UBL 2.0 customizations, extensions, versions, validation and interchange

G. Ken Holman, Crane Softwrights Ltd.
<gkholman@CraneSoftwrights.com>

Copyright © 2007 Crane Softwrights Ltd.

Status

This is a personal contribution by G. Ken Holman of Crane Softwrights Ltd. towards the development strategies for UBL 2 definition (UBL Compatibility and UBL Conformance), deployment (UBL Customization) and minor revisions (UBL 2.x), and has no official status and does not (necessarily) represent the views of the UBL technical committee nor any other of its members.

This paper presents concrete proposals of the processes that would be needed for a UBL customization definition and deployment. Considerations are presented for application deployment, for interoperability, for schema validation, for value validation and for UBL customization. If you wanted to skip the preamble, the final summary of the deployment recommendations is found in Section 8, “Customization deployment recommendations”, but it helps to know more about the artefacts described by reading the rest of the document.

This paper also presents a concrete proposal for minor versioning, that is, a methodology for the specification of incremental revisions to UBL. For example, how would UBL 2.1 constructs be defined in a UBL 2 environment, and UBL 2.2 constructs in a UBL 2.1 environment, etc. This is related to customization and conformance in the nature of the deployed applications acting on UBL instances, thus the discussion is appropriate when considering application issues for customization. If you wanted to skip the preamble, the minor versioning approach is described in Section 9, “Minor-versioning proposal”.

Feedback is eagerly sought from all readers regarding any incorrect assumptions or perceived technical inaccuracies in order to improve the methodologies that are proposed. The philosophical drawbacks or attractions of these methodologies is also welcome to be discussed. Please feel free to propose alternative approaches, but please defend your suggestions with demonstrations or reasoned arguments.

Not all of the development is completed, but I think I have a clear idea of what I think is needed (whether I can convey that or not is up to you). I can publish this for comment while I think about the development of the processes that remain to be created.

. . . . . Ken

Table 1. Summary of changes

Version	Changes
0.5	Address the distinction between "model-level compatibility" and "document-level conformance". Refine the filtering concept to three types of filter: namespace, global name and context filters. Relate the concept of "history of a document model" as described by one reviewer. Remove the "Assumptions" section as this was considered a distraction from the main arguments. Replace the use of "subset" with "customization". Assimilate the concepts of the impact section in the other sections and remove the impact section entirely
0.4	Remove discussion and proposal of minor-version namespace usage; add new section to summarize impact of translate-before-validate decision. Add "subset schema valid" to Section 6.4, “Customization UBL conformance”. Recommend in Section 8.2.2, “Application handling of an arbitrary UBL instance input” not to use full UBL validation in order for an implementation to be tolerant of minor-version information item additions to instance from future minor versions of the UBL schemas. Add definition of "UBL Customization" to Section 6, “UBL Customizations” and Section 2, “Definitions”.
0.3	Remove item-level extensibility considerations; elaborate on subset definition of code lists and business rules; elaborate on trading partner definition of constructs
0.2	Add example of minor-version namespace usage; clarify thoughts of "blind interchange" as a system design issue and not as a business issue
0.1	Original

Version 0.5 $Date: 2007/04/19 19:45:45 $(UTC)

Table of Contents

1. Introduction 3
2. Definitions 3
3. Considerations for UBL Compatibility and UBL Conformance 5
    3.1. UBL Compatibility 5
    3.2. UBL Conformance 5
4. XPath files 6
    4.1. XPath file vocabulary 7
    4.2. XPath reports 9
    4.3. Using XPath files for tests of presence and containment 11
5. Limitations of XSD 11
    5.1. Unable to declare derivatives of the extension point 12
    5.2. Unable to elide optional elements through derivation 12
    5.3. Unable to express different enumeration restrictions based on context 12
    5.4. Unable to express co-occurrence constraints 12
    5.5. Unable to maintain UBL conventions using XSD extension 13
6. UBL Customizations 13
    6.1. Customization definition and artefacts 13
    6.2. The customization fence 13
    6.3. The choice of XSD for schema expression 14
    6.4. Customization UBL conformance 15
    6.5. Context/value validation 15
7. Transform-before-validate process 17
    7.1. Namespace-based transform-before-validate 17
    7.2. Name-based transform-before-validate 17

7.3. Context-based transform-before-validate 17
8. Customization deployment recommendations 20
8.1. Customization artefact preparation 20
8.2. Application handling of customization instance input 22
8.3. Application handling of customization instance output 25
9. Minor-versioning proposal 26
9.1. The problem with a pure XSD approach to minor versioning 26
9.2. Translate-before-validate supports incremental network migration 26
9.3. Minor-versioning mechanics 27
9.4. Vocabulary history 28
9.5. W3C Technical Architecture Group (TAG) document 28
9.6. Conclusion 29
Bibliography 29

1. Introduction

Many requirements for versioning and customization have been expressed by members in meetings and in minutes and in some private correspondence and discussions. In this contribution I will summarize some of the topics in question regarding deployment, state the requirements as I understand them, expound on personal perspectives of the contradictions and offer some candidate considerations for bypassing the issues, as well as thoughts of the ramifications of certain decisions that may be considered expedient.

I heartily welcome refuting arguments accompanied by demonstrative examples of contrary positions that will bolster the ability to meet the user requirements. My objective is to bring these topics up for discussion with demonstrations of experimental implementation so as to inform the committee members considering how to build on UBL 2.0.

2. Definitions

exhaustive confirmation
- the process by which XPath files are used to exhaustively check all information items described by a given XML vocabulary as being described in a second XML vocabulary without missing any required information items described in the second XML vocabulary (see Section 4.3, “Using XPath files for tests of presence and containment” for details)

extension
- an information item that is not defined by UBL 2.0 and is not a candidate for a revision, say UBL 2.1, but is needed by users of a customization

information items
- actual elements and attributes in an XML document
- the available elements and attributes of an XML vocabulary as described by schemata

minor version or revision
- a subsequent release of UBL 2, say UBL 2.1, in which some of the information items have been augmented to have new optional information items where parent elements of those items are modified to have new constraints allowing the new optional information items

namespace URI string
- a Uniform Resource Identifier [URI] string used as a portion of the unique label for an XML information item; the complete unique label is the combination of the namespace URI string and the information item's local name (element type or attribute name)

NDR
- Naming and Design Rules dictate the synthesis of schema expressions from a collection of information item descriptions (however expressed); it hap-

	pens that UBL uses a spreadsheet to collect the information item descriptions, and commercial software translates these into XSD schemata following the UBL NDR
NVDL - Namespace-based Validation Dispatching Language	• ISO/IEC 19757-4 NVDL [NVDL] • an expression of the dispatching of fragments of an XML instance to different validation processes based on the namespace URI string portion of the labels of information items found in the instance
Schematron	• ISO/IEC 19757-3 Schematron [Schematron] • an assertion-based expression of constraints placed on the information items of an XML instance
serendipitous exchange	• an authorized transaction between trading partners where the common core of UBL constructs conveys sufficient information for a successful result, perhaps requiring out-of-band intervention but without requiring an application to change or the instance to have certain optional information items expected by either application (the serendipity being that the two trading partners can successfully do business without having to change either's applications that are already creating and consuming standardized UBL as a core of the interchanged messages regardless of their respective customizations)
UBL Customization	• a definition of XML instances and XML-based applications based on those instances that are somehow conformant or compatible with the standardized UBL 2.0 specification; this document sets out to describe the different ways in which UBL customizations can be described

validation	• the act of confirming the information items of an XML instance meet a set of expressed constraints
	  • structural validation confirms the nesting of elements and attached attributes (thus information is correctly found)
	  • lexical validation confirms the character-by-character structure of an item (thus information is correctly formed)
	  • value validation confirms the value expressed in an items (thus information is correctly understood)

XPath	• W3C XPath [XPath 1.0] is a data model and an addressing syntax for information items in an XML document
XPath file	• an enumeration of absolute XPath location paths for all of the information items described by an XSD schema expression following the UBL NDR • an enumeration of absolute XPath location paths for information items found in a given XML instance
XSD	• W3C XML Schema [XSD] (filename extension and common abbreviation) • an object-oriented type-based expression of constraints placed on the information items of an XML instance

XSLT	• Extensible Stylesheet Language Transformations [XSLT 1.0] is used to construct a structured information output from XML inputs

3. Considerations for UBL Compatibility and UBL Conformance

I think most everyone on the committee understands a "UBL Customization" is, in the generic sense, the description of XML instances or XML-based applications acting on those instances that are somehow based on or derived from the UBL 2.0 specification.

