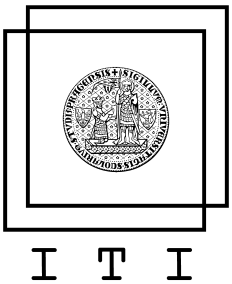
ITI Series

Institut Teoretické Informatiky
Institute for Theoretical Computer Science



2009-428

xmlprague

XML Prague 2009

Conference Proceedings

Institute for Theoretical
Computer Science (ITI)
Charles University

Malostranské náměstí 25
118 00 Praha 1
Czech Republic

http://iti.mff.cuni.cz/series/

XML Prague 2009 – Conference Proceedings

ISBN 978-80-7378-061-6

XML Prague 2009

Conference Proceedings

Lesser Town Campus
Prague, Czech Republic

March 21–22, 2009

Syntea software group, a.s.

(www.syntea.cz)

System integrator and supplier of IT solutions of large process-oriented systems based on the technologies developed by the company:

Xdefinition validation and processing of XML documents

GAM on-line batch-processes and distributed applications

XBPEL XWSDL workflow solution

AC LDAP user administration and add-on for LDAP

TRS secure WAN transport of large data files.

The company develops and uses these technologies already more then decade. Customers of the company are **financial and insurance institutions**, commercial subjects, **public sector** and operators of Internet portals in Czech and Slovak Republics.

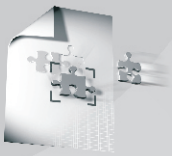
To support education of new IT experts Syntea runs two schools: the college "**Informatics Institute**" and the "**Digital Media Institute**". The professionals of the company also participate in teaching at other tertiary education institutions (Faculty of Mathematics and Physics of Charles University and Czech Agriculture University).

The company aspires to contribute in using technologies that simplify and make more precise the communication between users, analysts, architects and programmers.

Our goal is to accelerate implementation of IT solutions and increase the robustness of the applications in routine use.

<oXygen/> xml editor

<oXygen/> combines visual editing features suited for content authors with a mature and productive XML development environment.



XML Authoring and Publishing

- Ready to use support for document frameworks
  DITA, DocBook, TEI, XHTML
- One click publishing to different formats
  PDF, XHTML, Java Help, etc.
- Content Management Systems access
  WebDAV, custom plugins
- Support for reusable content
  Xinclude, Entities, DITA conref
- Comprehensive set of actions for
  Tables (HTML/CALS), Lists, Images, etc.
- Configurable and Customizable for any XML vocabulary
  Open Java API, Open source actions library

XML Development

- Schema editors
  XML Schema, DTD, RelaxNG, Schematron, NVDL
- XSLT 1.0/2.0 Basic and Schema Aware
  Debug, Profile, Edit, Transform
- Xquery
  Debug, Profile, XML Databases (MarkLogic, eXist, etc.)
- Other Supported Standards
  XPath, CSS, SVG, XSL-FO

www.oxygenxml.com

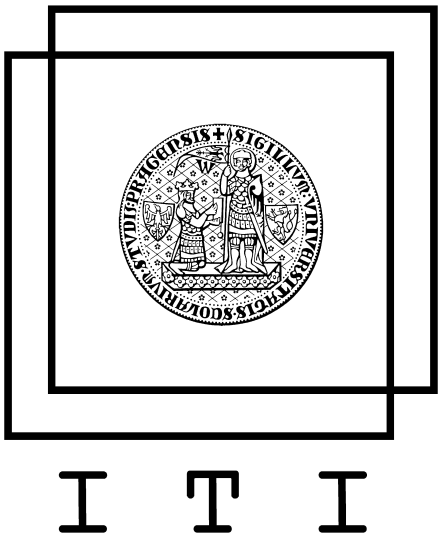
<oXygen/> is available in two editions:

- **<oXygen/> XML Author**, for the content authors, starting from 179 USD.
- **<oXygen/> XML Editor**, for developers, containing the complete development environment starting from 48 USD Academic / 299 USD Professional.

Both editions can run as a standalone application or as an Eclipse IDE plugin, on Windows, Mac OS X or Linux.

Table of Contents

Institute for Theoretical Computer Science



- Center of research in Computer Science and Discrete Mathematics funded by the Ministry of Education of the Czech Republic
- Established in 2000, current project approved for 2005–2009 with a view to extension to 2011
- Staff of 60+ researchers include both experienced and young scientists
- ITI is a joint project of the following institutions:
  - Faculty of Mathematics and Physics, Charles University, Prague
  - Faculty of Applied Sciences, University of West Bohemia, Pilsen
  - Faculty of Informatics, Masaryk University, Brno
  - Mathematical Institute, Academy of Sciences of the Czech Republic
  - Institute of Computer Science, Academy of Sciences of the Czech Republic
- For more information, see `http://iti.mff.cuni.cz`
- Publication preprints are available in ITI Series (`http://iti.mff.cuni.cz/series`)

General Information

Date

Saturday, March 21st, 2009
Sunday, March 22nd, 2009

Location

Lesser Town Campus of Charles University, Lecture Halls S5 and S6
Malostranské náměstí 25, 110 00 Prague 1, Czech Republic

Organizing Committee

Petr Cimprich, *Unity Mobile*
Tomáš Kaiser, *University of West Bohemia, Pilsen*
James Fuller, *Webcomposite*
Jirka Kosek, *xmlguru.cz & University of Economics, Prague*
Petr Pajas, *Charles University, Prague*
Václav Trojan, *Syntea software group a.s.*
Mohamed Zergaoui, *Innovimax*
Pavel Kroh
Vít Janota, *UTMG Consulting*
Martin Žák, *UTMG Consulting*

Produced By

XMLPrague.cz (http://xmlprague.cz)
Institute for Theoretical Computer Science (http://iti.mff.cuni.cz)
Webcomposite, s.r.o. (http://webcomposite.com)

Gold Sponsor

Syntea software group a.s. (http://syntea.cz)

Sponsors

oXygen (http://www.oxygenxml.com)
Florent Georges (http://fgeorges.org)
Xcruciate (http://www.xcruciate.co.uk)

Preface

This publication contains papers presented at XML Prague 2009.

XML Prague is a conference on XML for developers, markup geeks, information managers, and students. In its 4th year, XML Prague focuses on emerging trends in core XML technologies and their application in the real world. XML Prague conference will take place 21–22 March 2009 at Charles University in the beautiful city of Prague, Czech Republic.

The conference is hosted at the Lesser Town Campus of the Faculty of Mathematics and Physics, Charles University, Prague. XML Prague 2009 is jointly organized by the Institute for Theoretical Computer Science and XML Prague Committee in the framework of their cooperation supported by project 1M0545 of the Czech Ministry of Education.

The conference provides Academic and IT professionals with an overview of successful XML technologies, with the focus being more towards real world application versus theoretical exposition.

This is the fourth year we have organized this event. Information about XML Prague 2005, 2006 and 2007 was published in ITI Series 2005-254, 2006-294 and 2007-353 (see http://iti.mff.cuni.cz/series/).

— *James Fuller, XML Prague Committee*

XML Schema Moves Forward

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper gives an overview of the strengths and weaknesses of XML Schema 1.0 as background to the motivation behind the design of the new version, 1.1, which is now approaching the finish line. It provides an overview of the few features in the 1.1 version, with particular emphasis on assertions, and the impact that this particular facility will have on the way that schemas are written.

1. Introduction

XML Schema 1.0 [1] has been a W3C Recommendation since May 2001, and has been controversial ever since. It has probably received heavier criticism than any other specification in the XML family; but it has not suffered the fate of some proposed standards (XLink is an example) of being ignored by the user community. On the contrary, it is widely used, it has been implemented by all the major vendors (not to mention Saxonica), and it could be claimed that it has met all its original design objectives.

The first substantial revision of the specification, XML Schema 1.1 [2], is now close to completion. It entered its final period of public consultation in January 2009, and there is sufficient evidence of implementation (pre-release products from IBM and Saxonica) to meet W3C's criteria for transition to Recommendation status.

In the next section I will attempt an analysis of what XML Schema 1.0 does well amd what it does badly. Section 3 then gives an overview of what's new in the 1.1 version. Section 4 of the paper concentrates on what I consider to be the most significant feature of the new version, Assertions, and analyses the impact this facility is likely to have on the way schemas are used. Finally, the paper concludes with an assessment of the future.

2. XML Schema 1.0: Strengths and Weaknesses

The development of XML Schema 1.0 (or XSD 1.0 as I will refer to it) was motivated by a large number of requirements, and many of the strengths and weaknesses of the specification can be traced to the diversity of goals that it was trying to accomplish.

When XML was launched in 1998, it brought with it the DTD syntax inherited from SGML, despite widespread misgivings about its fitness for purpose. Tim Bray's *Annotated XML Specification* [3], for example, observes of its parameter entities:

> *Parameter entities (PEs) are things that show up in the DTD as references beginning with % and ending with ;. They are kind of hideous and hard to use, and especially, they are hard to read when you're looking at someone else's DTD. But for the moment, they are really the only way we have to build DTDs in a modular and maintainable way.*

DTDs made it into XML because its developers wanted to get something out quickly, and their policy for achieving this was to subset what was already available in SGML and avoid inventing anything new. The policy was remarkably successful, but it had the effect of perpetrating some pretty archaic syntax. Compatibility, as I was taught as a student by David Wheeler, means deliberately repeating other people's mistakes.

A replacement for DTDs was always on the agenda, which is probably the main reason that when namespaces were grafted on to XML in 1999, no-one worried too much that they were completely incompatible with DTDs.

Between 1998 and 2000 there were various experimental schema languages proposed: Eric van der Vlist [4] mentions XML-Data, XDR, DCD, SOX, and DDML. XML Schema 1.0 set out to supersede all of these efforts, and in order to succeed in this it had to offer the union of the capabilities of each of these proposals. (The very name XML Schema was chosen to stake a claim to be the one true schema language for XML, a claim which many found presumptious. The recognition that the language was only one schema language among many and therefore needed a proper name led to the working group officially adopting the designation XSD for the 1.1 revision.)

Specific factors motivating the design of XML Schema included:

1. The basic DTD concept of defining the structure of a document by means of a BNF grammar for each kind of element was retained
2. Explicit constructs were added to represent the popular ways of exploiting parameter entities in DTDs: for example, to allow content models or parts of content models to be shared between different elements, to allow content models to be extended or restricted, and to allow groups of elements (substitution groups) to be defined so that the content model need only refer to the group to permit any of its members to appear.
3. There was explicit retention of some of the more exotic aspects of DTDs, such as the definition of attributes typed as IDs, entity references, or notation names. Some of these concepts were also generalized, but many of the quirky restrictions remained. The ability not only to validate the instance document, but to expand it by supplying defaults for missing attributes, was also carried forward. This was done as part of an attempt to ensure that schemas could be a 100% complete

replacement for DTDs; however, the secondary function of DTDs in defining external entities was not carried forward, so in that respect this goal was not achieved.

4. A data type system was added allowing the content of elements and attributes to be constrained, for example to integer or date values, or to strings matching a particular regular expression. The actual collection of primitive types that ended up in the specification is manifestly the result of a rather undisciplined commmittee decision-making process. The fact that the specification itself asserts that the set of data types is "judiciously chosen ... to serve the widest possible audience" can be seen as a rather lame attempt at self-justification: "we know it looks completely arbitrary, but we were trying to keep everyone happy."
5. An important principle guiding the development, perhaps the most innovative aspect of the specification and one that perhaps is the origin of many of its problems, is the idea that a schema should not only distinguish valid documents from invalid ones, but should annotate the validated document with type information derived from validation. For example, it should not only check that an attribute holds a valid date, but should annotate the attribute as such.

   This idea explains why XML Schema can be used to control data binding, that is the mapping of XML instance documents to data structures in conventional programming languages, and also why it can be used to underpin the type system of XML processing languages such as XSLT and XQuery. It also accounts for a large share of the complexity in the specification, as well as many of the restrictions in its capability: for example, the notorious "unique particle attribute" constraint which limits the grammar of content models to be unambiguous without lookahead, is there largely to ensure that a type can be unambiguously assigned to every element in the document.
6. The specification took the approach that namespaces should be used as the primary mechanism defining the modularity of an XML vocabulary.

   There is some evidence that the language designers recognized another principle, namely that there is never a single schema for a namespace. For example, different messages used by an application might use elements from the same vocabulary, but with different validation rules, or the same instance document might be subject to different validation rules at different stages in its lifecycle. However, the language ended up with little of substance to offer users in this area, other than a certain vagueness in the definition of the relationship between namespaces and schema documents that left implementors perplexed and users facing interoperability problems.
7. The specification attempted to cover the spectrum of XML applications from narrative document processing through to application data interchange. Since some XML users sit firmly at one end of this spectrum or the other, it can be ar-

gued that no-one was entirely happy with the outcome; it is certainly true that both groups are burdened with many facilities that they perceive as unnecessary.

As this discussion tends to show, it can be argued that XML Schema's widespread adoption as well as its reputation for being overburdened with both unnecessary complexity and unnecessary restrictions share a single root cause: the desire to satisfy the widest possible set of users and requirements.

The criticisms of XML Schema 1.0 are well summed up by James Clark [5] in an internet posting:

1. The syntax makes schema documents painfully difficult to read and write
2. Lack of orthogonality, for example restrictions on the attributes of a type are defined completely differently from restrictions on the child elements
3. A specification that lacks both readability and formal rigour
4. Numerous missing features, for example cross-dependencies between attributes and interleaving of child elements
5. An arbitrary yet fixed set of primitive data types
6. No ability to declare a particular element as the document root

In this posting Clark was arguing against the decision of an IETF group to define an XML vocabulary using XSD rather than Relax NG. The fact is, however, that although these criticisms of XSD are all valid and are well known, many users and standards bodies have chosen to use XML Schema despite all its faults. There are two factors that explain this: firstly, despite all the weaknesses, it generally does the job; and secondly, it has wide backing in the industry. It's one of those standards which everyone uses mainly because everyone else does.

3. An Overview of XML Schema 1.1

Standards work is not like software development. Working Groups try to publish requirements and use them to steer their decision-making; they try to publish plans (or at any rate, a few milestone dates) and stick to them; but in the end, the agenda for each meeting is determined by what the members of the working group put on the table, and the decisions made are determined by how they vote on each proposal. Moreover, most of the decisions depend on individuals rather than the companies they represent: the big decisions for most companies are whether to participate, whom to send, and whether to sign off the finished spec at the end. Everything else typically depends on the priorities of the individual members.

The XML Schema Working Group (of which I have been a member since 2007) is well aware of the deficiencies of the 1.0 specification outlined above, but is constrained in how to address the problems. The constraints arise from the need to maintain compatibility, from the limited resources available, and from the need to

achieve consensus for any change — the fact that there is a recognized problem is no guarantee that anyone will propose a solution that everyone else will agree on.

Progress on XSD 1.1 has been painfully slow for a number of reasons. As happens to some software development teams, fixing the many problems in 1.0 has sometimes taken priority over new development (no less than 152 errata to the first edition were published, which were then consolidated in a second edition in 2004; some of these ran to several pages of new text. No further errata have been published since 2004, because the working group decided to focus its efforts on shipping XSD 1.1; but there are 108 open bugs awaiting attention.) Many users of XML Schema would be surprised how small the core team developing the specification is: in 2000/01, while the original specification was under development, there were generally over 30 people at each meeting, but the typical attendence today is more likely to be six to eight. Volunteers are welcome!

Nevertheless, the development of XSD 1.1 is now in its endgame phase. There will still be a few bugs to iron out, and perhaps a few political skirmishes to overcome, but the shape of the final Recommendation is now fairly clear. The following sections give an overview of the main areas of change. Most of the facilities are described in the form of a brief synopsis, because I want to devote most of my time and space to a discussion of what I consider the most important feature, the introduction of assertions.

3.1. Relaxations on Content Models

A number of rules on content models have been relaxed, or new facilities introduced, to allow more flexibility to schema designers, or more accurate descriptions of existing instance documents.

1. Named element particles can now overlap with wildcards; if an element in an instance document matches both a specific element particle and a wildcard, the specific particle is chosen in preference.
2. So called "open content models" define wildcards that are permitted implicitly as children of any element, either anywhere, or after all other children. The main reason for this is to allow extensibility.
3. It is also possible to define a default attribute group containing attributes (including wildcards) that are applicable to all elements.
4. Wildcards offer more flexibility: as well as defining the permitted namespaces, they can now specify a list of prohobited namespaces; they can also prohibit a specific list of element or attribute names.
5. Wildcards are allowed in `all` groups.

6. Arbitrary occurrence limits are allowed in `all` groups, allowing a content model for example in which up to four `<author>` elements can appear, without constraints on where they appear.
7. An element may now appear in more than one substitution group. For example, in the schema for XSLT, the `<xsl:variable>` and `<xsl:param>` may appear in both the substitution group for instructions and that for top-level declarations.

3.2. Derived types

The complex rules in XSD 1.0 that define whether one type is a valid restriction of another have been replaced by a simple rule that states that R is a valid restriction of B if the set of sentences legal in R is a subset of the sentences that are legal in B. This of course leaves the onus on the implementor to invent an algorithm for checking this rule, but this is an improvement on XSD 1.0 where the specification provided an algorithm that was incorrect. There is also an escape clause that allows the implementation to fall back to checking instances against both types, and only reporting the invalid restriction the first time it finds an instance that is valid against R but not against B.

It is now possible in a local element declaration to specify a target namespace different from that of the containing schema document. This makes it possible to define restrictions of content models whose elements are in multiple namespaces. It is also possible to give names to identity constraints, allowing them to be copied into a type derived by restriction.

As we will see below, I expect that many cases where types have traditionally been derived by restriction will in future be much more easily handled by using assertions.

3.3. Co-occurrence Constraints

A long-standing complaint about XSD 1.0 has been that it is not possible to define cross-validation rules between different attributes, or rules whereby the permitted content of an element depends on the value of one of the attributes (unless of course the attribute happens to be called `xsi:type`).

XSD 1.1 addresses this with a feature called *conditional type assignment*. The idea is that an element may have a number of alternative types (or content models), and which of these is chosen is a function of the attributes present on the element. For example an `<address>` element might have different content models depending on the value of its `country` attribute. The conditional type defines the attributes permitted on the element as well as the content model for its children, so it is possible for example to define that two attributes are mutually exclusive by defining two alternative types (each permitting one of the attributes) and selecting the type based on the presence of one of the attributes.

The following example illustrates this. The type `addressType` (which may be abstract) is the generic type, from which all the other types must be derived; it defaults to `xs:anyType`. The last alternative in this example omits the `test` attribute, and thus defines the default type, which in this case is `xs:error` meaning that all instances are invalid. If this line were omitted the default type would be `addressType`.

```
<xs:element name="address" type="addressType">
   <xs:alternative test="@country='us'" type="addressTypeUS"/>
   <xs:alternative test="@country='gb'" type="addressTypeUK"/>
   <xs:alternative type="xs:error"/>
</xs:element>
```

There is a deliberate asymmetry here between attributes and child elements; conditional type assignment can only be driven by the attributes of an element, and not (for example) by the value of its first child element. The reason for this is to preserve the principle that validation should be streamable, that is, it should be possible to perform validation in a single pass through the document. In fact it would be possible to still to do streaming validation without this restriction (by validating against all the permitted types in parallel and then deciding at the end which of the validity assessments to ignore), but the working group wanted to keep implementation simple.

The logic for deciding which of the conditional types to validate against uses a small subset of XPath. The subset is syntactically constrained, supposedly for ease of implementation, but more importantly, the XPath expression is provided with a view of the input document that only contains the relevant element and its attributes, with no children, siblings, or ancestors. The current Saxon implementation constructs this limited subset view of the document, but then allows the whole of the XPath 2.0 syntax to be used to query it. This only allows use of handy functions such as `starts-with()` which are not present in the subset syntax.

There is a new type `xs:error` which can be assigned when the conditional type assignment rules find a combination of attributes that is invalid. This is in effect an alternative to using assertions.

3.4. Changes to Data Types

There is a new facet available on all the date/time types to control whether the timezone is mandatory, optional, or prohibited. These could previously be done using regular expressions, but the new facet is a much better solution as it makes it possible for applications to determine whether timezones are expected or not, and for validators to give better error messages when the constraint is violated. One new built-in derived type has been created to exploit this facet: `xs:dateTimeStamp`,

which is a xs:dateTime with mandatory timezone. Other similar types, for example an xs:date with no timezone, can be defined by users.

The types xs:dayTimeDuration and xs:yearMonthDuration, previously introduced in the XSLT/XQuery specifications, have now found their way into the XSD specification itself.

A new type xs:precisionDecimal is available. This has been a rather controversial addition, since many people felt there were quite enough numeric types already, but IBM have been pushing very hard for it, and it's often the case with standards that if one party is dogged enough, others will eventually give way in the interests of moving on to other things. The xs:precisionDecimal type differs from xs:decimal in that the number of digits is significant: 1.00 does not mean the same as 1. We are all going to have to relearn the basic rules of arithmetic as a result.

An interesting departure is that the specification now allows implementors to define their own primitive types and facets. This was introduced partly as a way to prevent the xs:precisionDecimal impasse occurring again in the future; if one vendor believes strongly enough that a new type is needed, they can add it to their product and let the market determine whether other vendors feel obliged to follow suit. It can also be seen as an answer to Clark's criticism, cited earlier, of the arbitrariness of the chosen set of primitives.

The specification now makes a distinction between values that are identical and values that are equal. For example, the durations PT1M and PT60S are equal but not identical. This means that if PT1M is allowed in an enumeration of durations, the value PT60S will also be accepted. This is different from the case of the numbers 1e2 and 10e1, which are both equal and identical, meaning that the system does not retain any distinction between them. It also provides some kind of solution to the problem of NaN, which is identical but not equal to itself.

Other changes to data types can largely be categorized as tinkering and bug-fixing. Many previously unresolved questions now have an answer, for example the meaning of a union type with xs:ID as one of its members, and the question of whether two empty lists are equal when compared in an identity constraint. Types derived by restriction from union types now work the way they should, not the way that XSD 1.0 said they should: which means, for example, that such a type can be included as a member type of another union. The behaviour of the namespace-sensitive types xs:QName and xs:NOTATION is now much clearer.

3.5. Schema Modularity

The exact interpretation of the three elements xs:include, xs:import, and xs:redefine was not very precisely defined in the XSD 1.0 specifications. XSD 1.1 is an improvement, though not as radical an overhaul as some might have wished. For example, it's still rather fuzzy about how circular includes or imports should be

handled, and this is an area where different implementations are known to interpret the spec in different ways.

An area that was particularly difficult to sort out was `xs:redefine` (for example, what happens if two schema documents A and B both redefine C?). Rather than clean this up, the working group has chosen to leave the spec largely unchanged, bugs and all, and to describe the facility as deprecated. It's not clear what practical effect this will have, since conformant implementations are still obliged to offer the facility, but at least it warns users that they are sailing close to the rocks. In place of `xs:redefine`, a new `xs:override` declaration is provided. This is specified in terms of an XSLT transformation that is applied to the schema document being overridden, so there should be far less room for ambiguity. The other difference from `xs:redefine` is there there is no requirement that the new definition should be in any way compatible with the old. For example, you can override an industry-standard schema in which a `<personName>` element has children `<firstName>`, `<middleName>`, and `<lastName>` with a definition in which `<personName>` is simply a string.

Analogously with the `use-when` attribute in XSLT 2.0, or with conditionally included or ignored sections in DTDs, there is a set of attributes that allow a section of a schema to be ignored depending on particular properties of the implementation: for example the version of the XSD specification that it supports, or the presence or absence of particular vendor-defined types.

4. Assertions

I've left what I consider the most important new feature in XSD 1.1 until last: assertions.

The concept of assertions is unashamedly derived from Schematron [6]; though the idea of course has much older parallels in database technology, where the ability to define integrity constraints on the data using arbitrary query statements as conditions has been around since the 1970s.

4.1. Defining an Assertion

In XSD 1.1, one or more assertions may be associated with a type. The facility is available for both complex and simple types, though the syntax and semantics are slightly different in the two cases. The assertion is an XPath 2.0 expression that is applied to the instance being validated; if the effective boolean value of the assertion is false (or if it fails with an error), then the instance is invalid.

An important rule is that the assertion can only see the data that is being validated against the type. So an assertion applied to an address, for example, can only look at the contents of the address; it cannot navigate "up the tree" to examine the context in which the address appears. So it cannot apply different rules depending on

whether the parent of the address is a <shippingAddress> or a <billingAddress>. If you want such rules, you have to apply them at the next level up. One reason for this rule is to minimize the impact of assertions on streamability; without a streaming XPath processor, assertions can always be evaluated by building an in-memory subtree of the part of the document being validated, which will typically be much smaller than the whole document. Another reason is simply philosophy: there is a basic assumption that the validity of an element depends only on the content of the element and not on any extraneous factors. This invariant is essential to many of the environments in which XSD is used; for example it is fundamental in XSLT and XQuery that an element (that is, a subtree rooted at an element) can be copied and attached to a new parent element without affecting its validity or the type annotations attached to its nodes.

The assertion can be any XPath 2.0 expression. There was much debate about whether to define yet another subset of XPath for writing assertion, but in the end this was not done; the full syntax must be supported. However, there is a loophole in that the set of functions available for use within the expression is implementation-defined. This means that some implementations might support the `matches()` function for regular expression tests, while others omit it.

The specification carefully defines the context in which the XPath expression will be evaluated. This has the effect of making some XPath constructs fairly useless: for example even if the expression is allowed to call the `doc()` function, the call will always fail because the definition says that `available documents` in the dynamic evaluation context is an empty set. This rule is designed to ensure that validation is largely context-free: two people validating the same instance against the same schema will generally get the same answer. (There are some exceptions. If the `current-dateTime()` is available, for example, then it is possible to write assertions such that the validity of an element depends on the phase of the moon.)

An issue which caused many technical problems in defining the specification was whether the nodes being validated should be seen by the XPath assertion as typed or untyped. Being typed would imply that they have already been validated against the schema, which clearly is not the case. Similarly the question arises as to whether the XPath expression should have access to the schema type definitions. To provide such access creates a danger of circularity: what happens if the assertion for a valid part-number consists of the expression `$value castable as part-number`? The solution adopted is that the top-level element being validated has the type `xs:anyType`, but its children and attributes are fully typed against the schema. The XPath expression can refer to built-in types (such as `xs:dateTime`) but not to user-defined types.

When validating against a complex type, the context item (.) refers to the element being validated, and if the type has simple content, the variable `$value` refers to the textual content of the element. When validating against a simple type, there is no

context item, and `$value` represents the content, as an instance of the nearest built-in type. (The way this is defined, if the user-defined type containing the assertion is a restriction of a list of integers, then `$value` will be a sequence of integers. So you can assert that the list is in ascending order with the assertion `test="every $i in 2 to count($value) satisfies $value[$i] ge $value[$i - 1]`. It would be much more difficult to express this assertion if the value were untyped.)

4.2. Examples of Assertions

Assertions apply either to complex types or simple types.

Suppose a `schedule` element contains a sequence of `event` children, and the events must be in ascending order by date. This constraint can be described like this:

```
<xs:element name="schedule">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="event" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   <xs:assert test="empty(event[@date lt preceding-sibling::event[1]/@date])"/>
  </xs:complexType>
</xs:element>
```

A few observations on this example:

- The context node for evaluating the test expression is the element whose validity is being tested.
- Saxon treats assertions that use the `empty()` function specially: if the set of nodes being tested is not empty, the nodes in the set are individually reported in the error message, with line-number information.
- The assertion has to be defined at the level of the `schedule`, not at the level of an individual event, because the validity of an element can only depend on its content (children and descendants), not on its neighbours or ancestors.
- There is no problem using the `preceding-sibling` axis within the assertion, so long as it does not try to navigate outside the bounding subtree.
- The XPath expression relies on the fact that `A lt B` is false if either operand is an empty sequence.
- Unlike Schematron, there is no provision for customised error messages. This may prove to be an area where implementors add value through vendor-specific extensions.

The next example gives an assertion on a simple type. For simple types, assertions are facets, and behave in the same way as other facets (pattern, length, maxInclusive,

and so on.) This example defines a subtype of `xs:date` containing all dates that fall on a weekday (Monday to Friday).

```
<xs:simpleType name="weekDay">
  <xs:restriction base="xs:date">
    <xs:assertion test="($value - xs:date('1901-01-06'))
                        div xs:dayTimeDuration('P1D') mod 7 = (1 to 5)"
  </xs:complexType>
</xs:element>
```

Some notes on this example:

- The value being tested is available as the value of the variable `$value`. This will be an instance of the base type, in this case `xs:date`, allowing date arithmetic as shown here.
- The example shows why a user-friendly error message would really be quite useful! However, the same can equally be said of some other facets, notably the `pattern` facet.
- The XPath expression does not have any access to any nodes in the instance document. There is no context item; the variable `$value` is an atomic value, not a node. One effect is that validation of values against simple types can be done in an environment where there is no XML document, for example validation of fields on an interactive form.
- The reason the value is supplied as a variable, not as the context item, is because it can be multivalued. Specifically, if the simple type is a list type, `$value` will hold a sequence of atomic values.
- The element is called `xs:assertion`, rather than `xs:assert`, because there would otherwise be an ambiguity in the syntax when defining assertions on a complex type with simple content.

4.3. Grammar versus Predicates

The provision of assertions in XSD 1.1 gives you a choice of two ways to define constraints on your instance documents: by means of a grammar, or by means of boolean predicates.

I believe this is analogous to the situation that exists in the database world, where a distinction is made between structural constraints and integrity constraints. In a SQL database, for example, structural constraints are imposed by the definitions of the tables and their columns, while integrity constraints are defined by means of primary and foreign key definitions and boolean `CHECK` clauses.

There are some constraints that are much more conveniently expressed using a grammar for the content model, while others are more convenient to express as

boolean predicates. This is not so much a question of the expressive power of the two formalisms, as of their practical usability.

Arguably, the availability of assertions makes some of the features for defining content models superfluous. For example, in the case of an interleaved content model (`xs:all`) it might in many cases be simpler to define the grammar as an unlimited repetition of a choice, and then use an assertion to define any cardinality constraints. This style of definition will be particularly useful for a content model such as *A head element, followed by one or more of each of the following, in any order...* which cannot be easily written as a grammar because of the XSD restrictions on combining sequence and interleave operators in the same content model.

Many users currently use XSD to define structural constraints, and supplement this with Schematron to describe integrity constraints. Some have even suggested that this combination works sufficiently well that adding assertions to XSD is unnecessary. I think this point of view is mistaken, and underestimates the power that comes from having both styles of constraint available in a single integrated environment.

One area where I suspect integrity constraints (assertions) will completely displace grammatical constraints is in defining restricted content models. This is partly because the XSD syntax for defining a restricted model is unhappily designed: instead of defining the differences from the base model, you are required to define the restricted model from scratch, and have the system tell you whether your new model is a true restriction. This makes the restricted model difficult to write and difficult to maintain (any changes to the base model need to be reviewed to see whether they should be reproduced in the restricted model). By contrast, defining the restriction by means of an assertion avoids this repetitive coding, and only concerns itself with how the two models differ. As far as I can tell, most practical restrictions of content models can easily be expressed in an assertion that specifies the absence, or in a few cases the maximum cardinality, of selected child elements. There are theoretical cases that don't work this way, for example when the restricted type allows all the elements appearing in the parent type but constrains their order; but I doubt such restrictions appear very often in practice.

A particular case in point is what I refer to as a "deep restriction". For example, suppose a global corporation has a schema for internal purchase orders, and deep within this is a field for monetary amounts, with an attribute to indicate the currency in use. One division of the company might want to use a specialization of the internal purchase order schema that is restricted so that the currency is always USD. There are several ways of doing this with XSD today: one can define a restriction of the top-level IPO type, which varies by using a restriction of the types at the next level down, which themselves vary only in the types of their children, all the way down to the type defining the currency attribute, which itself is restricted by means of an enumeration facet. Alternatively, one can use `xs:redefine`, which has all sorts of operational problems, not least of which is that the organisation is now using two

schemas which use the same names to refer to different things. A third solution is to create one schema by copy-and-paste, or perhaps by XSLT transformation, from the other. With assertions, there is a much cleaner option: define a subtype of internal purchase order with the assertion `test="empty(.//@currency[. ne 'USD'])"`.

4.4. Performance and Interoperability

Some members of the XML Schema Working Group have been very concerned about the potential impact of assertions on the cost of validation, especially since an assertion defined on the top-level type of a document might cause the entire document to be constructed as a tree in memory. Personally, my attitude is to give users the rope they need and credit them with the ability to decide whether or not to hang themselves. There is no doubt that the cost of validation can be significant in many applications, but in my view the only person who can decide whether and when the cost is justified is the user.

It's also worth pointing out that validating using assertions in a schema, even if expensive, may turn out to be much more efficient than doing the same validation in a user-written application.

Many XPath expressions can in fact be evaluated in streaming mode, without requiring too much intelligence on the part of the XPath compiler. The example in the previous section, `test="empty(.//@currency[. ne 'USD'])"`, is an obvious case in point. Generally, my experience is that placing restrictions in a specification in the hope that it will make it easier to produce performant implementations is usually misguided. Sometimes it can have completely the opposite effect: for example Saxon's schema processor goes to great lengths to maintain the particles in an `xs:all` content model as a sequence rather than a set, purely in order to enfore an artificial rule that the XSD 1.0 spec explicitly states is there only to make life easier for implementors!

There's always a great temptation in a standards effort, when two participants disagree about a decision that has to be made, for the working group to end up "having it both ways" by leaving the choice implementation-defined. Sometimes this is the right thing to do, especially when there are genuine differences between the requirements of different groups of users or different processing use cases. More often than not, however, leaving things implementation-defined is a compromise reached because different vendors could not agree on the desirability of a particular feature. An obvious example is the optionality of the `preceding-sibling` axis in XQuery. Such a decision is never in the best interests of the user community: in my view it's better to make the feature an integral part of the specification, and recognize the fact that some implementors may not provide it in their early releases.

In the development of assertions in XSD 1.1, there has been a great deal of debate about how much of XPath to make available. I have argued that most implementors will use an XPath subsystem off-the-shelf and it's more effort to subset it than to

ship the whole thing. That's a somewhat facile argument, because of course I know full well that in large companies, achieving this kind of software reuse can be remarkably difficult. However, I do think it is entirely in the users' interests that the whole of XPath should be supported. With the specification in its current state, you can use all the XPath syntax, but I'm surprised to discover that the function library is entirely implementation-defined. I hope that proves to be an oversight and one that is corrected before the ink finally dries.

5. Conclusions

XML Schema has its problems, but it is widely used and it works. A new version that addresses some of the limitations is therefore long overdue. While XSD 1.1 will not solve all the problems, especially those arising from excessive complexity of the existing specification, it does plug most of the biggest holes where there is missing functionality, and should greatly help users who rely on it as their primary validation technology.

Assertions in particular promise, in my view, to change the rules of the game by enabling a far richer set of constraints to be defined in a schema, and by making it much easier to define variants of schemas so that different rules can be applied to the same documents in different processing scenarios.

References

[1] Henry S. Thompson. David Beech. Murray Maloney. Noah Mendelsohn. *XML Schema Part 1: Structures Second Edition*. *W3C Recommendation*. 28 October 2004. W3C. http://www.w3.org/TR/xmlschema-1/

[2] Shudi (Sandy) Gao. C. M. Sperberg-McQueen. Henry S. Thompson. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. *W3C Working Draft*. 30 January 2009. W3C. http://www.w3.org/TR/xmlschema11-1/

[3] Tim Bray. *The Annotated XML Specification*. 1998. http://www.xml.com/axml/testaxml.htm

[4] Eric van der Vlist. *XML Schema Languages*. *Which one?*. 2002. XML 2002. IDEAlliance. http://www.idealliance.org/papers/xml02/dx_xml02/papers/04-01-01/04-01-01.pdf

[5] James Clark. *RELAX NG and W3C XML Schema*. *Comments on draft guidelines for use of XML by IETF at http://www.imc.org/ietf-xml-use/*. 4 June 2002. http://www.imc.org/ietf-xml-use/mail-archive/msg00217.html

[6] *Schematron*. http://www.schematron.com/

Full validation of Atom feeds containing extensions

村田 真 (MURATA Makoto [FAMILY Given])
International University of Japan
<eb2m-mrt@asahi-net.or.jp>

Abstract

The RELAX NG schema in the Atom Syndication Format [4] does not provide full validation of Atom feeds containing extensions. Rather, this schema skips extension elements and attributes, even when extension elements further contain Atom feeds or entries. This document shows that ISO/IEC 19757-4 Namespace-based Validation Dispatching Language [7] allows full validation of atom feeds containing extensions. NVDL decomposes atom feeds containing extensions into (1) extension-free atom and (2) extensions so that (1) and (2) are validated separately. As an example, an NVDL script for Google Calendar is presented.

Note: All files used in this note are available at http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atomextensions.zip.

Keywords: NVDL, Atom, Extension

1. Introduction

The RELAX NG schema (hereafter `atom.rnc`) in the Atom Syndication Format [4] does not provide full validation of atom feeds containing extensions. Rather, the schema focuses on top-level constructs of atom feeds; it skips extension elements and attributes, even when extension elements further contain constructs of atom feeds.

Some specifications (e.g., Atom Threading Extensions [5]) for atom extensions provide schemas for extension elements and attributes. These extension schemas focus on extension elements and attributes, and are typically written in RELAX NG. However, such extension schemas are not referenced from `atom.rnc`. As a result, these schemas do not provide full validation of atom feeds containing extensions. They are useful for documentation, but they are not usable for validating atom feeds.

One might wonder whether `atom.rnc` and extension schemas can be combined to form a single RELAX NG schema against which atom feeds containing extensions are fully validated. Our earlier work [8] is the first attempt for such combined

schemas. We combined a variation of `atom.rnc` and three schemas for atom extensions thereby successfully providing full validation. However, we do not believe that this all-in-one approach provides a reliable basis for full validation of atom and its extensions. The all-in-one approach requires that (1) schema authors understand schema customization techniques (e.g., the combine feature of RELAX NG) very well, (2) they avoid pitfalls caused by wildcards, and (3) they understand customization points of all schemas to be combined.

Recently, as part of Google DATA APIs, Google has started to provide RELAX NG schemas for their extensions of atoms. These schemas are rewrites of the original `atom.rnc` so that Google extensions are validated rather than skipped. For example, [10] contains `events_atom.rnc`. It is a rewrite of `atom.rnc` with elements and attributes for Google Calendar Data API, Google Data API, and [9] added. Any of the specific APIs appears to provide such schemas. Although Google's attempt for validating (rather than skipping) their atom extensions is certainly welcome, it has some drawbacks. First, rewrites by Google have to be updated whenever the original schema is updated, since the definitions in the original are duplicated. Second, rewrites by Google disallow other foreign elements and attributes, thus eliminating further extensions. Third, validation of XML documents returned by Google Calendar against `events_atom.rnc` leads to some validation errors, as of 2009 February.

In this document, we advocate the use of ISO/IEC 19757-4 Namespace-based Validation Dispatching Language [7] for full validation of atom feeds containing extensions. Schema authors for atom extensions first create schemas dedicated to the extensions. They then create NVDL scripts for combining these schemas and atom.rnc. Controlled by NVDL scripts, the NVDL engine decomposes atom feeds containing extension elements or attributes into (1) extension-free atom and (2) extensions so that (1) and (2) are validated separately. As an example, an NVDL script for Google Calendar is presented.

2. NVDL scripts for Atom

We demonstrate the use of NVDL for Atom extensions step by step. If the reader is not familiar with NVDL, some tutorials (e.g., [2], [1], and [3]) are available.

2.1. Atom and foreign elements/attributes

We begin with a simple NVDL script. It invokes `atom.rng` and does not invoke other schemas. First, we show a list of schema files in thie NVDL script.

- RELAX NG
  - `atom.rng`[1] (`atom.rnc`[2])

[1] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rng
[2] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rnc

Extracted from [4] and converted to the XML syntax.

- NVDL
  - `open.nvdl`[3] (shown below)
- XML
  - `simpleExample.xml`[4]
    Extracted from [4].
  - `openSearchExample.xml`[5]
    Extracted from 7.1.2 of [9].

The point of this NVDL script is the `<alow/>` action in the `anyNamespace` element. This action allows foreign attributes and elements. Before an atom feed is validated against `atom.rng` (which is equivalent to `atom.rnc`), all foreign attributes and elements are removed by the NVDL engine.

