Position Paper: Modularity, Namespaces and Versioning

Author: (see Change History section)

Date: 2-26-2003

Filename: draft-burcham-modnamver-07.doc

1 Change History

Revision	Editor	Description
0.7	Bill Burcham	Fleshed-out Recommendations:Versioning. Finalized namespace URI's throughout.
0.6	Dave Carlson	Changes made to section "Options: Module Structure" to enumerate several options for module definition.
0.1-0.5	Bill Burcham	Baseline

2 Summary

There are many possible mappings of XML schema constructs to namespaces and to operating system files. This paper explores some of those alternatives and sets forth some rules governing that mapping in UBL.

3 Problem Description

Namespaces are a syntactic convenience supporting the association of a "context" with either a lexical scope (default namespace), or a shorthand identifier (namespace qualifier). This context, applied either implicitly (in a lexical scope) or explicitly (via qualified names) supports compression of what would otherwise be long identifiers. In the absence of namespaces, identifier names are all long.

It is common for an instance document to carry namespace declarations, so that it might be validated. Processing logic (such as a stylesheet) typically carries namespace declarations pertaining to the language in which it is specified in (XSLT) as well as the namespaces on which it *operates*. The latter must match namespaces in the instance document under translation in order for useful work to be carried out.

In practice, namespaces are often given names denoting a hierarchy. XML processing tools may or may not use this hierarchy information. This sort of hierarchical naming though can be useful for the human reader.

As with other significant software artifacts, schemas can become large. In addition to the logical taming of complexity that namespaces provide, we might like to also divide the physical realization of that schema into multiple operating system files.

Schemas change over time. UBL will be no exception. What sort of version information (if any) will a schema carry? How shall that information be carried so as to conveniently support the needs of users operating on document instances with XML processing tools.

This position paper will address these three topics related to namespaces:

1. **Namespace Structure**: What shall be the mapping between namespaces and XML Schema constructs (e.g. type definitions)?
2. **Module Structure**: What shall be the mapping between namespaces and XML Schema constructs and operating system files?

3. **Versioning**: What support for versioning of schema shall be provided?

In subsequent sections, we'll examine each topic in turn, presenting first the options, then a recommendation.

4 Assumptions

Much of this discussion will be based on the expected complexity of the UBL vocabulary. We structure systems into components in order to manage complexity.

4.1 Problem Size

How big will UBL be? How interconnected?

One source for complexity estimation is xCBL. TBD: how many type definitions, element declarations, "instance roots" in xCBL?

Another source for estimation is X12 that according to [NDR-MSG-88] has:

> a bit over 1,000 data elements (…) a smaller number of segments, and
> 300 or so transaction sets

Also from [NDR-MSG-88] we have EDIFACT:

- There are just under 650 data elements which are
- used in approx 200 composite structures (sort of equivalent to low level Aggregate Core Components (ACCs)).
- These elements and composites are reused within just over 150 segment structures (sort of equivalent to higher level ACCs).
- Combinations of all the above make up just under 200 messages (doc types).

So an estimate of 1000 types and 250 message types seems reasonable for UBL.

4.2 Optimal Component Size

We don't want to define 1000 types all in one XML namespace, nor would we want to define them all in one file. Such an approach would lack structure necessary for understanding both by maintainer and users. Additionally, performance would be far from optimal for instance documents that needed only a subset of the UBL types.

For these reasons we presume that we need to structure and divide UBL into a hierarchy of components. We will strive to balance coupling and cohesion between the components in order to:

- Manage the complexity of each component while not creating too many components[1]

[1] The "seven plus or minus two" rule [SEVEN-TWO] is a good, general rule of thumb. It's especially useful when you don't have any other rule. It says that if you want people to be able to keep a set of concepts in mind, then you are limited to about seven concepts.

- Provide for useful subsetting of components

We envision that many useful instance documents (messages) will be possible that require only a fraction of the overall UBL schema. In those cases it should be possible to avoid processing of the unneeded parts.

5 Options: XML Namespace Identification

This section presents some options for the form that UBL namespace names might take.