During the UBL TC face-to-face in Singapore November 2006 it became apparent there are two very distinct interpretations of how to describe and deploy a customization of UBL in different environments. A number of members focus on the business objects described by the UBL information model based on the CCTS information model. Other members focus on the XML instance and XSD schema constructs described by the UBL document model based on the CCTS schema model.

For the purposes of this paper, I propose and distinguish the terms *UBL Compatibility* on the information model and *UBL Conformance* on the document model. For the sections following the definitions contrasted in this section I will be focusing only on the document model and the interchange of XML instances based on XSD schemas for UBL conformance according to these distinctions and definitions.

3.1. UBL Compatibility

The Core Component Technical Specification [CCTS] presents a model for standardizing business semantics and a methodology for developing a common set of semantic building blocks that represent business data. Using these building blocks one can create one's own XML vocabulary with user-named constructs based on CCTS constructs with the standardized business semantics.

The UBL business objects are built on top of the CCTS building blocks and, as such, the UBL vocabulary is compatible at an information model level with other XML vocabularies and data representations also based on CCTS. The actual XML document-model-level vocabularies may be very different, but the semantic basis on which these vocabularies are built promotes compatibility of the information expressed using those semantics.

These UBL business objects can be, in turn, the basis upon which other information models and associated XML vocabularies are built, which promotes compatibility of the information expressed in those vocabularies with the information expressed in UBL instances. Such compatibility is at the application and information model level, however, and does not enforce or predict any document-level instance compatibility in the use of labeled XML information items as defined by UBL.

I propose that an instance or a system is said to be *UBL-Compatible* when its information item definitions (not labels) are based on the same UBL and CCTS business objects, thus promoting interoperability at an application and model level between two UBL-compatible systems that understand the underlying business objects.

3.2. UBL Conformance

An XML vocabulary defines a set of labels for element and attribute information items arranged in a tree hierarchy. Systems that are developed to interchange XML documents can validate the correct use of labels according to the labels defined by an XSD schema, and then populate internal programming structures with the information interchanged under each information item label. Through the use of XML Namespaces the labels are a combination of a local name and a namespace URI string thus ensuring the labels for a given vocabulary are not ambiguous with labels for another vocabulary.

I propose that an instance or a system is said to be *UBL-conformant* when at the document-model level the information item labels (and, by the document models, their constituent structure definitions) represent the same labels and structures as those published in UBL for information items (the "UBL Name" local name and UBL namespace URI strings). This promotes interoperability at a document interchange level between two UBL-conformant systems that interpret the labeled business objects.

There are two kinds of document-model UBL conformance to consider: instance conformance and system conformance, with two kinds of system conformance to consider.

System conformance is trickier than instance conformance, and it is difficult to decide where to draw the line between different levels of system conformance. I distinguish the adjectives *UBL-conformant* from *UBL-open* for systems, primarily indicating the support for the serendipitous exchange of UBL instances.

These characterization of conformance are focused on the receiving system. I'm assuming the UBL instances the sending part of the system produces have enough information in the UBL-standardized information items that the sending system accepts that a receiving system can process the documents even if the receiving system chooses to ignore any extensions or optional constructs that might be present in the UBL instance as defined by the sending system. Of course it would be incumbent on the receiver to archive all messages as received in case there are any disputes over the complete information sent. An archived unprocessed message can always be reprocessed, while an archived processed message won't provide the ability to reprocess the message as originally received.

3.2.1. UBL-conformant instances

I think instance conformance is straightforward: an XML instance is considered *UBL-conformant* if there are no document-level constraint violations of the labeled structures (hierarchy) and lexical structures (expressed content) when validating the instance against the published UBL XSD schemata element structures and data types.

The UBL document structures accommodate the presence of foreign-labeled constructs underneath an extension point. This is an element available in all UBL documents under which there are no constraints described by UBL, other than the child element not be in the UBL XML vocabulary. That child's descendent content can have any constraints defined by a user community and is ignored by the validation against the standardized UBL document models.

3.2.2. UBL-conformant systems

I think that a system should be considered *UBL-conformant* if (1) it produces UBL-conformant instances, and (2) it accepts UBL-conformant instances even if its business rules choose to reject a particular UBL-conformant instance if some optional UBL information required by the system is absent, some extension element required by the system is absent, or any other system-related business rules are violated.

3.2.3. UBL-open systems

I think that a system should be considered *UBL-open* if it supports serendipitous exchange, that is (1) it produces UBL-conformant instances, and (2) the UBL instances the receiving part of the system accepts can be processed without extensions or optional constructs the receiving system supports, perhaps violating receiving-system business rules.

Note that such successful receipt may require some out-of-band authorization or intervention when the receiving system finds missing extensions or constructs important to established business rules, but with such intervention the transaction can still successfully complete without needing the sender to change the information being sent or retransmit the information.

4. XPath files

The XPath recommendation [XPath 1.0] defines a data model for the information found XML instances. This model is agnostic to any constraints that may have been imposed on the creation or use of the instance. XPath 1.0 is based

only on XML syntax and XML Namespaces. The data model describes well-formed instances (which may or may not be valid). The model focuses on the information found in the instance and not in any way the syntax used in the instance to express the information.

The UBL Human Interface Subcommittee [HISC] project has created an XML vocabulary for enumerating information items in a catalogue of available XPath addresses from the document element to all items allowed by a given document model described by a schema or to all items found a particular XML instance. The normative instance of an XPath file for a given document model is an XML instance of the XPath file vocabulary [XPath File]. This instance can be machine-processed by any XML-aware application and can also be used to create human legible reports and diagnostic materials.

XPath files for UBL 2 schemata are publicly available [UBL-XPath]. There happens to be sufficient information in a UBL schema expression to derive the complete suite of information items. The combination of UBL NDRs happen to make it straightforward to create XPath files from the published XSD expressions. For example, the use of the "Garden of Eden" (all elements and types defined globally) approach to declarations, and only sequence groups (no choice groups), makes XPath files unambiguous for UBL, whereas XPath files might be quite insufficient for other document models and modeling conventions.

More research is required to come up with more information in the XPath normative files to accommodate different schema expression conventions and NDRs. Initial discussions with vendors indicates that XPath files are not sufficiently rich to express XML document structures for arbitrary constraints, only those constraints limited by the UBL NDRs.

Note that XPath files need not be generated from XSD schemas or XML instances. The UBL spreadsheets used to determine the contents of the XSD schemas (or any spreadsheets describing content nesting and definition) can be used as a source for creating XPath files. However created, the XPath files express in a programmatically processed form all of the possible combinations of XML hierarchy for the information items described by a document model, schema or instance.

4.1. XPath file vocabulary

The document element of an XPath file is `<XPath>`. Document-wide namespace prefix/URI associations are expressed in `<Namespace>` elements. XML element information structure is expressed in an XPath file as `<Element>` element children of parent `<Element>` elements. Similarly, attributes of elements are expressed as `<Attribute>` element children.