Example 1. `open.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
     startMode="root">
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng">
        <mode>
          <namespace ns="" match="attributes">
            <attach/>
          </namespace>
          <anyNamespace match="elements attributes">
            <allow/>
          </anyNamespace>
        </mode>
      </validate>
    </namespace>
  </mode>
</rules>
```

An extension-free feed (`simpleExample.xml`) and a feed containing OpenSearch extensions (`openSearchExample.xml`) are valid against this NVDL script.

[3] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/open.nvdl
[4] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/simpleExample.xml
[5] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearchExample.xml

Example 2. Validation of `simpleExample.xml` and `openSearchExample.xml`

```
makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh open.nvdl simpleExample.xml
Open the NVDL file: open.nvdl
Validate the instance file: simpleExample.xml
simpleExample.xml is a valid XML document.

makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh open.nvdl openSearchExample.xml
Open the NVDL file: open.nvdl
Validate the instance file: openSearchExample.xml
openSearchExample.xml is a valid XML document.
```

2.2. Atom, OpenSearch, and foreign elements/attributes

Our previous NVDL script does not validate [9] elements but merely skips them. Let us introduce an NVDL script that validates them.

- RELAX NG
  - `atom.rng`[6] (`atom.rnc`[7])
    Extracted from [4] and converted to the XML syntax.
  - `openSearch11.rng`[8] (`openSearch11.rnc`[9])
    Created from scratch.
- NVDL
  - `opensearch1.nvdl`[10] (shown below)
  - `opensearch2.nvdl`[11] (shown below)
    A named mode is shared.
  - `opensearch3.nvdl`[12] (shown below)
    It XIncludes `allow-foreign.nvdl`[13].
  - `allow-foreign.nvdl`[14] (shown below)
    It is XIncluded by `opensearch3.nvdl`[15].

[6] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rng
[7] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rnc
[8] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rng
[9] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rnc
[10] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/opensearch1.nvdl
[11] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/opensearch2.nvdl
[12] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/opensearch3.nvdl
[13] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/allow-foreign.nvdl
[14] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/allow-foreign.nvdl
[15] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/opensearch3.nvdl

  - `opensearch4.nvdl`[16] (shown below)
    An internal entity is used, instead.
- XML
  - `openSearchExample.xml`[17]
    Extracted from 7.1.2 of [9].
  - `openSearchIncorrectExample.xml`[18]
    An incorrect example.

Example 3. `opensearch1.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
     startMode="root">
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng">
        <mode>
          <namespace ns="" match="attributes">
            <attach/>
          </namespace>
          <anyNamespace match="elements attributes">
            <allow/>
          </anyNamespace>
        </mode>
        <context path="feed">
          <mode>
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng">
                <mode>
                  <namespace ns="" match="attributes">
                    <attach/>
                  </namespace>
                  <anyNamespace match="elements attributes">
                    <allow/>
                  </anyNamespace>
                </mode>
              </validate>
            </namespace>
            <namespace ns="" match="attributes">
              <attach/>
            </namespace>
```

[16] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/opensearch4.nvdl
[17] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearchExample.xml
[18] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearchIncorrectExample.xml

```
            <anyNamespace match="elements attributes">
              <allow/>
            </anyNamespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

This script is derived from the previous one by adding a `context` element, which controls the validation of foreign elements or attributes of `atom:feed` elements. The first namespace element within this `context` element specifies that `openSearch11.rng` be used for the validation of foreign elements of the namespace `"http://a9.com/-/spec/opensearch/1.1/"`.

Given an Atom feed, the NVDL engine first strips foreign elements and attributes. Foreign elements are validated against `openSearch11.rng`, if they are of the OpenSearch namespace and appear as children of `atom:feed`. Otherwise, foreign elements and attributes are allowed but removed before validation. Then, the atom feed without foreign elements and attributes are validated against `atom.rng`.

This script can be made more compact, since the child mode of `<validate schema="atom.rng">` and that of `<validate schema="openSearch11.rng">` are identical. We only have to create a named mode and reference to it with the attribute `useMode`.

Example 4. `opensearch2.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
    startMode="root">
  <mode name="allow-foreign">
    <namespace ns="" match="attributes">
      <attach/>
    </namespace>
    <anyNamespace match="elements attributes">
      <allow/>
    </anyNamespace>
  </mode>
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng" useMode="allow-foreign">
        <context path="feed">
          <mode>
            <namespace ns="" match="attributes">
              <attach/>
```

```
            </namespace>
            <anyNamespace match="elements attributes">
              <allow/>
            </anyNamespace>
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

Note that the `namespace` and `anyNamespace` elements in the anonymous mode are identical to the contents of the mode `"allow-foreign"`. If we use XInclude, we can make the script even more compact.

Example 5. `opensearch3.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
       xmlns:xi="http://www.w3.org/2001/XInclude" startMode="root">
  <xi:include parser="xml" href="allow-foreign.nvdl"/>
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng" useMode="allow-foreign">
        <context path="feed">
          <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

Example 6. `allow-foreign.nvdl`

```
<?xml version="1.0"?>
<mode xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
    name="allow-foreign">
  <namespace ns="" match="attributes">
    <attach/>
```

```
  </namespace>
  <anyNamespace match="elements attributes">
    <allow/>
  </anyNamespace>
</mode>
```

If XInclude does not work, we can use internal parsed entities.

Example 7. `opensearch4.nvdl`

```
<?xml version="1.0"?>
<!DOCTYPE rules [<!ENTITY allow-foreign-mode
"<mode name='allow-foreign'>
    <namespace ns='' match='attributes'>
      <attach/>
    </namespace>
    <anyNamespace match='elements attributes'>
      <allow/>
    </anyNamespace>
  </mode>">
]>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
     startMode="root">
  &allow-foreign-mode;
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng" useMode="allow-foreign">
        <context path="feed">
          <mode>
            &allow-foreign-mode;
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

A feed containing OpenSearch extensions (`openSearchExample.xml`) is again valid against these NVDL script.

Example 8. Validation of `openSearchExample.xml`

```
$ sh c:/nvdl/SnRNV/msvnvdl.sh opensearch4.nvdl  openSearchExample.xml
Open the NVDL file: opensearch4.nvdl
```

```
Validate the instance file: openSearchExample.xml
openSearchExample.xml is a valid XML document.
```

Consider an incorrect example `openSearchIncorrectExample.nvdl` created from by replacing

```
<opensearch:startIndex>21</opensearch:startIndex>
<opensearch:itemsPerPage>10</opensearch:itemsPerPage>
```

with

```
<opensearch:startIndex>twenty-one</opensearch:startIndex>
<opensearch:itemsPerPage>ten</opensearch:itemsPerPage>
```

Then, we have validation errors as below:

Example 9. Validation of `openSearchIncorrectExample.xml`

```
$ sh c:/nvdl/SnRNV/msvnvdl.sh opensearch4.nvdl  openSearchIncorrectExample.xml
Open the NVDL file: opensearch4.nvdl
Validate the instance file: openSearchIncorrectExample.xml
file:openSearchIncorrectExample.xml Line:12, Col:62, (Validate(openSearch11.rng)
 file:opensearch4.nvdl, Line:22, Col:76): "twenty-one" does not satisfy the "int" type
file:openSearchIncorrectExample.xml Line:13, Col:59, (Validate(openSearch11.rng)
 file:opensearch4.nvdl, Line:22, Col:76): "ten" does not satisfy the "int" type
2 errors in openSearchIncorrectExample.xml
```

2.3. Atom, OpenSearch, Threading Extensions, History, and foreign elements/attributes

Let us add two more extensions: [5] (Atom Threading Extensions), [6] (Feed Paging and Archiving). We use the XInclude version shown above as a basis.

- RELAX NG
  - `atom.rng`[19] (`atom.rnc`[20])
    Extracted from [4] and converted to the XML syntax.
  - `openSearch11.rng`[21] (`openSearch11.rnc`[22])
    Created from scratch.
  - `threadingElements.rng`[23] (`threadingElements.rnc`[24])
    Extracted from RFC 4685 (Atom Threading Extensions), slightly modified, and converted to the XML syntax.

[19] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rng
[20] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rnc
[21] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rng
[22] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rnc
[23] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/threadingElements.rng
[24] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/threadingElements.rnc

  - `threadingAttributes.rng`[25] (`threadingAttributes.rnc`[26])
    Extracted from RFC 4685 (Atom Threading Extensions) and converted to the XML syntax.
  - `archiveElements.rng`[27] (`archiveElements.rnc`[28])
    Extracted from RFC 5005 (Feed Paging and Archiving) and converted to the XML syntax.
- NVDL
  - `threading-and-history.nvdl`[29] (shown below)
- XML
  - `link-good.xml`[30]
    A correct example.
  - `inReplyTo-good.xml`[31]
    A correct example.
  - `link-bad.xml`[32]
    An incorrect example.
  - `inReplyTo-bad.xml`[33]
    An incorrect example.

Two `context` elements are added. The first one controls the validation of foreign elements or attributes of `atom:entry` elements. The second one, `atom:link` elements.

Three `namespace` elements are added. The first one appears within an existing `context` element. It handles the namespace "http://purl.org/syndication/history/1.0", to which `fh:archive` and `fh:complete` belong. The second one appears within the mode for the context `atom:entry`. It handles the namespace "`http://purl.org/syndication/thread/1.0`", to which `thr:in-reply-to` and `thr:total` belong. The third one appears within the mode for the context `atom:link`. Thanks to `match="attributes"`, this `namespace` element handles attributes of the namespace "http://purl.org/syndication/thread/1.0", specifically such as `thr:count` and `thr:updated`.

Example 10. `threading-and-history.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
```

[25] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/threadingAttributes.rng
[26] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/threadingAttributes.rnc
[27] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/archiveElements.rng
[28] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/archiveElements.rnc
[29] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/threading-and-history.nvdl
[30] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/link-good.xml
[31] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/inReplyTo-good.xml
[32] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/link-bad.xml
[33] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/inReplyTo-bad.xml

```
      xmlns:xi="http://www.w3.org/2001/XInclude" startMode="root">
  <xi:include parser="xml" href="allow-foreign.nvdl"/>
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng" useMode="allow-foreign">
        <context path="feed">
          <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng" useMode="allow-foreign"/>
            </namespace>
            <namespace ns="http://purl.org/syndication/history/1.0">
             <validate schema="archiveElements.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
        <context path="entry">
          <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace ns="http://purl.org/syndication/thread/1.0">
            <validate schema="threadingElements.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
        <context path="link">
          <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace match="attributes" ▶
ns="http://purl.org/syndication/thread/1.0">
              <validate schema="threadingAttributes.rng" ▶
useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

Two valid examples (`link-good.xml` and `inReplyTo-good.xml`) can be successfully validated.

Example 11. Validation of `link-good.xml` and `inReplyTo-good.xml`

```
$ sh c:/nvdl/SnRNV/msvnvdl.sh threading-and-history.nvdl link-good.xml
Open the NVDL file: threading-and-history.nvdl
Validate the instance file: link-good.xml
```

```
link-good.xml is a valid XML document.

makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh threading-and-history.nvdl inReplyTo-good.xml
Open the NVDL file: threading-and-history.nvdl
Validate the instance file: inReplyTo-good.xml
inReplyTo-good.xml is a valid XML document.
```

Errors in two invalid examples (`link-bad.xml` and `inReplyTo-bad.xml`) can be successfully detected.

Example 12. Validation `link-bad.xml` and `inReplyTo-bad.xml`

```
$ sh c:/nvdl/SnRNV/msvnvdl.sh threading-and-history.nvdl link-bad.xml
Open the NVDL file: threading-and-history.nvdl
Validate the instance file: link-bad.xml
file:link-bad.xml Line:16, Col:67, (Validate(threadingAttributes.rng) file:threa
ding-and-history.nvdl, Line:31, Col:83): attribute "thr:count" has a bad value:
"z10" does not satisfy the "nonNegativeInteger" type
1 errors in link-bad.xml

makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh threading-and-history.nvdl inReplyTo-bad.xml
Open the NVDL file: threading-and-history.nvdl
Validate the instance file: inReplyTo-bad.xml
file:inReplyTo-bad.xml Line:25, Col:51, (Validate(threadingElements.rng) file:th
reading-and-history.nvdl, Line:23, Col:81): unexpected attribute "ype"
1 errors in inReplyTo-bad.xml
```

2.4. Atom, OpenSearch, Google Calendar, and foreign elements/attributes

Google calendar XML documents are atom feeds containing extensions: OpenSearch, GData, and Google Calendar.

Permissible structures of Google calendar XML documents depends on the projection value used for retrieval. When the projection value is `"full"`, `gd:feedLink` elements have links to comments; when the projection value is "composite", `gd:feedLink` elements contain `atom:feed` elements, which represent comments. More about this, see [10].

An NVDL script for the projection value `"full"` is shown below.

- RELAX NG
  - `atom.rng`[34] (`atom.rnc`[35])

    Extracted from [4] and converted to the XML syntax.

[34] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rng
[35] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/atom.rnc

  - `openSearch11.rng`[36] (`openSearch11.rnc`[37])
    Created from scratch.
  - `gCal.rng`[38] (`gCal.rnc`[39])
    Extracted from [10], slightly modified, and converted to the XML syntax.
  - `gacl.rng`[40] (`gacl.rnc`[41])
    Extracted from [10], slightly modified, and converted to the XML syntax.
  - `gd.rng`[42] (`gd.rnc`[43])
    Extracted from [11], slightly modified, and converted to the XML syntax.
- NVDL
  - `full.nvdl`[44] (shown below)
  - `composite.nvdl`[45] (shown below)
- XML
  - `full20090207.xml`[46]
    An example document created by Google Calendar. It is available at `http://www.google.com/calendar/feeds/e1m39bcb04fuc79plrvmgmijho%40group.calendar.google.com/public/full`.
  - `composite.xml`[47]
    An example document created by Google Calendar. It is available at `http://www.google.com/calendar/feeds/e1m39bcb04fuc79plrvmgmijho%40group.calendar.google.com/public/composite`.
  - `compositeError.xml`[48]
    An incorrect example document created from `composite.xml` by replacing `updated` within `gd:feedLink` by `updaed`.

Example 13. `full.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
```

[36] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rng
[37] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/openSearch11.rnc
[38] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gCal.rng
[39] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gCal.rnc
[40] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gacl.rng
[41] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gacl.rnc
[42] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gd.rng
[43] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/gd.rnc
[44] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/full.nvdl
[45] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/composite.nvdl
[46] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/full20090207.xml
[47] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/composite.xml
[48] http://www.asahi-net.or.jp/~eb2m-mrt/atomextensions/compositeError.xml

```
     xmlns:xi="http://www.w3.org/2001/XInclude" startMode="root">
<xi:include parser="xml" href="allow-foreign.nvdl"/>
<mode name="root">
  <namespace ns="http://www.w3.org/2005/Atom">
    <validate schema="atom.rng" useMode="allow-foreign">
      <context path="feed">
        <mode>
          <xi:include parser="xml" href="allow-foreign.nvdl"/>
          <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
            <validate schema="openSearch11.rng" useMode="allow-foreign"/>
          </namespace>
          <namespace ns="http://schemas.google.com/g/2005">
            <validate schema="gd.rng">
              <context path="entryLink">
                 <mode>
                   <xi:include parser="xml" href="allow-foreign.nvdl"/>
                   <namespace ns="http://www.w3.org/2005/Atom">
                  <validate schema="atomEntry.rng" useMode="allow-foreign"/>
                   </namespace>
                 </mode>
              </context>
              <context path="feedLink">
                 <mode>
                   <xi:include parser="xml" href="allow-foreign.nvdl"/>
                   <namespace ns="http://www.w3.org/2005/Atom">
                   <validate schema="atomFeed.rng" useMode="allow-foreign"/>
                   </namespace>
                 </mode>
              </context>
            </validate>
          </namespace>
          <namespace ns="http://schemas.google.com/gCal/2005">
            <validate schema="gCal.rng" useMode="allow-foreign"/>
          </namespace>
        </mode>
      </context>
      <context path="entry">
        <mode>
          <xi:include parser="xml" href="allow-foreign.nvdl"/>
          <namespace ns="http://schemas.google.com/g/2005">
            <validate schema="gd.rng" useMode="allow-foreign"/>
          </namespace>
          <namespace ns="http://schemas.google.com/gCal/2005">
            <validate schema="gCal.rng" useMode="allow-foreign"/>
          </namespace>
          <namespace ns="http://schemas.google.com/acl/2007">
```

```
            <validate schema="gacl.rng" useMode="allow-foreign"/>
          </namespace>
        </mode>
      </context>
    </validate>
  </namespace>
 </mode>
</rules>
```

Let us validate an example document (`full20090207.xml`) created by Google Calendar. This document is obtained from `http://www.google.com/calendar/feeds/e1m39bcb04fuc79plrvmgmijho%40group.calendar.google.com/public/full`.

This document successfully validates against `full.nvdl`.

Example 14. Validation of `full20090207.xml`

```
makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh full.nvdl full20090207.xml
Open the NVDL file: full.nvdl
Validate the instance file: full20090207.xml
full20090207.xml is a valid XML document.
```

An NVDL script for the projection value "composite" is shown below. Observe that <atom:feed> elements within <gd:feedLink> elements are validated, while they are skipped in `full.nvdl`.

Example 15. `composite.nvdl`

```
<?xml version="1.0"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
       xmlns:xi="http://www.w3.org/2001/XInclude" startMode="root">
  <xi:include parser="xml" href="allow-foreign.nvdl"/>
  <mode name="root">
    <namespace ns="http://www.w3.org/2005/Atom">
      <validate schema="atom.rng" useMode="allow-foreign">
        <context path="feed">
          <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace ns="http://a9.com/-/spec/opensearch/1.1/">
              <validate schema="openSearch11.rng" useMode="allow-foreign"/>
            </namespace>
            <namespace ns="http://schemas.google.com/g/2005">
              <validate schema="gd.rng">
                <context path="entryLink">
                   <mode>
                     <xi:include parser="xml" href="allow-foreign.nvdl"/>
```

```
            <namespace ns="http://www.w3.org/2005/Atom">
             <validate schema="atomEntry.rng" useMode="allow-foreign"/>
            </namespace>
           </mode>
         </context>
         <context path="feedLink">
           <mode>
            <xi:include parser="xml" href="allow-foreign.nvdl"/>
            <namespace ns="http://www.w3.org/2005/Atom">
             <validate schema="atomFeed.rng" useMode="allow-foreign"/>
            </namespace>
           </mode>
         </context>
       </validate>
     </namespace>
     <namespace ns="http://schemas.google.com/gCal/2005">
       <validate schema="gCal.rng" useMode="allow-foreign"/>
     </namespace>
   </mode>
</context>
<context path="entry">
   <mode>
     <xi:include parser="xml" href="allow-foreign.nvdl"/>
     <namespace ns="http://schemas.google.com/g/2005">
       <validate schema="gd.rng">
         <mode>
           <namespace ns="http://www.w3.org/2005/Atom">
             <validate schema="atom.rng">
               <mode>
                 <namespace ns="" match="attributes">
                   <attach/>
                 </namespace>
                 <anyNamespace match="elements attributes">
                   <allow/>
                 </anyNamespace>
               </mode>
             </validate>
           </namespace>
           <namespace ns="" match="attributes">
             <attach/>
           </namespace>
           <anyNamespace match="elements attributes">
             <allow/>
           </anyNamespace>
         </mode>
       </validate>
```

```
            </namespace>
            <namespace ns="http://schemas.google.com/gCal/2005">
              <validate schema="gCal.rng" useMode="allow-foreign"/>
            </namespace>
            <namespace ns="http://schemas.google.com/acl/2007">
              <validate schema="gacl.rng" useMode="allow-foreign"/>
            </namespace>
          </mode>
        </context>
      </validate>
    </namespace>
  </mode>
</rules>
```

Again, let us validate an example document (`composite.xml`) created by Google Calendar. This document is obtained from `http://www.google.com/calendar/feeds/e1m39bcb04fuc79plrvmgmijho%40group.calendar.google.com/public/composite`. It successfully validates as previously.

Example 16. Validation of `composite.xml`

```
makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh composite.nvdl composite.xml
Open the NVDL file: composite.nvdl
Validate the instance file: composite.xml
composite.xml is a valid XML document.
```

Let us replace `updated` within `gd:feedLink` by `updaed`. Then, validation against `composite.nvdl` reports errors, while that against `full.nvdl` does not, since children of `gd:feedLink` are skipped.

Example 17. Validation of `compositeError.xml`

```
makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh composite.nvdl compositeError.xml
Open the NVDL file: composite.nvdl
Validate the instance file: compositeError.xml
file:compositeError.xml Line:194, Col:29, (Validate(atom.rng) file:composite.nvd
l, Line:46, Col:49): tag name "updaed" is not allowed. Possible tag names are: <
author>,<category>,<contributor>,<generator>,<icon>,<link>,<logo>,<rights>,<subt
itle>,<title>,<updated>, and more
1 errors in compositeError.xml

makoto@toraneko ~/atomextensions
$ sh c:/nvdl/SnRNV/msvnvdl.sh full.nvdl compositeError.xml
Open the NVDL file: full.nvdl
```

```
Validate the instance file: compositeError.xml
compositeError.xml is a valid XML document.
```

Let us compare advantages and disadvantages of our NVDL scripts and `events_atom.rnc`, which is provided as part of [10]. The RELAX NG schema `events_atom.rnc` is better than our NVDL scripts in several points.

1. Tighter constraints on the original Atom constructs are imposed, when necessary. For example, `div` elements of XHTML are disallowed, although they are allowed in the original `atom.rnc`. Meanwhile, our NVDL scripts cannot impose such constraints.
2. Permissible interactions of the original Atom constructs and Google extensions are specified precisely. For example, atom feeds embedded in `gd:feedLink` cannot have `atom:subtitle` elements, while top-level atom feeds can. Meanwhile, our NVDL scripts cannot impose such constraints.

Meanwhile, our scripts have some advantages.

1. Since `atom.rnc` is used as is, we do not have to modify our NVDL scripts, when `atom.rnc` is updated. Meanwhile, modifying `events_atom.rnc` accordingly is a non-trivial task.
2. Other extensions of Atom can be easily incorporated. Meanwhile, it is very difficult to extend `events_atom.rnc` for other extensions from third-parties. Meanwhile, it is prohibitively difficult to introduce further extensions to `events_atom.rnc`.

We believe that this comparison demonstrates the trade off between the all-in-one approach and the NVDL approach. The all-in-one approach is stricter, while the NVDL approach is more flexible.

3. Conclusions and Future work

We have presented the NVDL approach for validating Atom and its extensions. The NVDL approach cannot impose tight constraints but is more flexible. Specifically, an NVDL script for a particular extension can be easily extended to allow other extensions.

An interesting area for NVDL is validation of OOXML and ODF documents. In particular, ODF documents containing OOXML fragments, which are expected to provide round trip conversion, can be validated using NVDL.

One could argue that NVDL scripts are not very readable. Some syntax sugar can improve the readability. Specifically, James Clark suggested a new attribute `action` of `namespace` and `anyNamespace` elements, which allows compact representations such as <namespace ns="" action="attach"/> and <anyNamespace action="allow"/> rather than <namespace ns=""> <attach/> </namespace> and <anyNamespace

action="allow"> <allow/> </anyNamespace>. A drastic approach is to invent a compact syntax of NVDL, which is analogous to the RELAX NG compact syntax.

Bibliography

[1] David Pawson, An introduction to NVDL, ISO 19757-4, http://www.dpawson.co.uk/nvdl/

[2] Petr Nálevka and Jirka Kosek, NVDL Tutorial, http://jnvdl.sourceforge.net/tutorial.html

[3] Roger Costello, Tutorial on NVDL, http://www.dpawson.co.uk/nvdl/

[4] IETF RFC 4287, The Atom Syndication Format, http://tools.ietf.org/html/rfc4287.html

[5] IETF RFC 4685, Atom Threading Extensions, http://tools.ietf.org/html/rfc4685.html

[6] IETF RFC 5005, Feed Paging and Archiving, http://tools.ietf.org/html/rfc5005.html

[7] ISO/IEC 19757-4, DSDL -- Namespace-based Validation Dispatching Language, http://standards.iso.org/ittf/PubliclyAvailableStandards/c038615_ISO_IEC_19757-4_2006(E).zip

[8] Schema for the combination of Atom and its extensions, http://www.imc.org/atom-syntax/mail-archive/msg19894.html

[9] OpenSearch 1.1 Draft 3, Specifications/OpenSearch/1.1/Draft 3 - OpenSearch, http://www.opensearch.org/Specifications/OpenSearch/1.1#Example_of_OpenSearch_response_elements_in_Atom_1.0

[10] Google Calendar API Reference Guide (v2.0), http://code.google.com/apis/calendar/docs/2.0/reference.html

[11] Google Core Data API Common Elements: "Kinds", http://code.google.com/apis/gdata/elements.html

Introduction to Code Lists in XML

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

1. Controlled vocabularies

1.1. XML document interchange

An XML document describes a hierarchy of information items

- XML is only responsible for representation of information and not the meaning of information
  - how information is labeled allows it to be identified, not interpreted
  - up to applications to interpret the meaning of the labels and information so-labeled
- each item is labeled using the document's XML vocabulary
  - the item's value is expressed in an attribute's specification or an element's text value
- document constraints describe limitations on the contents of the XML documents
  - what is allowed to be used as item labels and where
  - what is allowed to be used as item values and where
  - a document isn't XML unless it is well-formed
    - rules govern the proper labeling of the information in the hierarchy
    - labels can be comprised of namespace URI strings to be globally unique
      - the metaphor for "labels in a namespace" is "words in a dictionary"
  - different document constraint languages provide different validation features
    - directives of the language engage validation semantics

Business documents have many information items whose values are controlled

- code lists have been used for hundreds of years
  - show up in historical documents, business records, passenger manifests, etc.
- codes, identifiers, any information item with a predetermined value set
  - like a label, a code represents the semantic, it doesn't "mean" the semantic
  - nothing in the value conveys understanding, only representation
  - still up to an application recognizing the code to be pre-programmed to interpret the semantic associated with that code

    - sender and receiver need to agree on the understanding of the value
- the information's value is limited to one or more of a set of fixed values
- item values do not impact on the structure of the document

Two distinct kinds of "vocabularies" for interchange

- the XML vocabulary of element and attribute labels
- a controlled vocabulary of code or identifier values
- in document interchange, the vocabularies represent the concepts and information for commonly-understood semantics
- in applications, internal representations of both may be very much richer than the interchange representation

1.2. Controlled vocabulary semantics

A controlled vocabulary is the set of agreed-upon values for a concept

- e.g. the list of country code abbreviations
    - e.g. "`CA`" for Canada, "`US`" for United States
- e.g. the list of currency code abbreviations
    - e.g. "`CAD`" for Canadian dollars, "`USD`" for United States dollars
- e.g. the list of transaction payment means
    - e.g. "`10`" for cash, "`20`" for cheque
- e.g. the list of units of measure
    - e.g. "`KGM`" for kilogram, "`MTQ`" for cubic meter
- e.g. identifiers for different kinds of dimensions
    - e.g. representing gross weight and net weight
- e.g. a company's private list of product and service identifiers
    - e.g. catalogue part numbers

Each value in a controlled vocabulary represents a particular semantic

- for obvious enumerated concepts, no semantic need be published authoritatively
    - e.g. the directions of latitude are either "`North`" or "`South`" of the equator
    - e.g. currency conversion operators are either "`Multiply`" or "`Divide`"
- for public vocabularies, the associated semantic is a published concept
    - managed by a public authority recognized as the trusted custodian of values
    - e.g. the International Organization for Standardization (ISO) list of country codes, currency codes, container sizes, etc.
    - e.g. the United Nations Economic Commission for Europe (UN/ECE) list of port codes, types of payment means, etc.

    - e.g. the Canadian Post Office list of Canadian province and territory abbreviations
- between trading partners, the associated semantic is an agreed-upon concept
    - e.g. the list of identifiers representing product and service offerings of a vendor
    - e.g. the document status codes accepted by a particular work flow specification

All parties implicitly agree to interpret the concepts in the same way in their independent applications

- by constraining the expression of the possible values to an agreed-upon set, both parties set expectations for interchange
- a formal expression of the constraints can form part of a business contract agreeing to limit values used to only the agreed-upon set
- traditional approaches to using W3C schema conflate the document constraints with the value constraints
    - new approaches are needed to layer value constraints on document constraints
- an important caveat: "obvious" values may not be obvious to all

A controlled value is necessarily unique in a single given list

- if a given string value represented more than one concept it would be ambiguous and there would be no way to distinguish which concept was desired
- list meta data for a value distinguishes ambiguous values in combined lists
    - the values may overlap when meta data is used to identify which list's distinct value is being used
- the unique value is analogous to a relational database table key
    - used as a lookup value
- another use of the "words in a dictionary" metaphor
    - the meta data of the list defines which dictionary the words are from
    - the meta data may distinguish different versions of the same dictionary

Each controlled value may have many associated values

- value meta data may be simple strings or compound values
    - compound information can be expressed in rich markup
- analogous to relational database table columns
- display string(s)
- non-normative synonyms
- language translations

- supporting detail and nuance
- meta data
  - derivation method
  - source of information

ISO parlance has been in use a long time for code lists

- "Code" refers to a value's unique key value within its list
- "Name" refers to that value's description with which meaning is intended to be expressed
- for some concepts, far more information needs to be associated with values

Controlled vocabularies are used in documents, databases, applications, messages

- by controlling the representation of a concept, a specified value can unambiguously identify the associated semantic
  - provided all users of the value understand the concept in the same manner
  - the burden is on the trusted custodian of the values to maintain the documentation of the list
- abbreviated values (codes) may provide a savings of effort or space when otherwise the expression of the concept is long-winded or wordy
  - the abbreviation is consistent
  - mnemonic values are typically biased to a particular language
    - e.g. "`USD`" mnemonic for "United States Dollars"
    - e.g. "`ES`" mnemonic for "Spain" ("España" is Spanish for "Spain")
  - non-mnemonic numeric values are often used as representations of abstract concepts without language bias
    - e.g. "`42`" non-mnemonic for "Payment to bank account" payment means
  - the mnemonic or non-mnemonic abbreviation is typically short
    - e.g. "`51`" non-mnemonic for "norme 6 97-Telereglement CFONB (French Organisation for Banking Standards) - Option A" payment means
- commonly used for centuries in messages to keep messages succinct

Promotes consistent interpretation of the value

- all applications can follow the published or agreed-upon semantics
- opportunity for misinterpretation through neglect or accident
- if each trading partner came up with their own abbreviations independently, it would be impossible to know that two different values represent the same concept
- removes language dependencies when abbreviating the same concept in two languages

    - though some codes are mnemonically derived from a native language, the rule governing that prevents the code from being derived differently in another language
- meta data columns can include various translations
    - promotes common interpretation

1.3. Facets of controlled vocabularies

1.3.1. Codes and identifiers

As a general rule of thumb (but not definitively), a controlled vocabulary information item is typically either a code or an identifier

- these are very symmetrically-defined constructs that are distinguished by arbitrary decisions of construction and use
    - guidelines and distinctions are not black and white
    - whether the values are characteristic or lookup can be twisted one way or the other
- these concepts are not always consistently applied
    - e.g. in UBL some identifiers could easily be codes and vice versa
- a code typically represents a unique concept, group or type using a *characteristic* value
    - e.g. a currency code for an account value - "`GBP`" (British pounds)
    - e.g. a unit of measure for a measurement - "`MTR`" (meters)
    - e.g. a shipping container's dimensions
        - this is an example of a set of coded values created by the application of a scheme on component parts of the value describing the container's height, width, depth and features
    - e.g. a method of transport
    - e.g. a document's type
- an identifier typically represents a unique thing or singleton from a group using a *lookup* value
    - may be synthesized by applying an algorithmic scheme
        - the range of identifier values may, however, be enumerated as members of a list
    - e.g. a particular account's identifier - "travel" or "supplies" or "`ABC0001`"
    - e.g. a particular dimension - "gross width" or "net width"
    - e.g. a particular aircraft's identifier

    - e.g. a particular catalogue item's identifier

Trading partners may wish to constrain either codes or identifiers or any other information item as a controlled vocabulary

- codes typically taken from a set representing known semantic concepts
- constraining an identifier would be from a fixed list of identifiers
    - e.g. a set of account identifiers
- open-ended identifiers would typically not be constrained
    - used to identify things that are being created

1.3.2. Code list registration authorities

The custodians of abstract code lists are typically the authorities with governance

- users of a list have a level of assurance regarding the maintenance of the sets of values
- stability is implied by the authority's governance of the concepts and expressions of the list members

The publishing of the list is different from the definition of the list

- the authority selects and defines codes in the abstract based on semantics (meaning)
- the list of codes is published hopefully with sufficient information to convey the semantics they represent so that all users interpret the codes as meaning the same concepts

The authorities can publish their code lists in many possible formats

- prose lists and descriptions
    - text files
    - word processing files
    - web site pages (HTML files)
    - algorithmic descriptions (e.g. ISBN checksum)
    - value assemblies (e.g. container height, width, depth and feature values)
- W3C schemas with annotations
- Comma Separated Values (CSV) files
- database tables
- colloquial XML expressions
    - using a bespoke document model invented by a community of users
    - an XML vocabulary not standardized outside of the community or users
- standardized XML expressions

  - using a published document model created by a committee effort
  - openness of process and access to results are important in assessing protection against private interests or encumbrances

Alternative expressions of lists may be made available in the absence of bona fide expressions

- a stop-gap measure to make up for the authority not having published the information in a useful form
- e.g. UBL has expressed a number of published abstract code lists using XML syntax until such time as the official custodians publish their own artefacts for public use

1.3.3. Identifying controlled vocabularies

Some controlled vocabularies are already officially maintained

- custodians are typically international standards organizations
- e.g. currency codes by ISO (ISO 4217)
- e.g. country codes by ISO (ISO 3166-1)
- e.g. payment means codes by UN/ECE (UN/ECE 4461)

Projects must establish which codes are applicable to their work

- community responsibility
  - manage expectations of individuals and trading partners
  - guide community in common understanding of concepts and representations
- a subset of codes from established lists
  - don't re-invent the wheel
- new codes for use where an established list is deficient
  - are extensions needed for the community to use?
- new lists of codes where there are no established lists
  - are entire new lists required for a set of community semantics?

Where new codes are needed,

- what do each of them mean?
  - what meta data might be associated with each?
- how are they coded?
  - mnemonic? numeric? arbitrary?
- which values are unconstrained?

List-level meta data identifies the list being used

- the needed list is an established list

- the list identification is the official list-level meta data

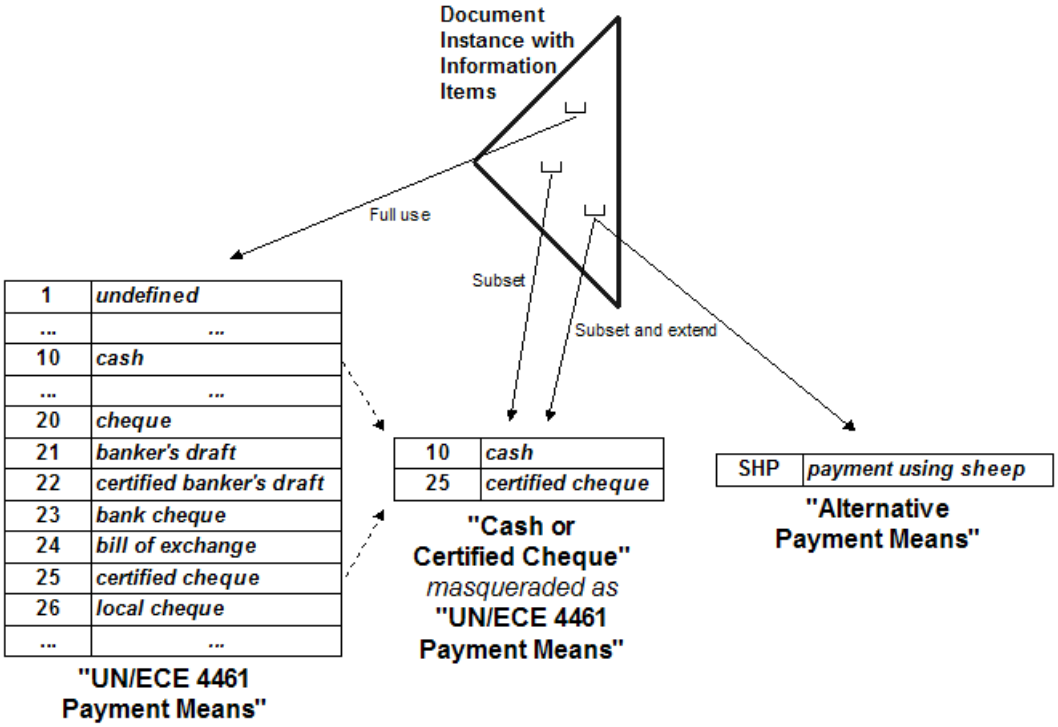


Figure 1. List-level meta data

Value-level meta data qualifies the code with more detail

- code and name are not always sufficient to identify information
- value-level meta data can be used to distinguish facets of the code

Code	Name	County	Subdivision	Port, rail, road, air, post, multi-modal, fixed, border	IATA	...
ADALV	Andorra la Vella	Andorra		3,4,6	ALV	...
...	...	...	...	...	...	...
USCB8	Columbus	United States	MT	2,3	CB8	...
USCBW	Columbus	United States	WI	3	CBW	...
USCLU	Columbus	United States	IN	3,4	CLU	...
USCMH	Columbus	United States	OH	4	CMH	...
USCSG	Columbus	United States	GA	3,4	CSG	...
USCUS	Columbus	United States	NM	3,4,B	CUS	...
USCZX	Columbus	United States	NC	3,6	CZX	...
USOLP	Columbus	United States	MO	3,6	OLP	...
USOLU	Columbus	United States	NE	3,4	OLU	...
USUBS	Columbus	United States	MS	3,4	UBS	...
USUCU	Columbus	United States	KS	2,3	UCU	...
USVCB	Columbus	United States	TX	3	VCB	...
USVDA	Columbus	United States	MI	4	VDA	...
USYBC	Columbus	United States	NJ	2,3,6	YBC	...
...	...	...	...	...	...	...
ZWWKI	Hwange	Zibabwe		4	WKI	...

"UN/ECE Rec 16 LOCODE"

Figure 2. Value-level meta data

Instance-level meta data qualifies the use of a code

- tells an application the list in which the code is found
- clarifies the meaning

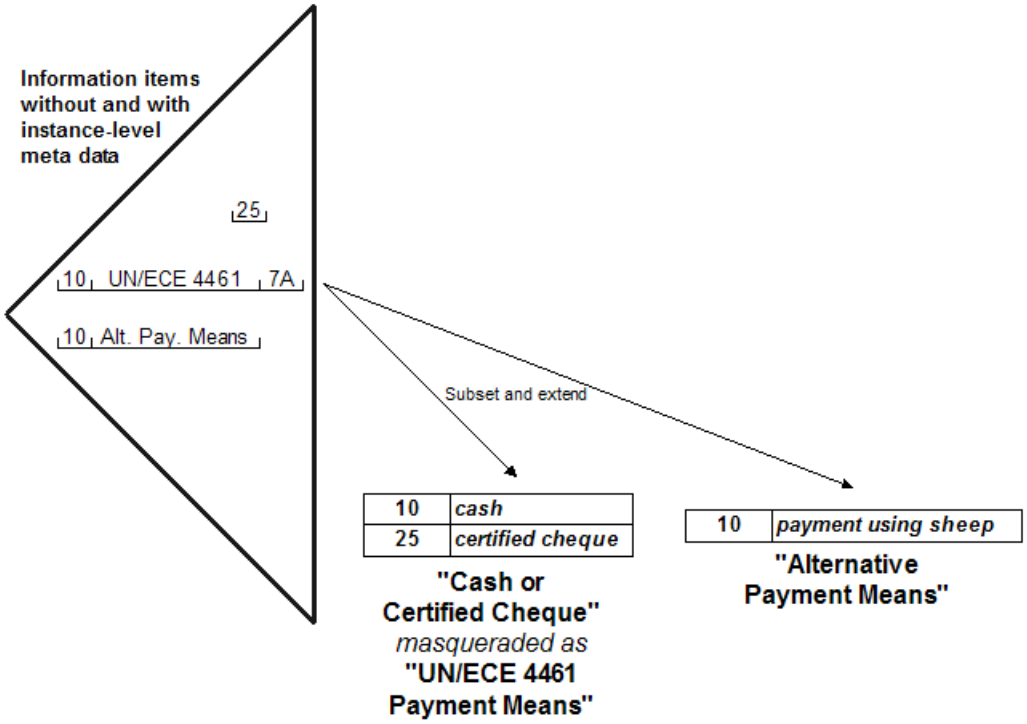


Figure 3. Instance-level meta data

Community responsibility when defining the XML vocabulary

- which instance-level meta data can the user specify?
- how does the user specify instance-level meta data?

Summary of controlled vocabulary meta data

- list-level meta data
  - distinguishes one list of values from another list of values
  - responsibility of the controlled vocabulary custodian
- value-level meta data
  - distinguishes one value from another value within the same list
  - responsibility of the controlled vocabulary custodian
- instance-level meta data
  - distinguishes from which list a value is being used
  - responsibility of the XML vocabulary designer
  - utilized by the XML document writer

1.3.4. Modeling controlled vocabularies

The organization of a set of associated values for all key values is tabular

- one row per semantic concept

- one or more key value columns uniquely identifying the concept
- zero or more meta data value columns defining the concept

The maintenance of list information necessarily needs to be tabular

- such a need distinguishes the available enumeration technologies as to their usefulness
  - i.e. a technology that does not support a tabular arrangement is not as useful as a technology that does support a tabular arrangement of list information

Maintaining an independent expression provides for re-use and change isolation

- the maintenance of a list of values will likely have a different life cycle than the contexts in which the values are used
- revising an external expression of a controlled vocabulary prevents having to change an expression in which a controlled vocabulary is embedded

Human language translations may help as supplemental information

- may reduce problems interpreting what a value represents

1.3.5. Expressing controlled vocabularies

Non-standard use of spreadsheets, word processing, comma-separated value (CSV) documents is common

- each custodial organization may have its own way of expressing the representation values and their associated semantics
- applications incorporating the values into their validation processes would need to accommodate ad hoc means with ad hoc measures
- the expression may not be well defined for maintenance

The interchange representation is independent of the internal representation

- though some applications may choose to use the interchange representation as the internal representation
- lookup strategies should be based on application requirements and be independent of interchange requirements

Artefacts for legal contractual agreements may be ambiguous

- using a standardized representation allows both parties to interpret all lists in the same fashion
- prose is often improperly used when meta data may be less ambiguous
  - especially when human language translation is involved

Standards exist with which one enumerates the defined values of a controlled vocabulary

- the choice of expression empowers the use of that expression in different circumstances
- some XML designers fervently believe that document values belong in a document schema
- some XML designers fervently believe that document values must never be in a document schema

W3C Schema enumerations

- designed for use only in validation with W3C Schema semantics
- the formalism only captures the key coded values in a standardized structure
  - the associated meta data may be expressed in a non-standardized structure
  - only one key can ever be used to distinguish the information in the list
- document-wide scope of re-use
  - e.g. every use of the code list incorporates every code of the code list
  - using only a subset of codes requires declaring a separate code list for those contexts where the subset is needed
    - there is a question of what meta data to use for the lists
    - the full list's meta data would be inappropriate for a subset list, yet instances might require the use of the meta data for the full list
- the expressions are intertwined with the expressions of structural constraints
  - to change an enumeration one must "touch" the schema files that participate in structural validation
  - risk of inadvertently modifying the structural constraints, or burden of proving that the structural constraints were not inadvertently modified

OASIS Genericode 1.0 (2007)

- designed for maintenance of the meta data of an enumeration and its members
- the formalism captures all information about values in a standardized structure
  - when there are multiple keys, the actual key needed can be chosen by an application
  - specifies standardized list-level meta data
    - all controlled vocabularies can be identifies using the same mechanisms
  - specifies mechanisms for arbitrary value-level meta data
    - each controlled vocabulary can satisfy its own requirements for value distinction and definition
- context-free scope of use
  - the definition of the codes is independent of the specification of where the codes are used

- the external XML-based expression is independent of any particular use
  - useful for validation or user-interface definition or any use
- still a role for schema specification of available instance-level meta data

1.3.6. Data entry of controlled vocabularies

By formally associating the use of value lists with document contexts, one can direct valid data entry

- directs a user interface
  - can be written to only allow entry of a value from the associated values
- value-level meta data is helpful
  - could be presented to the user to help them choose the right value to use in the data entry
- instance-level meta data records list-level meta data where needed for disambiguation
  - when one information item can have a value from two lists, and a code is needed that exists with the same value in each list, the list-level meta data distinguishes the code and needs to be recorded as instance-level meta data

1.3.7. Application development supporting controlled vocabularies

Controlled vocabularies are declarative while application program code is imperative

- easier (therefore cheaper?) to change an outboard declared list of allowed values than to change the inboard program logic
- non-programmer resources can be tasked with changing the declared vocabularies

An application can blindly support all values in a controlled vocabulary

- the application can support all possible allowed values and presume that prevalidation has rejected those instances where a supported value is not allowed
- the flexibility is in the filtering of allowed instances by dynamic application of value constraints during validation, without changing the programming in the application

The trading partner relationship constrains which values are allowed in a given transmission

- message filtering ensures only the messages with the allowed values for a given trading partner are passed for processing

Reduces application development to support new trading partners

- no need to change the program for every trading partner or new trading partners

Flexible to changing trading partner relationships

- as a relationship with a partner matures or changes, only the message filtering need change, not the application code

1.3.8. Validating controlled vocabularies

Having an expression of valid values enables the validation of specified values

- validation can reject an instance before engaging an application to act on the instance
- off-loads the validation responsibility (and possible error) from applications
- ensures consistent loading of database values

Values outside of the allowed set are considered invalid

- trading partners would not necessarily know what semantic an unexpected value represents
- legal agreements could not be entered into where the parties have arbitrary values possibly representing concepts outside of the agreement

Methodologies are published with which one confirms the proper selection of values in an XML information item

- traditional use of grammar- or type-based document schemas
  - e.g. XML DTD - grammar-based schema language
  - e.g. ISO/IEC 19757-2 RELAX-NG - grammar-based schema language
  - e.g. W3C Schema - type-based schema language
- alternative schema expressions
  - e.g. ISO/IEC 19757-3 Schematron - assertion-based schema language

Traditional approaches validate values at the same time as validating structure

- conflates structural validation with value validation
- inflexible to dynamically changing business requirements
  - no need to change the structural validation just because business relationships change
- business agreements impact on values but do not impact on document structures

UBL 2.0 separates UBL conformance from code list conformance

- to which version of UBL schemas do the structures conform?
  - e.g. `UBL-Invoice-2.0.xsd` for an invoice
- to which code lists do the values conform?
  - e.g. `defaultCodeList.xsl` for a suite of typical code lists

Layering value constraints on top of structural and lexical constrains

- can be used for any XML document structure, not only UBL
- can be applied to any information item with an enumerated set of allowed values
- not restricted to only codes or identifiers
- can be built on top of ISO/IEC 19757-3 Schematron
- separates structural/lexical validation from value validation

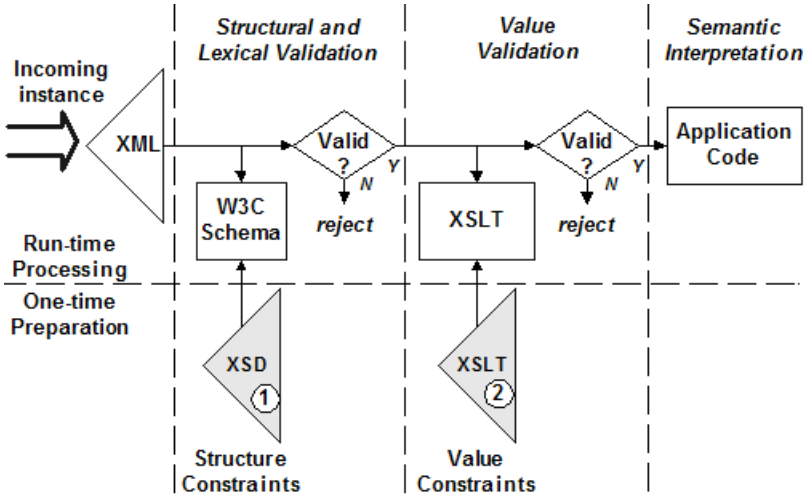


Figure 4. Two-step validation

Opportunity to incorporate many kinds of value constraints

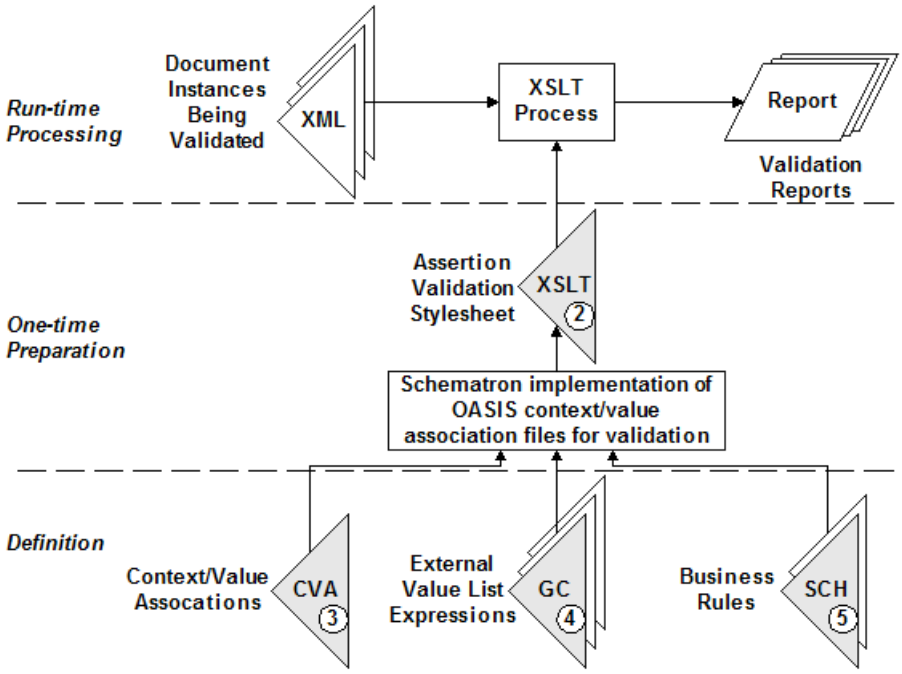


Figure 5. Second-pass value-validation artefact creation

Context/value associations establish which code lists apply where in the document

- gives flexibility to specify different codes for the same conceptual value used in different document contexts

External value list expressions in genericode

- the XML documents defining the controlled vocabularies
- includes list-level and value-level meta data

Business rules can express co-occurrence and algorithmic constraints

- more powerful than simple declarative approaches
- use Schematron for arbitrary XPath expressions

1.3.9. Semantic representation by fixed values

Assigned semantics

- each unique value represents an associated concept, label or longer value
- community of users agree on the association between specified value and represented value
- a value has two aspects of context in order to have some meaning
  - context of use/location
    - where in the document is the information item being used
  - context of definition/meaning
    - from which set of values is the information item value being obtained
  - without explicit context a value may be ambiguous or reliant on informal agreement
- especially important for non-mnemonic codes: e.g. "`42`" vs. "`USD`"
  - the meaning of some mnemonic codes might be guessed based on context of use, e.g. "`USD`" for a currency
  - non-mnemonic codes typically have no basis for guessing the meaning, e.g. "`42`" for a payment means

Instance-level meta data disambiguates a code when context is insufficient

- used for identification of values and definition of values
- list meta data identifies the collection from which the value is taken
  - information about the collection as a whole
  - gives context to the specified values
- value meta data helps define the semantics or details of the value itself
  - information about the one particular coded value

Changes in time can affect the interpretation/semantics of values

- the collection of values evolves creating a new version of an existing code list
  - identified by associated meta data
- the meaning of individual values evolves
  - described by associated meta data or prose
- migrating data from old to new may require simultaneous support of multiple versions of the same code list
  - requires flexibility not typically associated with traditional schema-based approaches to using codes
- unused and retired codes might get re-used later for new semantic concepts
  - e.g. country code "CS" was Czechoslovakia before 2003 and was reserved in 2006 for Serbia and Montenegro

1.3.10. Trading partners and agreements

Trading partners need to agree on the structure and content of interchanged documents

- so doing provides interoperability between independent systems acting on the information
- using XML isn't a magic bullet
  - it doesn't make our programs better, it makes our interchange of information more reliable
  - layers interchange constraints on top of implementation foundations
- structuring the information in a standardized fashion ensures the information is communicated
  - where to find information based on how it is labeled and where it is found hierarchically
- agreeing on the semantics behind values in the communicated information ensures the information is understood
  - what specified information values mean and represent in the abstract

Relationships between trading partners change, while standards do not

- trading partner requirements can be layered on top of industry standards
- trading partners can also anticipate future changes to standards

Published interchange specifications cannot pretend to know every value that trading partners need

- there is no standardization of many business concepts, just practical and pragmatic use for those concepts of import to trading partners

- therefore that can be no standardization of a set of values representing business concepts that are particular to trading partners

Industries can state what the semantics are behind values

- trading partners can agree on which values to use

Codes for some established business practices can be supplemented or subset by trading partners

- e.g. extending document status codes
  - a typical workflow will have typical status values for the progress of a document
  - particular workflow systems used by trading partners may use only some or maybe more document status values for a given document
- e.g. restricting payment means codes
  - payment can only be by certified cheque or credit card
- e.g. restricting and extending transportation status codes
  - the suite of status codes includes a subset of a standard suite in combination with non-standard additional values

Identifiers are especially important as they specify actual business objects and not business concepts

- e.g. account identifiers
  - every business will probably have a different set of accounts than other businesses
  - when engaging in a transaction, a trading partner can publish its accepted list of account identifiers so that a correspondent knows which values can be used
- e.g. measurement identifiers
  - a catalogue item's characteristics need to be identified unambiguously (e.g. "gross weight" is distinct from "net weight")

Additional business rules can be layered on top of value constraints

- e.g. validity of non-coded data values based on trading partner relationship
  - e.g. a maximum value for an order
- e.g. the nature of a product identified by its identifier may restrict the payment means by which it is paid for
  - e.g. no credit cards for certain products

2. Defining and using controlled vocabularies

2.1. Controlled value list maintenance and identity

A list of values has an identity in the abstract, regardless of how it is maintained

- e.g. ISO 3166-1 country codes
- e.g. UN/ECE 4461 payment means codes
- e.g. UBL 2.0 document status codes

The complete list may be maintained by hand or by a database or by any means

- the management of the values is important to long-term maintenance
- some lists may have tens of thousands of entries (e.g. vehicle model codes)
- a list may be synthesized by an algorithmic process
  - e.g. the 100 ISBN numbers assigned to publisher "978-1-894049"

The concrete expressions of the lists may vary based on purpose or contextual use

- e.g. complete lists
- e.g. restricted subset lists
- each list and list subset expression must necessarily be uniquely identified
  - a subset of a list cannot have the same identity as the complete list otherwise there would be confusion regarding which list is the "true" list
  - identity can be expressed as meta data for the list or list subset
- the concrete expression may take any useful form for the user
  - e.g. simple text
  - e.g. comma-separated values
  - e.g. XML files
  - having a standardized representation of lists would encourage the development of more widely-useful applications

The sender and receiver may have different identities for a list of identical values

- e.g. the sender specifies an ISO country code
  - the meta data for the list is that of the complete list
- e.g. the receiver only accepts a subset of ISO country codes as valid
  - the meta data for the subset list is necessarily different than that of the complete list
  - for validation purposes the subset list must masquerade as the complete list yet reject specified values outside of the subset list

2.2. Controlled value specification

A controlled value is, in fact, a multi-faceted value

- the list from which the code value is obtained
  - described by meta data for the list
  - the list identification itself may be multi-faceted
- the key code value itself
  - unique within any given list
- properties (value-level meta data) of the values themselves
  - helpful in understanding the semantics of the key code value

When an information item can be populated with a coded value, it should also be possible to specify the associated value list meta data

- even very stable lists of values will change over time
  - one may need to specify a chronologically-distinct interpretation of a given value
  - e.g. the list of provinces in Canada changed in 1999 when the Northwest Territories was split into two territories: Nunavut and the Northwest Territories
    - the Canadian postal province and territories indication of the Northwest Territories was and remains "NT" even though the definition of the territory changed
    - if the distinction is important to a trading partner, then provision for making the distinction must be made available
  - e.g. the list of country codes changes frequently
    - before 2003 "CS" represented Czechoslovakia and since 2006 "CS" is reserved for Serbia and Montenegro
- one information item value may be an item selected from one of a number of lists
  - if all of the values are mutually exclusive in separate lists, there is no risk of confusion other than changes over time for any given list as noted above
  - if the values in the lists are not mutually exclusive, meta data is required to disambiguate an ambiguous specified value

The risk is borne by the party encoding the information that the recipient can properly decode the intent expressed by the information

- list meta data is often optional and is often ignored when coded values are specified
- the more specific a specification is, the less opportunity for improper understanding of the intended meaning

3. Declaring controlled vocabularies

3.1. Declaring controlled vocabularies

Standards are in development for the non-schema-based representation of a list of coded values

- trading partners may wish to trim or augment the list of coded values acceptable
- trading partners may wish to use different controlled vocabularies for a given information item found in different document contexts
- the representation of individual coded values includes documentary information and metadata
  - for detailed value description
  - for long-term maintenance and understanding
- OASIS genericode 1.0
  - `http://docs.oasis-open.org/codelist/genericode`
  - an XML representation standardized by the OASIS Code List Representation Technical Committee
  - `http://www.oasis-open.org/committees/codelist/`
  - "Defining an XML format for interchange, documentation and management of code lists (a.k.a. controlled vocabularies or coded value enumerations) in any processing context"
    - not obliged to use XML format *inside* the application
    - very common to compile the XML interchange format into an internal processing format
      - e.g. conversion to XSLT
      - e.g. implementation in database stored procedures
    - XML is designed for interchange and is not always conveniently structured for real-time processing

One could use schema enumerations but ...

- too inflexible for globally-defined information items
  - cannot have different sets of values in different document contexts for a globally-defined information item
- modifying the schemas means using non-standardized schema expressions
  - not bad in and of itself but requires extra assurances for compatibility
  - structural and lexical validation is assured if the standardized schema expressions are treated as read-only

Meta-data-only code list are important as placeholders

- effectively an infinite set of all possible codes satisfying the lexical rules
- indicating that a particular information item's value is from a controlled vocabulary but that there is no controlled vocabulary listing a set of codes
- e.g. only 18 of 91 UBL code lists are published with values, 73 uniquely-categorized code lists have only meta data
- users have the option of restricting the infinite list into a finite list

3.2. Rendering controlled vocabularies

3.2.1. Rendering controlled vocabularies

Some audiences do not appreciate having to read raw XML to interpret the contents

- angle brackets are distracting
- the volume of XML markup overwhelms the information content of the instance

Crane Softwrights Ltd. has published free developer resources with which to render a genericode code expression to HTML

- XSLT 1.0 stylesheet: `Crane-genericode2html.xsl`
- standalone production of HTML from genericode file
- browser-based viewing of HTML from genericode file

3.2.2. Standalone production of an HTML rendering

Consider the HTML rendering of a simple one-item code list:

- ```
  java -jar saxon.jar -o Additional_PaymentMeansCode.html
  Additional_PaymentMeansCode.gc Crane-genericode2html.xsl
  ```

The resulting HTML file when rendered appears as follows:
  ```

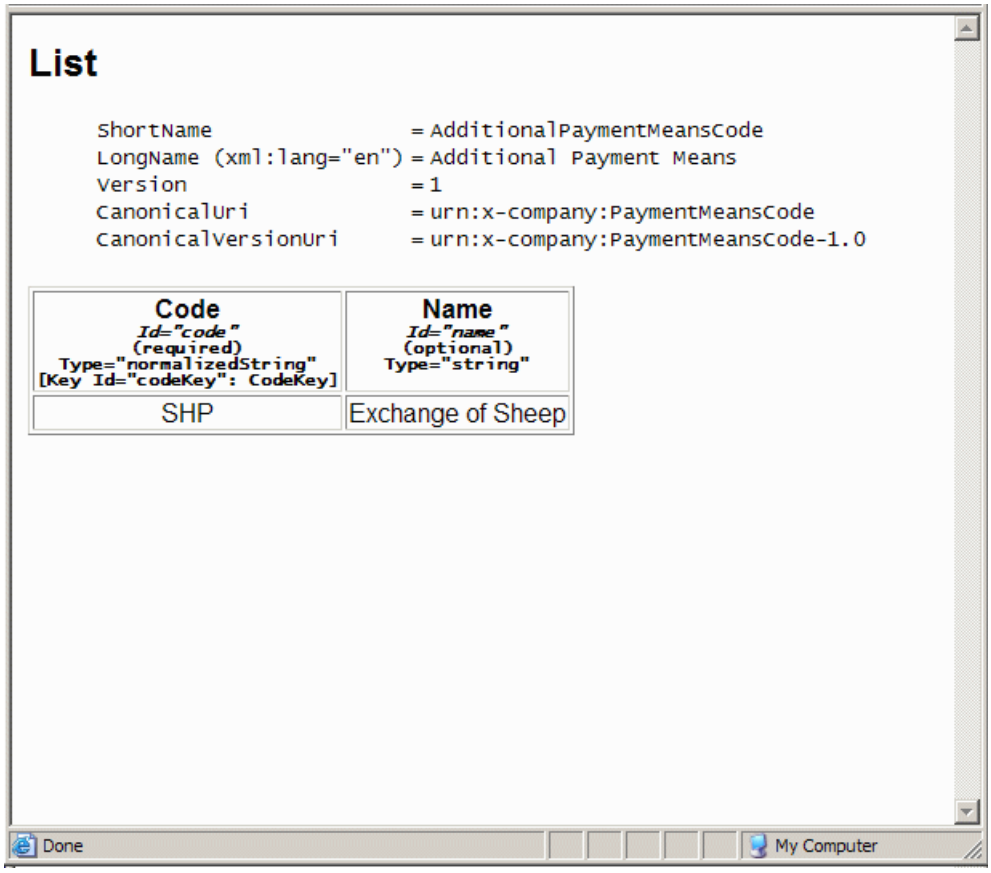


Figure 6. HTML rendering of a genericode file

3.2.3. Browser-based viewing of an HTML rendering

W3C XML stylesheet association standardized `xml-stylesheet` processing instruction:

- `http://www.w3.org/1999/06/REC-xml-stylesheet-19990629`
- `href=` points to the stylesheet file (modify as required)
- `type="text/xsl"` used to indicate the interpretation of the stylesheet
- in Windows drag the file from Windows Explorer to the browser canvas to render

The modified file includes the processing instruction recognized by XML processing tools:

- examples in the `samp/ss` directory

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="Crane-genericode2html.xsl"?>
<gc:CodeList
xmlns:gc="http://docs.oasis-open.org/codelist/ns/genericode/1.0/">
   <Identification>
```

```
      <ShortName>AdditionalPaymentMeansCode</ShortName>
      <LongName xml:lang="en">Additional Payment Means</LongName>
      <Version>1</Version>
      <CanonicalUri>urn:x-company:PaymentMeansCode</CanonicalUri>
      <CanonicalVersionUri>urn:x-company:PaymentMeansCode-1.0
</CanonicalVersionUri>
   </Identification>
   <ColumnSet>
      <Column Id="code" Use="required">
         <ShortName>Code</ShortName>
         <Data Type="normalizedString"/>
      </Column>
      <Column Id="name" Use="optional">
         <ShortName>Name</ShortName>
         <Data Type="string"/>
      </Column>
      <Key Id="codeKey">
         <ShortName>CodeKey</ShortName>
         <ColumnRef Ref="code"/>
      </Key>
   </ColumnSet>
   <SimpleCodeList>
      <Row>
         <Value ColumnRef="code">
            <SimpleValue>SHP</SimpleValue>
         </Value>
         <Value ColumnRef="name">
            <SimpleValue>Exchange of Sheep</SimpleValue>
         </Value>
      </Row>
   </SimpleCodeList>
</gc:CodeList>
```

4. Associating controlled vocabularies in XML documents

4.1. Constraining information items using controlled vocabularies

Three kinds of constraints to be validated for an XML document

- structural constraints ensure information items are correctly found
- lexical constrains ensure information items are correctly formed
- value constraints ensure information items are correctly understood

Constraining the document structure and lexical patterns is independent of business/value rules

- a community of users can publish an agreed upon schema to validate information items are correctly found and formed

Constraining information item use of controlled vocabularies is very dependent on business/value rules

- business/value rules implied by the nature of the information item
  - e.g. points of a compass will never change
- business/value rules imposed by a community of users
  - e.g. the document status codes for the condition of a document in a transaction
- business/value rules agreed upon between trading partners
  - e.g. identification of account numbers for particular purposes

Typical use of W3C Schema conflates structural and value constraints inflexibly

- one gets more flexibility by separating value constraints from structural constraints
- only structural constraints should be imposed across a community of users
  - standard should constrain how the information is found and how it is formed, not how it is valued
  - very infrequent changes to the structure of information being interchanged
  - changes imply big impacts on applications and processing
- value constraints should be selectively imposed
  - changes in trading partners
  - changes in business practices over time
  - possibly frequent changes to the values allowed by different parties
  - once programs accommodate a given set of values, changing the subsets of values in use doesn't change the applications
- business rules should be selectively added
  - private requirements could never be anticipated by standards committees

4.2. Context/value association

Context/value association files

- `http://www.oasis-open.org/committees/document.php?document_id=29990`
- an XML vocabulary for associating document contexts with specified values
- suitable for constraining document entry in a user interface
- suitable for document validation before application processing
- techniques for specifying, restricting and extending lists for the purposes of validation

Masquerading meta data when restricting a large list to a subset of values

- the validation needs to match an instance's use of large list meta data to a declaration of a subset list using subset list list-level meta data
- the subset list list-level meta data necessarily is different than the list-level meta data of the list from which it is derived
- the subset list masquerades as the list from which it is derived so that instance-level meta data doesn't use the custom subset list list-level meta data

ISO/IEC 19757-3 Schematron deployment

- as supplied, the methodology reports context/value constraint violations in simple text
- Schematron can alternatively be deployed with different available reporting techniques

The principles of context/value association are as follows:

- XML documents have information items that need to be validated
  - the locations (contexts) of those items can be addressed using XPath addresses
- genericode files have values and list meta data to use for validation
  - the locations of those files can be declared with URL addresses
  - the identity of each list is uniquely specified in order to be referenced multiple times
- an association marries a document context with a set of genericode files
  - each XPath document context is specified with the identities of the genericode declarations
- validation checks values found in document contexts against genericode files linked by the association for the document context
  - any present meta data in the document context is checked with the available genericode meta data

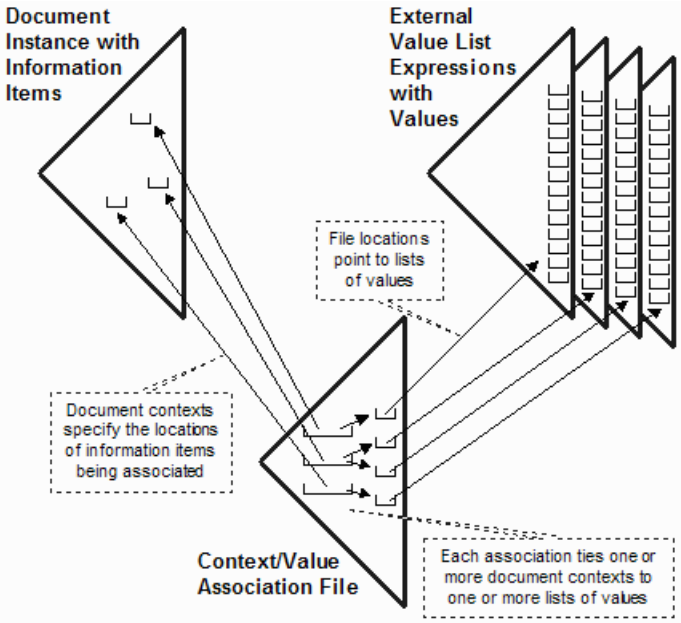


Figure 7. Context/value association

Appropriate for constraining data entry application user interfaces

- used as a front end to a user preventing the data entry of different values
  - drop-down lists
  - radio buttons
  - check boxes
- the end result of editing an instance is that the values are all from the associated lists
- the value-level meta data can be presented to the user
  - assists the user in choosing which value or values to use
- the options to include instance-level meta data should be offered
  - reflects the list-level meta data for the list from where the values are taken

Appropriate for constraining data validation

- used as a front end to an application that implements the logic for all possible values
- selective association for business scenarios prevents the application from acting on inappropriate values for a given transaction
  - relationships between specific partners may be different
  - different profiles of using documents may constrain particular values

Only the CVA vocabulary is standardized by OASIS, not how it is used

- the file format and the semantics represented by the elements and attributes are being standardized by OASIS

- any implementation is considered out of scope of the committee work

4.3. Using context/value association for validation

Separates structural/lexical validation from value validation

- an XML document is checked using a two-step process
- the first pass for structural and lexical validation passes
- the second pass reports that a coded value used for a currency is unexpected
- the document structure and lexical content can be constrained by standardization
  - e.g. the UBL technical committee publishes normative W3C schemas
- the document controlled-value content is constrained by business requirements between trading partners
  - e.g. the UBL committee publishes default coded value checks
    - `defaultCodeList.xsl`
  - trading partners can use this value validation methodology to create their own value checking second-pass process

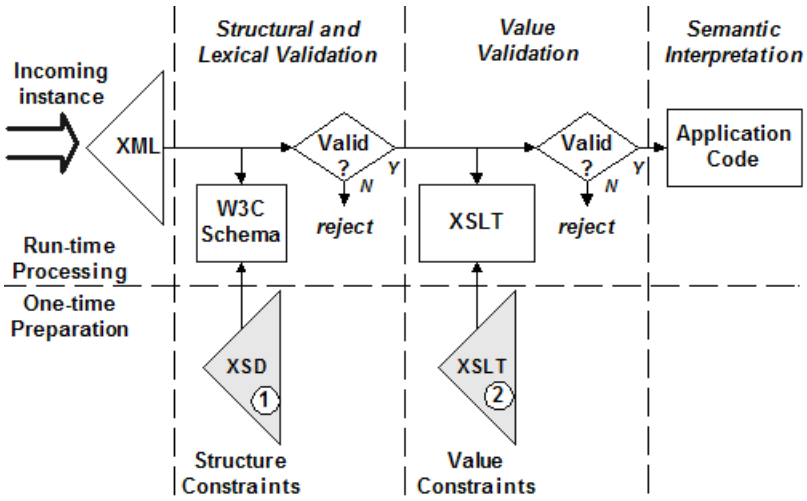


Figure 8. Two-step validation

Document arrives at application unchanged

- validation only confirms the use of structure and content, without modifying it

Second pass results meaningless without first pass being successful

- the values must be correctly found and correctly formed before checking the actual values produces an accurate result

`Crane-CVA2sch` package from Crane Softwrights Ltd. web site

- historically developed in the OASIS UBL Technical Committee

- moved into the OASIS Code List Representation Technical Committee
- moved out of the OASIS Code List Representation Technical Committee
  - the committee decided to focus on file formats and not methodologies
  - intellectual property returned to Crane Softwrights Ltd.
- Crane is donating CVA2sch to an Apache Schematron project

A methodology for code list and value validation based on ISO/IEC 19757-3 Schematron

- an information item is asserted to have one of an allowed set of predetermined values
  - a failed assertion is a value validation error
- assertions are derived from context/value associations

Schematron is usually implemented using the Extensible Stylesheet Language (XSLT)

- the supplied Schematron stylesheet for stylesheets is a copy of the publicly-available reference XSLT implementation
  - `http://www.schematron.com`
  - the methodology supplies a wrapper stylesheet for the reference skeleton
- other non-XSLT implementations of Schematron exist
  - e.g. Amara/Scimitar implements ISO Schematron in Python
    - `http://uche.ogbuji.net/tech/4suite/amara`
    - same architecture as reference XSLT implementation in that Scimitar is a Python program that writes a Python program that performs the validation

The XSLT generated to implement the Schematron assertions is used as the second pass of validation to test XML instances for having correct controlled-vocabulary values

- the testing relies on the first-pass structural validation, having already confirmed the structure and lexical values used in the instance
- without the first pass confirming the accurate presence of information items, the second pass is meaningless

The methodology supports the incorporation of any number of sets of Schematron assertions

- ISO Schematron supports the inclusion of multiple schema fragments into a single schema expression
- business rules related or unrelated to code lists may be expressed as Schematron assertions
  - the trading partner schema can then include business rules in addition to coded value rules

Overview of the process to prepare the second pass value validation XSLT stylesheet:

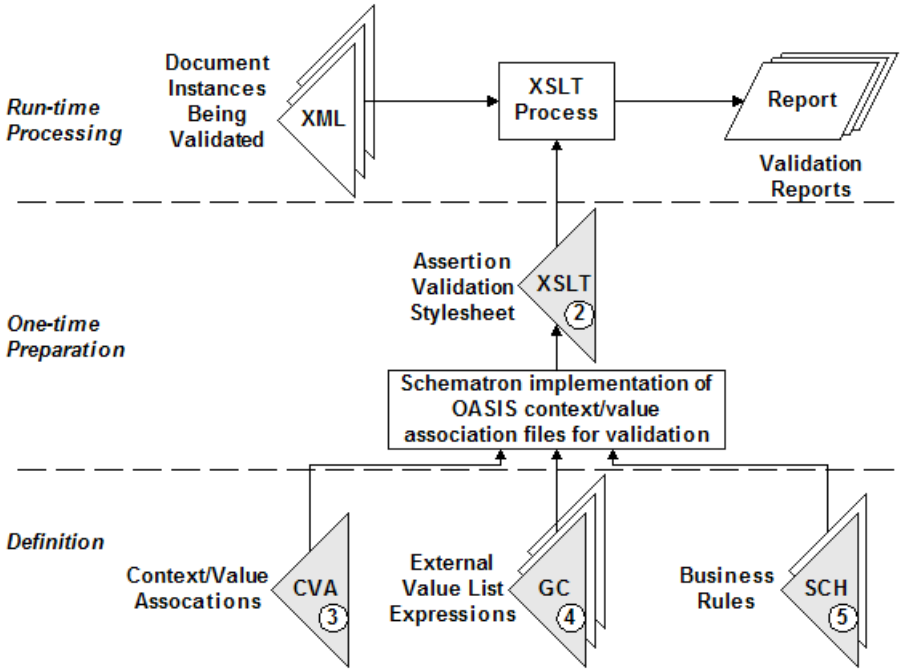


Figure 9. Second-pass value-validation artefact creation

- the circled labels in the diagram are indicated by the parenthesized numbers
- the inputs:
  - (3) the specification of contexts uses the context/value association XML vocabulary defined by the OASIS Code List Representation TC
  - (4) the specification of coded values uses the genericode vocabulary defined by the OASIS Code List Representation TC
  - (5) supplemental business rules are specified using ISO/IEC 19757-3 Schematron
- the output:
  - (2) an XSLT stylesheet (or some other implementation of Schematron assertion checking)

Recall Figure 8

- the XSLT created here (2) plugs in to the two-step validation process

Recall Figure 7

- all three documents on that diagram are shown here as instances being validated, the context value association files and the external value list expressions

4.4. Rendering context/value association files

4.4.1. Rendering context/value association files

Some audiences do not appreciate having to read raw XML to interpret the contents

- angle brackets are distracting
- the volume of XML markup overwhelms the information content of the instance

Crane Softwrights Ltd. has published free developer resources with which to render a context/value association file to HTML

- XSLT 1.0 stylesheet: `Crane-assoc2html.xsl`
- standalone production of HTML from context/value association file
- browser-based viewing of HTML from context/value association file

The rendering of the context/value association file includes the rendering of the referenced genericode files

- the stylesheet for the context/value association file automatically imports the stylesheet for a genericode file

See Section 3.2.3 for the technique of embedding a stylesheet association processing instruction

Instead of embedding simple text for documentation, one can use rich markup

- the Crane stylesheet supports the embedding of HTML
- note in the `order-constraints-doc.xml` example the declaration of the HTML namespace and use of the "`x:`" namespace prefix

```
<?xml-stylesheet type="text/xsl" href="Crane-assoc2html.xsl"?>
<ValueListConstraints
  xmlns="urn:oasis:names:tc:ubl:schema:Value-List-Constraints-0.8"
  xmlns:cbc="urn:oasis:...:CommonBasicComponents-2"
  xmlns:cac="urn:oasis:...:CommonAggregateComponents-2"
  xmlns:x="http://www.w3.org/TR/REC-html40"
  xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  id="urn:x-illustration"
  name="code-list-rules">
  <Title>
    Illustration of code list constraints -
    <x:samp>order-constraints.cva</x:samp>
  </Title>
  <Identification>
    <x:pre>
    $Id: order-constraints-doc.cva,v 1.1 2007/02/10 02:24:18
    G. Ken Holman Exp $
    </x:pre>
  </Identification>
```

```
<Description>
  <x:p>
    This is an illustrative example of all of the features of
    specifying the context/value constraints that one can express
    for XML documents.
  </x:p>
  <x:p>
    The validation requirements for this contrived scenario are as
    follows:
    <x:ul>
      <x:li>the UN/CEFACT currency code list is restricted to be
      only Canadian and US dollars:</x:li>
      <x:li>the seller must be in the US</x:li>
      <x:li>the buyer may be in either Canada or the US</x:li>
      <x:li>the definition for Payment Means is extended to include
      both UBL definitions and additional definitions</x:li>
    </x:ul>
  </x:p>
</Description>
...
```

4.4.2. Standalone production of an HTML rendering

The rendering of the example of a context/value association file with embedded HTML on Section 4.4.1:

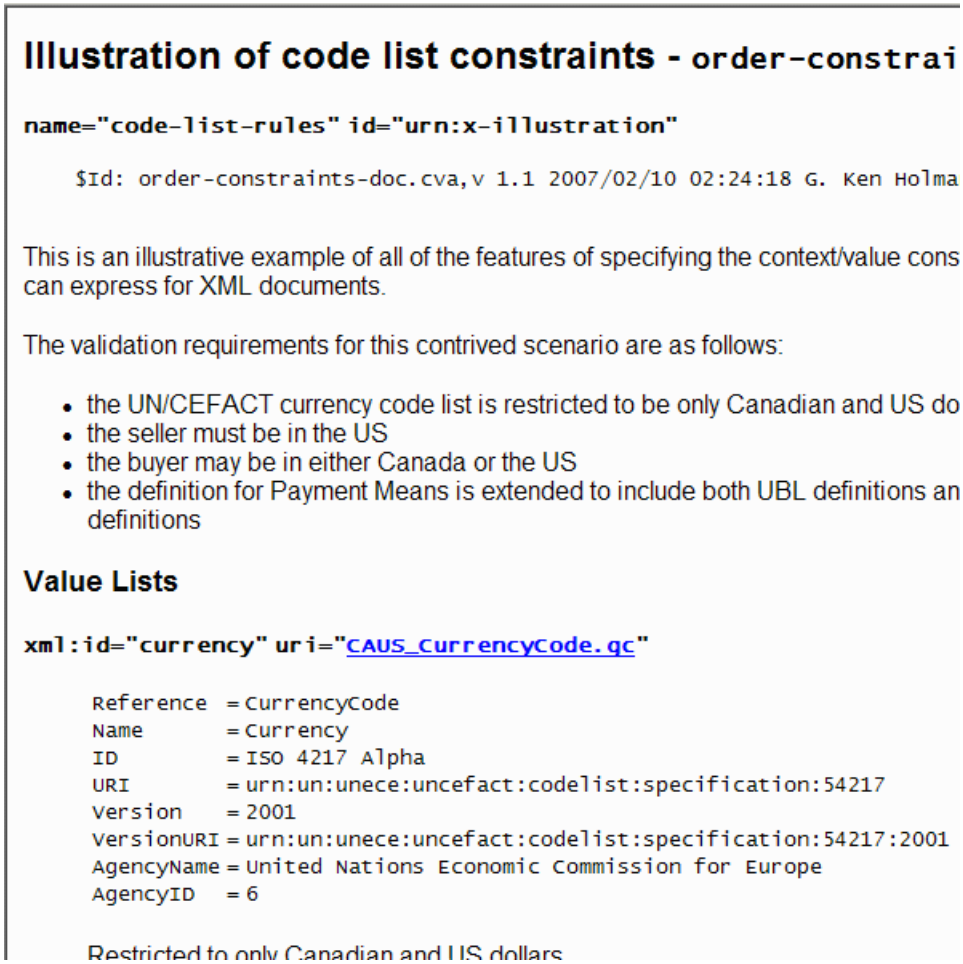


Illustration of code list constraints - order-constrai

name="code-list-rules" id="urn:x-illustration"

$Id: order-constraints-doc.cva,v 1.1 2007/02/10 02:24:18 G. Ken Holma

This is an illustrative example of all of the features of specifying the context/value cons
can express for XML documents.

The validation requirements for this contrived scenario are as follows:

- the UN/CEFACT currency code list is restricted to be only Canadian and US do
- the seller must be in the US
- the buyer may be in either Canada or the US
- the definition for Payment Means is extended to include both UBL definitions an
  definitions

Value Lists

xml:id="currency" uri="CAUS_CurrencyCode.gc"

Reference	= CurrencyCode
Name	= Currency
ID	= ISO 4217 Alpha
URI	= urn:un:unece:uncefact:codelist:specification:54217
Version	= 2001
VersionURI	= urn:un:unece:uncefact:codelist:specification:54217:2001
AgencyName	= United Nations Economic Commission for Europe
AgencyID	= 6

Restricted to only Canadian and US dollars

Figure 10. HTML rendering of a context/value association file containing HTML

Note how the HTML browser renders the embedded HTML constructs.

Testing XSLT

Tony Graham
Menteith Consulting Ltd
<Tony.Graham@MenteithConsulting.com>

1. Overview

Creating a working stylesheet may seem like an end in itself, but once it's written you may want it to run faster or you may not be sure that the output is correct (And if you are sure, how sure are you?).

Profilers, unit test frameworks, and other tools of conventional programming are similarly available for XSLT but are not widely used. This presentation surveys the available tools for ensuring the quality of your XSLT.

There is no one-size-fits-all solution when looking for tools. For example, if you are using Saxon and Ant, then you are looking for a different set of tools than if you are using libXSLT and Makefiles.

2. Profilers

Profilers are in wide use for conventional programming, and several XSLT and XSL profilers are available. However, it is a truism of XSLT programming that different XSLT processors have their own strengths and weaknesses, so if you are profiling your stylesheet, it is important to profile it running on the same XSLT processor as you will use in production.

Some XSLT processors, such as Saxon and libXSLT, have their own profiling mechanisms, and several XSLT editors or IDEs, such as <oXygen/>, Stylus Studio, and XML Spy, provide built-in profilers what work with a number of different XSLT processors.

Profiling XSLT is not an exact science since:

- The execution time for a template includes the execution times of all of the templates that it calls; the processor and IDE vendors do their best to separate the two when reporting timing.
- Results depend on the state of the machine; running the stylesheet multiple times in succession generally means the later runs are faster than the earlier as the program and its libraries are already in memory.
    - Both Saxon and xsltproc provide a command-line switch for running the transformation time multiple times to counteract irregularities in the timing of a single run.

- The time taken by a particular template may depend as much on the current node list as on the structure of the template.

The presentation will describe the features and applicability of the different XSLT and XSL profilers available as well as a few hints and tips about how to get the best out of your profiler: such as what to do when the profiler reports that your hottest hotspot is a literal result element instead of the complicated XPath selector that you expected.

2.1. XML IDEs

XML IDEs such as <oXygen/>, Stylus Studio, and XMLSpy support profiling and debugging of XSLT. Figure 1 shows <oXygen/> 8 profiling information display.

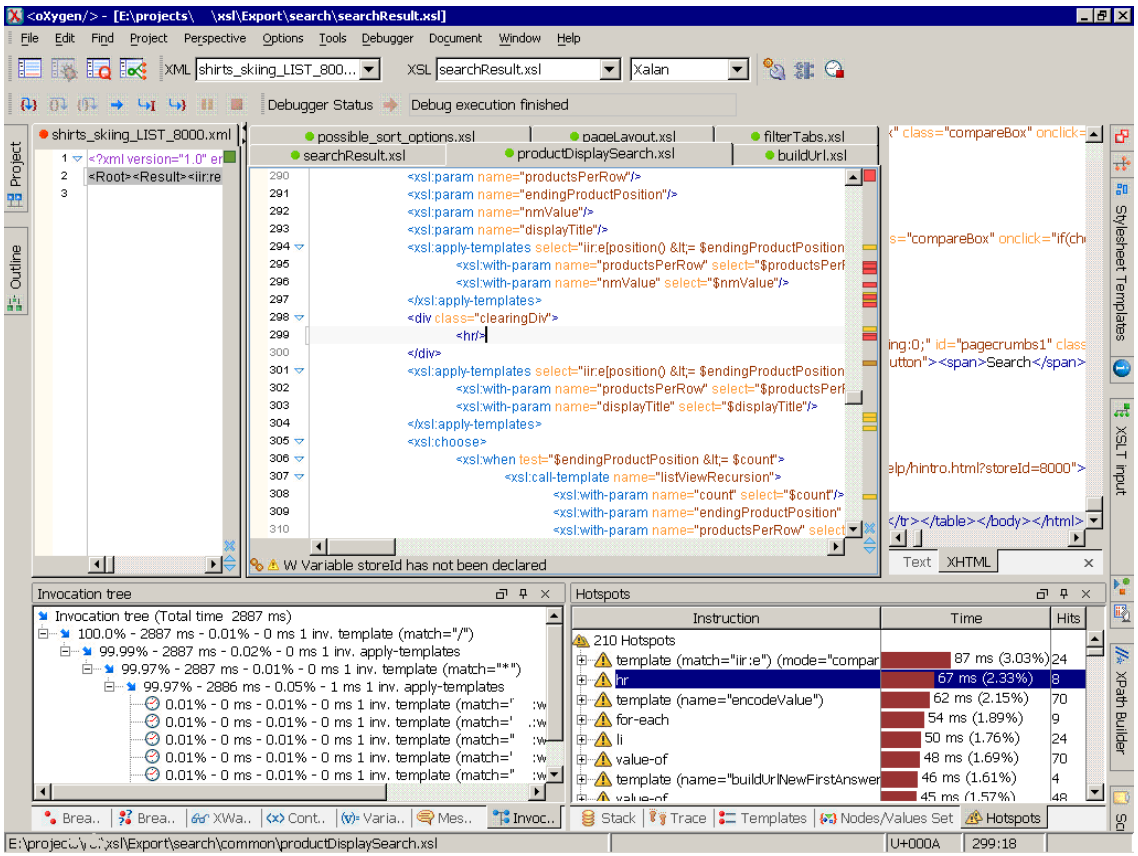


Figure 1. <oXygen/> 8 XSLT profiling

2.2. xsltproc

2.2.1. Profiler

The xsltproc XSLT 1.0 processor supports a --profile switch that triggers a dump of profiling information. The example below shows the first few lines of output from profiling one of the code-generating stylesheets for the xmlroff XSL formatter:

```
xsltproc --profile --stringparam dump-info dump-info.xml fo-context-dump.xsl ▶
xslspec.xml
number       match                name      mode  Calls Tot 100us Avg

    0            property-to-context-property-merge
                                                172 426224   2478
    1               fo-context-c-file              1 315848 315848
    2            property-to-slist-foreach-if
                                                172 143712    835
    3            property-to-context-property-copy
                                                172 107006    622
...
```

2.2.2. Debugger

xsltproc has a command-line debugger.

2.3. Saxon

2.3.1. Profiling

Saxon provides a profile through a two part process:

1. Run Saxon with the -TP switch, writing the error output to a file

   ```
   java -jar dir/saxon9.jar -TP source stylesheet 2>profile.xml
   ```

2. Run the timing-profile.xsl stylesheet from the saxon-resources.zip file (available from http://www.saxonica.com/) to create a HTML report:

   ```
   jjava -jar dir/saxon9.jar profile.xml timing-profile.xsl >profile.html
   ```

Figure 2 shows a screenshot of the report generated from running the same stylesheet as in the Section 2.2 example. Notice the first two "hotspots" are the same as for xsltproc, but after that the result start to diverge.

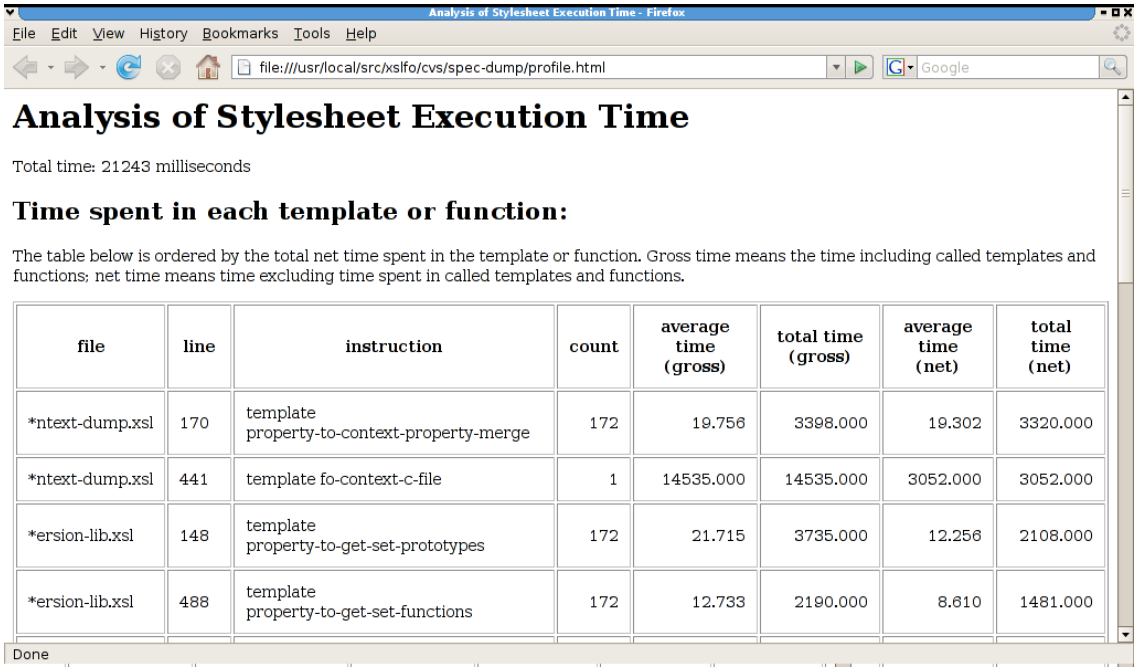


Figure 2. Saxon profile report

3. Test Frameworks

Not all testing frameworks are unit testing frameworks, and not all tests are unit tests.

Unit tests – tests written from the perspective of the programmer – came to prominence with the rise in popularity of the Extreme Programming (XP) methodology in the late 1990s. Programmers have always recognised that they should write tests for their code, but that hasn't always meant that they do. Writing tests before writing the code is central to XP, so the publicity about XP brought unit testing to the attention of many programmers, irrespective of whether they adopted all, some, or none of the XP methodology. The practise can be separated from the rest of the XP bag of tricks and referred to as Test-Driven Development (TDD).

Unit testing is perhaps most commonly associated with the Java programming language since XP's creators also wrote the JUnit unit testing framework and since Ant, the ubiquitous Java-based build tool, makes it easy to run JUnit tests and generate HTML reports of the results. Unit testing tools are available for a wide variety of programming languages, including XSLT, but current awareness of XSL and XSLT unit testing tools is limited.

Your choice of XSLT unit testing framework depends less on your XSLT processor than it does on your XSLT version and your testing approach. There are already several unit testing frameworks specific to XSLT 2.0 (which will limit your choice of XSLT processor), but otherwise your choice depends on whether you want to

work purely in XSLT, within Ant, with Ant and JUnit, with just JUnit. Again, your choice will depend on what other tools you are using.

XSLT unit testing frameworks include:

- XSLTunit – http://xsltunit.org/
- Juxy – http://juxy.tigris.org/
- Unit Testing XSLT – http://www.jenitennison.com/xslt/utilities/unit-testing/
- tennison-tests – http://tennison-tests.sourceforge.net/
- <XmlUnit/> – http://xmlunit.sourceforge.net/
- uft-x – http://utf-x.sourceforge.net/
- XTS – http://www.fgeorges.org/xslt/xslt-unit/

A testing framework of a different kind is the XSLV static validation tool available from http://www.brics.dk/XSLV/.

XSpec (http://code.google.com/p/xspec/) is a Behavior Driven Development (BDD) framework for XSLT.

3.1. Effectiveness of Unit Testing

Few people, if any, would claim that unit testing is the silver bullet that will kill all your software defects. The industry data summarised in Software Estimation: Demystifying the Black Art [1] indicates it is most often effective in removing 30% of the defects in the code being tested.

Table 1. Defect-removal Rates (from [1])

Removal Step	**Lowest Rate**	**Modal Rate**	**Highest Rate**
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modelling or prototyping	35%	65%	80%
Personal desk checking of code	20%	40%	60%
Unit test	*15%*	*30%*	*50%*
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%

Removal Step	Lowest Rate	Modal Rate	Highest Rate
High-volume beta test (>1,000 sites)	60%	75%	85%

As with so many things, you mileage may vary, and historical data from your own organisation will be a better indication of how effective this, or any technique, is for you.

3.2. Unit Testing Wish-List

My personal wish-list for an XSLT unit testing framework includes:

- Ability to assert that a message was (or was not) output.

  If all an xsl:template does is emit a message, the result from the template will, in XSLT 2.0 terms, be just an empty sequence. There are times when it is useful to check that the template took the path less travelled and emitted a message rather than just failed to produce a result.
- Works with keys

  When the unit testing framework and test input data are both in a stylesheet that imports the style sheet being tested, there is no input document for which to generate key data. Testing with uninitialised keys just leads to frustration.
- Test stylesheet as a whole

  A framework should be able to test by running the whole stylesheet as well as by exercising individual templates.
- Report results from multiple unit test files

  Most of the unit test frameworks are designed to run a single file containing the unit tests. When there are a lot of tests or when the spec for the transform has multiple parts, it makes sense to split the unit tests into multiple parts, each in a different file. A framework should be able to run multiple files of unit tests in one invocation and report on the combined results of all the tests.
- Maintainers respond to bug reports and enhancement requests
- Test data and assertions written in XML

  Several of the unit testing frameworks use tests written in Java and/or the tests are run using Ant. However, I do more than just jobs that use one or both of Java and Ant (what I do use at any point is often what my clients are currently using), so I personally prefer that tests can be written in XML and to have the option of running tests from the command line so that it's not necessary for users to know a particular programming language or use a particular build tool to be able to create and run unit tests.

3.3. Why use more that XSLT for testing XSLT?

Why should you prefer an XSLT unit testing framework that uses more that just XSLT? Two reasons: `xsl:message` and multiple output documents.

Most XSLT unit testing frameworks are written purely in XSLT; that is, the framework uses an XSLT processor and XSLT stylesheet to run the tests (or a massaged form of the tests) on the stylesheet under test. The test result output is typically XML that is processed by another stylesheet to produce HTML for viewing.

Using XSLT to test XSLT may be partly explained by XSLT being an XSLT practitioner's favourite tool and partly by Eric van der Vlist's XSLT-based XSLTunit being the first known framework. Jeni Tennison's possibly unnamed framework is also purely XSLT (though XSLT 2.0 instead of the XSLT 1.0 in XSLTunit), and it inspired the Tennison Tests framework, which automates running the multiple tests but hasn't tinkered with the XSLT nature of the tests, and also XTC.

By comparison, the frameworks that rely on compiling code to run tests are, it seems, much less well known. They include Juxy, which compiles Java code for JUnit tests, and XMLUnit, which has versions in Java for use with JUnit and C# for use with NUnit.

IMO, the extra-XSLT frameworks have the advantage that you can (or should be able to) make assertions about aspects of the stylesheet to which a pure XSLT framework is either blind or oblivious.

Inasmuch as you typically use `xsl:message` to output messages when there's an error in the source document or there's something that the stylesheet can't handle, you should want to make assertions about whether or not a message has been emitted. This is particularly true when you are writing "dirty" tests that you expect will trigger error handling (as opposed to "clean" tests that you expect to work), and even more true when you expect to execute a `xsl:message` that has '`terminate="yes"`'.

Pure XSLT frameworks don't provide a way to tell when a message is emitted, and any `<xsl:message terminate="yes">` will terminate the testing framework along with the stylesheet under test, leaving you with nothing. A good extra-XSLT framework will let you make assertions about `xsl:message` and will not evaporate when the stylesheet terminates itself.

When a stylesheet may create multiple result documents, you probably want to make assertions about the existence and content of those result documents. That could be left to a post-process, but if the file names and the content relate to the content of the source document, it is simpler and more self-contained when you could make your assertions in the same unit tests as you use to make assertions about the rest of the operation of the stylesheet. A pure XSLT framework can't provide any indication that a result document was or was not created, though it may be possible to access a secondary result document as part of the unit tests

(provided the XSLT processor finishes writing the secondary result document while the transformation-that-is-the-running-of-the-unit-tests is still in progress). An extra-XSLT framework typically lets you make assertions after the stylesheet-under-test has finished.

So while there are some aspects of XSLT processing, such as what happens on a `xsl:message`, that you as a stylesheet writer don't need to concern yourself about, you as a stylesheet tester do want to know about them, so you need more that just an XSLT processor in the framework for your unit tests.

3.4. XSLTunit

XSLTunit (http://xsltunit.org/) by Eric van der Vlist of Dyomedea is the grandfather of XSLT unit testing frameworks. When asked, Eric described it as "stable, rustic and mature". It has influenced, both positively and negatively, the development of several other testing projects.

XSLTunit tests are written in a stylesheet. The stylesheet imports both the stylesheet being tested and the XSLTunit "xsltunit.xsl" stylesheet. The unit tests are written within a template that matches the document root (so it is the first rule executed when the combined stylesheets are run).

This example from the XSLTunit web site illustrates how elements in the XSLTunit namespace do the work and an xsl:apply-template invokes the stylesheet being tested to produce a result that is to be compared against the expected result.

```
<xsl:template match="/">
  <xsltu:tests>
    <xsltu:test id="test-title">
      <xsl:call-template name="xsltu:assertEqual">

        <xsl:with-param name="id" select="'full-value'"/>

        <xsl:with-param name="nodes1">
          <xsl:apply-templates ▶
select="document('library.xml')/library/book[isbn='0836217462']/title"/>
        </xsl:with-param>
        <xsl:with-param name="nodes2">
          <h1>Being a Dog Is a Full-Time Job</h1>
        </xsl:with-param>
      </xsl:call-template>
    </xsltu:test>
…
  </xsltu:tests>
</xsl:template>
```

The result from running XSLTunit is an XML document indicating the success or failure of each test. When a test asserting equality with an expected result fails, the

output also includes a diff of the expected and actual results. The following is the result from running the sample files from the XSLTunit web site:

```
<xsltu:tests xmlns:xsltu="http://xsltunit.org/0/">
  <xsltu:test id="test-title">
    <xsltu:assert id="full-value" outcome="passed"/>
  </xsltu:test>
  <xsltu:test id="test-title-reverted">
    <xsltu:assert id="non-empty-h1" outcome="passed"/>
  </xsltu:test>
  <xsltu:test id="XPath-expressions">
    <xsltu:assert id="h1" outcome="passed"/>
    <xsltu:assert id="value" outcome="passed"/>
  </xsltu:test>
</xsltu:tests>
```

3.5. Juxy

Juxy (http://juxy.tigris.org/), by Pavel Sher, describes itself as "a library for unit testing XSLT stylesheets from Java". It states that it is best suited for the projects where both Java and XSLT are used simultaneously.

In contrast to XSLTunit[2] and its descendants, Juxy tests are written in Java. Tests can be written to run standalone, for use with JUnit, or for use with any other (probably Java-based) testing framework. Tests for use with JUnit are the most likely use for Juxy, both because the tests are less verbose and so are easier to both read and write and because many people are already using JUnit for testing Java code.

The input being tested can be from a file, a Document object or a String, as shown in the following JUnit-specific example excerpted from the Juxy website:

```
public class SampleTestCase extends JuxyTestCase {
  public void testListTransformation() {
      newContext("stylesheet.xsl");
      context().setDocument("" +
          "<list>" +
          " <item>item 1</item>" +
          " <item>item 2</item>" +
          " <item>item 3</item>" +
          "</list>");
      Node result = applyTemplates();
      xpathAssert("text()", "item 1, item 2, item 3").eval(result);
  }
}
```

[2] /trac_consulting/wiki/TestingXSLT/XSLTunit

3.5.1. XML Format

At the time of this writing, Juxy CVS contains a stylesheet, by Tony Graham, for converting tests in XML format into Java, which is then compiled in the same manner as conventional Juxy tests. The following example has the same effect as the previous Java example:

```
<test name="MoreThanOneElementInTheList_ApplyTemplates">
  <document select="/list"><list><item>first item</item><item>second ▶
item</item><item>third item</item></list></document>

  <apply-templates select="/list"/>
  <assert-equals>
    <expected>first item, second item, third item</expected>
  </assert-equals>
</test>
```

Note that the requirement to translate tests into Java methods for use with JUnit currently limits test names to strings that (when prepended with `'text'`) make valid method names for use with JUnit.

Figure 3 shows the JUnit report from running the test cases generated from the complete XML file for the above example.

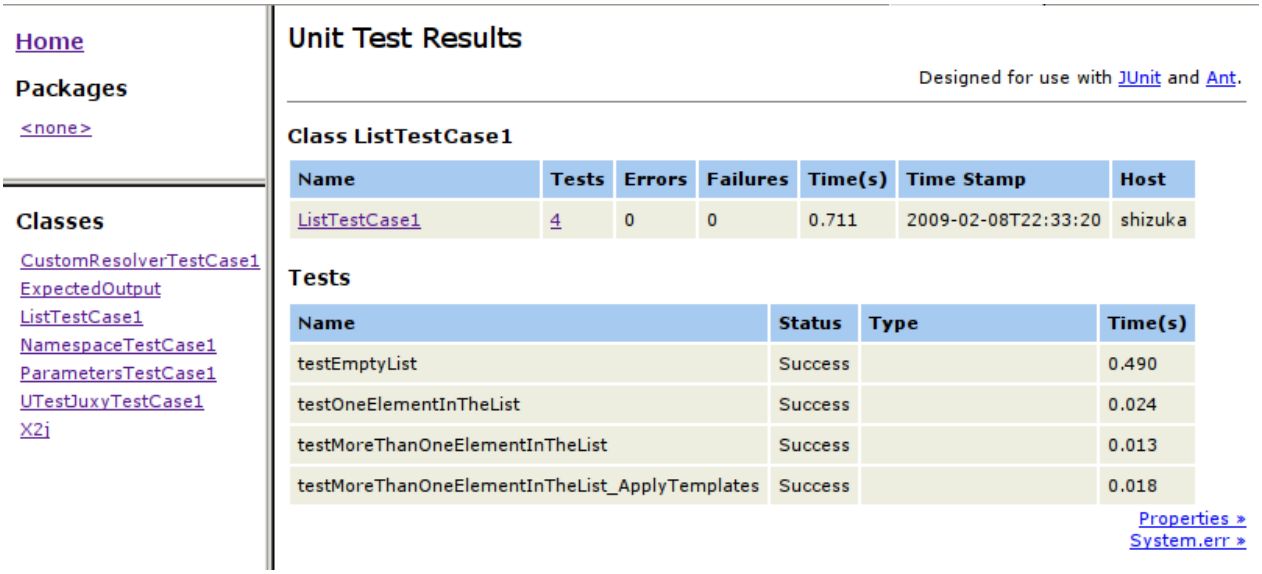


Figure 3. Juxy test report

The advantage of generating tests as Java and running the stylesheet under test from a non-XSLT framework is that the framework can catch error conditions that would blow away an all-XSLT framework. For example, the following test asserts that `x2j.xsl` will terminate when the source document is the content of the `<document>` element:

```
<test name="Stylesheet">
  <stylesheet href="xmlTest/x2j.xsl"/>
  <document><stylesheet href="href"><root/></stylesheet></document>
  <assert-error>
    <apply-templates/>
  </assert-error>
</test>
```

It is intended to add the ability to make assertions about the content of `xsl:message` output and the names of files that are read and written.

3.6. Unit Testing XSLT

Jeni Tennison's unit testing framework, available from http://www.jenitennison.com/xslt/utilities/unit-testing/ under the title "Unit Testing XSLT", is a pure XSLT 2.0 solution where unit tests may be either in the stylesheet being tested or in a separate file. The following example from Jeni's web site shows a simple test of an XSLT 2.0 function:

```
<test:tests>
  <test:test>
    <test:param name="number" select="2" />
    <test:expect select="4" />
  </test:test>
</test:tests>

<xsl:function name="eg:square" as="xs:double">
  <xsl:param name="number" as="xs:double" />
  <xsl:sequence select="$number * $number" />
</xsl:function>
```

A stylesheet containing tests and templates is transformed using a provided stylesheet to generate a new, standalone stylesheet that contains only the tests and that imports the original stylesheet. Running this stylesheet (irrespective of what you use for input) runs the tests and produces an XML result file. This result is then transformed using another provided stylesheet to produce a HTML report (defaulting to using Jeni's distinctive purple and green colour scheme) that summarises the results.

Figure 4 shows the report generated using the above test plus a second test that is forced to fail (since "2 * 2" does not equal "5") to show the details that are provided for failed tests.

TEST REPORT

Stylesheet: **usr/local/src/unit-testing/square.xsl**

Tested: 4 April 2007 at 23:48

Summary

Passed 1/2

Test ID	Title	Success/Total
square		1/2

Square

Test ID	Test Title	Success/Failure
square.1		Success
square.2		Failure

Tested XSLT

```
<xsl:function xmlns:xml="http://www.w3.org/XML/1998/namespace"
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:test="http://www.jenitennison.com/xslt/unit-test"
              xmlns:xsd="http://www.w3.org/2001/XMLSchemaAlias"
              xmlns:eg="http://example.com/"
              name="eg:square"
              as="xs:double">
   <xsl:param name="number"
                 as="xs:double" />
   <xsl:sequence select="$number * $number" />
</xsl:function>
```

Square.2

Parameters

Name	Value
number	2

Results

Expected Result	Actual Result
5	4

Figure 4. Unit test report

3.7. tennison-tests

The tennison-tests project on SourceForge marries Jeni Tennison's unit testing stylesheets with an Ant task for running one or more unit test files and producing reports. This framework is most likely to be useful to someone used to using Ant and already using it for unit testing non-XSLT code. As the tennison-tests web page puts it:

> *The Tennison Tests (XSLT Unit Testing) project aims to harvest the best of both worlds, allowing XSLT Developer's to write their tests in XML and appease the nUnit camp by providing an easy integration into automated build tools, specifically Ant.*

Using tennison-tests requires adding the task's definition and then using it in an Ant target, for example:

```
<!-- ============================================================= -->
<!-- Targets for running unit tests                                -->
<!-- ============================================================= -->

  <!-- Defines the 'tennison-tests' custom Ant task. -->

  <taskdef name="xslttest"
    classname="com.jenitennison.xslt.unittest.XSLTTest"
    classpath="${basedir}/ant-xslttest-1.0.0/ant-xslttest-1.0.0.jar"/>

  <!-- Executes all unit tests in 'test'. -->
  <target name="test" depends="init">
    <xslttest src="${basedir}/ant-xslttest-1.0.0/main/src/xslt"
       target="build/test"
       generate="true">
      <fileset dir="${basedir}/test">
     <include name="*.xml" />
      </fileset>
      <factory name="net.sf.saxon.TransformerFactoryImpl"/>
    </xslttest>
  </target>

  <!-- Executes a single unit test. -->
  <!-- Example usage:
          ant -Dtest=2-6.xml test.single
  -->

  <target name="test.single" depends="init">
    <xslttest src="${basedir}/ant-xslttest-1.0.0/main/src/xslt"
       target="build/test"
       generate="true">
```

```
      <fileset dir="${basedir}/test">
     <include name="${test}" />
      </fileset>
      <factory name="net.sf.saxon.TransformerFactoryImpl"/>
    </xslttest>
  </target>
```

The generated reports appear identical to those produced by Jeni's original stylesheets (apart from using a different colour scheme). While the Ant task can run multiple unit test files on one invocation, as yet it does not produce a summary report of all the individual unit test file's results.

3.8. <XmlUnit/>

<XmlUnit/>, from http://xmlunit.sourceforge.net/, is available in two forms: a Java framework (for use both with and without JUnit) and a less well-developed C# framework for use with NUnit. The Java framework can test assertions about XML documents (and even badly-formed HTML), validate documents, and compare two documents as well as test assertions about the result of XSLT transformations.

Tests are written as methods of Java (or C#) classes. The following abbreviated example from the <XmlUnit/> documentation shows a test where the result of an XSLT transformation is compared to the expected output.

```
public void testXSLTransformation() throws Exception {
    String myInputXML = "...";
    File myStylesheetFile = new File("...");
    Transform myTransform = new Transform(myInputXML, myStylesheetFile);
    String myExpectedOutputXML = "...";
    Diff myDiff = new Diff(myExpectedOutputXML, myTransform);
    assertTrue("XSL transformation worked as expected", myDiff.similar());
}
```

This example shows input and expected output coming from a String or a File. They may instead be a DOM node or a SAX InputSource.

3.9. Unit Testing Framework – XSLT (UTF-X)

UTF-X, available from http://utf-x.sourceforge.net/, is a Java-based framework where tests can be run from the command line, from an Ant build file, or using JUnit. Tests are written as XML. The following example from from the UTF-X web site asserts that the result of processing the content of the utfx:source element will match the content of the utfx:expected element.

```
<utfx:test>
  <utfx:name>sect1 with title only</utfx:name>
  <utfx:assert-equal>
```

```
      <utfx:source validate="yes">
        <section id="section1">
          <heading>Section 1</heading>
        </section>
      </utfx:source>
      <utfx:expected validate="yes">
        <a name="section1" />
        <h1>Section 1</h1>
      </utfx:expected>
    </utfx:assert-equal>
  </utfx:test>
```

UTF-X includes a test generator that can generate a test definition file for an existing stylesheet.

3.10. XTS

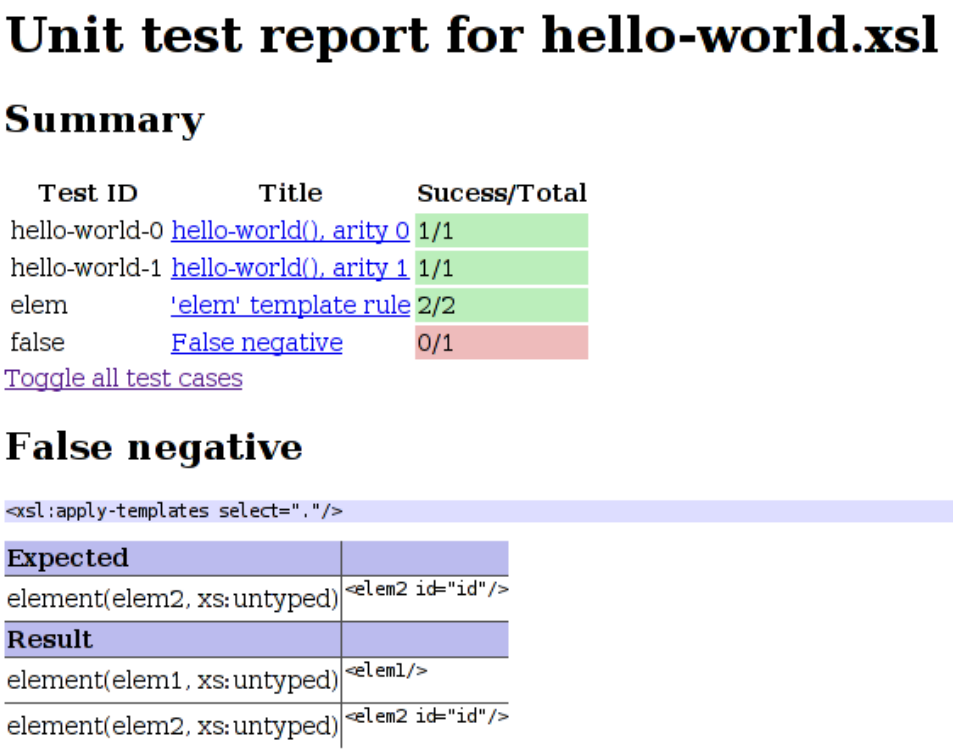
XTS, by Florent Georges, is available from http://www.fgeorges.org/xslt/xslt-unit/. XTS is an XSLT 2.0 framework that is capable of testing both XSLT 2.0 and XQuery. A single test comprises an assertion of the expected result followed by a sequence constructor, as in the following example from the XTS web site that illustrates testing a function from the stylesheet under test:

```
<t:tests>
  <t:title>hello-world()</t:title>
  <t:test>
    <t:expect select="'Hello, world!'"/>
    <xsl:sequence select="hw:hello-world()"/>
  </t:test>
</t:tests>
```

The sequence constructor – a single xsl:sequence in this case, though it could be more complicated – is compared for equality with the expected result. Alternatively, the expected result could be written as an XPath expression to evaluate or, when used with Florent's 'error-safe' extension for Saxon, an assertion of an error that should be thrown when evaluating the sequence constructor.

The XML file containing the tests is transformed into a test stylesheet that imports the stylesheet being tested. That stylesheet, when run, ignores its XML input and runs named templates corresponding to the tests in the original test XML file.

The output of the test stylesheet is an XML file that can be transformed into a HTML report. Figure 5 shows the sample report from the XTS web site.



Unit test report for hello-world.xsl

Summary

Test ID	Title	Sucess/Total
hello-world-0	hello-world(), arity 0	1/1
hello-world-1	hello-world(), arity 1	1/1
elem	'elem' template rule	2/2
false	False negative	0/1

Toggle all test cases

False negative

```
<xsl:apply-templates select="."/>
```

Expected	
element(elem2, xs:untyped)	<elem2 id="id"/>
Result	
element(elem1, xs:untyped)	<elem1/>
element(elem2, xs:untyped)	<elem2 id="id"/>

Figure 5. XTS report

3.11. XSpec

XSpec (http://code.google.com/p/xspec/), by Jeni Tennison and contributors, is a Behavior Driven Development (BDD) framework for XSLT. It is based on the Spec framework of RSpec, which is a BDD framework for Ruby.

XSpec consists of a syntax for describing the behaviour of your XSLT code, and some code that enables you to test your XSLT code against those descriptions.

Some aspects in which XSpec differs from Jeni's earlier Section 3.6 framework are:

- Tests are defined as 'scenarios' where you describe what should happen and define what to expect, whereas in the other framework, tests were identified by ID only.

  However, the ability to label tests and provide strings associated with assertions is not unique to XSpec.
- Scenarios may be nested, so multiple scenarios may inherit a common context
- Scenarios documents may import other scenario documents, promoting reuse
- Scenarios may be marked 'pending' (and a description may be provided): pending scenarios are not run, but are reported as pending in the test report, so they remain visible. This compares favourably to other frameworks where you would have to comment out any tests that can't be run at that time.

3.11.1. Example

For this stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <xsl:output method="xml"/>

  <xsl:template match="/">
          <m>Hello world!</m>
  </xsl:template>

  <xsl:template match="unused">
          <m>You can't see me!</m>
  </xsl:template>

</xsl:stylesheet>
```

This description document describes a single scenario:

```
<s:description xmlns:s="http://www.jenitennison.com/xslt/xspec"
    stylesheet="test.xsl">
    <s:scenario label="Processing document root">
        <s:context select="/"><anything/></s:context>
        <s:expect
          label="Should be 'Hello world'."><m>Hello world!</m></s:expect>
    </s:scenario>
</s:description>
```

Figure 6 shows the report from running XSpec on the description document.

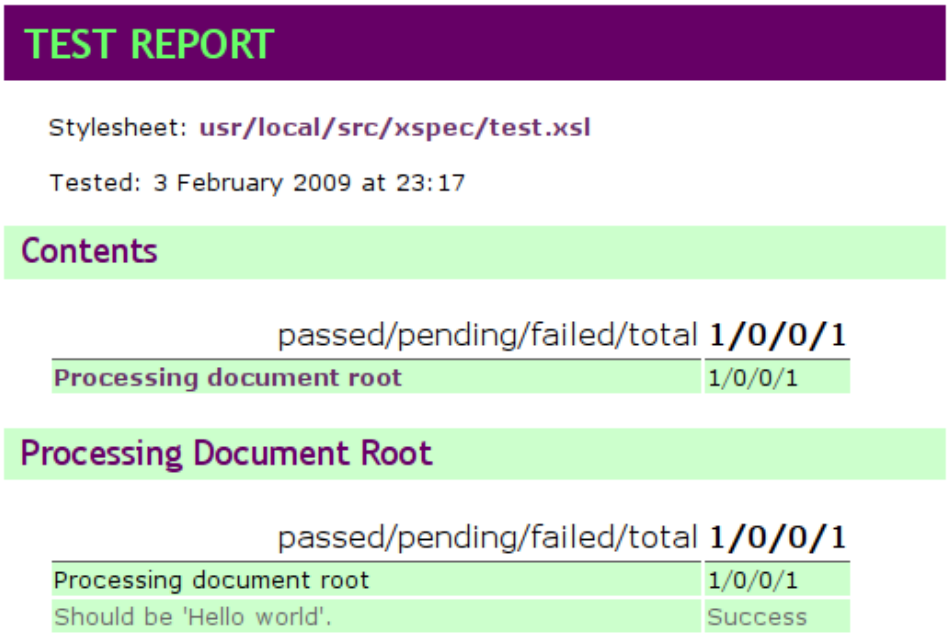


Figure 6. XSpec test result report

3.11.2. Coverage

XSpec also includes a coverage utility which highlights parts of the stylesheet under test that have not been exercised by the scenarios.

The coverage utility comprises a custom TraceListener for use with Saxon and a stylesheet for generating a report of the results.

The XSLTCoverageTraceListener produces an XML file recording the line number and module of each XSLT element in the stylesheet under test, producing an element each time an XSLT element is executed.

The stylesheet parses the stylesheet under test, and for each element, determines whether or not the element is listed in the XSLTCoverageTraceListener output. Those that are not are highlighted in the generated report.

Figure 7 shows the coverage report for the example stylesheet and description document used in the previous example. The unused template shows as white text on a red background.

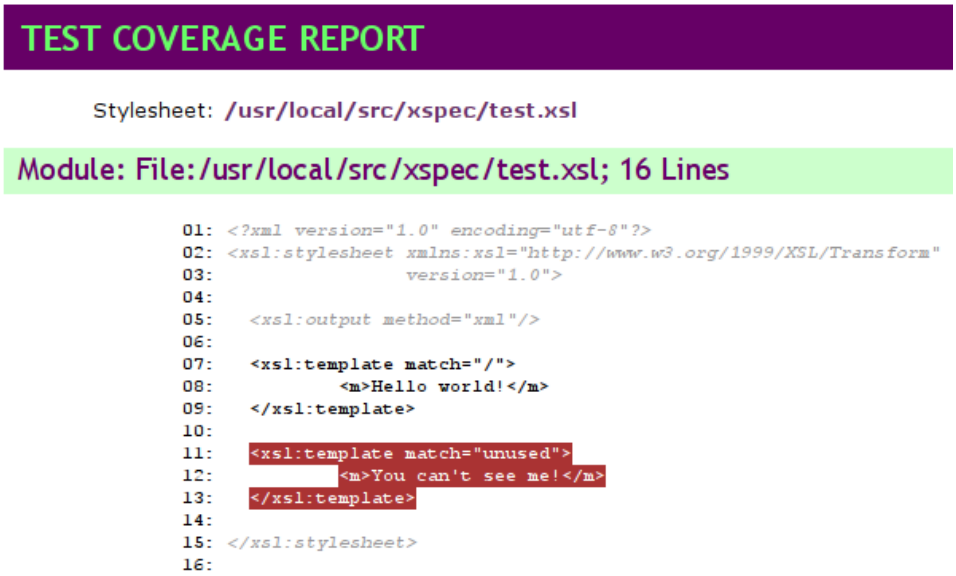


Figure 7. XSpec coverage report

4. Static Tests

4.1. XSLV Static Validation Tool

The XSLV tool for static validation of XSLT from the University of Aarhus, available online at http://www.brics.dk/XSLV/, is able to check that all output of a stylesheet at runtime is valid according to a specified output schema, assuming that the input is valid according to its specified schema. Schemas can be written as DTD, XML Schema, or a subset of RELAX NG.

Since XSLT is Turing complete, determining validity for all possible stylesheets is undecidable, so the tool applies some approximations. However, the designers wanted to be able to guarantee correctness, so the approximations err on the safe side: the tool reports some valid output as being invalid, but should never report invalid output as valid.

I consider that this type of static validation complements rather than replaces unit testing, firstly since the stylesheet output is likely to be invalid from its inception up until it is largely complete, and secondly since its all too easy to create output that's valid but still incorrect.

4.2. `xslqual.xsl`

`xslqual.xsl` (http://gandhimukul.tripod.com/xslt/xslquality.html), by Mukul Gandhi, runs a number of XSLT code quality rules. Each rule in this file is in a form of an XML fragment, something like following:

```
<rule name="DontUseDoubleSlashOperatorNearRoot">
    <message>
        Avoid using the operator // near the root of a large tree
    </message>
    <xpath>
        //(@match | @select)[starts-with(., '//')]
    </xpath>
    <priority>2</priority>
    <example>
        <!-- VIOLATION -->
        <xsl:variable name="x" select="//x/y/z" />

        <!-- REPAIR -->
        <xsl:variable name="x" select="/a/b/x/y/z" />
    </example>
</rule>
```

4.3. XSLT Metrics

XSLT Metrics (http://code.menteithconsulting.com/wiki/XSLTMetrics), by Tony Graham, is a stylesheet for some descriptive, rather than prescriptive or proscriptive, metrics about what's in a stylesheet. The purpose of the metrics is finding out what's in a stylesheet rather than telling you how to write your stylesheet.

The metrics stylesheet is written in XSLT 1.0 so it usable by the maximum number of people.

Current sample output from the DocBook XSLT stylesheets:

```
Stylesheets: 61
  With comments: 54
    Both preceding and containing comments: 0
    Preceding comments only: 0
    Containing comments only: 0
Templates: 1833
  With comments: 446
    Both preceding and containing comments: 56
    Preceding comments only: 151
    Containing comments only: 239
  Named templates: 421
    Recursive: 0
  Moded templates: 933
    Named moded templates: 2
Imports: 3
Includes: 57
```

4.4. debugxslt

debugxslt (http://code.google.com/p/debugxslt/), by James Fuller, is an XSLT lint checker that uses Schematron.

5. Coverage

Testing your XSLT is good. Knowing how much of your XSLT you've tested is even better. A coverage tool is able to report which portions of a stylesheet have been exercised and which haven't.

5.1. XSpec

XSpec (http://code.google.com/p/xspec/) has the only coverage utility currently known.

Bibliography

[1] Steve McConnell, Software Estimation: Demystifying the Black Art, ISBN 0-7356-0535-1, Microsoft Press, Redmond, Washington, 2006.

Testing XSLT with XSpec

Jeni Tennison
Jeni Tennison Consulting Ltd
<jeni@jenitennison.com>

Abstract

Test-driven development is one of the corner stones of Agile development, providing quick feedback about mistakes in code and freeing developers to refactor safe in the knowledge that any errors they introduce will be caught by the tests. There have been several test harnesses developed for XSLT, of which XSpec is one of the latest. XSpec draws inspiration from the behaviour-driven development framework for Ruby, called RSpec, and focuses on helping developers express the desired behaviour of their XSLT code. This paper discusses the XSpec language, its implementation in XSLT 2.0, and experience with using XSpec on complex, large-scale projects.

1. Introduction

Not long after a developer starts working with XSLT, they realise that testing is vital to XSLT development. Even with schema-aware XSLT 2.0, and even if you use type declarations religiously, it is impossible for an XSLT processor to check that your stylesheet makes sense. You may have mis-spelled an attribute name, performed a substring that will always return an empty string, forgotten a mode on a template. The only way to tell is to check and see.

So even if you don't practice agile development, a testing framework can save a lot of time and repetitive effort. Unsurprisingly, a number of testing frameworks have been developed for XSLT, and Tony Graham did a good job of summarising their different approaches, advantages and disadvantages at XTech 2007[1].

There are a number of different types and motivations for testing, from testing individual functions to testing entire applications. In the last few years, linked to the rise of Ruby on Rails, there has been a growing interest in behaviour-driven development[2]. This approach focuses on expressing the behaviour of code using domain-centric language, as well as (in the vein of test-driven development) describing the behaviour before writing the code that satisfies that behaviour.

[1] http://2007.xtech.org/public/schedule/detail/217
[2] http://en.wikipedia.org/wiki/Behavior_Driven_Development

XSpec is a framework that seeks to apply BDD principles to XSLT testing. It particularly follows the example of RSpec[3], used in much Ruby on Rails development, in the terms that it uses. The rest of this paper will describe how to describe the behaviour of an XSLT stylesheet using XSpec, the implementation of XSpec in XSLT, and some experience of using XSpec in large, complex applications.

2. XSpec

The aim of XSpec is to provide a flexible testing framework that supports whatever level of testing someone wants to do, while encouraging a behaviour-driven approach in which the tests act as descriptions of the behaviour of an application as well as runnable code.

A RELAX NG schema and other information about XSpec is available from http://code.google.com/p/xspec.

An XSpec document describes some aspect of the behaviour of a stylesheet application. It might describe a particular module, or a particular mode, or some other subset of the application, or it might describe the overall behaviour of the application. Accordingly, the document element of an XSpec document is a <x:description> element. (In this paper, the x prefix is used to denote the http://www.jenitennison.com/xslt/xspec namespace, which is used for all XSpec elements.) The stylesheet attribute holds a relative URI pointing to the stylesheet application whose behaviour the XSpec document describes.