5.1 Option 1: Namespace Name = Namespace Location

There is certainly precedent for this approach. See for example the ebXML Message Service schema `http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2_0.xsd`.

5.2 Option 2: Namespace Name is OASIS URN namespace

This option exemplifies the current best practice within OASIS. See RFC 3121 [OASIS-URN-NS] for details. See Namespaces in XML for background [NAMESPACE].

Under this option, the namespace names for UBL namespaces would have the following form while the schemas are at draft status:

```
urn:oasis:names:tc:ubl:schema{:subtype}?:{document-id}
```

When they move to specification status the form will change to:

```
urn:oasis:names:specification:ubl:schema{:subtype}?:{document-id}
```

Where the form of {document-id} is TBD but should match the schema module name (see section 7, Recommendation: Schema Location).

6 Recommendation: Namespace Identification

We pick Option 2: *Namespace Name is OASIS URN namespace.*

Will document-id include versioning information or will versioning be handled outside this identifier? See section 13, Recommendations: Versioning.

7 Recommendation: Schema Location

A question related to Namespace identification is schemaLocation. Schema location includes the complete URI which is used to identify schema modules.

Implications for XML for example might be: a type would define no more than seven (or so) elements, a namespace would define no more than about seven types, etc.

In the fashion of other OASIS specifications, UBL schema modules will be located under the UBL committee directory:

http://www.oasis-open.org/committees/ubl/schema/<*schema-mod-name*>.xsd

TBD does this recommendation need more justification?

Where <schema-mod-name> is the name of the schema module file. The form of that name is TBD.

There are two issues here. One is: how do we tell *users* to reach our schemas and two: what do we use internally – URN's or URL's. One is where/how do we publish our schemas.

8 Options: Namespace Structure

In this section we'll explore some mappings between XML Schema structures and namespaces.

8.1 Option 1: One Big Namespace

We could have one big namespace for UBL. On the plus side, it would be fairly easy to remember. The downside is that we would forfeit the opportunity to use hierarchical namespaces to communicate the structure of the vocabulary.

8.2 Option 2: One Namespace Per Type

This approach represents the other end of the spectrum. If you've got a namespace per type then why not just use the type name. The namespace fails to be shorthand for anything. It fails to be memorable, or to group related types together.

8.3 Option 3: Core Plus "Functional" Namespaces

This option represents a space between 8.1 and 8.2. There would be namespaces for "core" types and there would be namespaces for each of the functional areas e.g. Order, Invoice.

Purpose	**Namespace name**
Common Leaf Types	urn:oasis:names:tc:ubl:schema:CommonLeafTypes:*major-version*:*minor-version*
Common Aggregate Types	urn:oasis:names:tc:ubl:schema:CommonAggregateTypes:*major-version*:*minor-version*
Order Domain	urn:oasis:names:tc:ubl:schema:Order:*major-version*:*minor-version*
Invoice Domain	urn:oasis:names:tc:ubl:schema:Invoice:*major-version*:*minor-version*
TBD	TBD

This represents a top-level decomposition of the vocabulary into multiple vertical (functional) slices: Order, Invoice; and two (horizontal) slices – the so-called core, CommonLeafTypes and CommonAggregateTypes.

The downside of this approach is that with seven or so functional namespaces, they are going to get awfully "crowded" (on the order of one hundred types per namespace).

8.4 Option 4: Core Plus "Functional" Namespaces Plus Internal Structure as Needed

A refinement on 8.3, this option frees each of the functional and core namespaces to have their own hierarchy as necessary in order to further manage complexity.

Add explanation from the minutes here.

9 Recommendation: Namespace Structure

	Pro	Con
Option 1: one big namespace	Easy to remember namespace	When anything in UBL changes, all processing code must be changed (at a minimum to use new namespace name)
Option 2: namespace per type	Total compartmentalization	Why use namespaces at all? With this option the namespace ceases to provide useful contextualization.
Option 3: core plus "functional" namespaces	Allows parts of UBL to change independently. When a functional area changes, processing code depending on core needn't change.	Doesn't allow for intermediate structure. What if the functional namespaces may require further subdivision?
Option 4: core plus "functional" namespaces plus internal structure as needed	(same as Option 3)	By allowing intermediate namespaces, they will certainly flourish. Design rules must be developed to avoid regressing toward Option 2 over time.