An XPath file can be processed by an application to internalize all of the structures expressed so that the application can compare structures or do other processing with the information. The following is an excerpt from the beginning of the `AttachedDocument` document type:

```
<XPath xmlns="urn:oasis:names:tc:ubl:schema:XPath-1.0"
       id="urn:oasis:names:tc:ubl:XPath:AttachedDocument-2.0">
   <Namespace prefix="ad" uri="urn:...:AttachedDocument-2"/>
   <Namespace prefix="cac" uri="urn:..:CommonAggregateComponents-2"/>
   <Namespace prefix="cbc" uri="urn:..:CommonBasicComponents-2"/>
   ...
   <Element name="AttachedDocument" type="AttachedDocumentType"
            prefix="ad" minOccurs="1" maxOccurs="1">
      <Element name="UBLExtensions" type="UBLExtensionsType"
               prefix="ext" minOccurs="0" maxOccurs="1">
         <Element name="UBLExtension" type="UBLExtensionType"
                  prefix="ext" minOccurs="1" maxOccurs="unbounded">
            <Element name="ID" type="IDType"
                     extends="udt:IdentifierType" prefix="cbc"
                     minOccurs="0" maxOccurs="1" text="">
```

```
            <Attribute name="schemeAgencyID" use="optional"
                       type="xsd:normalizedString"/>
            <Attribute name="schemeAgencyName" use="optional"
                       type="xsd:string"/>
            ...
         </Element>
         <Element name="Name" type="NameType"
                  extends="udt:NameType" prefix="cbc"
                  minOccurs="0" maxOccurs="1" text="">
            <Attribute name="languageID" use="optional"
                       type="xsd:language"/>
         </Element>
         <Element name="ExtensionAgencyID"
                  type="ExtensionAgencyIDType"
                  extends="udt:IdentifierType" prefix="ext"
                  minOccurs="0" maxOccurs="1" text="">
            <Attribute name="schemeAgencyID" use="optional"
                       type="xsd:normalizedString"/>
            ...
         </Element>
         ...
      </Element>
   </Element>
   <Element name="UBLVersionID" type="UBLVersionIDType"
            extends="udt:IdentifierType" prefix="cbc"
            minOccurs="0" maxOccurs="1" text="">
      <Attribute name="schemeAgencyID" use="optional"
                 type="xsd:normalizedString"/>
      <Attribute name="schemeAgencyName" use="optional"
                 type="xsd:string"/>
      ...
   </Element>
   ...
   <Element name="CustomizationID" type="CustomizationIDType"
            extends="udt:IdentifierType" prefix="cbc"
            minOccurs="0" maxOccurs="1" text="">
   ...
   </Element>
   <Element name="ProfileID" type="ProfileIDType"
            extends="udt:IdentifierType" prefix="cbc"
            minOccurs="0" maxOccurs="1" text="">
      <Attribute name="schemeAgencyID" use="optional"
                 type="xsd:normalizedString"/>
      ...
   </Element>
   ...
</Element>
</XPath>
```

4.2. XPath reports

Two XPath reports are made available: a simple text report of absolute XPath addresses (that is, an XPath address that begins from the root node and document element) and a mockup XML instance, both of which have one of every information item described by an XPath file.

4.2.1. XPath text reports

An XPath text report is targeted to a human reader and can be a handy tool to overview the information found in an XPath file. Each XPath address is preceded by a reference ordinal (one ordinal for elements and a pair of ordinals for attributes), and the cardinality of the information item. An excerpt of the XPath text report of the above `AttachedDocument` XPath file is as follows (lines are wrapped to fit on this page, there is no line wrapping in the file):

```
1   1..1 /ad:AttachedDocument/
2   0..1 /ad:AttachedDocument/ext:UBLExtensions/
3   1..n /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
4   0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                            cbc:ID
4.1 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                              cbc:ID/@schemeAgencyID
4.2 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                            cbc:ID/@schemeAgencyName
4.3 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                               cbc:ID/@schemeDataURI
4.4 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                  cbc:ID/@schemeID
4.5 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                 cbc:ID/@schemeName
4.6 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                  cbc:ID/@schemeURI
4.7 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                             cbc:ID/@schemeVersionID
5   0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                          cbc:Name
5.1 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                cbc:Name/@languageID
6   0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                               ext:ExtensionAgencyID
6.1 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                 ext:ExtensionAgencyID/@schemeAgencyID
6.2 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                               ext:ExtensionAgencyID/@schemeAgencyName
6.3 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                  ext:ExtensionAgencyID/@schemeDataURI
6.4 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                     ext:ExtensionAgencyID/@schemeID
6.5 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                    ext:ExtensionAgencyID/@schemeName
6.6 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                     ext:ExtensionAgencyID/@schemeURI
6.7 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                ext:ExtensionAgencyID/@schemeVersionID
7   0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
```

```
                                                ext:ExtensionAgencyName
7.1 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                  ext:ExtensionAgencyName/@languageID
8   0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                                 ext:ExtensionVersionID
8.1 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                ext:ExtensionVersionID/@schemeAgencyID
8.2 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                              ext:ExtensionVersionID/@schemeAgencyName
8.3 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                 ext:ExtensionVersionID/@schemeDataURI
8.4 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                    ext:ExtensionVersionID/@schemeID
8.5 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                  ext:ExtensionVersionID/@schemeName
8.6 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                                   ext:ExtensionVersionID/@schemeURI
8.7 0..1 /ad:AttachedDocument/ext:UBLExtensions/ext:UBLExtension/
                             ext:ExtensionVersionID/@schemeVersionID

...
```

4.2.2. XPath instance report

The mockup XML instance created from an XPath file is called an XPath instance report and can be used to compare structures of XML documents containing the information items described by an XPath file. An excerpt of the XPath instance report of the above `AttachedDocument` XPath file is as follows; note how the corresponding reference ordinals are captured in the information items between exclamation marks, and because of this the instance cannot be validated by the UBL schemata (nevertheless the reference ordinals are valuable diagnostic tools in non-schema-aware systems such as XSLT 1.0 stylesheets):

```
<ad:AttachedDocument
  xmlns:ad="urn:...:AttachedDocument-2"
  xmlns:cac="urn:...:CommonAggregateComponents-2"
  xmlns:cbc="urn:...:CommonBasicComponents-2"
  ...
   <ext:UBLExtensions>
      <ext:UBLExtension>
         <cbc:ID schemeAgencyID="!4.1!" schemeAgencyName="!4.2!"
schemeDataURI="!4.3!" schemeID="!4.4!" schemeName="!4.5!"
schemeURI="!4.6!" schemeVersionID="!4.7!">!4!</cbc:ID>
         <cbc:Name languageID="!5.1!">!5!</cbc:Name>
         <ext:ExtensionAgencyID schemeAgencyID="!6.1!"
schemeAgencyName="!6.2!" schemeDataURI="!6.3!" schemeID="!6.4!"
schemeName="!6.5!" schemeURI="!6.6!" schemeVersionID="!6.7!"
>!6!</ext:ExtensionAgencyID>
         <ext:ExtensionAgencyName languageID="!7.1!"
>!7!</ext:ExtensionAgencyName>
         <ext:ExtensionVersionID schemeAgencyID="!8.1!"
schemeAgencyName="!8.2!" schemeDataURI="!8.3!" schemeID="!8.4!"
schemeName="!8.5!" schemeURI="!8.6!" schemeVersionID="!8.7!"
>!8!</ext:ExtensionVersionID>
         <ext:ExtensionAgencyURI schemeAgencyID="!9.1!"
schemeAgencyName="!9.2!" schemeDataURI="!9.3!" schemeID="!9.4!"
```

```
schemeName="!9.5!" schemeURI="!9.6!" schemeVersionID="!9.7!"
>!9!</ext:ExtensionAgencyURI>
         <ext:ExtensionURI schemeAgencyID="!10.1!"
schemeAgencyName="!10.2!" schemeDataURI="!10.3!" schemeID="!10.4!"
schemeName="!10.5!" schemeURI="!10.6!" schemeVersionID="!10.7!"
>!10!</ext:ExtensionURI>
         <ext:ExtensionReasonCode languageID="!11.1!" l
istAgencyID="!11.2!" listAgencyName="!11.3!" listID="!11.4!"
listName="!11.5!" listSchemeURI="!11.6!" listURI="!11.7!"
listVersionID="!11.8!" name="!11.9!">!11!</ext:ExtensionReasonCode>
         <ext:ExtensionReason languageID="!12.1!"
>!12!</ext:ExtensionReason>
         <ext:ExtensionContent>
         </ext:ExtensionContent>
      </ext:UBLExtension>
   </ext:UBLExtensions>
   <cbc:UBLVersionID schemeAgencyID="!14.1!"
schemeAgencyName="!14.2!" schemeDataURI="!14.3!"
schemeID="!14.4!" schemeName="!14.5!" schemeURI="!14.6!"
schemeVersionID="!14.7!">!14!</cbc:UBLVersionID>
   ...
```

4.3. Using XPath files for tests of presence and containment

Any system that is being built for UBL can utilize the XPath files to check for the presence of a given information item described by some other means. For example, a stylesheet environment created for the Danish Information Technology Ministry uses hand-edited annotations that are meant to express the XPath nesting of information items. These annotations can be programmatically checked for accuracy by using the normative XPath XML files to trace the hierarchy of information items specified.