BDD describes an application by detailing what happens in particular *scenarios*. Each scenario is an example of a situation. In XSpec, the <x:description> element contains a number of <x:scenario> elements, each of which describes a particular scenario. A crucial part of BDD is that the descriptions are human readable as well as automatically testable. So each <x:scenario> element has a label attribute that describes the scenario in natural language. For example:

```
<x:scenario label="when processing a para element">
  ...
</x:scenario>
```

In accordance with the BDD approach, it's a good idea to start scenario labels with the word "when". The label itself should describe the scenario; these labels will be used to identify the scenario later, in the test report and in any documentation generated from the XSpec document, so the labels need to be both descriptive and distinct from the labels used on other scenarios.

XSpec gives the tester the flexibility to describe scenarios at several levels. Scenarios fall into three main types:

[3] http://rspec.info/

- matching scenarios describe the result of applying templates to a node in a particular mode (and with particular parameters)
- function scenarios describe the results of calling a particular function with particular arguments
- named scenarios describe the results of calling a particular named template with particular parameters

Of these, matching scenarios provide the best abstraction away from the details of the code and towards the overall behaviour of the stylesheet. XSLT's rules about which matching template will be used with a particular element are complex, dependent on the match pattern, the import precedence, the priority of the template and the mode templates are applied in. So tying testing code to a particular template is futile.

2.1. Matching Scenarios

Matching scenarios use a <x:context> element that describes a node to apply templates to. The context can be supplied in two main ways:

- pointing to a node in an existing document by giving the document URI in the href attribute and, optionally, selecting a particular node by putting a path in the select attribute
- embedding XML within the <x:context> element; the content becomes the context node, although you can also select a node within that XML using the select attribute

The first method is useful when there are example XML documents that can be used as the basis of the testing. For example:

```
<x:scenario label="when processing a para element">
  <x:context href="source/test.xml" select="/doc/body/p[1]" />
  ...
</x:scenario>
```

Using this method without specifying a select attribute gives the ability to test the stylesheet as a whole, with a few caveats about the values of global variables which will be discussed later in the paper.

The second method is related to the concept of a *mock object*, which is commonly used in BDD: it is an example of some XML which is created simply for testing purposes. The XML might not be legal; it only needs to have the attributes, content or ancestry necessary for the particular behaviour that needs to be tested. For example:

```
<x:scenario label="when processing a para element">
  <x:context>
```

```
    <para>...</para>
  </x:context>
  ...
</x:scenario>
```

The <x:context> element can also have a mode attribute that supplies the mode to apply templates in.

2.2. Function Scenarios

Function scenarios hold a <x:call> element with a function attribute whose content is a qualified name that is the same as the qualified name of the function you want to call. The <x:call> element should hold a number of <x:param> elements, one for each of the arguments to the function.

The <x:param> elements can specify node values in the same way as the <x:context> element gets set (described above), or simply by giving a select attribute which holds an XPath that specifies the value. A name or position attribute can be used on each of the <x:param> elements; without these attributes, the order in which the parameters are specified will determine the order in which they're passed when the function is called. For example:

```
<x:scenario label="when capitalising a string">
  <x:call function="eg:capital-case">
    <x:param select="'an example string'" />
    <x:param select="true()" />
  </x:call>
  ...
</x:scenario>
```

will result in the call `eg:capital-case('an example string', false())` as will:

```
<x:scenario label="when capitalising a string">
  <x:call function="eg:capital-case">
    <x:param select="true()" position="2" />
    <x:param select="'an example string'" position="1" />
  </x:call>
  ...
</x:scenario>
```

2.3. Named Scenarios

Named template scenarios are similar to function scenarios except that the <x:call> element takes a template attribute rather than a function attribute, and the <x:param> elements within it must have a name attribute that supplies the name of the para-

meter. These parameters can also have a tunnel attribute to indicate a tunnel parameter. For example:

```
<x:scenario label="when creating a table with two columns containing three ▶
values">
  <x:call template="createTable">
    <x:param name="nodes">
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </x:param>
    <x:param name="cols" select="2" />
  </x:call>
  ...
</x:scenario>
```

In fact, you can use <x:param> in the same way within the <x:context> element in matching scenarios.

2.4. Expectations

Each scenario can have one or more "expectations": things that should be true of the result of the function or template invocation described by the scenario. Each expectation is specified with an <x:expect> element.

The label attribute on the <x:expect> element gives a human-readable description of the expectation, just as the label attribute on the <x:scenario> element gives a human-readable description of the scenario. In keeping with BDD practice, the label should usually start with "it should".

There are two main kinds of expectations:

- a value that the result should match, which may be
  an atomic value
  an XML snippet
- an arbitrary XPath test that should be true of the result

The select attribute is used to specify an atomic value. For example:

```
<x:scenario label="when capitalising a string">
  <x:call function="eg:capital-case">
    <x:param select="'an example string'" />
    <x:param select="true()" />
  </x:call>
  <x:expect label="it should capitalise every word in the string"
    select="'An Example String'" />
</x:scenario>
```

The content of the <x:expect> element can be used to specify some XML. For example:

```
<x:scenario label="when processing a para element">
  <x:context>
    <para>...</para>
  </x:context>
  <x:expect label="it should produce a p element">
    <p>...</p>
  </x:expect>
</x:scenario>
```

When comparing the actual result with the expected result, three dots in an element or attribute value within the expected XML means that the values aren't compared. If the actual result is:

```
<p>A sample para</p>
```

and the expected result is given as:

```
<p>...</p>
```

then these match. If the expected result is:

```
  <p>Some other para</p>
```

then they don't.

The test attribute can be used to specify an arbitrary XPath test. For example:

```
<x:scenario label="when creating a table with two columns containing three ▶
values">
  <x:call template="createTable">
    <x:param name="nodes">
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </x:param>
    <x:param name="cols" select="2" />
  </x:call>
  <x:expect label="it should have two columns"
    test="count(/table/colspec/col) = 2" />
</x:scenario>
```

The test attribute and the content of the <x:expect> element can be combined to test a portion of the result. For example:

```
<x:scenario label="when creating a table with two columns containing three ▶
values">
  <x:call template="createTable">
    <x:param name="nodes">
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </x:param>
    <x:param name="cols" select="2" />
  </x:call>
  <x:expect label="it should have two columns"
    test="count(/table/colspec/col) = 2" />
  <x:expect label="the first row should contain the first two values"
    test="/table/tbody/tr[1]">
    <tr>
      <td>A</td><td>B</td>
    </tr>
  </x:expect>
</x:scenario>
```

2.5. Nesting Scenarios

Scenarios can be nested within each other to group together similar scenarios. Descendant scenarios inherit the context or call from their ancestor scenarios. All the scenarios in a particular tree have to be of the same type (matching, function or named). Usually only the lowest level of the scenarios will contain any expectations. Here's an example:

```
<x:scenario label="when creating a table">
  <x:call template="createTable" />

  <x:scenario label="holding three values">
    <x:call>
      <x:param name="nodes">
        <value>A</value>
        <value>B</value>
        <value>C</value>
      </x:param>
    </x:call>

    <x:scenario label="in two columns">
      <x:call>
        <x:param name="cols" select="2" />
      </x:call>
      <x:expect label="the resulting table should have two columns"
        test="count(/table/colspec/col) = 2" />
```

```
      <x:expect label="the first row should contain the first two values"
        test="/table/tbody/tr[1]">
        <tr>
          <td>A</td><td>B</td>
        </tr>
      </x:expect>
    </x:scenario>

    ... other scenarios around creating tables with three values
        (with different numbers of columns) ...
  </x:scenario>

  ... other scenarios around creating tables ...
</x:scenario>
```

The labels of the nested scenarios are concatenated to create the label for the innermost scenario (for the purposes of documentation and reporting). In the above example, the third scenario has the final label "when creating a table holding three values in two columns".

2.6. Focusing Efforts

XSpec descriptions can get quite large, which can mean that running the tests takes some time. Although all the tests should be run before code is checked in, while working on a small part of a stylesheet application it can be helpful to focus on just the tests that relate to that part. There are three ways of dealing with this.

First, XSpec description documents can be imported into each other using <x:import>. The href attribute holds the location of the imported document. All the scenarios from the referenced document are imported into this one, and will be run when executed. For example:

```
<x:import xlink:href"other_xspec.xml" />
```

It helps if the imported XSpec description documents can stand alone; this enables you to perform a subset of the tests. To work effectively, the imported XSpec description documents should (through the stylesheet attribute on <x:description>) describe the same stylesheet as the main one, or a stylesheet module that's included or imported into that stylesheet.

Second, any scenario or expectation can be marked as "pending" by wrapping them within a <x:pending> element or adding a pending attribute to the <x:scenario> element. When the tests are run, any pending scenarios or expectations aren't tested (though they still appear, greyed out, in the test report). The <x:pending> element can have a label attribute to describe why the particular description is pending; for

example it might hold "TODO". If you use the pending attribute, its value should give the reason the tests are pending. For example:

```
<x:pending label="no support for block elements yet">
  <x:scenario label="when processing a para element">
    <x:context>
      <para>...</para>
    </x:context>
    <x:expect label="it should produce a p element">
      <p>...</p>
    </x:expect>
  </x:scenario>
</x:pending>
```

or:

```
<x:scenario pending="no support for block elements yet"
  label="when processing a para element">
  <x:context>
    <para>...</para>
  </x:context>
  <x:expect label="it should produce a p element">
    <p>...</p>
  </x:expect>
</x:scenario>
```

Third, you can mark any scenario as having the current "focus" by adding a focus attribute to a <x:scenario> element. Effectively, this marks every *other* scenario as "pending", with the label given as the value of the focus attribute. For example:

```
<x:scenario focus="getting capitalisation working"
  label="when capitalising a string">
  <x:call function="eg:capital-case">
    <x:param select="'an example string'" />
    <x:param select="true()" />
  </x:call>
  <x:expect label="it should capitalise every word in the string"
    select="'An Example String'" />
</x:scenario>
```

2.7. Global Parameters

Any <x:param> elements at the top level of the XSpec description document (as a child of the <x:description> element) can be used to override any global parameters

or variables that have declared in your stylesheet. They are set in just the same way as setting parameters when testing named templates or functions.

3. Implementation

XSpec testing has been implemented using a pipeline of two XSLT 2.0 stylesheets that are provided at http://code.google.com/p/xspec.

The first stylesheet, `generate-xspec-tests.xsl`, creates another stylesheet by translating an XSpec description document into runnable XSLT 2.0 code. When this automatically generated stylesheet is run, it creates an XML document that contains the results of the testing. The XML report can be transformed into an HTML test report using `format-xspec-report.xsl`.

Figure 1 shows an example test report with three failures. This example only contains one top-level scenario (listed in the 'Contents') and one leaf scenario with multiple expectations.

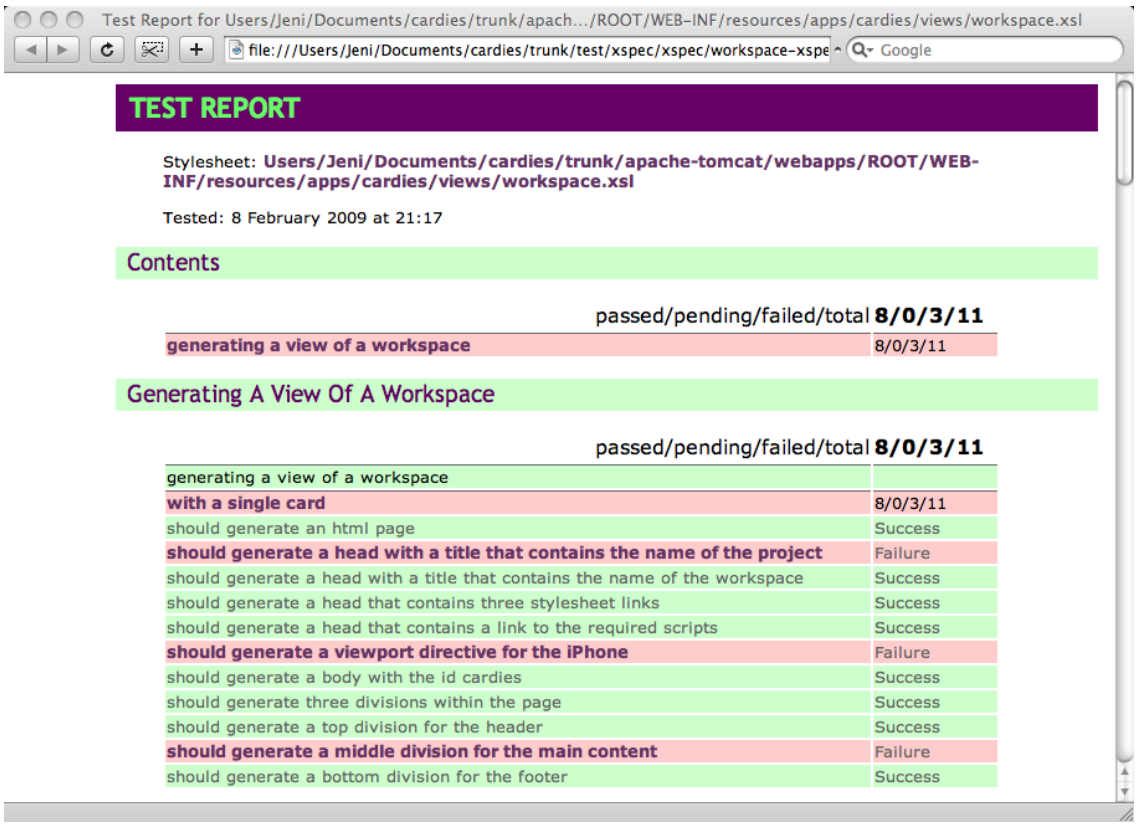


Figure 1. A XSpec Test Report

Clicking on an unsatisfied expection jumps to a side-by-side rendering of the actual and expected result. This is shown in Figure 2. Elements, attributes and text that doesn't match is highlighted in green.

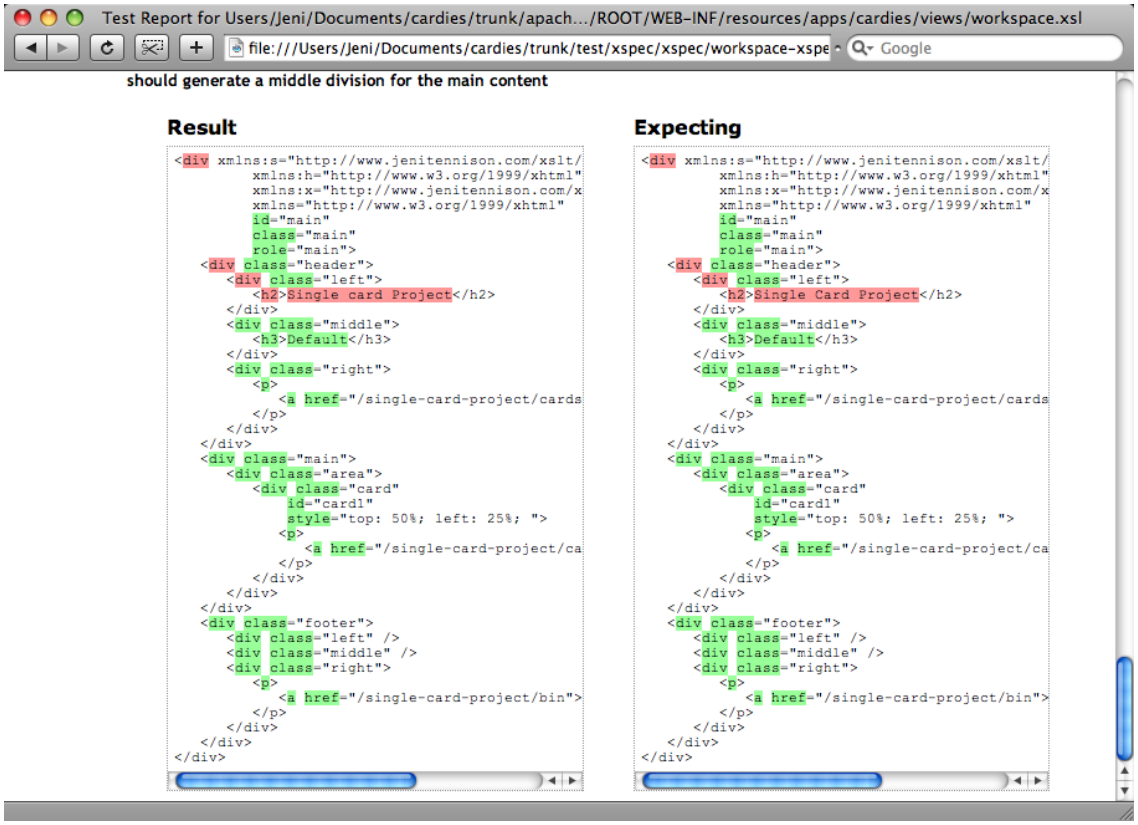


Figure 2. Detail of a Failed Expectation

The process of creating a test report is supported by a batch script for Windows and a shell script for other environments. The transformation steps could also be automated through an ant script or an XProc pipeline.

As well as producing a test report, the batch and shell scripts can be used to create a coverage report, which shows the parts of the code that are (and are not) run during the testing process. An example is shown in Figure 3, in which one template isn't executed during the running of the tests and is therefore highlighted in red. Stylesheet code that cannot be exercised by tests (such as the <xsl:stylesheet> element) is shown in grey, and italicised.

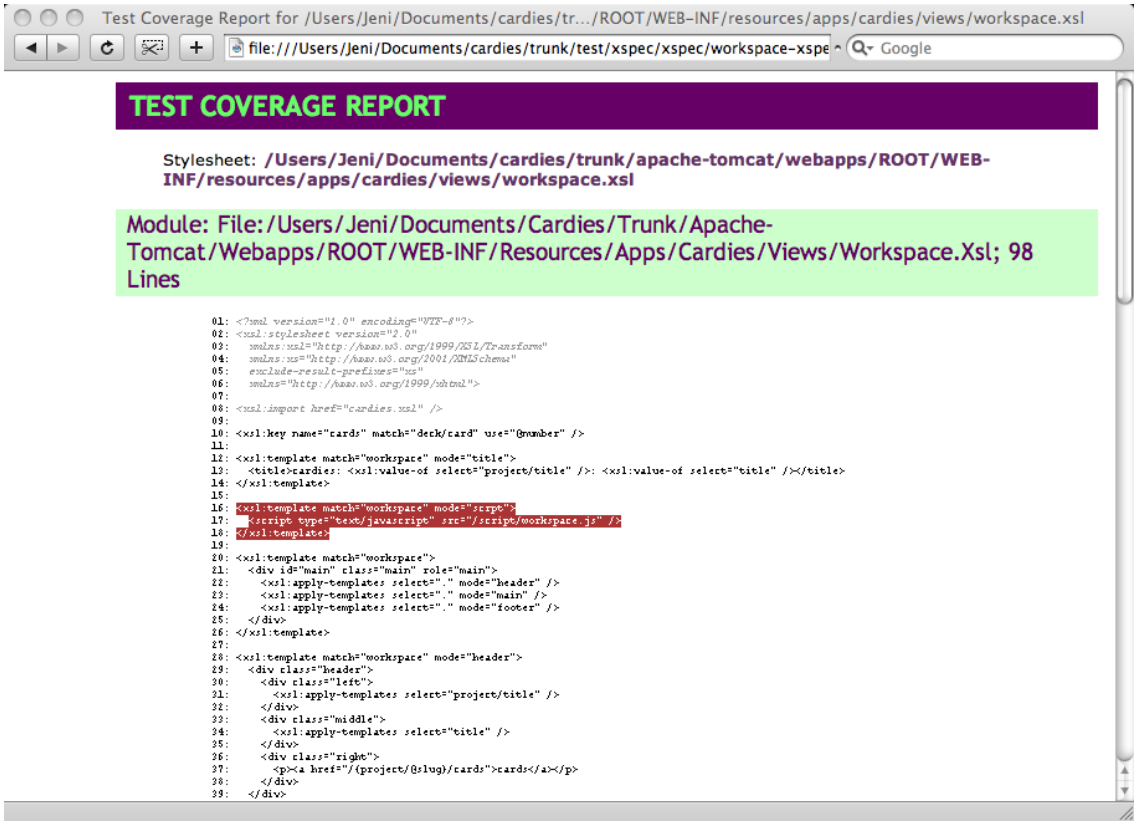


Figure 3. An XSpec Coverage Report

The coverage report is generated by creating an XML version of a trace report (using Saxon9) while the tests are run, and then tying this trace report together with both the tree of nodes within the stylesheet and its textual representation (so that it is rendered accurately within the HTML page). As with all coverage reports, it has to be read with caution: just because an instruction is executed during the tests does not mean that it is, itself, tested. Nevertheless, if a piece of code appears as read, the tests are lacking in that area (or the code itself is superfluous).

There are several disadvantages with any XSLT-based testing framework.

The first disadvantage is that it isn't possible to have different scenarios use different values for global parameters. Indeed, stylesheets that rely on global variables that depend on the source document are generally impossible to test, because during the testing itself there is no source document.

However, my experience is that the quality of the code, as well as its testability, increases if global parameters are avoided in favour of local (possibly tunnelling) parameters on any templates or functions that otherwise use global parameters. These can default to the value of the global parameter, but be set explicitly when testing. For example, if `$tableClass` is a global parameter, a testable template is:

```
<xsl:template name="createTable">
  <xsl:param name="nodes" as="node()+" required="yes" />
```

```
    <xsl:param name="cols" as="xs:integer" required="yes" />
    <xsl:param name="tableClass" as="xs:string" select="$tableClass" />
    ...
  </xsl:template>
```

As well as making the code more testable, using local parameters like this highlights the use of the parameter within the template and allows it to be called in other ways, improving its generality. Avoiding global parameters that rely on the source document also enables a stylesheet to be later used in an application that works over several such source documents.

The second disadvantage with using an XSLT implementation is that it prevents the testing of code that generates messages, and in particular that terminates the running of the stylesheet. XSpec does not yet have the facility to describe expected messages or terminations, but that does not mean that it would not be useful.

The third disadvantage is that it is not possible to test the generation of multiple result documents. Although the code for creating the content of such documents can be tested (if it is not nested inside the <xsl:result-document> element itself), there's no way to test that particular result documents are generated. As with messages, there is no facility for testing such documents within XSpec as a language anyway, but that does not mean it wouldn't be useful.

4. Experience

XSpec has been used successfully within at least two large-scale and complex projects.

One project was the conversion of WordML into a highly structured, semantic, XML document. This transformation involved multiple phases of tidying and grouping. Separate XSpec description documents covered different modes, reflecting the different phases of the transformation.

The other involved changing existing code that generated XHTML to add new functionality to include RDFa within the results. In this case, the existing code and some sample documents were used to automatically generate a set of XSpec regression tests. These tests were then expanded during the development of the new functionality. Separate XSpec description documents covered different kinds of source documents and enabled different developers to work on different aspects of the code concurrently.

Developing XSLT using XSpec seems to have three distinct phases.

During the first phase, the emphasis is on generating the basic structure of the result of the transformation. The tests are simple, high-level, and easy to satisfy.

During the second phase, the emphasis is on creating enough scenarios to cover all the elements that may have to be processed by the stylesheet. It is helpful, in this phase, to collect a set of example documents, to analyse the way that elements appear

within those documents (with which content, ancestry, and combination of attributes), and to decide how they should be transformed into some output. Scenarios created during this phase provide exemplars for simple mapping rules that are then naturally coded into templates. These kinds of scenarios naturally encourage matching templates.

During the third phase, a more exploratory testing cycle takes place, and it is during this phase that the hard work of the previous two phases pays dividends. During this phase, documents are tested against the stylesheet and examined to locate any problems with the output. When a bug is found, the developer writes a scenario that describes the circumstances that caused the bug, and the desired behaviour of the stylesheet. The stylesheet is then coded to operate correctly in those circumstances, with the existing tests providing an effective barrier against changes that introduce new bugs. This phase continues, even after the deployment of the stylesheet.

The biggest problems with using XSpec are keeping the descriptions current as the desired result of the code, and the design of the stylesheet, changes. The best tests don't repeat other tests, and they keep their expectations focused and simple. Using XML snippets within expectations, while convenient, often works against this; even though ... can be used to elide the values of elements or attributes, it can't be used as a general wildcard to stand in place of a number of elements in element content, or a number of attributes. So expectations that use XML snippets often test more than they need to, and sometimes have to be updated en masse.

Another problem that we encountered, and one that XSpec was improved to handle, was the difficulty in judging whether it mattered when there were differences in whitespace between the expected and actual result. For example, whereas normally whitespace in element content doesn't matter, in the test:

```
<x:scenario label="when processing a person's name">
  <x:context>
    <eg:Name><eg:Fn>Jeni</eg:Fn> <eg:Sn>Tennison</eg:Sn></eg:Name>
  </x:context>
  <x:expect label="it should generate some HTML with appropriate classes">
    <span class="fn"><span class="n">Jeni</span> <span ▶
class="sn">Tennison</span></span>
  </x:expect>
</x:scenario>
```

it does matter that the space between the <eg:Fn> and <eg:Sn> elements is translated into a space between the two elements in the expected result.

This experience led to the introduction of a preserve-space attribute in the <x:description> element, in which can be listed the names of elements within which whitespace-only-text-nodes should be preserved.

5. Future Work

While XSpec has proved its utility in existing projects, there are some specific gaps, mentioned above, which need to be filled. These are:

- The ability to have different values for global variables or parameters for different scenarios.
- Support for testing the messages generated by a stylesheet, even if they terminate the stylesheet.
- Support for testing the generation of multiple result documents by a stylesheet.

Each of these gaps can only be filled by an implementation that is written, at least partly, in something other than XSLT.

One promising possibility is to generate an XProc[5] pipeline based on the scenarios specified in the XSpec description document. Running the pipeline would generate a report document in the same way as the result of the current `generate-xspec-tests.xsl` stylesheet. As it runs the transformation, an XProc processor is able to set parameters, and to capture the messages and result documents that are generated, opening up the possibility of them being tested.

Now that XProc is nearing Recommendation, the next stage in the development of XSpec, then, is its implementation using XProc, and the addition of constructs that test messages and multiple result documents.

[5] http://www.w3.org/TR/xproc/

FunctX

A case study in end-to-end processing of XML

Priscilla Walmsley
Datypic
<pwalmsley@datypic.com>

Abstract

The FunctX function library is a set of reusable functions for XQuery 1.0 and XSLT 2.0. This includes commonly needed functions such as `substring-after-last`*, or* `index-of-node`*. Users can use the functions as is, or use them as examples to build on for their specific use cases.*

Behind the FunctX library is the FunctX application, open source software that can be used to manage, test and generate documentation for a function library. This talk will provide a tour of the FunctX application, showing how to use the application to create, test and document a custom library of reusable functions. It will also describe some lessons learned, and types and techniques gathered while writing an end-to-end XML processing application.

Keywords: XML, XSLT, XQuery

1. The FunctX Library

The FunctX XQuery/XSLT 2.0 function library is a set of reusable functions for XQuery 1.0 and XSLT 2.0. This includes commonly needed functions such as `substring-after-last`, or `index-of-node`. These functions are not generic or universal enough to be part of the XQuery 1.0/XPath 2.0 built-in functions, but are useful to a wide audience nonetheless.

A user can import the entire FunctX library into their XQuery 1.0 module or XSLT 2.0 stylesheet, or cut and paste individual function definitions as needed. The library also serves an instructive purpose, providing examples of XQuery and XSLT syntax that users can build on to create their own functions.

The library, available at http://www.functx.com, is currently in version 1.0 and consists of 154 functions in a variety of categories. In addition, the 130 functions built into XQuery 1.0/XPath 2.0 and XSLT 2.0 are documented on the site.

2. The FunctX Application

Behind the FunctX function library is the software used to manage, test and generate documentation for the library. FunctX provides an open source application and

framework for developers to easily create additional function libraries. It consists of the following components, depicted in the diagram below:

- an XML vocabulary for specifying XSLT and XQuery functions, along with their associated documentation and test cases
- a test harness for testing the outcome of XQuery and XSLT functions using various processors and reporting on results
- a documentation tool that generates human-readable, hyperlinked documentation for the functions in both HTML and PDF format

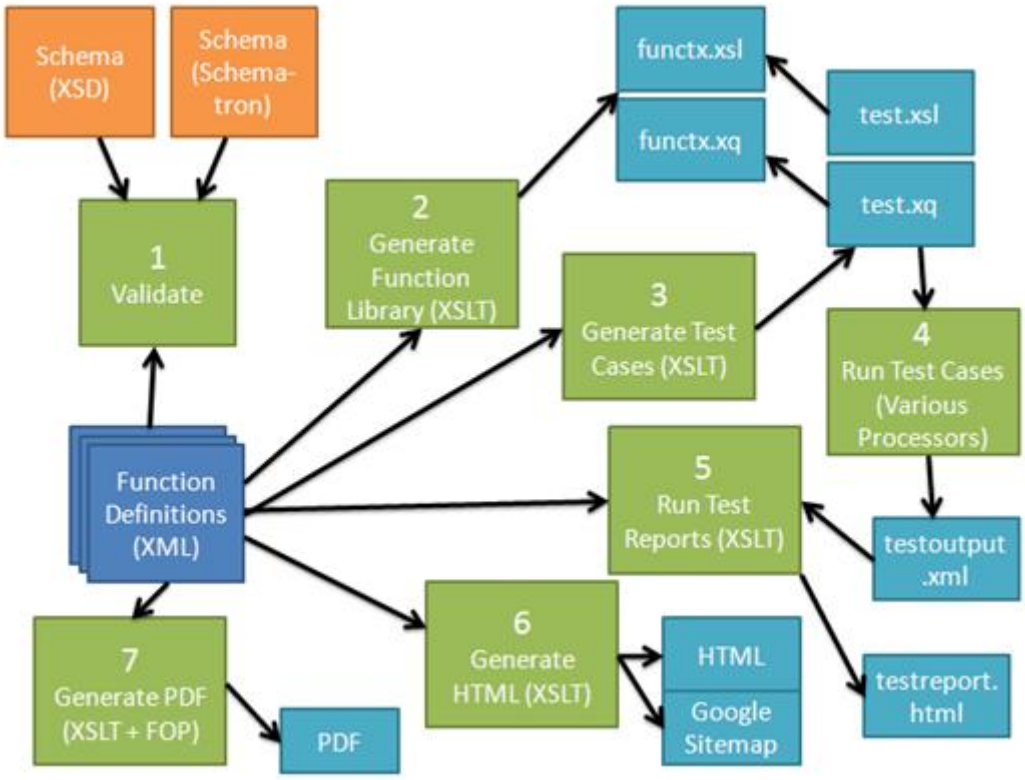


Figure 1. FunctX Process Flow

The FunctX application makes use of a variety of current XML standards. The core functionality of FunctX is implemented in XSLT 2.0, making heavy use of the new 2.0 features. In addition, W3C XML Schema and Schematron are used for validation, and XSL-FO is used to generate PDF documentation. The processing is controlled using Ant tasks to validate the function documents, run Saxon to generate tests and documentation, integrate various XQuery and XSLT processors to run the tests, and call FOP to generate PDFs.

2.1. The FunctX XML Vocabulary

An XML vocabulary was developed for FunctX to represent the function definitions. There is one XML document to describe each function. An example, for the `substring-after-last` function, is shown below.

Example 1. Function document, `functx-substring-after-last.xml`

```
<function xmlns:html="http://www.w3.org/1999/xhtml"
            xmlns:functx="http://www.functx.com"
            xmlns="http://www.datypic.com/xmlf"  ▶
xmlns:dxmlf="http://www.datypic.com/xmlf" >
       <functionName>functx:substring-after-last</functionName>
       <tracking>
          <libraryVersion>1.0</libraryVersion>
          <status>Tested</status>
          <published>2006-06-27</published>
          <lastUpdated>2007-02-26</lastUpdated>
       </tracking>
       <source>
         <personName>Priscilla Walmsley</personName>
         <organization>Datypic</organization>
         <eMail>pwalmsley@datypic.com</eMail>
         <url>http://www.datypic.com</url>
       </source>
       <briefDesc>The substring after the last occurrence of a ▶
delimiter</briefDesc>
       <inputLanguage>any</inputLanguage>
       <outputLanguage>any</outputLanguage>
       <relevantSyntax>xquery1 xpath1 xpath2 xslt1 xslt2</relevantSyntax>
       <keywords/>
       <arguments>
         <argument>
           <name>$arg</name>
           <type>xs:string?</type>
           <description>the string to substring</description>
         </argument>
         <argument>
           <name>$delim</name>
           <type>xs:string</type>
           <description>the delimiter</description>
         </argument>
         <return>
           <type>xs:string?</type>
         </return>
       </arguments>
       <longDesc>
```

```
        <html:p>The <html:code>functx:substring-after-last</html:code> function ▶
   returns the part of <html:code>$arg</html:code> that appears after the last ▶
   occurrence of <html:code>$delim</html:code>. If <html:code>$arg</html:code> ▶
   does not contain <html:code>$delim</html:code>, the entire ▶
   <html:code>$arg</html:code> is returned. If <html:code>$arg</html:code> is the ▶
   empty sequence, a zero-length string is returned.</html:p>
        </longDesc>
        <sampleCases>
          <sampleCase>
            <input>functx:substring-after-last('abc-def-ghi', '-')</input>
            <dxmlf:output xmlns="">ghi</dxmlf:output>
          </sampleCase>
          <sampleCase>
            <input>functx:substring-after-last('abcd-abcd', 'ab')</input>
            <dxmlf:output xmlns="">cd</dxmlf:output>
          </sampleCase>
          <sampleCase>
            <input>functx:substring-after-last('abcd-abcd', 'x')</input>
            <dxmlf:output xmlns="">abcd-abcd</dxmlf:output>
          </sampleCase>
        </sampleCases>
        <definition syntax="xpath2">
          fn:replace ($arg,fn:concat('^.*',functx:escape-for-regex($delim)),'')
       </definition>
       <dependsOn refs="functx:escape-for-regex"/>
       <seeAlso refs="fn:substring-after functx:substring-before-last ▶
   functx:substring-after-last-match"/>
   </function>
```

As shown in the example, the XML document for each function stores the following information:

- audit information, such as the history and source of the definition, in `tracking` and `source`
- a short and long description of the function, in `briefDesc` and `longDesc`
- a list of the arguments and return type of the function, in `arguments`
- test and example cases, in `sampleCases`
- the function body itself, in `definition`
- information on dependencies, in `dependsOn` and `seeAlso`

The general philosophy of the library is to try to support both XQuery 1.0 and XSLT 2.0 in the same definition, by defining the body of the function in XPath 2.0. In the example, the `syntax` attribute of the `definition` element indicates that the body is written in XPath 2.0. When this is not possible, more than one `definition` element can be included to support both XQuery and XSLT 2.0.

The FunctX application validates the function documents using XML Schema, Schematron and some XSLT 2.0 scripts.

2.2. The Test Harness

The FunctX application allows the function definitions to be tested with a variety of XQuery and XSLT processors. The first step in this process is to generate two documents:

1. `functx.xq` or `functx.xsl`: a single XQuery or XSLT document that contains all of the function definitions (it can also be scoped to a subset of the functions.)
2. `test.xq` or `test.xsl`: a test-case document that imports the function library and calls all of the functions with each of the test cases.

These documents are generated using XSLT 2.0 from the original function documents. A portion of the test-case document (for XQuery) is shown below.

Example 2. Snippet of the `test.xq` document

```
<function name="functx:substring-after-last">
    <test1>{local:formatResults( functx:substring-after-last('abc-def-ghi', ▶
'-') )}</test1>
   <test2>{local:formatResults( functx:substring-after-last('abcd-abcd', 'ab') ▶
)}</test2>
   <test3>{local:formatResults( functx:substring-after-last('abcd-abcd', 'x') ▶
)}</test3>
</function>
```

The test-case document is then run through the processor of choice using a provided Ant task, creating a file called `testoutput.xml`, an example of which is shown below.

Example 3. Snippet of the `testoutput.xml` document

```
<function name="functx:substring-after-last-match">
    <test1>ef-ghi</test1>
    <test2>d</test2>
    <test3>abcd-abcd</test3>
</function>
```

The final step in the testing process is to compare the results with what was expected for each test case. This is done using an XSLT 2.0 stylesheet that compares `testoutput.xml` with the original function documents. The result is a test report in HTML (`testreport.html`) that shows the tests that failed, an example of which is shown below.

Function	Num	Expected	Received
fn:matches	8	false	()
fn:normalize-space	8	zero-length string	()
fn:root	1	The document node of order.xml	()
fn:starts-with	7	true	false
fn:substring-after	5	query	zero-length string
fn:substring-before	7	zero-length string	()
fn:translate	7	zero-length string	()

Figure 2. Example Test Report

2.3. Generating Documentation

The FunctX application can also be used to generate documentation of the functions, in both HTML and PDF. For HTML, it will generate a single page per function, that is fully hyperlinked to other related functions. An example is shown below.

FunctX XQuery Function Library > Strings > Substrings >

functx:substring-after-last

The substring after the last occurrence of a delimiter

Description

The `functx:substring-after-last` function returns the part of `$arg` that appears after the last occurrence of `$delim`. If `$arg` does not contain `$delim`, the entire `$arg` is returned. If `$arg` is the empty sequence, a zero-length string is returned.

Arguments and Return Type

Name	Type	Description
$arg	`xs:string?`	the string to substring
$delim	`xs:string`	the delimiter
return value	`xs:string`	

XQuery Function Declaration

See XSLT definition.

XQuery Syntax for July 2004 - January 2007 (1.0):

```
declare namespace functx = "http://www.functx.com";
declare function functx:substring-after-last
  ( $arg as xs:string? ,
    $delim as xs:string ) as xs:string {
```

Figure 3. HTML Documentation: Function Page

Functions can be categorized within the library using a separate XML document, resulting in HTML that allows a user to choose a function by category, as shown below.

Figure 4. HTML Documentation: Category Page

In addition to HTML, a fully hyperlinked PDF can be generated from the function library, as shown below.

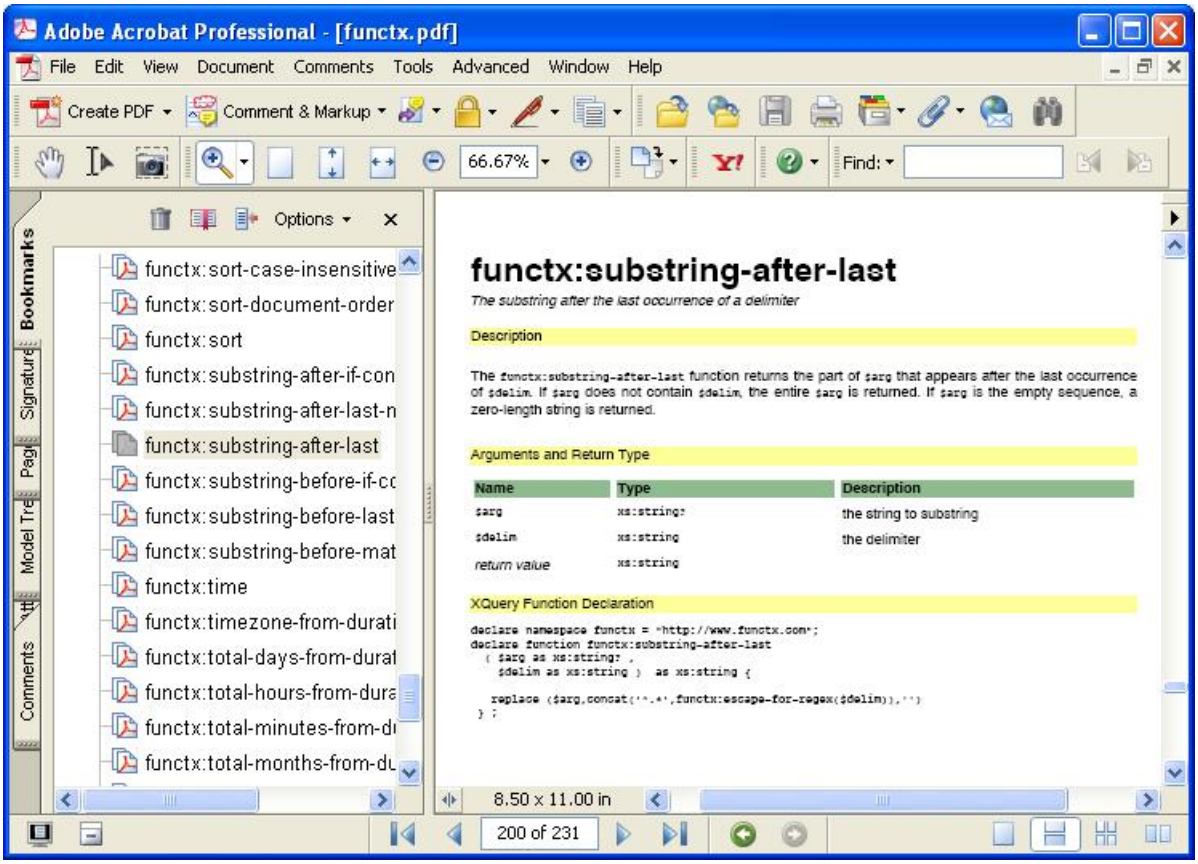


Figure 5. PDF Documentation

3. Conclusion

XQuery and XSLT functions are ideally organized into libraries to maximize their reuse and usefulness. Managing, testing and fully documenting these libraries can save an enormous amount of development and maintenance time. The FunctX application described in this paper provides a framework for developing and managing such a library.