Option 3 is recommended. We reserve the right to revisit this decision when we are further along in the process of defining types. If we find that we need more structure, we can move to option 4.

Option 4 is recommended now!

9.1 Into What Namespace Do Extensions Go

Extensions (by users) go into user-defined namespaces outside of UBL.

10 Options: Module Structure

This section describes options for decomposing schema definitions into modules, where modules are typically represented as operating system files. For XML Schemas, each file contains one schema document instance. A more general definition of “module” is as follows:

Definition: A Module is a <xsd:schema> document instance. In the UBL deliverable, each module is written to one operating system file. But in database storage (either RDBMS or XML native), a module would be recognized as an XML document instance.

The following options for module decomposition have been identified:

10.1 Option 1: One Module Per Namespace

This is the option used in the 0p70 UBL deliverable. It is the simplest rule to apply and works reasonably well for the size and scope of the 0p70 deliverable. However, it may not scale to a more mature library of several hundred reusable type definitions. The scalability concern is not due to technical issues, but due to difficulty of human users working with one very large file. Tool support will help to mitigate this problem, but even then some kind of logical modularity would be useful.

10.2 Option 2: One Module Per Object Class

This option would gather together all of the qualified variations of BIEs for each object class, as implemented by schema type definitions and their associated global elements. So BuyerParty, SellerParty, and so on would appear in one module.

A master schema must include all modules for a given namespace. Users of a namespace library would not import the individual modules, but only the master schema.

The primary motivation for this rule is to provide an easily automated decomposition strategy that does not require human intervention when generating schemas from a model or component repository.

A downside of this option is that the type definitions in a module do not include any definitions for closely related content element definitions.

10.3 Option 3: Modules based on Human Judgment of Related Functionality of Type Definitions

This option would gather together related type definitions based on functional similarity. For example, HazardousItem and its related child element content definitions would be collected in one module.

This might require substantial human analysis to determine the best decomposition of a namespace into modules. In particular, when leaf schema types (e.g. CountryType) are used by several modules, those shared types cannot be duplicated in functional modules.

10.4 Option 4: One Module per Type Definition

This is essentially the rule used for creating xCBL modules. Use of the schema files is only practical when they are opened in a schema design tool. A user would open the master schema, which must include several hundred small schema files.

11 Recommendation: Module Structure

This section describes the mapping of namespaces (as discussed in section 8 *Options: Namespace Structure*) onto XSD files. A namespace contains type definitions and element declarations. Any file containing type definitions and element declarations is called a SchemaModule.

Every namespace has a special SchemaModule, a RootSchema. Other namespaces dependent upon type definitions or element declaration defined in that namespace import the RootSchema and only the RootSchema.

If a namespace is small enough then it can be completely specified within the RootSchema. For larger namespaces, more SchemaModules may be defined – call these InternalModules. The RootSchema for that namespace then include those InternalModules.

This structure provides encapsulation of namespace implementations. To recap:

Import Rule: A namespace “A” dependent upon type definitions or element declaration defined in another namespace “B” imports B’s RootSchema. “A” never imports other (internal) schema modules of “B”.

Include Rule: The only place XSD “include” is used is within a RootSchema. When a namespace gets large, its type definitions and element declarations may be split into multiple SchemaModules (called InternalModules) and included by the RootSchema for that namespace.

The import rule presents a namespace as an indivisible grouping of types. A “piece” of a namespace can never be used without all it’s pieces. It is therefore important to strive to define namespaces that are minimal and orthogonal.

