

An approach to a XBRL taxonomy versioning strategy

Javier Mora González
XBRL Spain
javier.mora@xbrl.org.es

Abstract

The problem of management XBRL taxonomy versioning is essential to start successfully the different XBRL projects around the world. At the moment, different development groups are working with distinct numbered edition of the taxonomies, and as a consequence, a hard effort is necessary to debug errors. This paper suggests what methodology should be implemented to manage the numbering and namespace naming of these new editions, and what naming convention should be used for the relevant taxonomies so as to ensure access to all prior editions. Based on the analysis of this approach, we suggest a set of best practices to perform these objectives.

1. Introduction

In Open Standards, and specifically in those related to the XML[1] family, there is a problematic related to versioning. This problematic is especially important in the XBRL[2] vocabulary taxonomies, and their evolution depends to a large extent on finding a valid solution for this problem.

COREP and FINREP XBRL taxonomies are currently being implemented by a number of different National Supervisors in Europe.

Because all the supervisors did not engage in this implementation process at the same time, they are not all using the same numbered edition

of the taxonomies. The implementation process has highlighted a number of bugs and problems with the current editions of the taxonomies that are in circulation. In addition, there will be changes to the business rules underpinning the taxonomies that will need to be reflected in new editions of the taxonomies.

The solution is to be implemented urgently, because to create 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.

In this article we are going to suggest different strategies, with their advantages and disadvantages, and with the aim of trying to establish some best practices regarding the versioning of XBRL taxonomies.

2. Materials and Methods

Consider two cases for changes to XML schemas:

- *UPDATES*: For example, a construct that was valid and meaningful for the previous schema does not validate against the new schema.
- *EXTENSIONS*: The new schema extends the namespace (e.g., by adding new elements), but does not invalidate previously valid documents.

We have been considered the following approaches[5,6] to identify a new schema version:

1. Change the (internal) schema version attribute.
2. Create a schemaVersion attribute on the root element.
3. Change the schema's targetNamespace.
4. Change the name/location of the schema.

3. Discussion

Now we will describe each studied approach:

3.1. Change the (internal) schema version attribute.

In this approach one would simply change the number in the optional version attribute at the start of the XML schema. For example, in the code below one could change version="1.0" to version="1.1"

```
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified"
version="1.0">
```

ADVANTAGES:

- Easy. Part of the schema specification.
- Instance documents would not have to change if they remain valid with the new version of the schema.
- The schema contains information that informs applications that it has changed. An application could interrogate the version attribute, recognize that this is a new version of the schema, and take appropriate action.

DISADVANTAGES:

- The validator ignores the version attribute. Therefore, it is not an enforceable constraint.

3.2. Create a schemaVersion attribute on 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.

A) First, like the previous approach, this attribute could be used to capture the schema version. In this case, one could make the attribute required and the value fixed. Then each instance that used this schema would have to set the value of the attribute to the value used in the schema. This makes schemaVersion a constraint that is enforceable by the validator. With the example schema below, the instance would have to include a schemaVersion attribute with a value of 1.0 for the instance to validate.

```
<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>
```

ADVANTAGES:

- The schemaVersion attribute is an enforceable constraint. Instances would not validate without the same version number.

DISADVANTAGES:

- The schemaVersion number in the instance must match exactly. This does not allow an instance to indicate that it is valid using multiple versions of a schema.

B) 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 the first

approach (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.

Sample Schema (declares it's version as 1.3)

```
<xs:schema xmlns="http://www.exampleSchema"
targetNamespace="http://www.exampleSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified"
version="1.3">
<xs:element name="Example">
<xs:complexType>
....
<xs:attribute name="schemaVersion" type="xs:decimal"
use="required"/>
</xs:complexType>
</xs:element>
```

Sample Instance (declares it is compatible with version 1.2 or 1.2 and other versions depending upon the convention used)

```
<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">
```

ADVANTAGES:

- Instance documents may not have to change if they remain valid with the new schema version.
- Like first approach, an application would receive an indication that the schema has changed.
- Could provide an alternative to schemaLocation as a means to point to the correct schema version. This could be desirable where the business practice requires the use of a schema in a controlled repository, rather than an arbitrary location.

DISADVANTAGES:

- Requires extra processing by an application. For example, an application would have to pre-parse the instance to determine what schema version with which it is compatible, and compare this value to the version number stored in the schema file.

3.3. Change the schema's targetNamespace.

In this approach, the schema's 'targetNamespace' could be changed to designate that a new version of the schema exists. One way to do this is to include a schema version number in the designation of the target namespace as shown in the example below.

```
<xs:schema xmlns="http://www.exampleSchemaV1.0"
targetNamespace="http://www.exampleSchemaV1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
```

ADVANTAGES:

- Applications are notified of a change to the schema (i.e., an application would not recognize the new namespace).
- 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 both an advantage and a disadvantage.

DISADVANTAGES:

- With this approach, instance documents will not validate until they are changed to designate the new targetNamespace. However, one does not want to force all instance documents to change, even if the change to the schema is really minor and would not impact an instance.
- Any schemas that 'include' this schema would have to change because the target namespace of the included components must be the same as the target namespace of the including schema.

3.4. 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).

DISADVANTAGES:

- As with previous approach, this approach forces all instance documents to change, even if the change to the schema would not impact that instance.
- Any schemas that import the modified schema would have to change since the import statement provides the name and location of the imported schema.
- Unlike the previous options, with this approach an application receives no hint that the meaning of various element/attribute names has changed.
- 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. Therefore, relying on this attribute is not a good practice (with the current reading of the specification).

4. Conclusions

We can list the following *best practices* taking into account the advantages and disadvantages of each approach:

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

- 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.

Other XML vocabularies, like SBML[7], are already using these best practices in developing and implementation new versions of their taxonomies.

References

- [1] eXtensible Markup Language (XML) <http://www.w3.org/TR/REC-xml>
- [2] eXtensible Business Reporting Language <http://www.xbrl.org>
- [3] COmmon solvency ratio REPorting <http://www.corep.info>
- [4] FINancial REPorting <http://www.finrep.info>
- [5] XML Schema Versioning <http://www.xfront.com/Versioning.pdf>
- [6] XML Schemas: Best Practices – Versioning <http://www.stylusstudio.com/xmldev/200109/post10080.html>
- [7] Systems Biology Markup Language (SBML) <http://www.sbml.org>
- [8] Guide to Versioning XML Languages using XML Schema 1.1 <http://www.w3.org/TR/xmlschema-guide2versioning/>
- [9] XML Schema Versioning Use Cases <http://www.w3.org/XML/2005/xsd-versioning-use-cases/>
- [10] Framework for discussion of versioning <http://www.w3.org/XML/2004/02/xsdv.html>
- [11] An Approach for Evolving XML Vocabularies Using XML Schema <http://lists.w3.org/Archives/Public/www-tag/2004Aug/att-0010/NRMVersioningProposal.html>
- [12] Extending and Versioning Languages Part 1 <http://www.w3.org/2001/tag/doc/versioning.html>
- [13] Providing Compatible Schema Evolution <http://www.pacificspirit.com/Authoring/Compatibility/ProvidingCompatibleSchemaEvolution.html>