So an exhaustive proof of conformance and compatibility can be implemented using XPath files. I'm not familiar with any other scheme that employs exhaustive enumeration as a proof of the whole containment of one XML vocabulary within another XML vocabulary.

Using this proof, then, the containment of an edited or synthesized XPath file can be rigourously tested as being a proper subset of full UBL by programmatically checking that all of the XPath file entries of the smaller are found in the larger file, and that none of the mandatory items in the larger file are missing from the smaller. This programmatic checking allows the smaller XPath file to be created by any means, not just editing the larger file, since its resulting containment can be confirmed.

This was accomplished in the Danish project: the putative completeness of the proposed spreadsheets expressing the nesting of selected UBL constructs (even in different contexts) was measured by determining that all spreadsheet combinations of structure were found in the UBL 2 XPath file nested structures. This task revealed repairs to be made in the spreadsheets until no discrepancies were found. The end result is the assurance that the document structures expressed in the Danish spreadsheets were, in fact, structures found in UBL 2.0, thus they expressed a valid customization of UBL 2.0.

5. Limitations of XSD

It is widely touted that XSD derivation can be used to express restrictions of or extensions of a given XSD schema to produce a new set of constraints. Unfortunately, the modified sets of constraints anticipated to be needed by XSD users and customization preparers cannot be expressed using XSD redefinition or substitution semantics.

At this time I am unable to determine the XSD validation semantics for derivation to use to meet the business requirements listed in this section. Experts contacted through the XML-Dev and W3C Schema mail lists have been unable to provide mechanisms to satisfy these requirements through available XSD Schema semantics.

As a result, the recommendations of this methodology have been specified to support standalone XSD expressions of document constraints, provided these constraints follow the UBL NDR. Section 6.3, “The choice of XSD for schema expression” makes reference to tools with which one can create standalone expressions of document constraints, following the UBL NDR, and basing the work on published UBL concepts.

Therefore, the publication of a UBL revision or UBL customization can be safely created independent of the UBL definition on which it is based because exhaustive confirmation using XPath files can be used to confirm the independently-created artefacts.

5.1. Unable to declare derivatives of the extension point

It is not possible to express in an XSD extension or restriction of the published UBL schemata that a given extension element is allowed to be a child of the extension point. Consider the two possibilities based on the published UBL schemata defining the extension element with an `xsd:any` constraint of `##any` to allow any element of any namespace be a child of the element.

In the case of extension, a deriving schema attempts to add the definition of the customization extension element to the children of the UBL extension point (it is unclear how this is done because of derivation rules in XSD). A validating processor is obliged to first satisfy the base schema expression for the extension element before attempting to satisfy the extension constructs. But the processor will have already consumed all of the particles with the `##any` of the base schema before hitting the end of the extension children, thus when it attempts to validate the presence of the extension element, there are no particles left to be the extension element.

Similarly, in the case of restriction, a deriving schema attempts to restrict the definition of the UBL extension point to be elements of any namespace, followed by the customization extension element, followed by elements of any namespace. Again the use of `##any` directs the validating processor to consume all children of the extension point, and only when done will it then try to find an extension element which is not there.

5.2. Unable to elide optional elements through derivation

Should a customization definition wish to elide an optional element and make it totally unavailable, there is no way an XSD schema can restrict an existing content model to indicate that an optional element already declared in the base model is not included in the restricted model.

5.3. Unable to express different enumeration restrictions based on context

All elements in UBL are global thus those with enumerated data types necessarily have global scope across an entire instance. There is no way an XSD schema can restrict an existing content model to indicate that a contextual use of a data type has a different subset of enumerated values than in another contextual use.

5.4. Unable to express co-occurrence constraints

There is no way to express in an XSD schema a constraint on the existence of or the contents of information items based on the existence of or the contents of other information items.

5.5. Unable to maintain UBL conventions using XSD extension

In UBL all aggregate entities are structured with all basic entities listed first as children, followed by all associate entities listed next as children. A quick review of the NDR indicates that there is no specific rule requiring basic entities before associate entities ... have I missed one somewhere?

Nevertheless, XSD extension allows only additional constructs to be added after all of the base constructs. Should a revision to a UBL aggregate entity need a new child basic entity, this basic entity cannot be placed before child associate entities when using XSD extension.

6. UBL Customizations

A document-level UBL customization defines the use of UBL-conformant instances in a given context of a community of users. Customizations might be based on company size (as in the UBL 1 Small Business Subset SBS [SBS] already chartered as part of the UBL subcommittee), based on geographic users (as in the already chartered North European Subset NES [NES]), based on some as-yet-to-be-chartered sector basis (as in the example above for the aerospace industry use), or based on a company's private use (I plan to augment the UBL SBS with a private extension in order to support my company's legacy invoice layout).

6.1. Customization definition and artefacts

A *customization definition* defines the following four aspects of the UBL *customization instance* in three artefacts:

- the choice of predefined UBL information items of interest that are expected to be used (which must include all of the mandatory UBL information items, may contain any selection of optional items, and may choose not to even allow any selection of optional items), expressed in the customization XSD Schema artefact;
- the definition of the structure of a child element of the UBL extension point for constructs not in UBL but needed for the transaction (for example, line-item detail structures that provide more granular information about invoice line items than the standardized invoice line item), also expressed in the customization XSD Schema artefact;
- the specification of information item value rules based on context/value associations and controlled vocabularies (e.g. for code lists and identifier lists), expressed in a Schematron schema artefact as created by the UBL methodology for code list and value validation; and
- the specification of business rules about the co-occurrence constraints of the information items in the customization (for example, that the number of extension point line-item detail elements exactly matches the number of UBL-standard line item summary elements), expressed in a Schematron schema artefact.

The *customization schema* artefact is the document model of the customization instance and is expressed in an XSD schema expression following the UBL NDR. Section 6.3, “The choice of XSD for schema expression” makes reference to some tools with which one can create standalone expressions of document constraints, following the UBL NDR, and basing the work on published UBL concepts.

The *customization value rules* and *customization business rules* artefacts are Schematron schema expressions of Schematron patterns in isolation, used by trading partners and deployments as part of the post-structure-validation rules for value-validation.

6.2. The customization fence

The XPath files created from the customization schema enumerate which information items are important to the customization user community. These files inform applications and users of a kind of a "fence" in the set of information

items for full UBL. Senders of UBL customization instances can rely on everything within the fence to be meaningful to the recipient's system, but senders cannot rely on anything outside of the fence to be processed by the recipient claiming to support the customization. The corollary is that receivers of UBL customization instances need only be concerned with information items within the fence and are safe to ignore information items that lie outside of the fence.

The customization does not change the definition of UBL or UBL document-level conformance since anything added by the customization is added under the wildcard extension point, and anything removed by the customization must have been optional in UBL in the first place. This is a critical aspect of conformance and definition, and processes are being researched to exhaustively and robustly confirm, using the XPath files, that every information item expressible for a given customization is also an information item in the standard UBL.

It should be a stated guideline for customizations that any information item that most appropriately belongs in the standardized component should go in the standardized component and not in the customization extension.

6.3. The choice of XSD for schema expression

Given that the problems are centred around the extension and restriction, then the bespoke XSD expression of the customization schema can define precisely the information that is important to the application to receive. This schema expression also conveniently informs the instance creation process. And many creation tools and programming interfaces are keyed on the use of XSD Schema, so the bespoke customization schema would be instantly usable by all these tools.

Off hand I can think of two ways one would create a bespoke UBL XSD customization schema:

- edit an existing UBL schema expression or UBL customization schema expression either by hand or by a program (Crane Softwrights Ltd. makes "Simplified UBL schema customization" software available [Crane Resources] for this task); and
- edit a model representation of what you want in the schema and use a tool to translate the model into a schema expression; two available tools that do this for UBL 1.0 and are anticipated to have versions available for the use with UBL 2.0 are:
  - GEFEG.FX [GEFEG.FX] by GEFEG mbH in Germany;
  - UBLer [UBLer] by Invinet Systems in Spain; and
  - UBLish [UBLish] (UBL inter-schema helper) by SoftML in Singapore.