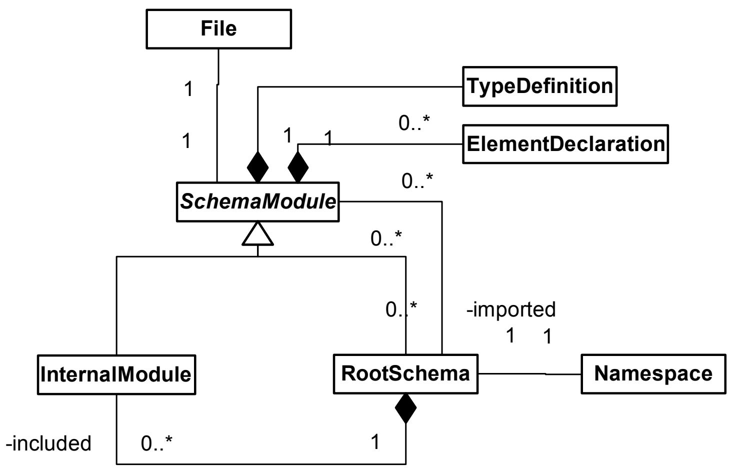
Spin out minimal and orthogonal a bit more.

It is not enough that a namespace be minimal in terms of its intrinsic size, but also in terms of the closure of all other namespaces it imports. By closure we mean namespaces it imports, and namespaces they import, and so on.

One good way to foster minimal namespaces is to dictate that there be no circular dependencies between them. The same statement can be made for SchemaModules. This rule has been applied successfully in many large systems[2].

(No) Circular Dependency Rule: There are no circular dependencies between SchemaModules. By extension, there are no circular dependencies between namespaces. This rule is not limited to *direct* dependencies – transitive dependencies must be taken into account.

Here is a depiction of the component structure we've described so far. This is a UML Static Structure Diagram. It uses classes and associations to depict the various concepts we've been discussing:



You can see that there are two kinds of schema module: RootSchema and "InternalModule". A RootSchema may have zero or more InternalModules that it includes. Any SchemaModule, be it a RootSchema or an InternalModule may import other RootSchemas.

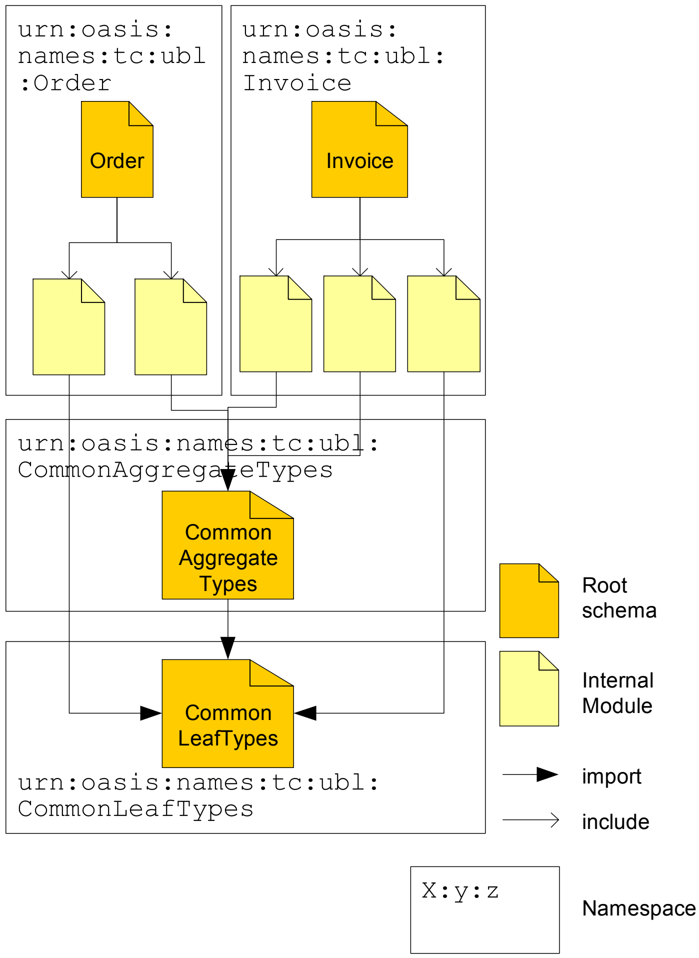
The diagram shows the 1-1 correspondence between RootSchemas and namespaces. It also shows the 1-1 correspondence between files and SchemaModules. A SchemaModule consists of type definitions and element declarations.