Designing XML/Web Languages: A Review of Common Mistakes

Robin Berjon
Robineko
<robin@berjon.com>

Abstract

The tremendous uptake that XML benefited from a decade ago led to a great many XML vocabularies being defined, notably in the area of Web technology. But unfortunately there was not enough markup experience around to help so many projects avoid pitfalls.

This talk will look at a number of examples of vocabulary design failures in the area of Web languages, and discuss why they are problematic and how to avoid them.

Keywords: XML, markup, design, error, SVG, XAML, Web, XLink

1. Introduction

XML is now over ten years old and can euphemistically be dubbed a success. That being said, I don't believe I need convince readers that not all of its uses have been successful. Over time, many bright minds have attempted to describe how to best make use of it when designing vocabularies, but I believe it is safe to say that those efforts, no matter how excellent, have not been sufficient in ensuring that all applications of XML are produced in an entirely sane manner.

Part of the reason for that is education and outreach: people will often just grab XML and run, without digging around for best practices. But a larger problem is that XML combines simplicity and flexibility in such a way that a set of best practices only gets one so far in avoiding pitfalls. This does not mean that we are doomed to repeat mistakes over and over again, simply that we need to learn from our experience.

That is why this paper does not try to define a nice and simple manual as an amulet against poor vocabulary design, but rather intends to show some mistakes so that we may learn from them. As such, its organisation is more that of a shopping list rather than a treatise on XML.

Much of the errors outlined here use SVG as their source. This does not mean that SVG is the only language to make those mistakes, neither does it mean that

SVG is a bad XML vocabulary — in fact, SVG rocks. While not at all SVG-specific, there are several reasons for me to pick it as a common example:

- Knowledge of SVG is quite widespread, the specifications are openly accessible to all, hundreds of thousands of examples are available on the Web, which makes verification easy.
- SVG is a rather successful language. This shows that there is a distinction between poor vocabulary design at the syntax level and at the application level. It is of course ideal to get both right and I look forward to SVG addressing some of its issues, but good vocabulary design is no substitute for getting everything else wrong.
- Being one of the earlier major Web languages to be created after XML came into existence, it stumbled upon many of the issues that should be avoided. Being such a "seasoned" language means it has also seen many of the potential errors.
- All of its specifications were done by a group of smart people, and reviewed not only by a large community but also by other W3C groups (in fact by many non-W3C standards groups too) including the XML working groups. This shows that there is no shame in making some of these mistakes, only shame in not learning from them.
- Finally, while I can certainly not take credit for all of SVG — far from it! — I was nevertheless deeply involved in its creation. I also use it on a very regular basis. This means that not only do I know it well, but also when I point fingers and laugh, I know that I have my share of responsibility in some of those decisions.

As a final note before we delve into these mistakes, I would like to make it clear that this domain does not deal in absolutes. There are cases in which one may consider these mistakes to be good solutions; and a few cases may even be controversial and considered by some as the right option in all cases. I do not see that as an issue: as a community we can discuss and disagree. What matters is that when choosing one way of designing a language over another, one be informed of the discussion so as to make one's own decision.

2. Namespace Issues

XML namespaces are one of the most hated aspects of the XML family. Not even XML Schema has received as much contempt, and it needed a lot more work and far longer specifications in order to get there. Maybe there will be a second version of the XML stack some day, and when that day comes we can hopefully address namespaces in a way that will cause less acrimony. In the meantime, whether you like or dislike them they are what we have.

2.1. Not using a namespace

One of the biggest mistakes one can make when dealing with namespaces is to not use them. Namespaces are the tool one uses to identify an element (and in some rarer cases other things) as being part of a language. Not using namespaces means that documents in a given vocabulary cannot easily be composed into another as it will then become impossible to distinguish between the inclusion of an another language, an error in the current language, or a future version of that same language.

The absence of namespaces also makes querying a mixed document difficult. For instance if XHTML and SVG were to not have namespaces, they could still be rendered: SVG is always inside an `svg` element when it appears inside XHTML, and XHTML inside SVG is always inside a `foreignObject` element. But since the composition of the two languages can be done to any depth, if you have SVG inside XHTML inside SVG inside XHTML and so on, it is going to be difficult to find all the `title` elements or all the `font` elements. And since they have different meanings in each vocabulary, getting one for the other is very likely to cause bugs.

Admittedly, there are cases in which you can forget about namespaces. The parallel is similar to the throwaway script that one writes to perform a single, simple task now and then and never plans to reuse. And if that script does become an important part of a system, starts getting some serious usage, and needs maintenance, it's usually not overly difficult to emulate the old poorly designed interfaces while nicer ones are being shifted in. Remember however that data which is being used is a lot harder to refactor than code. The cost of adding a namespace declaration and writing namespace-aware code is tiny, especially compared to the pain of using a poorly designed data format. So unless it really is for a throwaway document, not using namespaces is a mistake.

2.2. Using too many namespaces

Of course, there can be too much of a good thing. I wouldn't say that using too many namespaces is a mistake in and of itself, but there is a point at which it does make processing a document — not to mention authoring one — a fair bit difficult.

The first item that typically springs to mind when people think of an excess in namespaces is RDF, but that is not the best of examples. It is arguable that RDF could have made things simpler, perhaps notably by making sure that the RDF Schema namespace would never need appear in RDF instances, but overall it was designed to be a very open system which would freely mix properties from a large variety of independent sources — it is to be expected that it would use many namespaces.

A better example is one that, thankfully, did not come to pass. At some point in SVG's history, one participant argued that SVG should push the toolbox approach as far as possible. Since SVG is defined by multiple modules that cover different

features (structure, geometric elements, gradients, filters, animation, etc.) and since those modules can be reused independently, they should all be split up into separate languages — each of them with different namespaces. A quick count shows that that would define around 17 namespaces instead of one, and that even the simplest documents would require at least seven or eight. A less radical but similarly painful idea is that each new version of a language should put its new features in a different namespace.

From a processing point of view that would cause no problem, but it would make the language impossible to author. There is no value to placing language modules in different namespaces, and there are far better solutions to versioning. To paraphrase, one should use as few namespaces as possible, but no fewer.

One mistake that SVG did make however is not taking the notion of a host language far enough. Some languages are designed to be reused in other languages, and have little reason to exist on their own. Two good examples are SMIL Animation, and XML Events. There is little point in identifying these languages as separate when they are embedded in another: indeed, what is the point in a generic animation processor understanding that an animation is being applied to a `circle` element if it does not understand what a `circle` element is? In such cases there is no point in placing those elements in their own namespace and they can simply be "hosted", which is to say that basically their semantics and processing are defined in another language, but they are absorbed into the host. SVG did the right thing with SMIL Animation by hosting it thus, but the wrong thing by not doing the same with XML Events. As a results, in SVG 1.2 one needs to declare an extra namespace just for events, even though there is no way in which a `listener` element will have any independent meaning.

2.3. Non-HTTP namespaces

One cause of confusion around namespaces is that many of them use a URI with the `http` scheme. People expect there to be something at the end of that URI. The fact is: there should be.

While there is no generally accepted way of defining namespace documents that live at the location pointed to by the namespace URI, the good practice is to place an HTML document there with links to documentation about the language, to schemata describing it, perhaps even style sheets or tools to go with it.

Not only is that the friendly thing to do so that humans can easily find the information they need, but it is also forward-looking: if at some point there is enough momentum behind reaching agreement on a way of embedding information in such documents then programs could automatically retrieve information for various purposes (e.g. an authoring tool could see a namespace it doesn't know and go fetch a default style sheet and schema for it).

That is something that can't be done with URNs, and even less with made-up schemes such as `antlib:` or `clr-runtime:`. A sadly common subset of this mistake is to have the namespace point directly to a DTD or XML Schema (e.g. `http://www.abisource.com/awml.dtd`): the day you decide to switch to another schema language, or to update the language version without changing the namespace the tight integration is going to be an issue.

A particularly nasty variant on this mistake is to use namespaces to point directly to an implementation of the language they describe. The canonical example here is from Microsoft Silverlight (which uses XAML). In XAML, custom components reside in a namespace which points to the DLL assembly that is to be used to render them.

Example 1. A "custom" namespace in XAML

```
xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"
```

This nullifies the whole point of shipping XML rather than code in the first place, and is basically as close as one can get to implementing a locked system in XML. The only way in which the tight coupling that this supposes could be worsened would be by specifying a path to it.

Example 2. Worse than XAML: trying hard

```
xmlns:custom="my-namespace:Foo;implementation=/usr/bin/my-cool-code.pl"
```

2.4. Processing namespaces differently

Once in a while someone decides that they dislike namespaces enough that their program should implement them differently. Technically, that's not entirely wrong: since the Namespaces in XML specification was built separately from the XML specification, some exegesis could declare it correct. But being technically right never prevented anyone from being daft, and this is one case in which the two coincide.

The more frequent such inventive processing is to declare that all attributes of an element that have no prefix shall be considered to be in that element's namespace. There is no good reason to make it so, and people who opt for that approach often do so because they short-sightedly believe that it should have been that way from day one. An example is enough to show where this starts being problematic:

Example 3. What happens when prefix-less attributes inherit the element's namespace

```
<doc xmlns='http://example.com/ns/ook#' xmlns:ns='http://example.com/ns/ook#'>
  <elem attr='foo' ns:attr='bar'>
```

```
    ...
  </elem>
</doc>
```

The XML parser will report no error here, but we now have two attributes with the same fully qualified name, and different values. Which one takes precedence? XML parsers aren't required to return attributes in their original order, so it's hard to specify. Of course, the Namespaces specification has the same issue with two attributes with different prefixes resolving to the same namespace URI, but it defines that as an error and processors implement it. This means that with this new rule, one needs to re-implement lower-level processing that is usually taken care of inside the XML parser. If one is not reusing the XML infrastructure, one might just as well not use XML at all.

There are even more creative deviations from namespaces. I will not go into the details of this since the same conclusions apply, but the rules for namespace processing in the Ant build tool at some point reached a rare degree of confusion. Not only was a default namespace applied without being declared, but elements from the default namespace sometimes had to be put in another namespace to be recognised when they were the child of an element from outside of the default namespace. If anything, this probably shows that one should be very careful before "simplifying" namespaces — many such simplifications have unpleasant side-effects.

2.5. Not allowing foreign namespaces

It can be at times rather ironic to see language authors go to the trouble of properly using namespaces throughout their design, only to forbid that elements or attributes from other namespaces appear within their own (or sometimes, making it so they can only appear at designated places, much like smokers in airports).

Whenever such limitations are imposed, the value of XML is decreased as the extensibility it promised is taken away. There are two primary reasons for this unfortunate situation. One is bad validation technology: there is a good case to be made for saying that since XML Schema does not consider arbitrary namespaces valid by default (and makes it quite difficult to specify a language in which they are accepted), it should really be called ML Schema. NVDL addresses this, but it isn't used widely enough yet.

Another is bad processing rules: it is not overly difficult to specify how foreign namespaces should be handled (skip over them for processing, but include them at the XML level if it is expose, e.g. in the DOM), but it is often overlooked. These rules are generally worth specifying as they are often the same rules that make versioning possible: ignoring what is not understood a processor can still handle future versions of its own language.

3. XML is for Humans

XML was intended to be human-readable. While that idea may make some people chortle, it is still a worthy goal to design with human readability — and writability — in mind. After all, if all one needs is a way to dump data that is only to be readable by machines in a format available anywhere other options will be faster and simpler, e.g. JSON or YAML.

Many of the mistakes in this section are by no means limited to XML, and tend to apply to other contexts — notably programming — as well, but they are nevertheless worth recalling.

3.1. Unreadable names

There are two primary ways in which one can make element and attribute names hard to read.

The first is to make them too short when they are not common elements. It is a good idea for instance to use `p` for paragraphs as it is an extremely common element, but it is more dubious to use `s`. Is that going to be for strike-through or sup text? Should it have been kept for `span`?

The other is compound names. Those can be difficult to read for native speakers of the language from which the names come from (often English) even though they have a natural feel for word boundaries, they often become hell for people who do not know the language well. The most common offender in this category is DocBook (I believe largely for historical reasons, and then for consistency). To wit: `personblurb`, `personname`, `audioobject`, `imageobjectco`, `inlinemediaobject`, `qandadiv`, `classsynopsisinfo`, `citebiblioid`, `simplemsgentry`... the `itemizedlist` goes on.

3.2. Hard to memorise names

Good language design should make it harder for people to make mistakes. One of the most basic part of that is using regular, easy to remember identifiers so as to avoid typos.

With that in mind, something in the following non-exhaustive list should strike one as wrong:

- http://www.w3.org/1999/xlink
- http://www.w3.org/2000/svg
- http://www.w3.org/1999/xhtml
- http://www.w3.org/2002/xforms
- http://www.w3.org/2001/xml-events
- http://www.w3.org/1999/02/22-rdf-syntax-ns#
- http://www.w3.org/2000/01/rdf-schema#

- http://www.w3.org/2002/07/owl#
- http://www.w3.org/2001/XMLSchema
- http://www.w3.org/2001/XMLSchema-instance
- http://www.w3.org/1998/Math/MathML
- http://www.w3.org/1999/XSL/Transform
- http://www.w3.org/1999/XSL/Format
- http://www.w3.org/2003/05/soap-encoding
- http://www.w3.org/2003/05/soap-envelope
- http://www.w3.org/2003/06/wsdl

W3C is an easy target in this area, but they are by no means alone:

- http://schemas.microsoft.com/winfx/2006/xaml/presentation
- http://schemas.microsoft.com/client/2007
- http://schemas.microsoft.com/office/word/2003/wordml
- and many more

There are arguments in favour of impossible to remember namespace URIs, but as anyone who's had to produce an XSLT style sheet outputting XHTML, SVG, XLink, XML Events, and MathML can attest these arguments are not grounded in pragmatic reality. They are too easy to get wrong, and — in part due to broken tools — lead to bugs that are sometimes difficult for users to uncover. Thankfully, the W3C is now on a more mnemonic namespace assignment policy.

3.3. Naming without respect to context

One of the great things about trees is that they provide natural context for content. And when people edit content inside a tree, they are aware of that context. That is the reason why an author will know that a `title` element inside a `circle` element will be the title for that circle, and not for the entire document.

Failing to use context in naming elements or attributes entails a loss in language fluidity. Some vocabularies that have strong roots in SGML have an excuse for this approach since DTDs named elements globally. This is visible in DocBook for instance: it feels daft to call every item in a list a `listitem` since it appears as the child of one of DocBook's many list elements, but that was necessary in the early days.

More recent languages have no such excuse. For instance SVG has a long list of elements beginning with "fe": `feDistantLight`, `feSpotLight`, `feColorMatrix`, `feConvolveMatrix`, `feGaussianBlur`, `feTurbulence`... That "fe" is meant to signify "filter effect". But the fact is that these elements can only appear as children of the `filter` element — they are always going to be in a filter effect defining context. The prefix just makes them more clumsy, as if all SVG elements began with "svg".

3.4. Human-readable text in attributes

It is often tempting to place text intended for humans inside an attribute, perhaps so as to "attach" it more directly to the element, or to make authoring more terse. The archetypal example of this being:

Example 4. Text in an attribute

```
<img ... alt='photo of a dahut in the fog'/>
```

The issue here is that this approach breaks down as soon as one starts requiring structure inside the string. For instance, if instead of using a title element to specify the titles of sections DocBook had chosen a title attribute on the section element, it would be impossible to have the title be "The `Foo` interface".

That might seem like an acceptable limitation, but it gets worse: if the text is expected to be potentially in any language, there will be cases in which it will require structure. For instance, some Chinese or Japanese text requires what are known as ruby annotations (basically text that is rendered on top or to the right of the primary text to indicate the pronunciation). Similarly, it can be useful to specify the writing direction when mixing languages that go in different directions (e.g. Arabic and French). Also, due to limitations in Unicode, some characters will not render correctly (i.e. will be rendered with the wrong glyphs) unless you specify which language the text is in — that is notably the case of the CJK set in which Unicode gave some Chinese, Japanese, and Korean text the same code-point even though they are depicted differently in each language. For this case one could place a `lang` attribute on the element to get the right effect on the text inside the attribute, but that would set the language of the entire element, not just of the text. It's an extreme case, but if one had an `img` element pointing to an image of a wine label from France, with an `alt` attribute in Korean, and set the `lang` to `kr` so that the `alt` renders right, then the language of the label would be said to be Korean too.

Given the technicalities involved in getting I18N right, and given the greater extensibility of the element approach, it should be inferred that text intended for human consumption should only occur in element content. That being said, the original argument — terseness — has some merit for authors. In the case in which it is desired, it is therefore possible to define a two-tiered approach in which such text can occur in either an attribute or a child (with the child taking precedence). That approach however has drawbacks, and needs to be used with caution.

3.5. XML for cyborgs

Some XML languages are clearly designed to match a processing approach or a data model, and are unfortunately later pitched as intended for human consumption. There are many such examples, and they tend to exhibit several of the errors ex-

amined in this document. XAML is a very clear example of this, but unfortunately it is such a verbose language that there would be no room here for an example of just how mind-boggling it can be. A quick display of one of its bizarre traits should suffice.

Example 5. Simple XAML

```
<Canvas x:Name='Foo' Height='300' Width='500' Background='White'
        xmlns="http://schemas.microsoft.com/client/2007"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Rectangle Width="200" Height="100" Canvas.Top='20' Canvas.Left='100' ▶
Stroke="Black" StrokeThickness="3">
    <Rectangle.Fill>
      <ImageBrush BitmapSource="C:\dahut.jpg" />
    </Rectangle.Fill>
  </Rectangle>
</Canvas>
```

The above shows two things: compound properties (the `Rectangle.Fill` that is a child element acting as if it were an attribute of its parent but with extra structure), and attached properties (attributes on an element that actually "talk" to its parent, in this case `Canvas.Top` and `Canvas.Left`). There is no doubt that there is a logical model behind this approach, but it belongs to the implementation that supports it, and makes things complex for the user. Indeed, how am I to know that the position of a rectangle is a property of its container, and not something intrinsic to it? Likewise, why is the fill of a rectangle a property with the name `Rectangle.Fill` even though it contains a complete image, should the rectangle itself be a property of the canvas and be called `Canvas.Rectangle`?

That is the sort of issue that arises when the underlying model is made too explicit in the language.

4. Language Issues

Language design sits above the previous considerations yet is hard to set entirely apart from them. Some of the examples here mesh into other previous ones, and some apply outside of XML.

4.1. Inconsistent naming

This is one of the simplest to get right and yet is often wrong as the result of things being specified on the fly and not revisited later. The typical example is SVG having `circle`, `path`, `ellipse`, etc. but `rect` instead of `rectangle`. It rarely has a serious impact, but it does make a language harder to learn.

4.2. Incoherent features

One thing that is difficult when introducing a new feature inside a language is to make sure that it works coherently with all the rest of the language's features. Such errors are often difficult to detect without extensive testing.

A good example is the different treatment applied to shapes and animations in SVG. The following will work and display a second rectangle inside the `g`.

```
<svg ...>
  <defs>
    <rect ... xml:id='dahut'/>
  </defs>
  <g>
    <rect x='10' y='42' .../>
    <use xlink:href='#dahut'/>
  </g>
</svg>
```

But the following, while valid, will not cause the `g` element to be animated:

```
<svg ...>
  <defs>
    <animateTransform ... xml:id='dahut-anim'/>
  </defs>
  <g>
    <rect x='10' y='42' .../>
    <use xlink:href='#dahut-anim'/>
  </g>
</svg>
```

Instead the `defs` will be animated, which will do nothing. This sort of discrepancy confuses authors, many of whom will try the latter at some point. While extensive testing is often too costly to put into effect, one low-tech approach that can help is to have a matrix tabulating all the features against all the others, and for each one that is added to look at how they interact. This would have made it apparent that animations applied to `use` work (they can animate it), but `use` applied to animations doesn't. How to address this, if only by clearly documenting it, is then a matter that will depend on the situation.

4.3. No lacunae values

A lacuna value is the value of an attribute or element that is used when that attribute or element is not specified, or when it is specified but its content is not understood by the language. This is different from a default value, which is what is used simply when an attribute or element is absent. For instance, if the `x` attribute on `rect` has a lacuna value of "0", then the value of x will be "0" for both `<rect/>` and `<rect`

`x="babe, like, you know!"/>` whereas the default value would only apply in the first case.

Not specifying lacunae values will hurt error processing (and its cousin versioning), and lead to interoperability issues.

Imagine for instance that an author specifies the fill of a rectangle to be `#99` instead of the intended `#999` because of a typo. The processor can interpret that in several ways:

- Decide it's too big an error, and melt the motherboard.
- Realise it's wrong, and pick a default colour of its choosing (often black, but could be fuchsia).
- Pad with the last character, making it `#999`, with 0 (`#990`), or with something else that could make sense.
- Throw it away, and apply a colour inherited from a `fill` attribute higher in the tree.
- Throw it away, and consider it null (transparent).
- Take the inherited colour, and only change the red and green components.

Most of the above will produce different colours. And since they'll produce different results in different implementations, there will be no interoperability. (The throwing up approach might sound appealing to XML heads, but as we will see later what is good at the syntax level is not necessarily as good at the language level).

The only solution is for each value to specify the behaviour when it is missing or in error. Not doing so is a great part of what created the HTML mess that we have today.

4.4. XML error handling

Just because the XML parser does its own error checking doesn't mean that you can ignore it entirely. Unless your language's processing model is defined to apply only on a complete document (which is unrealistic in many cases, but was adopted by SOAP for instance), it is conceivable that a processor will have processed a lot of an XML document before the parser realises there is an error.

It is usually agreed that trying to recover from an XML parsing error is, in the general case, a bad idea. In fact, it nullifies a lot of the value that XML brings to the table. That being said, given that a processor is likely to process XML in a streaming manner for performance reasons, the question remains of what to do when an error happens midway, or even if the error is simply the closing element missing.

One approach is to be transactional and throw away everything that was done based on the erroneous document — that is often the only option in systems that require some form of integrity. Another is the best effort approach, which is more suited for rendering systems: display everything that you understood up to the error,

then stop. Neither of these options is always right or wrong, the one to choose depends on context but it is important to specify which one applies, and in the best effort case one also needs to be clear about what precisely happens: if we render up to the error is an error message shown? Is a partial DOM created that script can act on? Is a load event dispatched?

4.5. Language error handling

In some cases lacunae values will not be enough to recover from errors, for instance because an element appears in the language's namespace that the processor does not know about, or because an element that it knows about appears at the wrong place in the tree.

Again this is a case in which if the rules for processing such errors are not defined, implementations are guaranteed to differ. There are multiple approaches here: flag an error and give up, ignore the element and process what is inside it as if it hadn't been there at all (or process some of it), or ignore the entire subtree contained inside that element.

Being thoroughly strict and throwing up upon a language-level error is the right thing to do in some cases, notably when one wants to be sure that the entire document is understood. SOAP has a `mustUnderstand` attribute that smartly flags such cases, and likewise SVG has a `switch` element that can provide alternatives to content that isn't understood. But as a rule of thumb, it is usually better to ignore the element for language-specific matters, but make it accessible in the DOM if there is scripting or in downstream versions of the document if it is being passed on. The reason for that is versioning, as we will see next. But everything else aside, what is key here is that behaviour in the face of the unknown be fully specified lest interoperability issues crop up.

4.6. No versioning strategy

Versioning languages is hard, and there are many ways of getting it wrong. There is not room enough here to go into a full discussion of versioning, but suffice it to say that not thinking about it in version 1 will be the cause of many headaches in version 2. And there always is a version 2.

It is, of course, difficult to predict the many directions in which a language may evolve, and there are cases in which a given evolution of a language will require changes so radical that making it compatible with older content will be a bad idea. It is not those cases that one needs to worry about as they will be easy enough to address by creating something entirely new. What language designers need to focus on is the next dot release.

When a language is being created it often starts with a big brainstorm of all the features that it could support, which are then trimmed down to the smallest number

of features that will make it useful (the second phase is, sadly, too often overlooked). A good exercise to try out to see if a language one is designing is ready for evolution is to start with a typical v1 document, and then start adding made up markup corresponding to what it would look like if it made use of all the features that were pushed off for version 2. The question then is: will that document still be processed exactly the same way as the unadorned document in a version 1 processor? If not, and if the intent is not to throw up unless everything is understood, then you have effectively painted the language into a corner.

If the intent is that x.n+1 should roughly work with a little loss in x.n implementations, then at the very least one should make sure that lacunae values are defined, and that unknowns are ignored (as a side note, not putting the version number in the namespace will also be less confusing since it cannot be changed without losing all previous implementations). Providing a way to fall back to alternate content when a feature is missing is a plus as it allows users to handle some cases, but it cannot be relied on as the sole mechanism as it has a very high cost in terms of content production.

5. Wishful Thinking and Doe-Eyed Beliefs

The heart of language designers is often in the right place; the problem has more to do with what their hands are doing.

5.1. Overcomplexity

Overcomplexity is a really general issue that applies equally to software architecture, and arguably to life in general. There is however something specific to be said about vocabulary design here: adding an element to a specification is simple.

This is not a force to be ignored. XML being by nature extensible, it is tantalisingly easy to believe that functionality can be added to a language simply by adding an element. When working on a specification, one will often hear: "Well, it's simple. Look. Just add in a `<feature param1='this' param2='that'/>` element and wow! BAM! It rocks!"

That it is not the case is obvious when looking at the work of others, but all of a sudden becomes less so when it's the feature one wants. As a rule of thumb, before proposing any new feature, one should thumb through old drafts of SVG Full 1.2 and meditate a little on the purpose of life.

5.2. Reusing the useless

At times it feels like the XML family of technologies is just that: one big family. And you wouldn't want to forget anyone when your baby language is born.

That feeling (and a general sense that reuse is good) leads people to want to reuse as many parts of the XML stack as possible when creating a new language. That is a good feeling, and certainly one that should be listened to carefully — there are indeed many good and useful technologies to reuse.

However not all of them fall in that basket. One such example is XLink. For certain, it sounds like a good idea that the links in any XML document should be understandable so that generic processors could spider the whole XML Web. But that only works if everyone plays, and furthermore the cost of using XLink has to be taken into account. First, a whole new namespace is needed. Second, the distinction between `href` and `src` requires a second attribute. And then there are issues with parts of XLink being useless for (or detrimental to) one's needs, which entails specifying that parts of it should be used but not others, or that on such and such element when one XLink attribute isn't present it defaults to something specific not in the XLink specification, etc.

Originally I have nothing against XLink, but SVG used it and experience shows that it was a bad idea. It confused users, it pushed SVG into defaulting attributes in ways that aren't entirely kosher, and it brought a cost that no one needed. As it turns out, very few people use XLink. Core XML specification produced by the W3C such as XSLT or XML Schema don't use it even though they have linking elements.

Reuse of other languages should be done where needed, and when the cost does not exceed that of reinvention. The good feeling one gets from reusing another technology should not be part of the balance.

5.3. Naïve versioning

Versioning, as explained above, is important enough that it deserves to be done right. Yet some languages have taken a rather naïve approach to it typically consisting in a `version` attribute on the root element. That is fine if the purpose is to die immediately when a given version is not supported (in which case simply changing the namespace would be less verbose and just as effective), but will not produce any useful effect if the intent is to allow processors to work across versions.

Indeed, what is such a processor to do if it see a version attribute with a value greater than the version it supports? Nothing useful comes to mind, short of warning the user that there may be rendering issues, a message which said user will either ignore, or will cause him to panic, but will not yield any useful result. Conversely, if the version attribute points to an earlier version, should features from later versions be ignored? That would make implementations unduly complex.

Versioning through the inclusion of version metadata is largely useless, if only because most documents in the wild tend to have the wrong version.

5.4. Relying on the external subset

It can be tempting to provide default values through the external subset rather than, or in addition to, through the specification. After all, even if the specification defines lacunae values, it would be nice if generic XML processors could also benefit from that information. This is generally useless, and occasionally harmful. It is useless because as explained above a lacuna value is different from, and more powerful than, a default value. Therefore, specifying default values for generic processors will lead to a mix up where the values are sometimes right but not always.

It can be harmful if, as was done at one point in SVG, the external subset is used to default namespace declarations. That will lead to elements that are in a different namespace depending on whether the external subset was processed, which is not only optional but, in the case of Web technologies, rare.

Another such reliance on the external subset that will cause no end of trouble is to expect it to define entities. The typical example here is XHTML, which will regularly trip parsers that do not fetch the external subset and subsequently complain about undefined "nbsp" entities.

6. Miscellaneous

As inelegant as a "Miscellaneous" section may be, reality is such that not all sets of items can be grouped in an n-point plan. This section lists issues common enough that they deserve mention, but that nevertheless could only be made to fit elsewhere with contrived acrobatics.

6.1. Markup in CDATA sections

This problem is simple and requires very little explanation. It occurs altogether too often, and the following example should be familiar to most.

Example 6. XML embedded inside a CDATA section

```
<doc xmlns='...'>
  ...
  <other-document>
    <![CDATA[
      <some-other-document>
        ...
      </some-other-document>
    ]]>
  </other-document>
  ...
</doc>
```

There are cases in which this can be needed, oftentimes it is used to embed poorly formed HTML, and on some cases, because XML is not entirely composable (the XML declaration and DTD prevent that), it is justified even for XML. But generally speaking, it is a bad idea: if the content is XML, then it should be part of the tree and not require some indirect further level of parsing.

6.2. Excessive microparsing

Microparsing is a term covering the use of an extra non-XML syntax inside of XML, usually in attribute values. It has been the subject of heated debate in the earlier days of the XML community. The idea here is not to flag microparsing as always bad since in fact there are numerous cases in which it is a good idea. Rather, there should be a rule of thumb separating the good uses of it from those in which it is simply hiding away structure that should be in the tree as in the CDATA example above.

Microparsing is generally good when it is designed with the author in mind. For instance, XPath is much better as it is than if one had to turn `book[/book-store/@specialty=@style]|//author[alias[2]]` into an XML tree. This is essentially the same argument that goes in favour of supporting a regular expression language within a larger language rather than having to express the same concept with a long series of method calls.

There are however cases in which microparsing does not help the author much. For instance:

Example 7. A extract from an SVG path

```
M363.73 85.73 C359.27 86.29 355.23 86.73 354.23 81.23 C353.23 75.73 355.73 ▶
73.73 363.23 75.73
C370.73 77.73 375.73 84.23 363.73 85.73 zM327.23 89.23 C327.23 89.23 308.51 ▶
93.65 325.73 80.73
C333.73 74.73 334.23 79.73 334.73 82.73 C335.48 87.2 327.23 89.23 327.23 89.23 ▶
zM384.23 48.73
C375.88 47.06 376.23 42.23 385.23 40.23 C386.7 39.91 389.23 49.73 384.23 48.73 ▶
zM389.23 48.73
C391.73 48.23 395.73 49.23 396.23 52.73 C396.73 56.23 392.73 58.23 390.23 56.23
C387.73 54.23 386.73 49.23 389.23 48.73 zM383.23 59.73 C385.73 58.73 393.23 ▶
60.23 392.73 63.23
C392.23 66.23 386.23 66.73 383.73 65.23 C381.23 63.73 380.73 60.73 383.23 59.73 ▶
zM384.23 77.23
C387.23 74.73 390.73 77.23 391.73 78.73 C392.73 80.23 387.73 82.23 386.23 82.73
C384.73 83.23 381.23 79.73 384.23 77.23 zM395.73 40.23 C395.73 40.23 399.73 ▶
40.23 398.73 41.73
C397.73 43.23 394.73 43.23 394.73 43.23 zM401.73 49.23 C401.73 49.23 405.73 ▶
49.23 404.73 50.73
```

```
C403.73 52.23 400.73 52.23 400.73 52.23 zM369.23 97.23 C369.23 97.23 374.23 ▶
99.23 373.23 100.73
C372.23 102.23 370.73 104.73 367.23 101.23 C363.73 97.73 369.23 97.23 369.23 ▶
97.23 zM355.73 116.73
C358.73 114.23 362.23 116.73 363.23 118.23 C364.23 119.73 359.23 121.73 357.73 ▶
122.23
...
```

Some people, including yours truly, can read and even write the above. But they should be discarded as bad guinea pigs. The reason for using such a syntax for paths in SVG was two-fold (and is the same reason used in other similar situations): file size, and DOM size (whereby if an element had been used for each path command, the DOM would have been supposedly much larger). Where file size is concerned, the structure of such path data is so repetitive that a good compression algorithm (such as gzip, or of course EXI) will produce similar compressed sizes whether the microsyntax or elements are used — and since SVG path data is usually big anyway, one wants to use compression (support for which is mandated). And where the DOM size is concerned, one has to keep in mind that it is merely an API. A generic DOM will be larger, but the DOM inside an SVG implementation should be able to have a very similar footprint since whether path data is in an attribute or in elements should have no effect on internal storage.

So the rule of thumb in this situation is that microparsing is for authors, not for implementations.

Practical Reuse in XML

Ari Nordström
Condesign Operations Support AB
<ari.nordstrom@condesign.se>

Abstract

Reuse is often the key selling point for XML authoring systems. This presentation examines reuse from various points of view, from the author's to the developer's, offering practical strategies for reuse of content. Markup design is discussed, as are necessary prerequisites for making such a system work. What should be reused, and when? How do you uniquely identify a resource, and what, exactly, is a resource anyway? How do you design a user interface that helps the author instead of hindering her? From a practical point of view, how do you design a publishing process that works?

A demonstration of such a system illustrates the points made.

Keywords: applic, authoring, cross-reference reuse, cross-references, fragment identifiers, hyperlinks, ID attributes, inclusion links, inset links, linking, naming schemes, profiling, publishing, reuse, unique identifiers, URN, XLink, xlink:actuate, xlink:href, xlink:role, xlink:show

This paper is about being lazy. You see, that's what reuse is about for me, being lazy without appearing to be so, reusing instead of rewriting, linking instead of copying and pasting. In fact, this is how I often present XML to my clients: it's about allowing me to be as lazy as possible, without ever having to be sloppy.

1. Constructing Reusable Components

What can be reused? How do we identify a resource for reuse? How can we reliably point at it? What kind of markup should we use? What if what we wish to reuse is mostly OK, but there's that little, tiny piece that is specific to one single product...?

This chapter discusses the prerequisites and basic mechanisms for reuse.

1.1. What Can Be Reused?

Anything that can be linked to. There are three distinct types of links, namely the following:

- Images

- Cross-references
- Inclusions of document fragments

Depending on context, the above three types will blur into each other or split into subtypes, but my basic point is that anything we can link to can be reused, and these three types are what we usually reuse.

1.2. Naming and Uniqueness

If you want to find a book in a library, you need the book's exact location: “room five, left wall, top shelf, third book from the left”. Books are frequently misplaced, however, so you might end up needing the book's title and the author's name in order to locate it.

But if the book you're looking for is the Yellow Pages, if it is in a room full of Yellow Pages books (one copy for each area code), and if that room contains a copy for yeach year the books have been published, the problem is more serious. In that case, to locate the telephone number of a company that was active twenty years ago, you need to not only know the exact area code but also the right year.

A further complication occurs if the books have been translated to a dozen languages. Would any translated version for the right area code and year do, or would you need the original language?

Finally, let's say that you're looking for a piece of paper taped to a page of a specific book. You'd need to know the right area code, the right year, the right language, and preferably the page number, or spend a very long time looking. If this was a Document Management System (DMS) and that piece of paper was the fragment you wanted to reuse, you'd probably never find it, and certainly never find it twice.

We can conclude that in order to find (and reuse) anything, there are prerequisites:

- We need names rather than locations.
- The names need to be unique.
- We need to handle any versions there might be..
- We need to handle any languages we expect to encounter.
- If the target is something inside the resource rather than the resource itself, we the target's location within the resource.
- And we still need the current location of the resource we're looking for.
- We need a controlled environment to keep track of everything we wish to store. For a book, it's a library; for an XML fragement, it's a DMS.

1.3. What's in a Name?

Since names created by human authors will most likely not be unique, the names must be generated automatically to ensure uniqueness. Databases generate IDs for each object they handle, but do very little to help identifying different versions or languages of a stored object (we need to use meta-data and relational data for that), however, so we need a more intelligent naming scheme, an abstraction layer that uniquely identifies the basic document, and adds versioning and language information to it. Since we are in the standards business, I suggest a *URN scheme*.

"URN" ("Uniform Resource Name") is an IETF specification designed to define a naming scheme that guarantees uniqueness within a well-defined namespace. A basic URN looks like this:

```
<URN> ::= "urn:" <NID> ":" <NSS>
```

The URN identifier consists of at least three fields separated by colons. The first is a fixed value, `urn`, that tells us that what we see is a URN. The second, `<NID>`, stands for a namespace identifier for this particular namespace, and the third, `<NSS>`, is a string constructed using rules valid only in the `<NID>` namespace.

In practical terms, you register a namespace with IETF and then define rules for naming resources in that namespace. IETF guarantees that there will only be exactly one namespace with the `<NID>` name, ever, and the namespace owner guarantees that the naming scheme provides uniqueness for the `<NSS>` string.

A local URN scheme that includes version and language identifiers might look like this:

```
<NSS> ::= <DID> ":" <LANG> ":" <VERSION>
```

In this example, `<DID>` is a document-specific identifier, `<LANG>` a language (and possibly a country) code, and `<VERSION>` a version counter.

In this naming scheme, `<DID>`, the document identifier, identifies the semantic document. The language (and country) codes identify different *renditions* of that same semantic document, and the `<VERSION>` string is used to indicate the progress made when editing (and publishing) the document[1].

1.3.1. A Word About ID Attributes

Remember the paper taped to that Yellow Pages page in my example? To find that piece of paper, we'd need the page number in addition to the right area code and year. In XML, the equivalent is an `ID` attribute. The combined reference (`URN#ID`) helps us find the exact location within a resource.

[1]Compare this to different renditions of an image. A JPEG version of an image different from a PNG or SVG version? I'd say no. Similarly, what separates a translation from the original is simply the language used; the information is the same.

The URN part is described above, and `ID` attributes are a well-known concept in XML. A few pointers about `ID`s, however:

- As with document names, you should not rely on humans to create `ID` values.
- An `ID` generation application is not difficult to write. However, it is useful to add a string that includes the element type and date, first in the `ID`, because while the `ID`s shouldn't be created by humans, being able to recognise an `ID` value is useful when writing.
- The Copy-Paste operation in the editor needs to be rewritten to generate new `ID`s in place of old ones when pasting or a non-valid XML document will result.

1.4. Markup (How We Link)

I've always been partial to XLink. It's my favourite spec, actually, because it's all I've ever needed in terms of XML linking. A Simple XLink covers about 99% of my needs. A cross-reference, for example, is easy:

```
<ref xlink:href="some-doc.xml#some-id"/>
```

Now, the process model isn't defined in the spec but basically, this variant of the above can be interpreted as an inclusion:

```
<ref xlink:href="some-doc.xml#some-id"
     xlink:show="embed"/>
```

Of course, it could also be used for an image; adding `xlink:actuate` is usually expected, however:

```
<ref xlink:href="my-picture.svg"
     xlink:show="embed"
     xlink:actuate="onLoad"/>
```

As I mentioned, XLink does not define the processing model for the `show` or `actuate` attributes. I do consider this approach to be a bit of a cheat, however, and suggest different element types for different links instead. This greatly simplifies processing while making the author's job easier. For our basic needs, the following three declarations cover a majority of the cases:

```
<!ELEMENT ref  EMPTY>
<!ATTLIST ref  %xlink.basic.atts;>

<!ELEMENT inset  EMPTY>
<!ATTLIST inset  %xlink.basic.atts;
                 xlink:show #FIXED "embed">

<!ELEMENT image  EMPTY>
<!ATTLIST image  %xlink.basic.atts;
```

```
xlink:show #FIXED "embed"
xlink:actuate #FIXED "onLoad">
```

Since the processing occurs either in an XML editor where you will normally need to declare things like an image element and its attributes as such, or in the publishing process when you will handle the links very differently depending on what type of links they are (targeting different XSL-FO structures), these declarations not only work but make both authoring and developing easier.

1.4.1. What About Target Markup?

The target, when an XML fragment, should be well-formed. When pointing to a place *within* a resource, the target element type should have an `ID` attribute, something that can uniquely identify it.[2]

Basically, anything that works as a wrapper is reusable; therefore, just about any element will do. It is often wise to declare block-level and inline wrappers specifically for use as containers for reusable content. A `block` element type that allows any block-level components in its content model, for example, is very useful when grouping reusable content.

Also, it is sometimes useful to include a section-level wrapper that groups sections to reusable components but does not add a section level:

```
<!ELEMENT section-group  (section|section-group)+>
<!ELEMENT section  (title, (%block.content;)*,(section|section-group)*)>
```

The `section-group` element, above, while grouping `section` elements, does not add a numbered level if included in content. It is merely a convenient container.

1.4.2. Multi-level Links

An inclusion link will often point at a fragment that contains another inclusion link. That fragment might in turn have an inclusion link to another fragment, and so on.

This is not really a problem unless there is a loop reference (in other words, a link that wants to include the document it is placed in, or if another document links "backwards", either directly or after a few levels), or if the multi-level inclusions result in a normalised document that isn't valid. Some processing will take care of both these cases.

Loop references should be checked for when normalising the document before publishing. See Section 3.1.

For the normalised, non-valid, document, parsing that document before publishing it is probably enough, *if* the error message that results is clear to the author. He

[2]With an `ID` value generation mechanism, it shouldn't be too hard to ensure uniqueness throughout the system.

or she would then be required to correct the problem before attempting to publish again.