But if the bespoke schema cannot be programmatically derived from the UBL schemata due to XSD schema semantics, then how is one to be assured there are no mistakes or incompatibilities? Provided the bespoke schema is written following the UBL NDRs then I think the answer is through the XPath files. As discussed in Section 4.3, “Using XPath files for tests of presence and containment”, one can programmatically determine that all of the information items described by the bespoke instance are found in the correct order and acceptable cardinality in the XPath files created from the original UBL schemata.

This gets us around the derivation limitations of XSD and gives us a pure-XSD set of schemata and a certifiable method of programmatically confirming the expressiveness of this bespoke schema is acceptable to UBL systems.

But in a deployment, how would such a pure-XSD-based system accommodate a "full" UBL instance that has ignorable elements not of interest to the customization? The instance as it arrives at the system can be massaged by a transformation that removes the information items not desirable to the customization, and then the customization-only instance can be sent to the program interfaces that are fashioned around the pure-XSD expressions. This is summarized in Section 8, “Customization deployment recommendations”. The transformation itself can be programmatically derived from the XPath files.

Given that a community of users is supported by the definition of a customization, the organization responsible for defining the customization could be responsible for creating reference and programmatically-certified implementations

of all of the support files needed to work with the customization. The pure-XSD expressions, the associated XPath files, the business rules of the customization, the transformation implementations (stylesheets, programs, etc.), and the evidence of programmatic verification of the processes could all be published and made available. The community of users can then just consistently use the customization artefacts in their implementation.

6.4. Customization UBL conformance

Section 3.2, “UBL Conformance” has some important implications for systems specified to implement customizations of UBL that may have extensions, in that it mandates the following in addition to being simply customization-schema valid:

- a customization instance must be UBL-conformant;
- I suspect that most customization systems will typically be specified to be UBL-conformant (though this is optional if one is willing to forego certain checks) and will enforce the presence of customization-required information items under the extension point and business rules that are either optional in full UBL or apply to under the UBL extension point; and
- I hope that most customization systems are specified to be UBL-open by making the extensions optional and keeping all UBL optional information items optional in the customization, thus the system will only take advantage of any optional constructs and extensions when present, but these systems would not reject a UBL-conformant instance, and would do the best it could with the core UBL constructs found. This may not be palatable to a community of users, but to support the serendipity factor, it should be considered.

A customization schema cannot be used for validation directly on received UBL instances, even if the instances pass UBL schema muster. This is because there could very well be optional information items defined by UBL that are present in the received instance but are not of interest to a customization user, so it would not be in the customization schema. This necessitates a *transform-before-validate* process (see Section 7, “Transform-before-validate process”) to elide all information items not enumerated by the customization schema. Such a transformation task can be automatically synthesized from the information in XPath files and then applied to incoming instances..

There are no predefined co-occurrence constraints (also termed above as "business rules") in using the normative UBL schemata. The use of a customization schema, however, will typically have business rules to ensure particular values are being used or particular associations can be made between the core content of UBL and any extensions. If a system first accepts an instance when it validates against the customization schema, but then decides based on the its business rules that it cannot accept the instance, there should be a distinction returned to the sender of the instance.

Perhaps the difference is that a system that receives a document that doesn't pass customization validation muster rejects the instance altogether, while a received document that does validate but doesn't meet business rules triggers the application to return an application response document indicating that the document content is unacceptable.

This would distinguish a return of "unable to be processed" from "unacceptable to be processed". The customization designer can change the kind of response returned in the an absence of a particular information item from "unable to be processed" into "unacceptable to be processed" by changing the type of validation of the information item from being a schema-based validation structure requirement into a business-rule-based validation assertion requirement.

6.5. Context/value validation

A methodology [UMCLVV] has been proposed for the validation of information items constrained by coded values such as in code lists and identifier lists. This methodology incorporates lists of files of associations between information items and files describing the values allowed for the information items.

Customization preparers may wish to define their own value lists as a basis for a community of users. They would take the default or other existing UBL code list files, make their required modifications with default values needed for the

community, and then make these artefacts available to their users to then customize with their own requirements in trading-partner agreements.

This discussion paper does not go into detail regarding value list validation. Version 0.8 of the methodology has been published based on feedback from readers and users of version 0.4. From version 0.6 the value list rules are packaged as standalone Schematron pattern schemas that are included into an including full Schematron schema. With the customization business rules and the customization value list rules packaged themselves as Schematron pattern schemas, the full Schematron schema can now include all possible sets of constraints into a single post-XSD-schema-validation pass with a single Schematron-generated XSLT value validation pass.

6.5.1. Code list context report

While not directly related to customization, note that those defining customizations might be interested in a third report, created from the normative XPath XML files, summarizes all of the contexts of code-list-based information items. There are many reports created for UBL 2.0 [UBL Code List Support]: one for each document type and one for an aggregate of all document types. A context report enumerates all of the unique minimal relative XPath addresses of the information items based on code list data types:

```
AttachedDocument: (20 uses of code list data types;
                   20 unique parents; 463 unique contexts)

cbc:AddressFormatCodeType: (1 unique parent; 36 contexts;
                            extends udt:CodeType)
  cbc:AddressFormatCode (36 contexts)
    cac:ReceiverParty/cac:AgentParty/cac:PartyLegalEntity/
                      cac:CorporateRegistrationScheme/
                      cac:JurisdictionRegionAddress/
                      cbc:AddressFormatCode
    cac:ReceiverParty/cac:AgentParty/cac:PartyLegalEntity/
                      cac:RegistrationAddress/cbc:AddressFormatCode
    ...
    cac:SignatoryParty/cac:PostalAddress/cbc:AddressFormatCode

cbc:AddressTypeCodeType: (1 unique parent; 36 contexts;
                          extends udt:CodeType)
  cbc:AddressTypeCode (36 contexts)
    cac:ReceiverParty/cac:AgentParty/cac:PartyLegalEntity/
                      cac:CorporateRegistrationScheme/
                      cac:JurisdictionRegionAddress/
                      cbc:AddressTypeCode
    ...
    cac:SignatoryParty/cac:PostalAddress/cbc:AddressTypeCode

cbc:ChannelCodeType: (1 unique parent; 6 contexts;
                      extends qdt:ChannelCodeType)
  cbc:ChannelCode (6 contexts)
    cac:ReceiverParty/cac:AgentParty/cac:Contact/
                      cac:OtherCommunication/cbc:ChannelCode
    cac:ReceiverParty/cac:Contact/cac:OtherCommunication/
                      cbc:ChannelCode
    cac:SenderParty/cac:AgentParty/cac:Contact/
                    cac:OtherCommunication/cbc:ChannelCode
    cac:SenderParty/cac:Contact/cac:OtherCommunication/
                    cbc:ChannelCode
```

```
    cac:SignatoryParty/cac:AgentParty/cac:Contact/
                        cac:OtherCommunication/cbc:ChannelCode
    cac:SignatoryParty/cac:Contact/cac:OtherCommunication/
                        cbc:ChannelCode
...
```

7. Transform-before-validate process

The essence in Section 6.4, “Customization UBL conformance” of transforming an incoming instance removing unexpected constructs for a given customization before validating and acting on the instance can be implemented in three different fashions based on the nature of the customization.

Termed in this documentation (and in the diagrams) as a "Customization Filter", this is a process that massages XML creating from the input XML document a new XML document with only expected constructs.

7.1. Namespace-based transform-before-validate

A customization that only adds extension constructs and does not remove any standardized constructs can implement transform-before-validate by using a namespace-based filter. UBL allows multiple extensions with multiple namespaces under the UBL extension point, yet a customization schema and customization-aware application is expecting only to validate and act on extension constructs in its known namespace.

This type of customization filter reads in the incoming XML instance and only allows constructs with the expected namespaces through the filter to then be processed by the customization schema and customization-aware application.

Such a filter is an easy application of the SAX interface or an XSLT stylesheet with which the input instance is massaged into an instance expected by downstream processes.

I'm also of the opinion I can borrow from NVDL [NVDL] to specify the filter process required, but this needs some experimentation.

7.2. Name-based transform-before-validate

A customization that adds extension constructs and also removes standardized optional constructs can implement transform-before-validate by using a name-based filter. UBL allows multiple extensions with multiple namespaces under the UBL extension point, and a valid UBL instance could easily have those optional constructs not used by a customization community, yet a customization schema and customization-aware application is expecting only to validate and act on known extension constructs and expected UBL constructs.

