefully phase in new schema versions and phase out old versions.

This strategy is intended to address these concerns:

Partner may not be able to update their applications on the same schedule that the data providers implement a schema change; prevent schema changes from breaking client code

As the number of schema versions increases, support becomes more difficult - the data providers need to keep the number of schema versions to a level that can be supported

Partner application changes cannot be made on short notice – data providers must provide notice before a schema version is inactivated, and support prior versions for some period of time

Why are schema changes necessary?

As XML schema constructs are typed, any modification that would result in a type change is a potential incompatibility. For example, we might currently express a measurement or a count in integer values (xs:int), but in a future enhancement we might need a way to allow decimal fractions for the same value (xs:double). If your application is expecting a field to return an integer at runtime and we start returning a double instead, your application could fail. So, we need a way to avoid changing the type directly.

Another example of a type change would be adding an element to an existing complex-type. If this new element is required and your client application does not include it, or expect it in a response that client application could fail. A similar problem would occur when removing an element.

Changing the multiplicity of an element is also a potential incompatibility. For example, we might currently allow a single Name element, but in a future enhancement we might allow repeating Name elements (maxOccurs="unbounded"). As some toolkits interpret repeating fields as arrays, this would be equivalent to a type change.

Note: We try to avoid making purely cosmetic changes to schemas that would break client code.

Strategy

3 main types of schema.

Simple Type Schemas: These schemas contain enumerated simple types that are used to define a list of valid values for a given element or attribute. These schema are never used on their own but are always imported into and used by another schema. Simple Type schemas can change over time as values are added, modified or removed. These changes do not result in versioning upgrades.

Base Schema: This schema contains type definitions that are used by other schema when defining messages. Like Simple Type schemas this schema is never used on its own but is always imported into and used by another schema.

Message Schemas: Schemas that we refer to as Message Schemas define the content of messages that are either input to, or output from, a service. Message schemas are named with a suffix containing the base schema version that they are associated to along with the version of the message schema. The top level element of every message defined within a message schema has an attribute named schemaVersion. This attribute will carry the version number of that message schema.

The version of a message schema that is used is driven by the consumer of a particular service.

When changes occur that would result in a type change the following steps are taken to prevent breaking existing client code.

Any types contained in the base schema that require changes are copied and given a new name within that schema. Then the change is made to the type with the new name and an annotation is added to the old type indicated that it has been deprecated. For example a complex-type named ChargeType may need to have a new element added to it for some business reason. A new copy of the ChargeType complex-type would be added to the base schema with the name ChargeType2.

If an element or type contained within a message schema requires a change, a new copy of that message schema will be created with its message version number incremented and the prior version will be deprecated.

Versioning Support Policy

The support status of message schemas will fall into 5 categories.

Draft: These include schemas that are still being developed and being reviewed both internally and by partners. These schemas are likely to change before implementation.

Pre-Release: These include schemas that have not been published in production yet. Generally these will be available for testing in our development environment though they may be still considered in a draft state.

Supported: These include schemas that are the current version, the prior version or any prior version that has been deprecated within one year. These schemas are fully supported. Full support includes the following:

When regression testing is done all supported schemas for a particular service are tested.

Issues that are identified with a service and a supported version of the schema receive the highest priority for troubleshooting and resolution.

Documentation for a particular service will always reflect the current version of its message schema.

Unsupported: These include schemas that have been deprecated for more than a year and are not the prior version to a current schema. Unsupported means the following:

This message schema version is still available for use.

Any regression testing that is done will not include this version of the schema.

Issues that are identified with a service and an unsupported version of the schema will have a lower priority for troubleshooting and resolution, and depending on the problem may not be corrected. (The resolution may involve the partner migrating to the current version of the schema).

Inactivated: These include versions of message schemas that can no longer be used. Message schemas will generally not be inactivated unless there is a particular reason to do so, such as a business change that makes them either not support the current business process, causes them to inaccurately represent their area of court information, or are no longer used by any consumers. We will attempt to provide a minimum of 6 months notice when a version of a message schema is about to become inactivated.

Suggested Best Practices

The following best practices are suggested for use of message schema versions:

Use supported versions. As a part of your applications normal maintenance schedule attempt to keep it current with using only supported versions of message schemas. This provides the greatest likelihood that system changes will not negatively impact your application (because of regression testing) and assures that the support that you receive from the court will be at its highest level.

When you have a need to change your application for some reason also migrate to the current version of any message schemas that are being used. If your application has fallen behind more than one version there is no need to migrate to the intermediate versions.

For new development always use the current schema version.