1.4.3. Special Considerations

There are some special considerations when including XML. Most importantly, the element type of the included fragment should be valid where the linking element is valid. Consider the following:

```
<!ELEMENT book  (chapter+)>
<!ELEMENT chapter  (p|list|inset)*>
<!ELEMENT inset  EMPTY>
<!ATTLIST inset  %xlink.atts;>
```

The resulting instances are assumed to be processed by XSL stylesheets that normalise any links, replacing the links with their target structures, and then use XSL-FO for PDF output, assuming a straight-forward chapter structure that matches the above. Consider this little instance:

```
<chapter>
  <inset xlink:href="inset.xml"/>
</chapter>
```

A valid `inset.xml` would look like this:

```
<p>Some text.</p>
```

This `inset.xml`, on the other hand, would crash the publishing process:

```
<chapter>
  <p>Some other text.</p>
</chapter>
```

And yes, DITA, XInclude, and similar systems have all solved this because of a parse option. On the other hand, it's not a difficult problem; when creating the link, the linking software should check the target element type and see if it is allowed in the context.[3]

A final consideration involves reusable phrases (as discussed in Section 1.6.1). It is often desirable to use the latest version of a phrase, instead of creating a link that will point at a specific version. However, in a mixed system, where some reusable components are phrases while others are block- or section-level structures, always using the latest version is not a good idea. Not only will it be impossible to keep and publish old manuals of old product variants, but the legal ramifications of not being able to do so is frightening enough for any product owner.

Using URNs, wildcards can be an option:

[3]This assumes that the target structure follows the same DTD or schema as the main document. In my experience, this is practically always the case.

```
<inset xlink:href="urn:x-paper:r1:doc123456:en-GB:*"/>
```

The system assumes, in this case, that the asterisk ("*") implies that the latest version is to be used.

A URN parser that can handle wildcards in this manner is harder to construct, however, and can be misused. A better alternative is to use a separate element for phrase reuse, leaving the choice to the processing application:

```
<phrase xlink:href="urn:x-paper:r1:doc123456:en-GB:0.58"/>
```

The `phrase` element is used when linking to phrases, while the `inset` element is used when linking to larger structures.

1.4.4. Reusing Cross-References

A special case of reuse, somewhat out of scope in this paper but nevertheless worthy of discussion, is the reuse (or rather, presentation) of cross-reference online. Have a look at this little example:

```
<p>Discussing rabbits is outside the scope of
this document. For more info, see
<locator xlink:href="#target-id"/>.</p>
```

On paper, this should translate to something like:

Discussing rabbits is outside the scope of this document. For more info, see *Section 3, "More Info", Page 31*.

Online, however, the results would be a bit awkward at best (let's pretend that the empasised words below are a hyperlink):

Discussing rabbits is outside the scope of this document. For more info, see *More Info*.

It gets worse: What if the link wasn't valid in online context and we had to do away with the link? A script could remove the link:

Discussing rabbits is outside the scope of this document. For more info, see.

There are solutions, however. Let us introduce a wrapper element:

```
<p>Discussing rabbits is outside the scope of
this document.<xref> For more info, see
<locator xlink:href="#target-id"/>.</xref></p>
```

On paper, the result is identical to the above, and if the link wasn't valid online, we could simply instruct a script to remove all `xref` elements, resulting in:

Discussing rabbits is outside the scope of this document.

Online (assuming for a moment that the link *is* valid online), the result would still be awkward, as shown above.

If we added a hyperlink element, we could use that instead for any hyperlinks we needed to create, and *always* remove the `xref` element online, making it into a construction applicable for paper publication only. The source XML would become:

```
<p>Discussing <hlink xlink:href="#target-id2">rabbits
</hlink> is outside the scope of this document.
<xref>For more info, see
<locator xlink:href="#target-id"/>.</xref></p>
```

The online output would then look like this:

Discussing *rabbits* is outside the scope of this document.

Which pretty much solves our problem.

1.4.5. What About Other Linking Systems?

There are other linking mechanisms out there, of course. A common technique for inclusion links is *XInclude*. It does pretty much what my `inset` element with its XLink attributes does, above. Earlier drafts even accepted the fragment identifier construct:

```
href="my-document.xml#my-id"
```

In the final recommendation, the fragment ID is no longer allowed, and in its place is an XPointer attribute that includes the `ID`. The `xpointer` attribute can also a relative pointer, however, and we've already established that in the context of reuse, addresses are less than ideal.

My biggest gripe about using XInclude, however, is that it is specifically about including XML in XML. It does not cover other options so if I were to use it for inclusions, I'd have to define other linking mechanisms for the other types of links. I'm not saying that it's wrong to have two (or more) linking systems but it creates more work and more code. In my view, every type of link should use the same basic mechanism and XLink fits the bill.[4]

1.5. Profiling and Filtering

When writing a reusable section, it is necessary to be as generic as possible. Any information specific to a product or product variant (or target market, or any other criteria) will limit its reuse.

A practical way to include model-specific information while keeping the section reusable is to *profile* the model-specific parts. Consider the following:

```
<doc>
  <p>Information common to products A and B.</p>
  <p>Information about product A.</p>
  <p>Information about product B.</p>
</doc>
```

[4]I'm the first to admit the advantages of DITA's `conref` processing, just as I readily admit the benefits of the XInclude `parse` model. Yes, it is sometimes an advantage; my point, however, is that I want to make do with just one linking mechanism.

The above XML fragment includes information about two products, A and B. One paragraph is about both products but the other two concern only one at a time. This fragment is only usable if published in a context that includes both products.[5]

Let's introduce logic to the fragment, and make it reusable for any of the three products:

```
<doc>
  <p>Information common to products A and B.</p>
  <p applic="A">Information about product A.</p>
  <p applic="B">Information about product B.</p>
</doc>
```

The `applic` ("applicability") attribute helps the publishing process determine what the fragment's various structures are about. The attribute identifies the *profile* of the node and its descendants. A profiled node is only applicable if the `applic` value(s) set for the root element match those of the node's. No `applic` attribute means that the node and its descendants are applicable in all contexts.

This approach is very useful. When publishing the fragment in context "A", we'd mark up the fragment like this:

```
<doc applic="A">
  <p>Information common to products A and B.</p>
  <p applic="A">Information about product A.</p>
  <p applic="B">Information about product B.</p>
</doc>
```

The result would be:

Information common to products A and B..

Information about product A.

Similarly, context "B" would only include the common paragraph and the "B" paragraph.

This model also handles several profiles at once:

```
<doc applic="A B">
  <p>Information common to products A and B.</p>
  <p applic="A">Information about product A.</p>
  <p applic="B">Information about product B.</p>
</doc>
```

Here, the context is "A or B", meaning that every node with `applic` values matching either "A" or "B" (or both) are included. The values are separated with spaces, making them easy to process.

A paragraph such as the following would also be included in the output since "B" is included in the list of profiles:

[5] That is, if you want to avoid the all-too-typical situation with a manual that includes every possible accessory available to the product. Consider driver's manuals for cars.

```
<p applic="B C D">Information about B, C, and D.</p>
```

While some profiles could easily be declared as an ennumerated list in the DTD, it is wiser to declare the attribute as `NMTOKENS` or `CDATA`. Product lines change, variants are included and excluded, and it is pointless having to change the DTD every time. Instead, the DMS should provide the author with the allowed `applic` values when authoring.

1.5.1. Variables

It is often desirable to use the name of a product in an otherwise generic section to "personalise" it, for example in a sentence like "thank you for buying *product X*". The profiling discussed earlier can help. Consider a `product-name` element:

```
<!ELEMENT product-name  EMPTY>
<!ATTLIST product-name  applic NMTOKENS #IMPLIED>
```

The welcome sentence would in context A look like this:

```
<p>Thank you for buying <product-name applic="A"/>.</p>
```

The `applic` attribute could either be set by the author when writing, or by the application when publishing the document. It is preferable to have the `applic` values supplied by the DMS. But what about the case when the profile of the published document includes more than one product?

```
<doc applic="A B C">
  <p>Thank you for buying <product-name applic="A B C"/>.</p>
</doc>
```

When publishing the welcome sentence, it is trivial to write an XSLT template that presents the `product-name` values as a list, with the word "and" separating the last items in the list:

Thank you for buying A, B, and C.

However, this also illustrates the danger with the approach. It is possible to produce grammatically incorrect combinations:

```
<p>The <product-name applic="A B C"/> is a high-powered
vehicle for cross-country driving.
```

The A, B, and C is a high-powered vehicle for cross-country driving.

1.5.2. Profiling Markup Considerations

What types of structures should one be allowed to profile? In most of my DTDs I've elected to include just about everything on block level, and everything on section and chapter level, and above.

Inline profiling, however, is trickier. For example, it is dangerous to include profiling on an `emphasis` element:

```
<p applic="B">Click <emphasis applic="C">twice</emphasis> to
abort the self-destruct sequence.</p>
```

Oops. I hope they didn't read the manual too carefully...

Instead, I'd suggest a more generic inline element for inline profiling inline. A simple `wrapper` element that *always includes a complete sentence* is usually enough. It's important to avoid having incomplete grammatical constructs result from the careless use of profiling.

A special case inline is the profiling of link elements for cross-referencing. Cross-referencing a text on product “B” when publishing in context “A”, for example, would create content that cannot be reused. Therefore, when profiling cross-references, one should either always include a link for every conceivable profile, or always profile a cross-reference wrapper element (as discussed in Section 1.4.4) instead, ensuring that only complete sentences are profiled.

On block level, profiles usually break very little. There are a few important exceptions, however. Here's a CALS table row that will cause problems if published outside context “B”:

```
<row>
  <entry><p>Some text.</p></entry>
  <entry applic="B"><p>Some text.</p></entry>
  <entry><p>Some text.</p></entry>
</row>
```

Publishing the table outside context “B” results in the row in the example having two columns instead of three, breaking the publishing process. Therefore, when writing table customisations, allowing profiles on `entry` elements is not a good idea.

Generally speaking, any profile that can break a structure is probably a bad idea. Not every such occurrence will crash a publishing process but they sometimes create awkward situations.

One of my pet peeves when writing list structures is to only construct lists that include at least two list items:

```
<!ELEMENT list  (p?, list-item, list-item+)>
```

A profile on `list-item` can in this case produce a document that is not valid.

Of course, declarations such as these are also dangerous:

```
<!ELEMENT figure  (graphic, caption)>
<!ELEMENT graphic  EMPTY>
<!ATTLIST graphic  applic NMTOKENS #IMPLIED
                   %graphic.atts;>
```

The sloppy use of a profile will result in a `figure` element without the `graphic` element, again resulting in a document that is not valid.

Any markup designed for profiling should therefore be relatively "loose". Any construct using the "required" form should probably be made "optional" if one were to allow profiling.

1.6. Size Matters... Or Does It?

Now that we know what to link to, and how to do it, what *should* we link to? Is there a limit to size? How large (or small) should the smallest reusable component be?

1.6.1. Phrase-Based Reuse

Many authoring systems today provide reuse of "phrases", small chunks of information no larger than a sentence. Typically, phrase-based reuse is common in specialised structures of limited complexity, for example, vehicle diagnostic procedures and installation instructions. The advantages include standardised language and low translation costs. They also find use in systems where the author is not a technical writer but instead someone with little experience in writing documentation.

Phrase-based reuse becomes more difficult when a document grows in complexity. For example, if several reusable phrases are required to form one "block" of information, some combinations of phrases will inevitably result in awkward language and translations.[6]

A partial solution to the problem of combining phrases is to use a style guide to ensure the style used in the original language. A more extreme solution is to use a controlled language such as *Simplified English*.

Another (serious) problem is that the smaller the phrase size, the harder a phrase is to locate. With enough phrases, how do you present them to the user so she can browse and locate the required phrase? With a couple of hundred phrases, searching for the right one becomes time-consuming. The time saved by not having to write or translate the phrase is lost because it takes time to locate it.[7]

The duplication of phrases is also a problem and usually results from not being able to find an exisitng phrase. In time, duplicates become common, further reducing the usefulness of phrase-based reuse.

1.6.2. Reusable Sections

On the other end of the scale, the reuse of large pieces of information, for example, chapters and sections, is common. For example, when authoring a manual for a car, sections on safety, changing tyres, etc, can easily be reused if they are profiled in

[6] Some translations resulting from phrase-based systems with automated translation can produce unwanted hilarity; witness the many Japanese VCR instruction leaflets out there.

[7] A modern translation tool with phrase recognition functionality can automatically translate any phrase already stored in its phrase memory, thus reducing the need to reuse phrases to save translation costs.

the way outlined above (see Section 1.5). However, the larger a section is, the more likely it is that some information specific to the model or product at hand will be included. Engine specs or tyre pressures are typical examples. They are specific to the model, and extensive profiling is required. With a complex enough section, it becomes very difficult to keep track of all the special cases and all the necessary profiling.

We can conclude that a large enough section is not easily reusable because there will be parts specific to a certain model or variant, regardless of any profiling done.

1.6.3. Block-level Reuse

In the systems I've been part of designing, block-level reuse is by far the most common, usually coupled with reused sections that include a basic (and fairly generic) text body, adding links to the blocks specific to product variants when required.[8]

Warnings and other types of admonitions comprise a typical example of block-level reuse. These admonitions are frequently written by a legal department rather than technical authors; linking to them ensures that the admonitions are used verbatim.

Block-level reuse in itself is pretty straight-forward. You can easily reuse anything from paragraphs to admonitions, figures, etc. The DTD will obviously decide what can be reused, but there is seldom any need to limit block-level reuse.

Problems sometimes arise when trying to find the the right "block", however, because there are too many of them. A very simple solution is implied by the linking mechanism itself: the fragment identifier in `URN#ID` allows us to create documents that serve as containers for, say, every warning that can be reused.[9]

The following is a simple container document for warnings:

```
<doc>
  <warning id="warning-1"><p>...</p></warning>
  <warning id="warning-2"><p>...</p></warning>
  <warning id="warning-3"><p>...</p></warning>
  ...
</doc>
```

The references to the warnings take the form `URN#warning-1`, `URN#warning-2`, and so on.

In some cases the use of a containing structure for reusable fragments becomes a disadvantage since the container document is version handled as a unit, instead of version handling the fragments separately. This is a problem if the fragments are

[8] Sometimes the sections are little more than skeleton documents with links.

[9] This approach makes it easier for a legal department to handle the admonitions as one editable unit rather than dozens, maybe hundreds, of separately stored fragments.

updated often; the links made will more often than not point to old versions of the document. A practical problem also arises if many persons are involved in updating the fragments; in most DMSs documents are checked out exclusively by one person at a time.[10]

2. Working with Reusable Material

Working with reusable components can be very easy or impossibly challenging, depending on the help provided. This chapter discusses the editing side of things.

2.1. Editing

Authoring documents that reuse other documents should be made as easy as possible. Making a link, whether it's a cross-reference or an inclusion, should not require the author to do more than absolutely necessary.

Here's what I'd regard as necessary when including a fragment or making a cross-reference:

- A link dialog with functions supporting every aspect of creating the link.
- Search functionality to locate the target fragment, from within the link dialog.
- Help to point out an element inside the included target fragment, if required.
- Creating the necessary element and its attributes.

Anything beyond the above serves only to complicate the author's task. Since the above applies equally to inclusions *and* cross-references, it also demonstrates the need for one linking system instead of several, in my opinion.[11]

Figure 1 shows such a linking dialog. A warning, located in a separate document and identified with a URN, is included in the main document. The target document can be located by clicking the Target Document button.

[10]Some systems solve this by using “optimistic checkout” where the document is not locked when checked out. Other authors can check out and work on the same document at the same time. The system informs the second and subsequent authors of the document's checked-out status, forcing the authors to talk to each other.

[11]The amount of coding required when aspiring for identical interfaces while using different linking systems is not trivial. Programmers may quite reasonably ask why different systems are used; often, they will have solved a problem for one linking system only to have to solve a similar problem for a different system later.

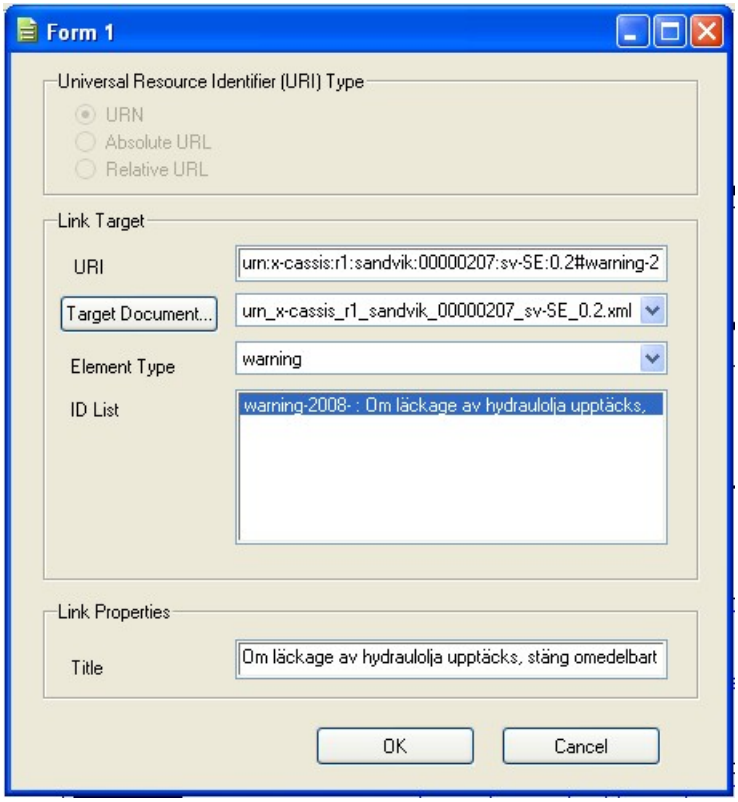


Figure 1. Making a Fragment Inclusion

The application allows including parts of the target document, which allows storing all warnings in a single collection document (as described in Section 1.6.3), making it easy to find them.

Note the Title field in the dialog. It contains the text node(s) of the warning that has been linked to. This information is placed in the `xlink:title` attribute when the link is created and provides a very useful visualisation of the inclusion in the editor. Figure 2 shows how the `xlink:title` contents may appear in an editor.

Figure 2. Included Content in an Editor

The included contents can be accessed by double-clicking the inset link (the `block-inset` element in the picture, above). The target document will open if it is checked out from the database.

Cross-references are created using that same dialog because what happens behind the scenes is identical to the inclusion case. The dialog is invoked by choosing an element and *only creates and edits the XLink attributes*, leaving the process of creating the linking element and attaching the XLink attributes to the element insert function.

Note the Element Type drop-down list in the dialog (see Figure 2). This list reads the elements present in the target document and lists the instances in the ID List list box for an element type.

Creating images is a bit different even though XLink is used here as well (see Figure 3). Inclusions and cross-references both process XML fragments while image links do not, but a user-friendly image dialog should also show a preview of a selected image file.

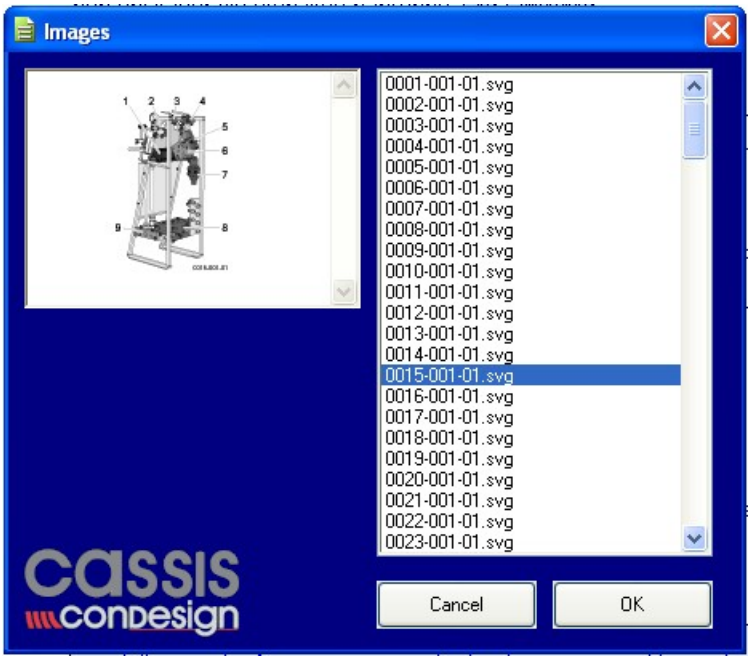


Figure 3. Inserting an Image

The mechanism behind the scenes is identical to the handling of other types of links. There is one addition, however: to be able to *show* the image in the XML editor, the current image URL is placed in the `xlink:role` attribute.[12]

A final linking case involves the reuse of phrases. While the link dialog described above would do the job, an interface requiring fewer steps for link creation is preferable. A dialog that lists the available phrases and allows drag & drop from it to the editor is a possibility.[13]

2.2. Profiling Information

Being able to easily profile a reusable fragment is essential. Thus, while an approach that involves manually editing an `applic` attribute is feasible, it is not practical. What if the profiling values had two levels: "if A, then B and C are possible; if D, A and B are forbidden, but E, F, and G are possible". This is quite common but almost impossible for authors to keep track of.

Figure 4 shows an interface that handles profiling.

[12]The XLink specification describes the `xlink:role` attribute as a URI that gives the link a *role* desribed by the URI but leaves out the processing model. Here, we've interpreted the `xlink:role` attribute as a pointer to a rendition of the semantic resource.

[13]The *mechanism* that creates the actual link is the same as the one behind the link dialog above.

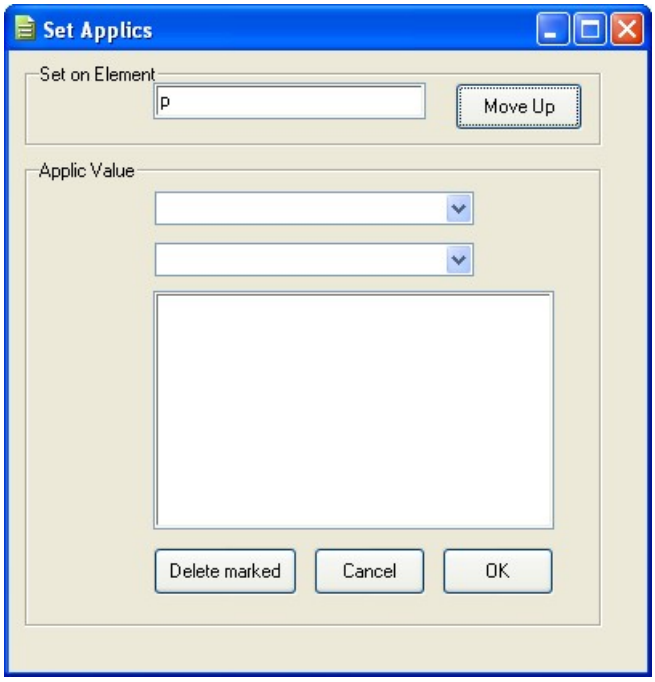


Figure 4. A Profiling Dialog

The Set Applics dialog keeps track of two levels of profiles, one level depending on the other. The values are fetched from the database through a web service chosen with drop-down lists. The dialog also shows which node the profiles are applied on. If the profile(s) should be set on an ancestor element instead, the dialog also allows moving up in the XML structure, picking the nearest allowed ancestor.

A final consideration is to *visualise* the profiles made. The profiled nodes should be clearly formatted and identified as such. Figure 5 illustrates the visualisation of profiled nodes.

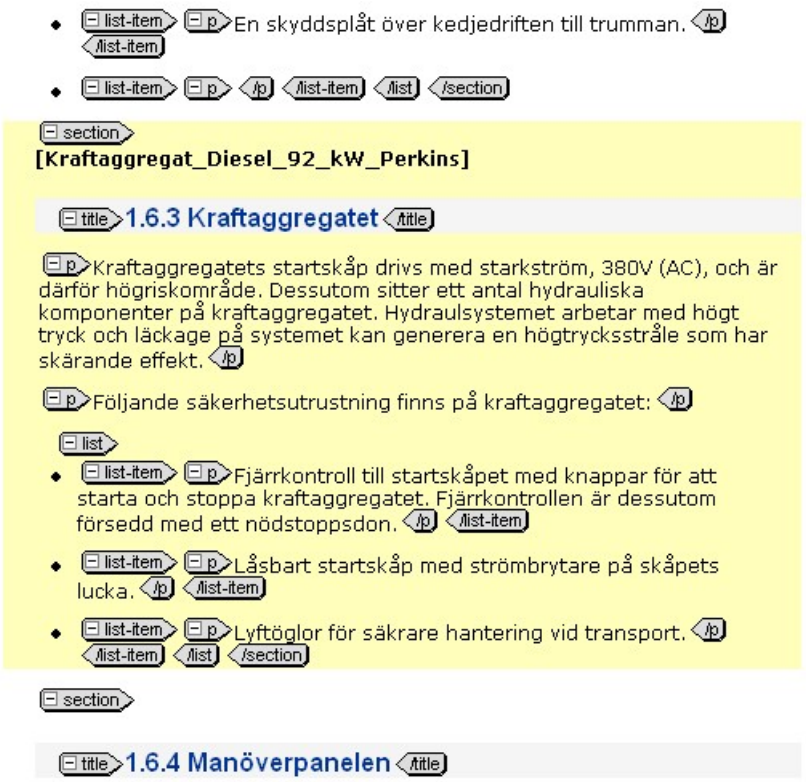


Figure 5. Profiled Nodes Visualised in an Editor

The formatting is achieved using CSS styling. The presence of an `applic` attribute sets the background colour of the `section` element to a pale yellow, and the `applic` attribute's value is displayed right after the start tag.

2.3. Keeping Track of It All

When reusing, a good search mechanism is essential. A DMS should include enough meta-data for each object to facilitate a search based on that meta-data. A search *within* each object might be more difficult, however; an XML-based database should then be considered.

In addition to a search function, a *visualisation* of the links is useful. The normalisation process (described in Section 3.1) used when publishing can also be used to list any and all links, in every participating resource. These links can be visualised in a tree-like structure, using, for example, mind mapping software such as the open-source *FreeMind*[14].

[14]Adding language, version, and workflow status information to each node is also useful. The link tree allows authors, translators, project managers, and many other groups to easily visualise an existing document or plan a new one.

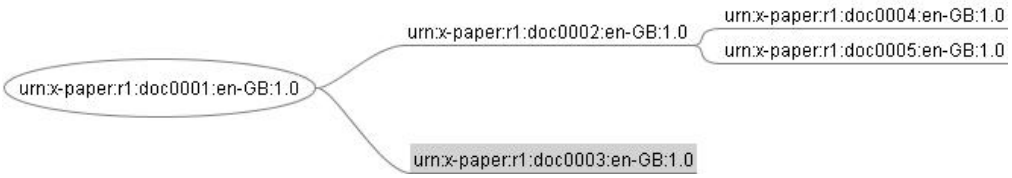


Figure 6. A Mind Map

Of course, even without mind-mapping software, link maps that show the relationships between documents are very easy to present in, say, HTML tables.

3. The Publishing Process

The basic publishing process can be split into these main parts:

- Normalising the linked fragments into one file
- Filtering the contents according to the profiles chosen
- Publishing the combined file

3.1. Normalisation

When publishing a document that contains inclusions, you should first normalise every included fragment into a single file. In this way, it is far easier to write a stylesheet that produces formatted output.

The normalisation process is not complex but does require some processing. If every resource is identified with a URN, one needs to point out the main document, open it in a temporary folder, parse it for any inclusions, locate and open those resources in the same folder, parse these for any inclusions, process them, and so on, until there are no further inclusions to process. The URNs in the `xlink:href` attributes should obviously be replaced with the file names.

Cross-references are easier. A typical cross-reference takes the form `URN#ID` but will in principle only need the `#ID` part because the `ID` is unique and situated in the same physical file.

It is a trivial matter to use an XSLT stylesheet to include all of these documents in a single normalised file.

3.1.1. ID Clashes

ID clashes occur when normalising a document that has linked to the same resource twice. It results in a normalised instance that is not valid. There are several ways to handle this:

First of all, it's not as common a problem as would seem to be the case at first glance. The same content does not often have to be included more than once, in a majority of cases.

However, *if* the same content is indeed included more than once, it only becomes a problem *if the ID values are copied into the normalised document.* This is required only very rarely because the only reason you'd want to copy an ID value to the normalised document is if you'd made a cross-reference to that node. An included resource that is the target of a cross-reference twice (or more times) creates ambiguity and means that an error was made earlier, during authoring.

It is quite possible to unwittingly create such a situation. Extensive profiling, for example, can be the cause of many strange structures, but nevertheless, it should be regarded as an error that needs to be handled before the document can be published.

3.2. Processing Profiled Documents

The profile(s) of a document can be processed either during the normalisation process or when feeding the normalised document to the publishing engine. The former is usually the better idea, for several reasons.

For one thing, the publishing process takes less time if the document has already been filtered. It is smaller and quite possibly, many time-consuming processes (for example, large image files) will have been filtered out at this stage. More importantly, however, the publishing process should only have to deal with publishing.

If the publishing process is XSL-FO and the stylesheets are large enough or split into several modules, the profiling instructions will be spread throughout the stylesheets, making them more complicated to maintain. Here's a typical example of profile processing:

```
<xsl:template match="p">
    <xsl:if test="not(@applic) or contains(@applic,$applic)">
       ...
    </xsl:if>
 </xsl:template>
```

The `xsl:if` instruction handles the profile processing. This sort of instruction would have to be included in every element template that needs to handle an `applic` attribute.

Better is to include the profile processing in the normalisation stylesheet. That stylesheet is small enough to be just one file, which makes maintaining the mechanism a lot easier. Note that the normalisation stylesheet is small mostly because all it does (well, more or less) is to copy XML structures into one large file. The basic copy mechanism looks like this, and only occurs in a select few templates:

```
<xsl:template match="*">
   ...
   <xsl:if test="not(@applic) or $print='yes'">
       <xsl:element name="{name(.)}">
          <xsl:call-template name="attribute-copy"/>
          <xsl:apply-templates/>
       </xsl:element>
   </xsl:if>
 </xsl:template>
```

The profile processing sets the `$print` to the value "yes" if the profile matches, and results in the node being copied.

3.3. Publishing in Other Languages

Let's say that we have a document with the URN `urn:x-paper:r1:doc0001:en-GB:1.0` that links to chapters `urn:x-paper:r1:doc0002:en-GB:1.0` and `urn:x-paper:r1:doc0003:en-GB:1.0`:

```
<doc>
  <inset xlink:href="urn:x-paper:r1:doc0002:en-GB:1.0"/>
  <inset xlink:href="urn:x-paper:r1:doc0003:en-GB:1.0"/>
</doc>
```

How do we publish this document in other languages, say, Swedish? When following the principles outlined in this paper, locating the Swedish versions in the database would consist of nothing more than replacing the `en-GB` language-country codes in the URNs with the Swedish ones and asking the database for them. The Swedish version of the document would be this:

```
<doc>
  <inset xlink:href="urn:x-paper:r1:doc0002:sv-SE:1.0"/>
  <inset xlink:href="urn:x-paper:r1:doc0003:sv-SE:1.0"/>
</doc>
```

The point here is that since we regard the different language versions of a document as different renditions of the same semantic document, finding the documents and processing the links are very straight-forward tasks. This skeleton document could be generated on the fly, if need be; with a DMS geared for URNs, finding the translated chapters could be performed when normalising the skeleton.

If the included chapters do not exist in Swedish versions, the normalisation process stops because the links cannot be processed. A clever enough process will instead offer to create a package of the missing documents, ready to be translated.

But what about country- or market-specific profiling? Doesn't this URN scheme make such customisations difficult? It could be argued that a translated version is a rendition by *our* definition so if country- or market-specific customisations are made during translation, we could simply define the customised and translated

version as a rendition of the original. Who would know the difference? The problem with this is that, as I've indicated earlier (see Section 3.3), the translations are not version handled separately; they always have the same version as the original. Therefore, any customising for a specific market or country would have to be handled using either no version handling at all, or a version handled kept separate from the main system.

An easier, and more true to form, approach is to handle market- and country-specific customisations as profiles like any other. A customisation for a specific market is profiling and should be treated as such. Anything using a language/country code (including an `xml:lang` attribute, if used, and certainly the language/ country code used in the URNs) is, in fact, meant to handle content that will appear in a document *after* filtering and customisation has already been performed.[15]

4. Shouldn't Everything Be URNs?

Early on in a project, a programmer asked me if URNs should be used for images, too. My first effort at implementing URNs for resources did not include images. I saw the image links as URLs to the image database and little more, and he quite rightly pointed out that I was violating my own principles. I only had to think for the briefest of moments: of course the images should be handled using URNs. They are version handled, and they sometimes require translation. Why shouldn't they be handled as resources in the exact same way as everything else?

The programmer went one step further, however. He proposed that profiles should also be handled using URNs. The idea is that there should be an abstraction layer between a profile consisting of several separate values for different products, markets, and so on, allowing the easy update of those separate values without ever touching the actual profile string. An additional bonus would be to be able to define a boolean condition as a single profile ("A *and* B *and* C"); this combination would equal exactly one URN.[16]

I'm not entirely clear about how a profiling URN scheme would look like but I do like the idea. It would be an advantage to use profiles as "meta-resources", and the code used to resolve URNs in other places would come in handy here as well.

5. Conclusions

In conclusion, here's what I would list among best practices when developing an XML system that allows reuse:

[15] If market-specific customisations are common, a better approach might be to allow for two separate sets of profiling markup: one attribute would handle product-specific profiles while another might focus on market-specific customisations.

[16] The profiles "A", "B", and "C" would also merit URNs of their own.

- Use names rather than addresses, and if at all possible, use URNs rather than your own schemes.
- Have the DMS generate URNs and ID values.
- Consider defining your translations as renditions of the original document.
- Use one linking system instead of many.
- Use profiling to increase reuse.
- Have your DMS handle the profiles.
- Avoid phrase-level reuse in more complex structures.
- Always avoid any profiling or linking mechanisms that might break a resulting normalised file.
- Don't assume your authors are familiar with URNs, profiling details, or various XML constructs; offer user interfaces that ease their tasks instead.

Exploring XProc

Norman Walsh
MarkLogic
<norman.walsh@marklogic.com>

Abstract

This presentation will explore the current state of XProc: An XML Pipeline Language through a combination of slides and live demos. Particular attention will be paid to demonstrating pipelines that are, or could be, useful to solve real world problems.

Optimizing XML Content Delivery with XProc

Vojtěch Toman
EMC Corporation
<toman_vojtech@emc.com>

Abstract

As XProc implementations are becoming more mature, the standard is attracting growing interest in the XML community. We will discuss the benefits of using XProc from an application developer point of view, and how it can make XML applications more robust and reliable. We will then briefly introduce the EMC's XProc processor implementation, which has been successfully deployed in an XML content delivery platform. Using a number of use cases from the content delivery domain, we will illustrate how XProc pipelines can be used for implementing the relevant functionality.

1. Introduction

XProc, or the XML Pipeline Language [6], has every potential to become one of the most useful new XML technologies around. The language has recently become a W3C Candidate Recommendation and is attracting growing interest in the XML community, both from users and implementers.

Following the progress of the specification, and the evolution of the language to its present form, an XProc processor implementation has been developed at EMC. The processor has been successfully deployed in a dynamic content delivery platform, where XProc has quickly proved its strengths and established itself as the primary technology for XML data manipulations.

The benefits of using a declarative XML processing model over traditional approaches are many. XProc streamlines the development of XML applications and makes their architecture cleaner and more robust. XProc bridges the gaps between different XML technologies — and, in turn, bridges the gap between XML and application developers. But perhaps most importantly, XProc can make XML processing fun again.

2. Application Development with XProc

Manual XML programming has always been a mundane and an error-prone task, with quite a steep learning curve as well: the beginning developer has to get famil-

iar with the available processing data models (and pick the most appropriate one for his needs), learn the programming APIs, and then, after some experimentation, rely on the help of "pro's" when things don't work as expected ("How do I move a DOM node from one document to another?").

Additional problems may arise when an application needs to integrate multiple technologies or tools for different kinds of XML processing. In order to make these tools work together, the developer often needs to write code that converts the output of one tool to structures that can by accepted by another. The need for this conversion is obvious in cases when heterogeneous data models are used (such as a combined relational and XML-based storage), but a translation of some kind is often unavoidable also in pure XML environments: the different tools may be based on different paradigms or processing models, or their APIs are just not directly compatible.

Integrating different XML processing tools can be a non-trivial task, and there is also a great danger of mistakes that can lead to unnecessary performance bottlenecks — or worse, to fatal error conditions in the application.

Another problem is that one-to-one mapping between different models is not always possible. Some information may get changed — or lost — during the conversion, because there is no natural representation for it in the target domain. This phenomenon is often referred to as *impedance mismatch* and has always been a source of problems when developing more complex XML applications.

With XProc, many of the issues described above can be reduced significantly. Emphasizing a declarative approach to XML data manipulations, XProc shields the developer from the complicated (and from the XML perspective often unimportant and distracting) details of the underlying XML frameworks and tools. In XProc, you specify *what* actions (and in what order) should be performed on XML data, but the actual implementation of *how* this is done is left to the XProc processor.

This is a very important aspect of XProc, since it can make the XML applications much easier, and therefore cheaper, to develop. Looking from a different perspective, XProc-based applications will also likely be more stable and reliable, simply because the amount of actual XML programming is reduced, and therefore the risk of bugs in the application code is lower. To put it bluntly, XProc can protect the applications from poor XML programming practices.

Using a declarative XML processing model has also a positive effect on the maintainability of the applications — it is usually easier to detect and fix problems in an XProc pipeline than in the application code, which is often hard to understand or entangled with other pieces of the application.

Another interesting benefit of using XProc is that it can make the applications much easier to customize and extend. Very often, new functionality can be introduced by simply adding new pipelines to the application (or by modifying the existing ones), with no or very little changes to the application itself. In the traditional model, especially with larger applications or application frameworks, customizations

are often quite costly (if possible at all), simply because the internal model is not flexible enough.

3. Enabling other XML Standards

XProc can play an important role as an *enabling technology* for other XML standards that require a certain level of XML processing capabilities.

A nice example of this is the XForms specification [5]. The standard has been around for some time, but has never quite taken off on a larger scale, even though there are a number or XForms implementations available these days.

From the ground up, XForms is entirely based on the XML model. While this clearly is an advantage from the design perspective, it can also be seen as one of the main reasons of the low adoption of the standard. The XForms model and the XForms submission data are XML documents, and to manipulate them, the ability to process XML data is necessary.

With the increasing proliferation of native XML offerings, and growing adoption of the XQuery language [8] in these tools, things are slowly getting in motion for XForms — as demonstrated, for instance, by the recent buzz around the XRX (XForms/REST/XQuery) web application architecture [9]. In short, XRX makes it possible to deploy XForms in end-to-end XML environments, eliminating the need for using non-XML models, such as middle-tier objects or (often costly) conversions of the submission data to relational structures.

XProc is a natural fit for the XRX architecture. By substituting XQuery by XProc, the resulting XForms/REST/XProc scheme is as powerful (XQuery is supported in XProc), while adding all the benefits of the declarative XProc processing model.

4. Calumet: The XML Peace-Pipe

> Bathe now in the stream before you,
> Wash the war-paint from your faces,
> Wash the blood-stains from your fingers,
> Bury your war-clubs and your weapons,
> Break the red stone from this quarry,
> Mould it and make it into Peace-Pipes,
> Take the reeds that grow beside you,
> Deck them with your brightest feathers,
> Smoke the calumet together,
> And as brothers live henceforward!
>
> — *Henry Wadsworth Longfellow, The Song of Hiawatha, Book I: The Peace-Pipe*

Calumet is the codename of the XProc processor being developed at EMC. The processor implements most of the required features demanded by the XProc specification, as well as a number of optional features. You can follow the progress of the implementation at the XProc Test Suite page [7].

The processor, together with a suite of other XML tools, will be made available through the EMC Developer Network [3], free for developer use.

The processor is written in Java, using a number of open-source XML components. Figure 1 shows the architecture and the individual components of the system.

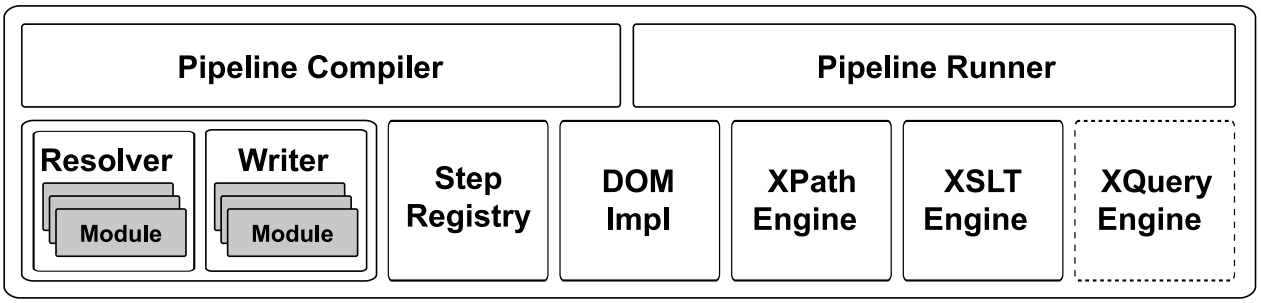


Figure 1. Overall Architecture

Conceptually, the processor consists of two main modules: the *pipeline compiler* and the *pipeline runner*. The pipeline compiler parses XProc pipelines and "compiles" them into a Java object representation that can be then executed by the pipeline runner. A compiled pipeline is a self-contained, reusable structure that can be run multiple times with different input data and parameters — an idea borrowed from the Java XML Transformation API [4], where similar functionality is supported through the `Templates` interface.

Resource resolution in Calumet is URI-based. The processor uses a pluggable system of so called *resolver* and *writer* modules, one for each supported URI scheme. By default, resources can be resolved from the file system, from the Java classpath, and over HTTP.