The diagram unfortunately fails to express the (No) Circular Dependency Rule.

Another way to visualize the structure is by example. The following informal diagram depicts instances of the various classes from the previous diagram.

---

[2] For example [LARGE-SCALE] introduces the concept of "levelization" as an organizing principal for very large C++ systems. Those systems, due to the nature of the language often have an explosion of type definitions (due to the presence of parameterized types). As a result, solutions to the management of type systems in C++ could be viewed as exemplary for our purposes.



The preceeding diagram shows how the order and invoice RootSchemas import the "CommonAggregateTypes" and "CommonLeaf Types" RootSchemas. It also shows how e.g. the order RootSchema includes various InternalModules – modules local to that namespace. The clear boxes show how the various SchemaModules are grouped into namespaces.

11.1 Message Types

If preferring type definitions over global element definitions is good, why not take it to the extreme [NDR-MSG-70]. **The type of the root element of a UBL document (message) is a global type** (not an anonymous type).

11.2 Number of Message Types

In some cases, various actions in the protocol (create vs. delete) will have totally different document structure requirements. But in some cases (create vs. update), the content might be identical. However, we still think we should design in favor of more document types rather than less, e.g. one for each transmission (a la RosettaNet). It avoids confusion on the part of developers to have a separate document type for each thing. We might then decide to optimize some of them by merging them together.

12 Options: Versioning

[XFRNT-VER] does a great job of laying out the problem and solution space for schema versioning as it is traditionally practiced. The options presented in that document are not really disjoint rather they are building blocks. If you look at the recommendations in that document, you will see that the options are used in concert.

12.1 Option XF-1: Change the (internal) schema "version" attribute

12.2 Option XF-2: Create a "schemaVersion" attribute on the root element

12.2.1 Usage A: Conformance enforced by validator

12.2.2 Usage B: Conformance enforced by an extra processing pass

12.3 Option XF-3: Change the schema's target namespace

12.4 Option XF-4: Change the name/location of the schema

12.5 Option 5: Schema Version as Context Classifier

In [NDR-MSG-13] the point was made that schema version might just be another context classifier.

13 Recommendations: Versioning

The following table summarizes the tradeoffs between the options.

	Pro	Con
Option XF-1: Change the (internal) schema "version" attribute		Not enforced by validator
Option XF-2-A: Create a "schemaVersion" attribute on the root element -- Conformance enforced by validator		Conformance requires exact match on version string
Option XF-2-B: Create a "schemaVersion" attribute		Extra processing step.

on the root element -- Conformance enforced by an extra processing pass		
Option XF-3: Change the schema's target namespace	Schema validation ensures that an instance conforms to its declared schema. There are never two (different) schemas with the same namespace URI.	With this approach, instance documents will not validate until they are changed to designate the new targetNamepsace. 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. +Include problems.
Option XF-4: Change the name/location of the schema		Ugh!
Option 5: Schema Version as Context Classifier	Leverages the context machinery	Requires the context machinery

We will use Option XF3 as a starting point for UBL. A UBL namespace URI is divided into two parts, one that describes the **purpose** of the namespace and another that captures **version** information.

The version information will in turn be divided into *major* and *minor* fields. For example, the namespace URI for the Invoice domain has this form:

```
urn:oasis:names:tc:ubl:schema:Invoice:major-version:minor-version
```

The *major-version* field is "1" for the first release of a namespace. Subsequent major releases increment the value by 1. For example, the first namespace URI for the first major release of the Invoice domain has the form:

```
urn:oasis:names:tc:ubl:schema:Invoice:1:0
```

The second major release will have a URI of the form:

```
urn:oasis:names:tc:ubl:schema:Invoice:2:0
```

The distinguished value "0" (zero) is used in the *minor-version* position when defining a new major version. In general, the namespace URI for every major release of the Invoice domain has the form:

```
urn:oasis:names:tc:ubl:schema:Invoice:major-number:0
```

Subsequent minor releases begin with *minor-version* 1. For example, the namespace URI for the first minor release of the Invoice domain has this form:

```
urn:oasis:names:tc:ubl:schema:Invoice:major-number:1
```