UBL elements are globally defined, thus the namespace-qualified name of a UBL element and its attributes can be used to determine if a given input construct is or is not in a given customization.

This type of customization filter reads in the incoming XML instance and only allows constructs with the expected namespace-qualified element names and attributes through the filter to then be processed by the customization schema and customization-aware application.

Such a filter is an easy application of the SAX interface or an XSLT stylesheet with which the input instance is massaged into an instance expected by downstream processes.

7.3. Context-based transform-before-validate

There are customizations (e.g. North European Subset [NES]) that have contextual differences in element definitions. This structural validation is two-step in that the published UBL schemas are used for overall validation and then a

profile-specific Schematron schema reports the presence of unexpected constructs in order to flag a structural error of unexpected element definition. A filter must, therefore, take an arbitrary instance and remove the contextually-invalid constructs in order to create a pure NES customization instance for validation.

Other customizations may be creating standalone XSD schemas for describing the constraints on a UBL instance for the given customization community. Perhaps spreadsheets are used to express the contextual differences. Perhaps the schemas are created by hand. By whatever mechanism, a set of schemas is created and a filter must, therefore, take an arbitrary instance and remove any constructs not found in the schemas in order to create a pure customization instance for validation.

However created, confirming that the customization schema itself is a valid subset of the UBL schema is a problem in and of itself. A rigourous methodology must be applied to ensure the expression of each and every information item in the customization is a valid information item in the complete UBL schema.

The type of customization filter needed by these customizations reads in the incoming XML instance and only allows constructs with the expected namespace-qualified element names and attributes in expected contexts through the filter to then be processed by the customization schema and customization-aware application.

Such a filter may be quite lengthy in that it enumerates all of the possible elements and attributes in all differing contexts. Maintaining such a filter by hand may be unmanageable.

7.3.1. Measuring customization schema document-level conformance

I posit that XPath files can be used to measure a customization schema for wholly being a strict subset of the full UBL schema.

Consider Figure 1, “Customization schema artefact preparation overview” showing in overview the processes called "Exhaustive Confirmation" and "Structure Confirmation" and the artefact called "Customization Filter".

Figure 1. Customization schema artefact preparation overview

The UBL schema is compared to the customization schema in the process called "Exhaustive Confirmation", exhaustively checking all possible customization schema information items as being information items in the UBL schema.

The "Make Filter" process synthesizes the customization filter from the customization schema.

The "Test Filter" process takes in information from the UBL schema and the customization filter and produces output. The "Structure Confirmation" process examines the filtered output with information from the customization schema to ensure the filter is functional.

Shown in more detail, consider Figure 2, “Customization schema artefact preparation detail” showing these processes and artefacts being created and used. This also introduces alternative methods of describing a customization, that being by pruning the UBL XPath file into the customization XPath file under the control of some customization definition, or synthesizing the UBL XPath file arbitrarily from some customization definition.

Figure 2. Customization schema artefact preparation detail

Note that the basis for "Exhaustive Confirmation" is comparing the customization XPath file with the complete UBL XPath file.

I posit that a customization filter can be synthesized from a customization XPath file without human intervention. This filter can be successfully tested by filtering an exhaustive UBL 2.0 instance in order to produce a result that is then checked for structural equivalence with an exhaustive customization instance.

8. Customization deployment recommendations

I believe the following transform-before-validate strategy is worth end-to-end prototyping and implementation for evaluation by the TC and user communities such as the NES to see if requirements and compatibilities are achieved. It may be revealed that enhancements are required to XPath files in order to streamline the process.

8.1. Customization artefact preparation

A number of artefacts for each document type should be produced by the preparers of a customization in support of the deployment of a customization definition.

8.1.1. Customization structural/lexical constraints

Artefacts expressing the structural/lexical constraints on a document are prepared. Typically this would be a schema artefact such as an XSD expression. The NES project uses a combination of the UBL XSD and a Schematron schema for refinement to accomplish the same task.

8.1.2. Customization filter

Depending on the nature of the customization, one of three possible filters are prepared. If the customization is expressed entirely by namespaces, see Section 7.1, “Namespace-based transform-before-validate” for details regarding the necessary filter. If the customization is expressed entirely by global names, see Section 7.2, “Name-based transform-before-validate” for details regarding the necessary filter. If the customization is expressed by contextual differences of global constructs, see Section 7.3, “Context-based transform-before-validate” for details regarding the necessary filter.

8.1.3. Supplemental value-validation constraints

See Figure 3, “Customization value validation artefact preparation” for a depiction of the data flows for producing the artefacts related to the co-occurrence constraints, the value lists and the business rules of customization documents, accommodating at the same time the constraints, codes and rules that may have been agreed-upon between trading partners.

Using the same *default code lists and associations* process used for the UBL definition of value list validation artefacts from the UBL schemata , the customization preparers can prepare the same artefacts for the customization. Building on the default value list expressions, selected external value list expressions created for UBL can be used and new external value list expressions of interest to the customization preparers can be used.

The customization preparers will probably also create a set of *customization business rules* that check co-occurrence constraints and possibly other business requirements that need to be specified. For example, consider the situation where the customization defines detailed line-item-level extension information that correlates to summary line-item-level information in the UBL-defined constructs. Such a correlation might be based on position, thus requiring the co-occurrence constraint that the count of detailed line-item extension elements to be exactly the same as the summary line-item standard elements. Alternatively, if there is a referencing mechanism, then the co-occurrence constraint checks that there exists a referenced line-item summary element for each line-item detail element. Such constraints are expressed in a Schematron pattern schema, which itself is not a complete Schematron schema but can be included by a complete Schematron schema.

Note

Reviewing the NES layered Schematron structural constraints, I believe these constraints can be expressed as customization business rules and used in the UBL methodology as part of the second pass value validation. Currently the NES Schematron schemas are expressed as standalone schemas and not as patterns for use through inclusion by other Schematron schemas. I will be suggesting to NES to publish their Schematron schemas as two-part files: an including schema and an included pattern, thus the included pattern can be useful by users of the UBL methodology.

Trading partners using the customization schema may have their own business rules, their own context associations for value lists and their own coded values for value lists. These would be expressed using the value validation methodology, each being a Schematron pattern schema.

The trading partner deploying the customization then uses the stylesheets of the value validation methodology of creating a *validation shell* Schematron schema to include the customization business rules, the customization value lists, the trading partner business rules and the trading partner value list rules creating a single complete Schematron schema expressing the entire set of value check, co-occurrence constraint and business *validation rules*.

This Schematron result is translated into an XSLT *trading partner and customization value lists and business rules* stylesheet that can be used to check an instance for all of the various checks and balances. This same XSLT file is referred to more succinctly in some of the diagrams in this paper as the *context/value and business rules* stylesheet.

Figure 3. Customization value validation artefact preparation

8.2. Application handling of customization instance input

A trading partner deploying an implementation of a customization UBL system needs to decide if the system is going to be hardwired only to pure customization instances, or is going to accommodate all UBL instances and act only on the pure customization portion of an arbitrary UBL instance. Deciding which to use will be based on the ability to engage the transform-before-validation process described in Section 6.4, “Customization UBL conformance”.

This decision determines if the deployed application is going to be able to handle only a pure customization instance or if it will also be able to handle an arbitrary UBL instance on input.

8.2.1. Application handling of a pure customization instance input

See Figure 4, “Customization handling of pure customization instance input” for a depiction of the possible data flows for a system handling an incoming pure customization instance. Such a system is the simplest deployment of UBL, but it is not tolerant of any instance of UBL that has any foreign content not prescribed by the customization schema.

Figure 4. Customization handling of pure customization instance input

When an incoming customization instance arrives at a receiving system, it is first archived for auditing, exception handling and dispute resolution purposes. Next it should be validated against the customization schema so as to ensure the input is suitable for the application code to handle. Any validation problem rejects the instance as "Unable to process".

At this point the information items are known to be correctly positioned. Next, the instance is checked against the trading partner's value lists and business rules XSLT stylesheet checking for violations of the value, co-occurrence and business rule constraints prepared in Section 8.1.3, “Supplemental value-validation constraints”. Trading partners will make changes when they have their own codes, their own code constraints, and their additional business rules. Any problem rejects the instance as "Unacceptable to process".