Monitor the Court Integration Services website, and Integration Services update notification emails for changes to the support status of message schemas. When a new version of a schema is published and an old version is deprecated, or when a version is slated to be inactivated, the change in status for that schema will be reflected on the CourtXML schema web page and will be announced in an Integration Services update email.

Design your software to be able to handle, or gracefully ignore, new elements that we add to the schema. This will simplify migration to a new version.

Make the schema version a configuration setting so if a new version does not change anything that is being used then migration to a new version only requires a configuration change.

## Schema versioning

<http://sindbad.gsfc.nasa.gov/xfdu/pdfdocs/xmlworkshop0914.pdf>

XML Schema Best Practices, Namespace, and Versioning Issues, 2005, Louis Reich (NASA/CSC MOIMS/IPR WG Chair): presentation for Consultative Committee for Space Data Systems (http://www.ccsds.org), Atlanta,Georgia, September 14,2005 (accessed 2011-01-22)

* Scenarios..
  + New schema changes interpretation of element
  + New schema extends the namespace (e.g. new elements)

### Versioning Approaches

Very Similar version of this discussion also at http://www.xfront.com/Versioning.pdf

#### • Option 1: Change the schema version attribute

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

attributeFormDefault="unqualified"

version="1.0">

* Pros
  + Easy
  + Instance documents do not change if schema only extended
  + Applications could interrogate ‘version’ attribute and take action
* Cons
  + Validator ignores version attribute – not enforceable

#### • Option 2: Change the schemaVersion attribute of the root element

With this approach an attribute is included on the element that introduces the namespace. In the examples below, this attribute is named ‘schemaVersion’. This option could be used in two ways.

Option 2A: Change the schemaVersion Attribute of the Root element

<xs:schema xmlns="http://www.exampleSchema"

targetNamespace="http://www.exampleSchema"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:element name="Example">

<xs:complexType>

….

<xs:attribute name="schemaVersion" type="xs:decimal" use="required" fixed="1.0"/>

</xs:complexType>

</xs:element>

* Defined in Schema as required attribute
* Instance documents forced to set
* Schema validator can enforce
* Advantage
  + Instances would not be valid without same version
* Disadvantage
  + Does not allow instance to be valid against multiple versions Schema

Option 2B: Change the schemaVersion Attribute of the Root element

The second approach uses the schemaVersion attribute in an entirely different way. It no longer captures the version of the schema within the schema (i.e., it is not a fixed value). Rather, it is used in the instance to declare the version (or versions) of the schema with which the instance is compatible. This approach would have to be done in conjunction with option 1 (or an alternative indicator in the schema file to identify its version).

The schemaVersion attribute’s value could be a list or a convention could be used to define how this attribute is used. For example, if the convention was that the schemaVersion attribute declares the latest schema version with which the instance is compatible, then the example instance below states that the instance should be valid with schema version 1.2 or earlier.

With this approach, an application could compare the schema version (captured in the schema file) with the version to which the instance reports that it is compatible.

<xs:schema … version="1.3">

<xs:element name="Example">

<xs:complexType>

…

<xs:attribute name="schemaVersion" type="xs:decimal" use="required"/>

</xs:complexType>

</xs:element>

<Example schemaVersion="1.2" xmlns="http://www.example" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.example MyLocation\Example.xsd">

* Version NOT set in root element’s attribute in schema
* Revert to Option 1 (or Similar) to set version in schema
* Use application logic to verify version in schema vs. instance
* Pros
  + Caters for compatibility across multiple versions
  + App knows about changes
* Cons
  + Extra processing by application-- pre-parse instance document to determine schema version with which it is compatible, and compare this value to the version number stored in the schema file.

#### • Option 3: Change the schema’s targetNamespace

Change targetNamspace for each new version

<xs:schema xmlns="http://www.someAuthority/exampleSchema/1.0" targetNamespace=" http://www.someAuthority/exampleSchema/1.0” xmlns:xs=”http://www.w3.org/2001/XMLSchema”

elementFormDefault=”qualified”

attributeFormDefault=”unqualified”>

* Pros
  + Application knows about change
  + Requires action to assure that there are no compatibility problems with the new schema. At a minimum, the instance documents that use the schema, and schemas that include the relevant schema, must change to reference the new targetNamespace. This can be both an advantage and a disadvantage. [note this is not a problem if mapping from original namespace to schema location does not change]
* Cons
  + Instance documents must upgrade to utilize features of new schema version.

#### • Option 4: Change the name/location of the schema

This approach changes the file name or location of the schema. This mimics the convention that many people use for naming their files so that they know which version is the most current (e.g., append version number or date to end of file name).