In UBL, the major-version field of a namespace URI 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. Regardless, at a minimum any change to any schema module constituting the namespace necessitates some change to the namespace URI. Said another way, **once a namespace URI is published by UBL it must never change.**

This approach yields non-obvious, yet beneficial effects when the interdependencies of namespaces are considered. UBL is composed of a number of interdependent namespaces. For instance, namespaces whose URI's start with `urn:oasis:names:tc:ubl:schema:Invoice:*` are dependent upon the common leaf and aggregate namespaces, whose URI's have the form `urn:oasis:names:tc:ubl:schema:CommonLeafTypes:*` and `urn:oasis:names:tc:ubl:schema:CommonAggregateTypes:*` respectively. If either of the common namespaces changes 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 if the importing schema changes then its namespace URI in turn must change. The outcome is twofold:

- There is never ambiguity at the point of reference. A dependent schema imports precisely the version of the namespace that is needed. The dependent never needs to account for the possibility that the imported namespace can change.

- When a dependent is upgraded to import a new version of a schema the dependent's version (in its namespace URI) must change.

The question now arises: what is meant by "major" versus "minor". What kind of change may a minor version introduce? When is it necessary to incur a new major version? Why are the answers to these questions even interesting?

To answer these questions you must start by understanding that UBL's use of major and minor version number borrows from a long tradition software tradition. In that tradition, a minor version declared it's "compatibility" with previous minor versions (of the same major version).

Since this sort of versioning scheme was applied to libraries, applications and even whole operating systems, the definition of the term "compatibility" in those various contexts necessarily varied widely.

Its historical use in shared libraries probably comes closest to the intended UBL use. A new release of a library (namespace) must specify a new major version number if it breaks compatibility with the previous version of the library (namespace). In the case of object libraries examples of breaking compatibility were 1) calling interface changed or 2) behavior (semantics) of interface changed.

Implicit in this major/minor scheme is that there is some benefit in breaking the version information into two pieces. The benefit in the traditional shared library paradigm is that objects dependent upon those shared libraries could still function properly with subsequent minor releases. Those minor releases might add new functionality of repair defects – but they wouldn't break the "contract" identified by the major version number.

The level of formal specification of this "contract" has varied in historical practice ranging from informal and undocumented to human-readable interface specification. UBL leverages XML schema itself as the means to capture this contract. Here's how it works…

A minor revision to a major release (of a namespace) *imports* the schema module for the major release. For instance, the schema module defining:

`urn:oasis:names:tc:ubl:schema:Invoice:1:2`

Must import the namespace:

`urn:oasis:names:tc:ubl:schema:Invoice:1:1`

The 1:2 revision may define new complex types by extending or restricting 1:1 ones. It may define brand new complex types and elements by composition. It must not use the XSD redefine element to change the definition of a type or element in the 1:1 version.

The opportunity exists in the 1:2 version to rename derived types. For instance if 1:1 defines Address and 1:2 specializes Address it would be possible to give the derived Address a new name, e.g. NewAddress. This is not required since namespace qualification suffices to distinguish the two distinct types. **The minor revision may give a derived type a new name only if the derived type represents a semantic distinct from that of the base type.**

For a particular namespace, the minor versions of a major version form a linearly-linked family. Each successive minor version imports the schema module of the preceding minor version. The process is bootstrapped by the first minor version importing the namespace defining the major version of interest. E.g.
`urn:oasis:names:tc:ubl:schema:Invoice:1:2` imports
`urn:oasis:names:tc:ubl:schema:Invoice:1:1` which imports
`urn:oasis:names:tc:ubl:schema:Invoice:1:0`.

The outcome of this usage of XSD import is that schema validation enforces these constraints:

1. forward compatibility of instances: an instance document valid in version M:m will be valid in any version M:m+n.

2. backward compatibility of reused components: an instance document that is valid in version M:m may contain constructs defined in M:m-n. Processing logic implemented in terms of version M:m-n will process those constructs properly since those constructs are valid with respect to version M:m-n.