At this point the information items are known to be correctly positioned and that no business rules have been violated, thus unburdening such checks from the application. The application itself is cognizant of the customization schema, possibly even incorporating pre-generated code synthesized from a programmatic analysis of the schema. The application handles the content and acts on the semantics of the information in the customization instance.

Considering Section 3.2.2, “UBL-conformant systems”, such a system would neither be UBL-open, nor even UBL-conformant, because one is not able to process an arbitrary UBL instance. Even with wildcard last siblings and wildcard extension points, a customization schema can have UBL optional elements removed from the middle of siblings. A full UBL instance, then, that has one of these removed elements would trigger a validation error.

8.2.2. Application handling of an arbitrary UBL instance input

See Figure 5, “Customization handling of arbitrary UBL instance input” for a depiction of the data flows for handling an incoming arbitrary UBL instance.

Figure 5. Customization handling of arbitrary UBL instance input

When an incoming complete UBL instance arrives at a receiving system, it is first archived for auditing, exception handling and dispute resolution purposes.

Next it is processed by the customization filter based on the nature of the filtering needed: either namespace-based, name-based or context-based (see Section 7, “Transform-before-validate process”). This outputs a customization instance from the complete UBL instance.

At this point, the flow is the same as for a pure customization instance because by now the instance has been so purified. The instance should be validated against the customization schema so as to ensure the input is suitable for the application code to handle. Any customization schema validation problem rejects the instance as "Unable to process".

At this point the information items are known to be correctly positioned. Next, the instance is checked for violations of the value list value, co-occurrence and business rule constraints prepared in Section 8.1.3, “Supplemental value-validation constraints”. Trading partners will make changes when they have their own values, their own value constraints, and their additional business rules. Any problem rejects the instance as "Unacceptable to process".

At this point the information items are known to be correctly positioned and that no business rules have been violated, thus unburdening such checks from the application. The application itself is cognizant of the customization schema, possibly even incorporating pre-generated code synthesized from a programmatic analysis of the schema. The application handles the content and acts on the semantics of the information in the customization instance.

Considering Section 3.2.2, “UBL-conformant systems”, such a system would be UBL-conformant, allowing any instance of UBL to be sent to the application and the application receives the massaged set of constructs it is expecting. In addition any programmed business rules that are violated would cause an otherwise valid instance to be rejected.

Considering Section 3.2.3, “UBL-open systems”, such a system would be UBL-open if, after allowing any instance of UBL to be sent to the application, the application accepts the massaged set of constructs even when some of the business rules are violated. While out-of-band intervention may be required to allow the transaction to continue, there is sufficient base information in the instance not to require the sending party to send new information. In this way the business rules do not impede the progress of a instance with a trading partner that either is not yet in a position to satisfy the rules or is in a temporary situation where doing so is not as important as concluding the transaction.

8.3. Application handling of customization instance output

See Figure 6, “Customization handling of output” for a depiction of the data flows for these processes.

The application is informed by the customization schema, possibly even incorporating generated code synthesized from a programmatic analysis of the customization schema. Instances produced this way should be valid but they could be validated to be sure, either by the customization schema or the full UBL schema, along with value validation by the context/value and business rules.

Figure 6. Customization handling of output

9. Minor-versioning proposal

The concept of translate-before-validate opens up an opportunity for smooth migration of incremental revisions to the UBL 2 document model in minor versions such as 2.1, 2.2, ... 2.x.

As a prelude I should note that I am a big fan of XML Namespaces and have embraced them from the beginning. I see XML simply as organizing information in a labeled hierarchy and I see namespace URI strings simply as a method of globally and uniquely specifying those labels to avoid ambiguity in XML structures. Nothing more. That they are also proving useful for resource discovery is outside of the mandate of XML namespaces.

With this perspective in mind, I am aware that over the years namespaces have been used to identify whole XML vocabularies, and I'm familiar with the OASIS Namespace URI versioning policy [OASIS Namespaces]. This policy states that namespace URI strings of newly-introduced optional (thus backward-compatible) global constructs are not different than the namespace URI strings of the vocabularies being updated.

However, in this proposal I am introducing what I first wrote in version 0.1 of this document and presented at the UBL Montréal face-to-face in mid-2006 as a new approach to deploying the use of namespace URI strings in revisions to an XML vocabulary being used across a large network of deployed XML systems written to recognize XML documents of a particular vocabulary. This approach will not mesh with the established OASIS policy. But I think it warrants consideration as an available approach for technical committee such as UBL to use, because we anticipate that there will be large networks of systems supporting UBL that will need a viable migration strategy supporting minor versions..

If the UBL TC also feels this approach warrants consideration, we will probably need dispensation from OASIS to use it. I encourage you to consider the features of this approach and the problems it solves. It is important for you to shoot holes in this approach if you can so that we can develop an ironclad version to present to OASIS for our use in minor-versioning.

9.1. The problem with a pure XSD approach to minor versioning

Consider a deployment of thousands of UBL 2.0 systems and tens of thousands of UBL 2.0 users creating instances for these systems. The UBL TC issues a new 2.1 with only optional features being added. OASIS policy states that adding new 2.1 constructs would utilize and not change the namespace URI of the 2.0 vocabulary.

However, when a user creates a 2.1 instance with a new 2.1 construct, they are only going to be able to do business with systems that accept 2.1 instances. Systems that have not yet migrated to the new schemas, but are willing to tolerate the absence of optional information, cannot accept the 2.1 instance because in a pure XSD validation approach the system rejects the 2.1 instance for not having validated with their built-in 2.0 schemas.

There is no opportunity for incremental migration across a network where systems can over time be upgraded to new 2.1 instance support while still supporting the 2.0 subset of any 2.1 instances that may be communicated.

9.2. Translate-before-validate supports incremental network migration

Consider the same deployment of a network of thousands of systems where each system has implemented a customization filter as described in Section 8.2.2, “Application handling of an arbitrary UBL instance input”. The UBL TC issues a new 2.1 with only optional features being added.

Now when a user creates a 2.1 instance with a new 2.1 construct, they can send the instance to any system on the network and those systems that have implemented the 2.1 semantics and document model act on the 2.1 information items, while those systems that are still implementing 2.0 semantics act on the 2.0 information items found in the 2.1

instance. There are no schema constraint violations because the customization filter has removed any new 2.1 constructs and let through only the known 2.0 constructs as implemented by the application.

Systems can migrate on their own schedule based on their own criteria for implementing new behaviours (which may or may not be impacted by users demanding 2.1 semantics). But during the migration period they can still do business with the 2.0 semantics of the 2.1 instances that arrive. And these systems need not be implemented for even 2.2, or 2.3 or any other minor version as the customization filter lets through what it knows about and removes what it doesn't know about.

Note that a UBL instance has an element named `UBLVersionID` in which 2.1 instances would have the value "`2.1`". The customization filter would still allow the authored value to be sent to the applications. Systems based on 2.0 are in a position in the application to flag the receipt of an instance of a "higher" version for exceptional behaviour if desired.

9.3. Minor-versioning mechanics

For this incremental migration of a network of systems to work, the removal of newly-added constructs in a minor version must not disturb the labeling of the constructs in the base or previous version.

Traditional approaches to minor-versioning change the namespace URI string of all constructs or of all constructs that have a changed definition. A ripple effect of any change at any level will trigger that change's parent element to have a new definition, which requires that element's parent element to have a new definition, and so on, creating a whole new vocabulary of labels using new namespace URI strings. A 2.1 vocabulary would, therefore, have very different labels than a 2.0 vocabulary because many levels of the document structure will have changed definition, thus mandating a change in namespace URI, thus creating, at the document level, new labels for elements that are different than the labels in 2.0. For consistency, some choose to put all constructs into a new namespace if they have changed or not.

OASIS policy dictates the newly-introduced 2.1 constructs would not be in a new namespace and would have the same namespace URI string as all of the constructs in 2.0.

This paper proposes a different approach to the introduction of new constructs in a minor-version revision to a vocabulary: use a new minor-version namespace URI string only on newly-introduced constructs, without changing the namespace URI string of any preexisting construct from any prior revision.