* Pros
  + None
* Cons
  + The schemaLocation attribute in the instance document is optional and is not authoritative even if it is present. It is a hint to help the processor to locate the schema.
  + Some consider it a security risk

### Terminology

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

The term "backward compatiable" will mean that an instance document defined by an old schema can be processed by an application that handles the new schema.

The term "forward compatiable" will mean that an instance document defined by a new schema can be processed by an application that handles the old schema.

## XML Schema Versioning Best Practices

From <http://www.xfront.com/Versioning.pdf>. This work was developed collaboratively by members of the xml-dev list group. This material is made freely available in the public interest. Any use of this material must acknowledge the xml-dev list group.

1. Capture the schema version somewhere in the XML schema.
2. Identify in the instance document, what version/versions of the schema with which the instance is compatible.
3. Make previous versions of an XML schema available.

This allows applications to use previous versions. It also allows users to migrate to new versions of the schema as compatibility is assured. One way to do this is to have applications pre-parse the instance and choose the appropriate schema based on the version number. For example, one could have the schemaLocation URI point to a document that includes a list of the locations of the available versions of the schema. A tool could then be used to obtain the correct version of the schema. The disadvantage of this approach is that this pre-parsing requires two passes at the XML instance (one to get the correct version of the schema and one to validate).

1. When an XML schema is only extended, (e.g., new elements, attributes, extensions to an enumerated list, etc.) one should strive to not invalidate existing instance documents. For example, if one is adding new elements or attributes, one could consider making them optional where this makes sense.

Also, one could come up with a convention for schema versioning to indicate whether the schema changed significantly (case 1) or was only extended (case 2). For example, for case 1 a version could increment by one (e.g., v1.0 to v2.0) whereas for case 2 a version could increment by less than one (e.g., v1.2 to v1.3). In this case, a possible approach would be to do the following with respect to the schema:

* + Change the schema version number within the schema (e.g., option 1).
  + Record the changes in the schema in a change history.
  + Make the new and previous versions of the schema available (therefore, one would want to change the file name/location as well).

1. Where the new schema changes the interpretation of some element (e.g., a construct that was valid and meaningful for the previous schema does not validate against the new schema), one should change the target namespace. In this case, the changes with respect to the schema are the same as with [4], with one addition:
   * Change the target namespace. In this case there are also required changes with respect to the instances that use this schema.
   * Update the instances to reflect the new target namespace.
   * Confirm that there are no compatibility problems with the new schema.
   * Change the attribute that identifies the version/versions of the schema with which the instance is valid.
   * Update the schema name/location if appropriate

## Impact of XML Schema Versioning on System Design

**by Roger L. Costello and Melissa Utzinger (no date, accessed 2011-01-22)** [**www.xfront.com/SchemaVersioning.html**](http://www.xfront.com/SchemaVersioning.html)

Creating a new version of an XML Schema may have effects that ripple through many parts of a system. Managing these effects can be expensive. So it is worthwhile to examine ways to mitigate the costly ripple effects of new versions of a Schema.

Frequently, Schema versioning is considered in isolation from the rest of the system. However, as noted, Schema changes may impact other parts of a system, so we recommend that Schema versioning be part of an integrated system evolution plan. Schema versioning is one of the drivers for system evolution.

As a strategy for facilitating system evolution we focus on these three parts of a system - Schemas, instance documents, and applications. To treat these three parts in a holistic fashion we make the following recommendations:

To minimize impact to existing instance documents and applications as new versions of XML Schemas are created, we make the following recommendations:

**Recommendation 1:** To avoid breaking namespace-aware applications with each new version of an XML Schema use the same namespace for all versions.

**Recommendation 2:** To prevent breaking old instance documents give the new Schema version a different filename or a different URL location or both.

**Recommendation 3:** To facilitate an application in recognizing that an element's content has changed, don't use anonymous types. Instead, use named types.

**Recommendation 4:** If you change a type when you create a new version of a Schema then give the type a different name.

**Recommendation 5:** Change the name of an element's type only if its *immediate* content has changed.

**Recommendation 6:** Use a version attribute on the root element. If an instance document is a compound document - that is, an assembly of XML fragments - then place a version attribute on the root of each fragment.

**Recommendation 7:** Applications should use the tag names to locate data in instance documents. Applications should be designed to anticipate that the order of tags may change.

**Recommendation 8:** Define a system-wide protocol (e.g., fault reporting mechanism) to be used when an application is unable to process an instance document it receives from another application.