3. backward incompatibility of new constructs: new constructs defined in version M:m will not be valid in M:m-n therefore processing logic would not be expected to operate on them.

4. potential backward compatibility of extended constructs: Extensions (of complex types) defined in M:m are *valid* in M:m-n however, processing logic implemented in terms of M:m-n will not be aware of extension elements. Care must be taken in the construction of processing logic to maximize the potential for compatible extension. In particular, processing logic that copies element content should do so in such a way that extension elements will be copied too.

When the changes to a namespace are such that it doesn't fit conveniently into this scheme (of importing the previous minor version namespace), a new major version is created. A new major version does not import previous version namespaces, nor does it make any representation as to compatibility with old versions. The purpose of the major version is to free the UBL designers to make significant, incompatible changes to the library.

It bears stating explicitly again that UBL is composed of a number of interdependent namespaces. It is not a single monolithic component. While it is expected that UBL *releases* will be assigned version identifiers of some sort e.g. UBL 1, UBL 2, this should not be confused with the versioning of the UBL namespaces discussed in this section. It would be perfectly reasonable, for example, for a release called "UBL version 2" to contain namespaces with URI's whose major version is not 2. Namespace versioning as described here is a fine-grained, technical mechanism for declaring and enforcing compatibility between interdependent namespaces over long periods of time (years).

14 Definitions

Backward compatibility – TBD.

BIE – Business Information Entity. A description of a business concept. Represented as an XML schema by a *root schema.*

extension a.k.a. customization – specification of new BIE's with well-defined, enforced relationships to old BIE's. Relationship types include: restriction, extension. In some cases processing logic will need to treat the base and the extension as the same, in other cases it will need to distinguish between them.

Forward compatibility – TBD

Namespace – a name that scopes a related group of XML type definitions.

processing logic – software logic that operates on BIE instances to achieve some business function

root schema – A *schema module* that directly, or via inclusion of other schema modules, defines all types for a particular namespace. This is the XML Schema representation of a BIE. (Compare that definition, with the one we came up with last week in Menlo Park: *A schema document corresponding to a single namespace, which is likely to pull in (by including or importing) schema modules*. **Issue**: Should a root schema always pull in the "meat" of the definitions for that namespace, regardless of how small it is?)

schema document – as defined by the XSD specification – per that specification, a schema document defines types into exactly one namespace, the target namespace.

schema module – A *schema document*. A schema module need not define all types in a particular namespace. Contrast with *root schema*. (Compare that definition, with last week's: *A "schema document" (as defined by the XSD spec) that is intended to be taken in combination with other such schema documents to be used.*)

versioning – reification of revisions to BIE's in order to support coexistence in a system, of two or more revisions of a BIE.

15 References

LARGE-SCALE	*Large Scale C++ Software Design,*	

	John Lakos, 1996, Addison-Wesley.	
NAMESPACE	*Namespaces in XML*	http://www.w3.org/TR/REC-xml-names/
NDR-MSG-13	*schema version as context classifier*, Burcham, Bill; Maler, Eve; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00013.html
NDR-MSG-70	*Fwd: Straw Man on Namespaces, Schema Module Architecture,etc.*, Rawlins, Mike; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00070.html
NDR-MSG-88	*Fwd: Straw Man on Namespaces,Schema Module Architecture, etc.*, Probert, Sue; Maler, Eve.; a post to the UBL-NDR mailing list.	http://lists.oasis-open.org/archives/ubl-ndrsc/200111/msg00088.html
OASIS-URN-NS	IETF RFC 3121 *A URN Namespace for OASIS*	http://www.faqs.org/rfcs/rfc3121.html
SCHEMA-PRIM	*XML Schema Part 0: Primer*	http://www.w3.org/TR/xmlschema-0/
SEVEN-TWO	*The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information*, George A. Miller, Psychological Review, 63, 81-97.	http://psychclassics.yorku.ca/Miller/
XFRNT-VER	*XML Schema Versioning,* MITRE Corporation and xml-dev list group members.	http://www.xfront.com/Versioning.pdf
XML-URI-LIST	*XML-URI List* at w3.org	http://lists.w3.org/Archives/Public/xml-uri/