Consider a very simplified example of the following XML instance of the ABC version 2.0 vocabulary with only one namespace:

```
<abc2:docelem xmlns:abc2="urn:x-ABC:2">
  <abc2:ABCVersionID>2.0</abc2:ABCVersionID>
  <abc2:ID>12345</abc2:ID>
  <abc2:Value>1000.00</abc2:Value>
</abc2:docelem>
```

An open ABC system could have a customization filter allowing only constructs in the ABC 2.0 namespace. Thousands of such systems could then be deployed, each filtering their incoming instances with the customization filter.

Now the ABC committee issues ABC version 2.1 with a new optional piece of information, called "`Scope`". An instance with this new element might look like:

```
<abc2:docelem xmlns:abc2="urn:x-ABC:2" xmlns:abc21="urn:x-ABC:2.1">
  <abc2:ABCVersionID>2.1</abc2:ABCVersionID>
  <abc2:ID>23456</abc2:ID>
  <abc21:Scope>All</abc21:Scope>
  <abc2:Value>2000.00</abc2:Value>
</abc2:docelem>
```

Note how the parent element `docelem` and all elements from the 2.0 definition of the ABC vocabulary have their labels still in the 2.0 namespace, and only the newly-introduced element is in the 2.1 namespace.

Systems without a customization filter will reject the 2.1 instance because it will not pass structural validation with the 2.0 schema. This validation happens before an application even begins its behaviour. The entire application process is cut short and the system rejects the instance and no transaction occurs.

However, systems with a customization filter will pass through the constructs it is expecting, and end up with a 2.0-valid subset of the 2.1 instance, but with the 2.1 version indicator intact:

```
<abc2:docelem xmlns:abc2="urn:x-ABC:2">
  <abc2:ABCVersionID>2.1</abc2:ABCVersionID>
  <abc2:ID>23456</abc2:ID>
  <abc2:Value>2000.00</abc2:Value>
</abc2:docelem>
```

Note that this would happen regardless of the kind of customization filter, be it namespace-based, name-based or context-based. The end result instance will be comprised of only those constructs that are acceptable to the applications that then act on the instances.

That some new semantic information may be missing can be recognized by inspecting the `ABCVersionID` element, but the application can still choose to act on the 2.0 subset of the information without throwing an error.

Now as more and more systems in the network embrace the 2.1 semantics, users of 2.1 instances will get those new semantics acted upon by more targets. But until that happens for any particular system that has not yet embraced the new semantics, users of 2.1 instances can still get the old semantics of their new instances acted upon by a system without the system rejecting the instance. The application still gets its chance to work on the document.

9.4. Vocabulary history

One early reviewer of this proposal (thanks, Jan!) observes that this approach "builds in history" of a vocabulary into an instance. At a glance one can tell which constructs of a given instance were introduced at a particular point in the vocabulary's history. This was of value to the reviewer because it is clear which parts of a given instance are going to be respected by which systems in a network support that level of the vocabulary, without any of the systems falling over dead because of the presence of unexpected constructs that have been introduced but not yet implemented.

9.5. W3C Technical Architecture Group (TAG) document

Writing up the details of this strategy in April 2007 I have today discovered that last month the W3C Technical Architecture Group (TAG) published its latest revision of a document on extending and versioning languages [W3C TAG Versioning], and the strategy matching my proposal is outlined in section "2.1.2 Forwards Compatible".

This summary seems to sum up the items I brought up in Montréal and in this submission I'm writing. Two selected sentences from that section of the document:

- Forwards compatible means that producers should be able to extend existing texts with new texts without consumers having to change existing implementations.
- The instance containing the extension, which isn't known by the consumer, must be transformed into an instance which is of a type known by the consumer.

Other subsections talk about alternatives such as changing the software to ignore content that is not understood. My use of transform-before-validate is done independently of the application because some applications that utilize schema expressions to synthesize data structures are not in a position to ignore data components that do not validate against the schemas or populate the data structures. In particular, W3C XSD schema validation software expressly signals the

presence of unknown constructs as validation errors. Promoting the use of outboard validation tasks relieves applications from implementing structural and lexical validation. For these reasons the use of validation software precludes the concept of ignoring content that is not understood. Only a transform-before-validation processing model will allow a system based on validation to implement forwards-compatible support for evolving XML schemas.

9.6. Conclusion

This minor-versioning approach addresses document-level conformance and the namespace-qualified labels of information items important in systems utilizing schema validation in advance of or incorporated as part of application execution. The processes in place for customization filtering effect the necessary transformation for incremental minor-version revision resilience in deployed systems.

A network of heterogeneous implementations can migrate to homogeneity incrementally without having to be in lock-step with users who are themselves migrating incrementally to support new versions of a vocabulary, but only if they implement a transform-before-validate strategy and processing model.

The W3C TAG document does not talk about the type of implementation of any particular processing model. I have identified three types of transform-before-validate customization filter suitable as such a processing model. These are based on the nature of the filtering needed: either namespace-based, name-based or context-based (see Section 7, “Transform-before-validate process”). All three support my proposal to use new namespace URI strings only for newly-introduced constructs in minor version revisions to the UBL vocabulary.

However using an alternative approach for identifying newly-introduced constructs would reduce the choice of implementation of this processing model by precluding the namespace-based filtering. Therefore, for maximum choice, I believe the UBL committee should consider asking OASIS for dispensation to implement this approach for minor version revisions of the UBL vocabulary.

Bibliography

[CCTS] *Core Components Technical Specification - Part 8 of the ebXML Framework [http://www.unece.org/cefact/ebxml/CCTS_V2-01_Final.pdf]* Version 2.01 2003-11-15

[Crane Resources] Crane Softwrights Ltd. *Free developer resources [http://www.CraneSoftwrights.com/links/resublo.htm]*

[GEFEG.FX] GEFEG, Germany *GEFEG.FX [http://www.gefeg.com/en/standard/xml/ubl.htm]*

[NES] *North European Subset [http://www.nesubl.eu]*

[NVDL] Makoto Murata *ISO/IEC 19757-4 NVDL [http://www.nvdl.org]*

[OASIS Namespaces] *OASIS Namespace URI Versioning Policy [http://docs.oasis-open.org/templates/rddl.html]*

[Python] *Python programming language [http://www.python.org]*

[HISC] *UBL Human Interface Subcommittee [http://www.oasis-open.org/committees/ubl/hisc]*

[SBS] *UBL Small Business Subset [http://docs.oasis-open.org/ubl/cs-UBL-1.0-SBS-1.0/]*

[Schematron] Rick Jelliffe *ISO/IEC 19757-3 Schematron [http://www.schematron.com]*

[UBL Code List Support] *UBL 2.0 code list support files [http://www.oasis-open.org/committees/document.php?document_id=23531]*

[UBL-XPath] *UBL 2.0 XPath files [http://docs.oasis-open.org/ubl/submissions/XPath-files/]*

[UBLer] Invinet Systems, Spain *UBLer [http://lists.oasis-open.org/archives/ubl/200612/msg00030.html]*

[UBLish] SoftML, Singapore *UBL inter-schema helper [http://www.softml.net/jedi/ubl/sw/UBLish/UBLish-1.0/index.html]*

[UMCLVV] *UBL Methodology for Code List and Value Validation (version 0.8) [http://www.oasis-open.org/committees/document.php?document_id=22591]*

[URI] *Uniform Resource Identifiers (URI): Generic Syntax [http://rfc.net/rfc2396.html]*

[W3C TAG Versioning] *Extending and Versioning Languages Part 1 [http://www.w3.org/2001/tag/doc/versioning-20070326.html]* Draft TAG Finding 26 March 2007

[XPath 1.0] James Clark, Steve DeRose *XML Path Language (XPath) Version 1.0 [http://www.w3.org/TR/1999/REC-xpath-19991116]* 1999-11-16

[XPath File] G. Ken Holman *XPath file models [http://www.oasis-open.org/committees/document.php?document_id=23525]* 2007-04-14

[XSD] Henry S. Thomson, et al.*XML Schema Part 1: Structures Second Edition [http://www.w3.org/TR/2004/PER-xmlschema-1-20040318]* 2004-03-18

[XSLT 1.0] James Clark *XSL Transformations (XSLT) Version 1.0 [http://www.w3.org/TR/1999/REC-xslt-19991116]* 1999-11-16