The *step registry* manages the implementations of all atomic XProc steps that are available to the processor. Using the registry API, application developers can provide implementations of custom atomic steps.

The processor is based on the DOM processing model; most of the XML manipulations are implemented using DOM operations. The processor supports XPath 1.0 as the expression language and XSLT 1.0 for transformations.

The architecture is open in the sense that application developers can register plug-ins with the processor, which makes it possible to customize the default behavior of the system or add new functionality. Besides providing implementations of additional XProc steps, plug-ins can also be used for extending the processor's I/O capabilities (by registering custom resolver and writer modules) or, for instance, for registering a custom DOM implementation.

The ability to use a custom DOM implementation is a useful feature of the processor. By default, the processor uses Apache Xerces as the underlying DOM implementation, but nothing prevents the application developers from switching to another, possibly more efficient, DOM implementation. It is possible, for instance, to deploy the processor on top of a native XML database that provides a DOM interface. Doing so may not only boost the performance, but it may also bring additional functionality, such as XQuery support, XML data indexes, or transaction control to the processor.

5. XML Content Delivery Use Cases

Modern XML content delivery systems allow for dynamic publishing of highly personalized content. Often leveraging the power of native XML databases, XQuery and other XML-related standards, they provide such functionality as support for constructing dynamic content assemblies from existing (or generated) content, or advanced content profiling driven by the preferences of the end-user (or by the content itself).

From the XML processing perspective, it is only natural to incorporate the XProc pipeline model in the content delivery context. In this section, we present a number of simple use cases and illustrate how XProc can be used for implementing the relevant functionality. For each use case, we start by formulating an introductory scenario (usually with a very simple solution) to describe the problem — and to show that even basic XProc pipelines often provide sufficient functionality. After that, we move on to a more challenging scenario, which will require using more complex XProc pipelines.

5.1. Content Publishing: Scenario I

The first example shows a likely candidate for the most frequently used XProc pipeline of all — a pipeline that will probably be present, in one form or another, in most XProc-based XML applications.

The pipeline takes an XML document and an XSLT stylesheet on its input and returns the result of applying the stylesheet to the document. The pipeline also supports passing parameters to the stylesheet through an (implicit) parameter input port.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            name="main">

  <p:input port="stylesheet"/>

  <p:xslt>
    <p:input port="stylesheet">
```

```
      <p:pipe step="main" port="stylesheet"/>
    </p:input>
  </p:xslt>

</p:pipeline>
```

5.2. Content Publishing: Scenario II

The pipeline in this section demonstrates the use of the `p:xquery` XProc step for publishing dynamically generated content.

The example assumes a model where application users can add comments to the XML content, by inserting `comment` elements in the documents:

```
<section>
  <comment user="jnovak123">
    <text>The text needs updating.</text>
  </comment>

  <para>...</para>
</section>
```

The pipeline runs a simple XQuery on a collection of input documents to find all comments by a particular user. The user name is a parameter to the XQuery, and is expected to appear on the `parameters` parameter input port of the pipeline.

The XQuery results are transformed into an XSL-FO document which is then passed to an XSL formatter. The resulting PDF report is written to the location specified by the *output-uri* option of the pipeline.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:c="http://www.w3.org/ns/xproc-step"
                name="main">

  <p:input port="source" sequence="true"/>
  <p:input port="parameters" kind="parameter"/>
  <p:option name="output-uri" required="true"/>

  <p:xquery>
    <p:input port="query">
      <p:inline>
        <c:query>
          declare variable $user as xs:string external;
          <comments user="{$user}">
          {
            for $doc in collection()
            let $comments := $doc//comment[@user=$user]
            return (
              for $comment in $comments
```

```
                return <comment doc="{base-uri($doc)}">
                       {$comment/text}
                       </comment>
              )
            }
            </comments>
          </c:query>
        </p:inline>
      </p:input>
    </p:xquery>

    <p:xslt>
      <p:input port="stylesheet">
        <p:document href="comments2fo.xsl"/>
      </p:input>
      <p:input port="parameters">
        <p:empty/>
      </p:input>
    </p:xslt>

    <p:xsl-formatter>
      <p:with-option name="href" select="$output-uri">
        <p:empty/>
      </p:with-option>
    </p:xsl-formatter>

  </p:declare-step>
```

5.3. Content Assembly: Scenario I

One of the great advantages of the XML storage model is that it allows for a natural reuse of content, both on document and node level. Componentizing the content into independent, reusable units not only reduces the duplication of information, but it also increases the flexibility of applications in terms of delivering highly personalized dynamic content.

In this example, we assume a componentized content model where individual components can be reused using XInclude references:

```
<chapter xmlns:xi="http://www.w3.org/2001/XInclude">
  <title>Chapter XXXVII. - In which it is Shown that Phileas Fogg Gained
     Nothing By His Tour Around the World, Unless It Were Happiness</title>

  <xi:include href="sect1.xml"/>
  <xi:include href="sect2.xml"/>
  <xi:include href="sect3.xml"/>
</chapter>
```

XInclude is supported out-of-the box in XProc. The pipeline below takes the input XML document, applies XInclude processing, and returns the resulting XML document.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc">
  <p:xinclude/>
</p:pipeline>
```

5.4. Content Assembly: Scenario II

DITA [2], or the Darwin Information Typing Architecture, is an XML standard for creating technical documentation. DITA is based on a componentized model: content is organized into topics, self-contained units of information from which structured documentation is constructed. This is done using so-called DITA maps, "table of contents"-like documents with references to individual topics.

Topic references are a form of reuse on the document level. In addition to that, DITA supports also reuse of parts of topics such as, for instance, common warning statements or notes. Each element in DITA documents can contain a `conref` attribute that points to the content unit that should be reused. The behavior of a conref is that the referenced content appears in place in the topic as if it was copied there, replacing the element that contained the reference.

There are rules as to what content can be referenced and when (for instance, the new content must be allowed in the target context), and also how the attributes of the source and target element are processed.

To be able to address individual elements in the documents, each element can be given a unique identifier. Identifiers in DITA are scoped: topic elements must be given an identifier (and this identifier must be unique among all topics) and the contained elements must use identifiers that are unique within the topic. DITA uses the following syntax for conrefs:

```
filename.xml#topic-id/element-id
```

In the case of local references (within a topic), the file name component can be omitted — as can be seen in the following example DITA topic:

```
<concept id="telephone">
  <title>Telephone</title>
  <conbody>
    ...
    <p>
      <note id="hangupphone">You must hang up your phone before
         you can make another call.</note>
    </p>
    ...
    <p>
```

```
      <note conref="#telephone/hangupphone"/>
    </p>
    ...
  </conbody>
</concept>
```

The example below presents a simple pipeline that resolves local conrefs. The pipeline is recursive (because conrefs can be nested) and makes use of `p:viewport`, a powerful construct in the core XProc language. (The pipeline does not address all issues that can arise when resolving conrefs, such as proper handling of element attributes or detection of loops. It also does not test whether resolving a conref results in valid DITA content.)

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            xmlns:c="http://www.w3.org/ns/xproc-step"
            xmlns:ex="http://xmlprague.cz/ns/xproc-examples"
            type="ex:resolve-local-conrefs" name="main">

  <p:viewport match="*[@conref]">
    <p:variable name="conref" select="/*/@conref"/>
   <p:variable name="topic-id" select="substring-before(concat($conref, '/'), ▶
'/')"/>
    <p:variable name="element-id" select="substring-after($conref, '/')"/>

    <p:try>
      <p:group>
        <p:output port="result"/>
        <p:identity>
          <p:input port="source" select="/*[@id=substring-after($topic-id, ▶
'#')]">
            <p:pipe step="main" port="source"/>
          </p:input>
        </p:identity>

        <p:identity>
          <p:input port="source" select="//*[@id=$element-id]"/>
        </p:identity>

        <p:delete match="/*/@id"/>
      </p:group>

      <p:catch>
        <p:error code="ex:CONREF-ERROR">
          <p:input port="source">
            <p:inline>
              <message>Unresolvable conref</message>
            </p:inline>
```

```
          </p:input>
        </p:error>
        <p:identity>
          <p:input port="source">
            <p:empty/>
          </p:input>
        </p:identity>
      </p:catch>
    </p:try>
  </p:viewport>

  <p:choose>
    <p:when test="//*[@conref]">
      <ex:resolve-local-conrefs/>
    </p:when>
    <p:otherwise>
      <p:identity/>
    </p:otherwise>
  </p:choose>

</p:pipeline>
```

5.5. Content Profiling: Scenario I

In DocBook [1], elements can use the `condition` attribute for specifying the effectivity information for individual pieces of content. The semantics of the `condition` attribute is application-specific.

Suppose the DocBook documents can be profiled for internal and external use. In his example, this is achieved by defining two possible values for the `condition` attribute: `internal` and `external`.

The pipeline below declares an option named *effectivity*. The option is not required and is set to `external` by default. The pipeline processes the input document and deletes all elements with an `condition` attribute whose value is different from the specified effectivity.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc">
  <p:option name="effectivity" select="external">
    <p:empty/>
  </p:option>

  <p:delete match="*[@condition != $effectivity]"/>

</p:pipeline>
```

5.6. Content Profiling: Scenario II

In real-life applications, effectivity filtering rules are typically much more complex. For instance, in an application for delivery of product documentation, the end-result of the publishing process may depend on a whole range of criteria — the configuration of the product, the target audience (product user, maintenance personnel), etc.

This example uses an assertion-based effectivity model. Each element in the input document can contain an `effectivity` child element that specifies a list of assertions:

```
<section>
  <effectivity>
    <assert name="product-name" value="Tatra T600"/>
    <assert name="audience" value="collector"/>
  </effectivity>

  <title>Tatra T600</title>

  <para>The T600 had a monocoque streamlined six-seater saloon body
     with a drag coefficient of just 0.32. It was powered by an air-cooled
     flat 4 cylinder 1,952 cc rear engine. 6,342 were made, 2,100 of them
     in Mladá Boleslav.</para>
</section>
```

The `assert` elements specify the names of the effectivity attributes and the values required in order to consider a content item effective.

When processing a document with effectivity information, the values for effectivity attributes must be provided by the application (or the end-user). In this example, this information is represented using a simple configuration document:

```
<configuration>
  <attr name="product-name" value="Škoda 1000 MB"/>
  <attr name="audience" value="mechanic"/>
</configuration>
```

The pipeline below takes a document with effectivity information and a separate configuration document, and performs effectivity filtering on the content.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            xmlns:c="http://www.w3.org/ns/xproc-step"
            xmlns:ex="http://xmlprague.cz/ns/xproc-examples"
            type="ex:process-effectivity" name="main">

  <p:input port="configuration"/>

  <p:import href="eval-effectivity.xpl"/>

  <p:viewport name="viewport" match="*[effectivity]">
```

```
    <ex:eval-effectivity>
      <p:input port="effectivity" select="/*/effectivity"/>
      <p:input port="configuration">
`<        <p:pipe step="main" port="configuration"/>
      </p:input>
    </ex:eval-effectivity>

    <p:choose>
      <p:when test="/c:result='true'">
        <p:delete match="/*/effectivity">
          <p:input port="source">
            <p:pipe step="viewport" port="current"/>
          </p:input>
        </p:delete>

        <ex:process-effectivity>
          <p:input port="configuration">
            <p:pipe step="main" port="configuration"/>
          </p:input>
        </ex:process-effectivity>
      </p:when>

      <p:otherwise>
        <p:identity>
          <p:input port="source">
            <p:empty/>
          </p:input>
        </p:identity>
      </p:otherwise>
    </p:choose>
  </p:viewport>

</p:pipeline>
```

To keep the source code readable, the actual effectivity evaluation is implemented in an external pipeline (`ex:eval-effectivity` in `eval-effectivity.xpl`). This pipeline matches the effectivity information of a content item with given configuration and decides whether the content item is effective. The boolean result is wrapped in a `c:result` document.

A possible implementation is presented below. For each `assert` element in the effectivity document, the pipeline tries to find a corresponding `attr` element in the configuration document. If no such element can be found, the content item is not effective and the pipeline returns `false`.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
```

```
            xmlns:c="http://www.w3.org/ns/xproc-step"
            xmlns:ex="http://xmlprague.cz/ns/xproc-examples"
            type="ex:eval-effectivity" name="eval-effectivity">

  <p:input port="configuration"/>

  <p:for-each>
    <p:iteration-source select="/*/assert"/>

    <p:variable name="aname" select="/*/@name"/>
    <p:variable name="avalue" select="/*/@value"/>

    <p:choose>
      <p:xpath-context>
        <p:pipe step="eval-effectivity" port="configuration"/>
      </p:xpath-context>

      <p:when test="/*/attr[@name=$aname and @value=$avalue]">
        <p:identity>
          <p:input port="source">
            <p:empty/>
          </p:input>
        </p:identity>
      </p:when>
      <p:otherwise>
        <p:identity>
          <p:input port="source">
            <p:inline><no-match/></p:inline>
          </p:input>
        </p:identity>
      </p:otherwise>
    </p:choose>
  </p:for-each>

  <p:count/>

  <p:string-replace match="c:result/text()" replace=". = '0'"/>

</p:pipeline>
```

6. Conclusion

XProc provides a powerful declarative model for XML processing. With the emergence of first XProc implementations, and with their increasing maturity, we can expect a growing adoption of XProc as a tool for XML data manipulations in XML applications.

We discussed some of the benefits of using XProc in XML application development. We briefly introduced the EMC's XProc implementation, and presented a number of simple use cases that demonstrated how XProc can be leveraged in XML content delivery environments.

References

[1] *DocBook.org*. http://www.docbook.org.

[2] Don Day, Michael Priestley, and JoAnn Hackos. *Darwin Information Typing Architecture (DITA) Architectural Specification v1.0*. OASIS Standard. 9 May November 2005. http://docs.oasis-open.org/dita/v1.0/archspec/ditaspec.toc.html.

[3] *EMC Developer Network*. http://developer.emc.com.

[4] *Java API for XML Processing*. https://jaxp.dev.java.net.

[5] John M. Boyer. *XForms 1.1*. W3C Candidate Recommendation. 29 November 2007. http://www.w3.org/TR/xforms11/.

[6] Norman Walsh, Alex Milowski, and Henry S. Thompson. *XProc: An XML Pipeline Language*. W3C Candidate Recommendation. 26 November 2008. http://www.w3.org/TR/xproc/.

[7] *XProc Test Suite*. http://tests.xproc.org.

[8] Scott Boagg, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C Recommendation. 23 January 2007. http://www.w3.org/TR/xquery/.

[9] *Web Development with XRX*. http://en.wikibooks.org/wiki/XRX.

A practical introduction to EXSLT 2.0

Florent Georges
fgeorges.org
<fgeorges@fgeorges.org>

EXSLT is a library of extension functions and instructions for XSLT 1.0. It defines several modules, providing features like regular expressions, dates & times manipulation functions, or dynamic evaluation of XPath expressions provided as strings. The most popular of them was without any doubt the node-set() function, allowing one to use a result tree fragment as a node-set.

With the new version of XSLT, some of the extensions provided by EXSLT are not needed anymore. Regular expressions for instance have been included in XPath 2.0 and XSLT 2.0, and node-set() does not make sense anymore because result tree fragments do not exist in XSLT 2.0. Other extensions are still usefull as trigonometric functions. Or the dyn:evaluate() function. But the later should be redefined and adapted to the new data model, and the new definitions of context in XSLT 2.0.

And besides those adaptations, the new features in XPath and XSLT 2.0 brought their own needs for new extensions. For instance sequences, that are the core of the new data model. They cannot be nested, and that is consistent with the need for compatibility with XSLT 1.0. But that limitation also prevent us to define more complex data structures that do not require copies and losing information.

Eventually, new abilities broaden the scope of possible applications. Because we can do more complex treatments more easily, we just want to be able to do more complex stuff yet. This kind of extensions encompasses the ability to parse HTML or XML fragments from strings, to send HTTP requests, to get information from a file system, or to handle ZIP files. And much more.

This introduction will show some live examples with two new proposed extensions. The first one is an HTTP Client to send HTTP requests and use their responses. This allows one to get resources from the Internet, to ask web services (SOAP and REST, like Google APIs,) and even to build a wsdl2xslt stylesheet. The second one is an extension to deal with ZIP files, to be able to handle ODF and Open XML documents.

High-performance XML: theory and practice

Alex Brown
Griffin Brown Digital Publishing Ltd.
<alexb@griffinbrown.co.uk>

Andrew Sales
Griffin Brown Digital Publishing Ltd.
<andrews@griffinbrown.co.uk>

Abstract

At the 2006 Extreme Markup conference in Montreal Alex Brown presented a paper [2] outlining a method of XML processing based around "frozen streams" which seemed to promise better memory usage and execution time for common XML processing operations. This paper briefly revisits the theory, presents the results of implementing it, enumerates the lessons learned and suggests new ways forward.

1. Introduction

As XML becomes an increasingly pervasive technology for data storage and processing, many adopters of the technology face a practical problem caused by the perceived slow performance of many XML processing operations, particularly in comparison to tried and trusted RDBMS-based and native bespoke solutions that are being replaced.

This paper will consider whether indeed XML processing operations are "slow" and if so what the fundamental causes of this slowness are. In particular it will consider the case of parsing larger (i.e., more than a few megabyte) documents when using XML system programmed with Java.

The paper will then outline a method, termed a "frozen stream" of in memory-storage of XML that differs from the common tree based approach and instead relies on a view build on stream-based XML parsing methods The approach is described, and it features and options set out. A benchmark is shown for comparison with other methods of in-memory XML representation The Paper goes on to demonstrate how this approach may be integrated with existing XML processing systems, reports on real-world implemention experience, and considers further enhancements that may be made in future.

2. Technical Background

This paper considers some of the problems of XML performance in general, but in particular is concerned with performance considerations when processing XML into an in-memory representation for the purpose of performing operations on it; principally read operations. The "real-world" problem that led the authors to investigate this problem area was the need for high-performance implementations of the Schematron (ISO/IEC 19757-3:2003) schema language.

The language and platform used in all the examples in this paper is Java™/ Java is a popular choice of programming language for developing applications which process XML. While there are alternatives, it is usually Java that has the most depth and breadth of implementation for the various XML technologies, and anybody working with XML in commercial development will usually find the requirement to implement solutions in Java occurs fairly frequently, not least because of JVMs' reasonable cross-platform abilities. While the use of Java imposes its own particular flavour to this study; the questions raised are applicable to other languages and platforms too.

3. The Current Situation

To arrive at some idea of the kind of memory footprint and speeds XML users might experience, here are some benchmarks for carrying out some common XML tasks. The document is the main content (document.xml) of the Ecma 376 standard, available for download from http://www.ecma-international.org/publications/standards/Ecma-376.htm, a document of approximately 60 MB.

Table 1. Memory use and time taken for common XML tasks

Task	Time Taken	Memory needed
Build DOM tree	14.1s	231 MB
Saxon identity transform	40.7s	237 MB
SAX Parse	5.7s	< 2 MB

The table above shows the time and memory use for some common XML operations performed by applications running within a Java virtual machine. Rather than use diagnostic tools to measure memory use, the "memory required" figure was determined by finding the smallest amount of memory in which the operation could be performed, using trial and error on the command line by manually setting the maxium heap size for the JVM. This "transient usage" figure is a meaningful one when considering users' experiences of XML efficiency – as it represents a firm requirement. If there are not (for example) 237 MB of memory that can be allocated to this XSLT process, it simply cannot be complete

4. Why in-memory models?

The primary motiviation for developing an efficient in-memory model was for use with implementations of the Schematron language. In our experience Schematron was becoming a more and more used validation technology, often deployed at numerous points throughout XML-based digital workflow to ensure the quality of the documents flowing through them. The continual need to build memory inefficient in-memory models for executing the XPath-based queries described by Schematron was putting a heavy load on the infrastructure used in such workflows.

Of course, for some subset of Schematron queries and for some subset of other kinds of activity (notably, XML transformations, it is be possible to implement a streaming approach, and much research in the area is ongoing.[14]. However, it currently appears that for a fully conforming XPath implementation, a fully-streaming approach is not achievable.

5. Memory and Execution Time

It is often held in computing that there is a trade-off between memory used and speed of execution. However, [1] challenges this general assertion ("It has been my experience more frequently […] that reducing a program's space requirements also reduces its run time") and proposes that rather than accepting any trade-off that a path of "mutual improvement" should be pursued (p. 7).

Could this hold for XML document processing too?

5.1. The Object overhead

XML models, DOM in particular, are such that they tend to impose or imply certain things about implementation. The lynchpin of the DOM model is the org.w3c.Node Interface, and implementing classes using its method signatures are steered towards an Object-rich representation in Java simply because they return and expect as parameters Object-subclassed objects. Indeed a naïve implementation of this interface would have a Node with many String properties, for its name, Namespace URI, etc.

As we shall see below, there are good reasons for resisting this when it comes to implementing XML in-memory schemes using Java, and indeed for bypassing what might be seen as a central Java tenet, the central role that Object and its subclasses play.

5.1.1. Java bloat

The major contributory factor to the bloatiness of Java DOMs is an underlying bloat associated with the creation of Java objects. Consider the following simple Java program:

```
class Objs
{
  public static void main( String[] args )
  {
    // create one million empty Strings
    String[] objs = new String[ 1000000 ];
    for( int i = 0; i < 1000000; i++ )
    {
      objs[ i ] = ( "" + i );
    }
  }
}
```

The heap space necessary for this program to execute to completion is 50 MB, to create strings which collectively contain only 5.89 million characters.

Note
All timings/memory usages in this paper are taken running from the command line on a 2.4GHz Pentium desktop machine using a Sun JVM (1.6.0_11-b03) running Microsoft® Windows™ XP

So even allowing, in this example, for the memory taken by the array's references themselves, we can still as a rule of thumb reckon that every Java String costs approximately 40 bytes of memory in addition to its native charactee content. Clearly, any memory-efficient XML storage implementation needs to work around this inefficiency.

6. The Frozen Stream

Given the XML document:

```
<root a='value'>
  <e>foo</e>
  <e>bar</e>
  <e>zxc</e>
  </root>
```

Then the usual programmatic representations of this document are as a tree (when using, for example, a DOM parser), or as a series of events (when using a SAX based parser).

As we have seen, the disadvantage of the tree-based models as implemented in Java is their profligacy with memory; the disadvantage of stream-based models is that they are intractable — we need to represent the entire document in memory for it to be efficiently queried.

We have also seen that an approach which implements XML document as a large number of Java Objects is doomed to use several times as much memory as the serialised document.

Thus a key component of a more memory approach must be to minimise the number of Java Objects used. More primitive types must instead be used.

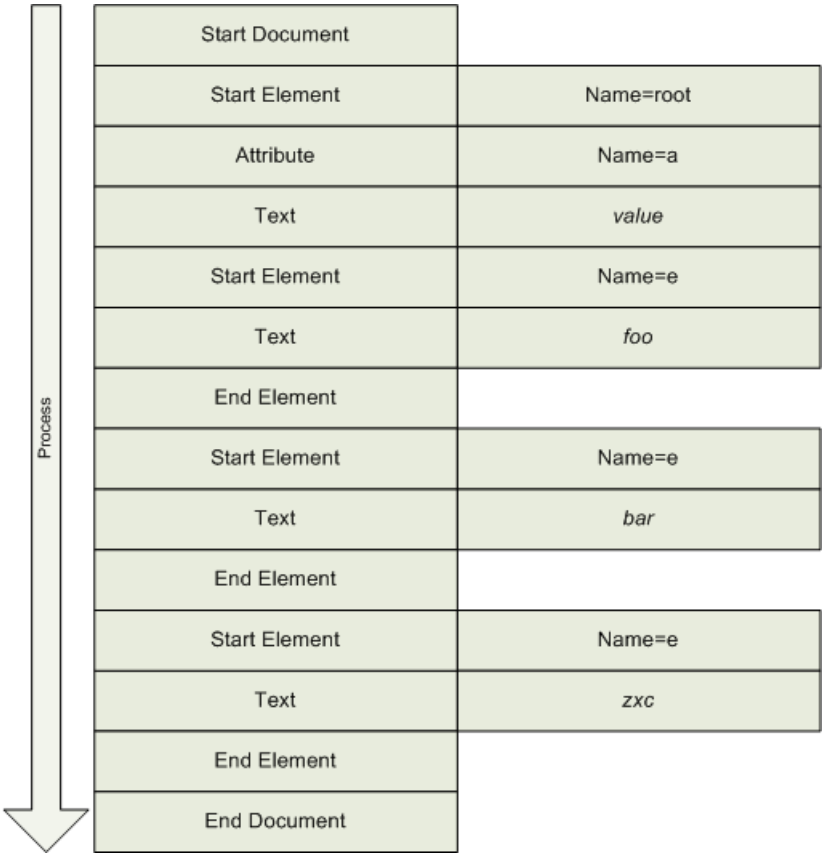


Figure 1. XML document as stream

Notice that the events recorded here are more finely-grained than the events generated by SAX parsing. Where in SAX the start of an element is a composite of all that element's namespace and attribute information, in the model above the start element event is separated out so that (for example) each attribute and attribute value is an event also. This is necessary so that every phenomenon in the XML document being parsed can have a corresponding event generated so that the entire infoset can be represented in a granular form without information loss.

Note

The view of an XML document as a fine-grained event sequence is taken even further by Simon St.Laurent's “Ripper” parser [10].

Notice also that many events have some additional text information associated with them, which are represented in the diagram by a box to their right.

6.1. Fine-grained Events

In order to avoid the Object overhead discussed above the stream representation must be held in memory using primitives. A natural choice is to use one byte per event, with certain values corresponding to certain events.

So a "start document" event might be represented by the byte 0x80, a "start element" by the byte 0x81, etc. Bytes in the range 0x80 - 0xFF are reserved for representing events. The purpose of bytes in the range 0x00 - 0x7F is discussed below.

6.2. Pooled String Storage

A large proportion of most XML documents is the textual data of various names (e.g. element and attribute names) and character data content (e.g. element text and attribute values). By representing these as indexes into some dictionary it is possible to reduce the amount of storage space required, particularly if the XML document has many re-used strings. Such an approach is taken by the Fast Infoset initiative [4] in which many string values are replaced by indexes into a number of string tables.

An extension of this method, and one that is particularly suitable for the frozen stream model, is to replace all strings with indexes into a table of all the distinct strings encountered during parsing.

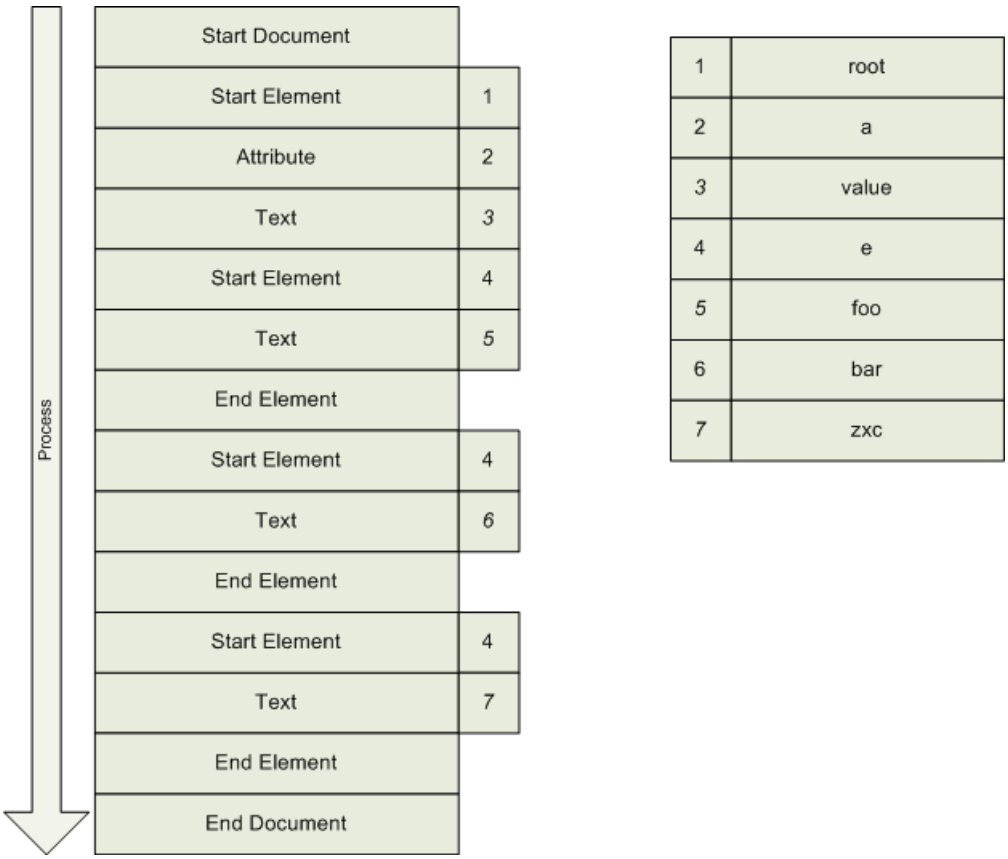


Figure 2. XML document as stream, with pooled string storage

As can be seen, the stream becomes one in which certain events have a string associated with them. In memory these are stored immediately following the event with which they are implicitly associated: so, the start of a element named "root" is represented by the byte value for an element start (0x81 say) and then a index into the string table for the string "root" – 1 in the example above. The model is similar to that of instructions and operands in assembly language programming, whereby certain instructions affect the interpretation of subsequent bytes.

When the number of an index value is greater than can be held in a single byte, some form of encoding is required. Because the values 0x80 - 0xFF are reserved for bytes indicating events, 7 bit values (i.e. bytes in the range 0x00 - 0x7F can be used to encode index values. Two advantages of this mechanism are that an iterator over the bytes can clearly identify the extent of bytes in an encoded value (any values with an eighth bit set acts a delimiter), and that only as many bytes as are necessary to efficiently encode the index value, are used.

7. Implementation Experience

An implementation of the frozen stream in Java shows its memory usage and speed compares favourably to DOM:

Table 2. Memory use and time taken for common XML tasks

Task	Time Taken	Memory needed
Build Frozen Stream	10.9s	117 MB

7.1. Scanning

However, performance when performing real-world operations was disappointing slow. The poor performance was a consequence of the large amount of time taken scanning bytes. This performance issue can be addressed through the introduction of "signpost" pseudo-nodes into the stream.

7.2. Signpost pseudo-events

These pseudo-nodes are items in the stream which use reserved values (i.e. with the high bit of each byte set) to indicate that they should be interpreted as events, not as data, but they do not correspond to anything in the XML infoset. Instead, their purpose is to route iterators more efficiently through the stream. So, for example, a signpost pseudo-event might express "next following sibling - 5000 bytes forward". An iterator searching for such a following sibling would thus be able to move straight to that position without any intervening scanning.

Again, these signposts take an instruction-followed-by-operand like form as bytes in the stream representation of the XML, with the data being an encoded form of an integer indicating a byte offset.

This approach *does* trade-off memory use and performance. Introducing such signpost events obviously increases the size of the stored stream. It also increases the computation necessary when building the stream. The payoff however, is a dramatic decrease in the amount of memory scanning required when iterating an in-memory stream.

Currently, experimentation has led us to introduce the following events, including psudo-events marked with an asterisk (the inclusion of theseis reflected in the above timing and memory use report):

- start element
- end element
- attribute name
- attribute value
- *following sibling
- *no following sibling
- *preceding sibling
- *no preceding sibling

- *parent
- namespace uri
- processing instruction name
- processing instruction value
- comment value
- CDATA section

(Optionally, we also introduce items for line and column numbers, but this is for niche use and not considered here.)

7.3. Making it useful

An efficient in-memory representation of XML is all very well, but is of little use if all it does in get built and occupy memory. Clearly, to be useful there should be some means of making it usable with other XML processing software.

For example, the popular Jaxen XPath engine [7] provides mechanisms whereby it can be used to query any in-memory XML document representation irrespective of how it is implemented. Unfortunately Jaxen is heavily predicated on the use of objects, so major surgery on the code base has been necessary to adapt it for use with frozen streams. The result of this activity is presented in the slides which accompany this paper.

8. Future Directions

8.1. Hardware assistance

If byte-by-byte scanning is too slow, one approach is simply to increase the amount of stream examined for each iteration during scanning. So, if say we were examining 64 bits at a time, rather than 8 (i.e. 8 bytes rather than one), then our scanning time would be reduced if the underlying system was capable of performing such multi-byte operations efficiently - as modern computer systems are.

This is where the beauty of using bit masks comes to the fore: because we know that any byte representing (say) a start element event has a distinctive bit pattern, it is possible to construct multi-byte masks that may be logically compared with a multi-byte section of stream in order to ask the question: is there a start element event in here? The result of such a logical operation will tell us if there is, and if so which byte(s) are matched.

While this approach is possible with high-level languages, it lends itself most naturally to an assembly-language (or even hardware) implementation. Parallel pipelines and dual core chips open up further possibility for a highly optimised assembly language stream scanner.

Is there also potential for using some of the multimedia-targetted features of modern computer systems for stream processing, since the XML 'stream' might be thought of as close to a video or audio stream? Interestingly, for example, copying a subtree when using streams becomes merely a memory copy (blit) operation.

8.2. In-memory model for mainstream application software

Although XML has become the prevalant serialisation format adopted in many IT domains, it risks being caught in this very niche. Many progreammers, especially those unused to XML, view XML as "just" a serialization format, a persistent form of what they more naturally view as in-memory typed properties. Without getting into a discussion of the damage this friction has caused to the XML world, I think it is safe to assert that the poor performance of older XML in-memory models may be part of the cause of such thinking. Nobody who wanted to write a high-performance office application would use, for example, a DOM as a backing.

Using a native XML store as the backing for application is not just a question of dogma, but may bring real benefits. For example the two standardised office document formats (ODF and OOXML) do not describe the *in-memory* representation of office documents. So we cannot, for example, run an XPath or XQuery expression against a "live" office document, but need to serialize it before running the query. Similarly, we cannot update document "in place" using idiomatic XML procedures. Allowing live processing of standardised XML formats would increase the interoperability of office application software, just as this has (nearly) achieved with in-browser XML.

8.3. An Iterator API?

While it may be argued there are already a good number of APIs for programmatic treatment of XML, and it is certainly true that designing good ones is hard, the frozen stream model naturally suggests its own kind of API, very different from tree-based, pull-based, or SAX-based ones.

The stream representation suggests an iterator (or cursor) based API, since we only want to reify a single event as a node at any one time. This fits well with some likely scenarios for usage of this efficient model, particularly for XPath querying of content which - in theory - requires just such an API for an implementation.

At its simplest such an API needs a single Iterator class (perhaps implementing the java.util.Iterator Interface), and the ability to specify, in XPath terms, which axis is being iterated. Further enhancement would see the ability to set a filter on the iterator (so it iterating only certain events), and to have an "iterate-to" feature, whereby one could set a condition which when met, stopped iteration.

Certainly such a cursor-based API would be a new and unusual way of approaching XML processing. It remains to be seen whether it would be popular with human users, though machines, of course, do not care for an API's style!

9. Conclusions

A general conclusion is that in 3 years since first starting on this work, the situation has improved. Java is more optimized, and DOMs are leaner and faster than they were with a new Java installation. Informal testing has also shown that among custom in-memory models, the Saxon XSLT, XQuery, and XML Schema processor [9], that Rolls-Royce of XML toolkits, consistently sets a benchmark that is hard to better. However, that is not to say there is not further room for improvement and study in this space, bearing in mind the following conclusions:

- Many existing in-memory models tend to bloat
- Java implementations have a particular tendencies to bloat when large numbers of Objects are used
- Much existing XML software implies or required large numbers of Objects
- Efficiency can be required by building a system around known limitations - in the case of Java by using primitives
- There may be scope for fresh thinking about XML APIs
- There is scope in such an implementation for architecture aware optimisations that may bring substantial performance benefits
- The frozen stream model is a viable means of achieving better XML performance
- The stream can be tuned to balance memory/speed requirements according to the usage scenario

Bibliography

[1] Jon Bentley. Programming Pearls. 2nd Ed., Addison-Wesley, 2000.

[2] Alex Brown. Frozen streams: an experimental time- and space-efficient implementation for in-memory representation of XML documents using Java. Extreme Markup Languages 2006. http://www.idealliance.org/papers/extreme/proceedings/xml/2006/Brown01/EML2006Brown01.xml.

[3] W3C DOM Working Group. Document Object Model. http://www.w3.org/DOM/DOMTR

[4] Paul Sandoz, Alessando Triglia and Santiago Pericas-Geertsen. Fast Infoset. Sun Developer Network. http://java.sun.com/developer/technicalArticles/xml/fastinfoset/.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissidies. Design Patterns. Addison-Wesley, 1995.

[6] Elliotte Rusty Harold. Processing XML with Java. Addison Wesley, 2003.

[7] Jaxen: universal XPath Engine http://jaxen.org/

[8] Bertrand Meyer. Object-oriented Software Construction. 2nd Ed., Prentice Hall 1997.

[9] Michael Kay. Saxon XSLT, XQuery, and XML Schema processor. http://www.saxonica.com/products.html

[10] Simon St.Laurent. What can you do with half a parser? Extreme Markup Languages 2003. http://www.mulberrytech.com/Extreme/Proceedings/html/2003/StLaurent01/EML2003StLaurent01.html

[11] Thread on xml-dev, Subject "XML Performance in a Transacation". Initiated by David Carver 22 March 2006.

[12] Wikipedia. http://wikipedia.org/.

[13] The Xerces2 Java Parser. http://xerces.apache.org/xerces2-j/.

[14] Mohamed Zergaoui. State of the art on streaming. Extreme Markup Languages 2008. http://www.balisage.net/Proceedings/html/2008/Zergaoui01/Balisage2008-Zergaoui01.html.

Imagining, building and using an XSLT virtual machine

Mark Howe
Jalfrezi Software Limited
<mark@cyberporte.com>

Tony Graham
Menteith Consulting Ltd
<Tony.Graham@MenteithConsulting.com>

1. The allure of all-XML server technology

In recent years XSLT has become the canonical language for XML transformation. There are good reasons for this. XSLT standards are supported by [9] and respected by multiple proprietary and open-source vendors[4]. The technology provides a higher-level programmer model than DOM and SAX. Because XSLT is itself an XML vocabulary, representation of XML within XSLT often displays a lisp-like elegance[5], and the transformation of XSLT by XSLT – for example when producing meta-stylesheets [6] – allows lisp-like program generation.

XSLT has most frequently been used for batch-type processing. The input XML is produced by hand or by non-XSLT technology. The transformed XML is written to file or piped to another program. XSLT itself is stateless between transformations, in the sense that the content of an input document has no effect on the processing of a subsequent input document. There are many situations in which this behaviour is exactly what is wanted, for example, when transforming database content into XSL-FO in order to produce printed output. Cocoon [2] is one example of a framework designed to optimise successive XSLT transformations of XML content.

In 2007 Mark Howe wrote an XSLT-driven website for a French estate agent. It generates a range of interactive search and display options directly from data exported in XML format from a proprietary offline database. While almost all the processing was handled by XSLT, some bespoke procedural code was still needed to turn HTTP requests into XML and to dispatch the output document as an HTTP response. The sending of email also had to be handled using non-XSLT technology. Nevertheless, this project confirmed that much of what is traditionally achieved using procedural languages and database back ends can be handled by XSLT.

[4] Eg LibXSLT, Saxon and Xalan

[5] See eg [7] p595: "XSLT, although not specifically designed as a generic or functional language [like lisp, ML or Haskell], has inherent capabilities to enable [higher-order functions]."

Another project involved the creation of a content management system for a social networking site[6]. The project started off in Perl CGI scripts which produced XML markup that was transformed into HTML by XSLT. Gradually more algorithmic functionality such as access control crept into the XSL stylesheet. This led to duplication of data, since XSLT could not easily make SQL requests and Perl could not process serialized XML efficiently.

Another part of the social networking site required a rich-media chat room. The server was written in object-oriented Perl, with I/O in XML over persistent sockets. The solution worked, but when questions were raised about adding new features and about potential security issues it became clear that the global semantics of the server could be obscured by the use of object-oriented programming to represent virtual world states.[7] A language such as XSLT would make transformations explicit, but there was no off-the-shelf way to use XSLT within a real-time server.

Together, these experiences engendered excitement about the possibilities of algorithmic processing with XSLT, tempered by the often inelegant reality of trying to interface XSLT technology to databases, multiple sockets and DOM-based technology. What would need to happen for XSLT to be able to provide all the functionality needed to run server applications?

Dimitre Novatchev's work on FXSL [4] has shown that XSLT "... is in fact a full-pledged functional programming language." However, we decided that a practical real time server solution required a less pure and more pragmatic approach. We therefore began exploring the viability of building an XSLT virtual server, using Perl and LibXSLT to test our ideas. That exploration eventually led to the Xcruciate project [10], based around a virtual machine called Xacerbate.

Our initial architecture consisted of two processes, which we referred to as the outer and inner loop. The outer loop collected socket input, turned it into XML and submitted it to inner loop for processing using XSLT. The output of the XSL transformation was piped back to the outer loop, which split up the output document and dispatched it to sockets as necessary. However, we soon realised that XSLT needed information not related directly to XML input, from the passage of time, through socket connections and disconnections to OS signals. We therefore moved towards an event-based model. Each input document describes an event. Each output document describes zero or more actions, such as output to one or more sockets or a request to disconnect one or more sockets.[8]

The system described above is stateless between transformations, as is usually the case with XSLT. Statefulness was added via an output option that rewrites XML

[6] http://www.stpixels.com

[7] This point is developed by [1] pp327-328: "... The objects of virtual worlds don't correspond well to the objects of an object-oriented programming language... Designers often think in terms of objects and methods when they should be thinking of multiple inheritance hierarchies and commands."

[8] Not all input results in output, and not all output is generated by socket input. For example, timer events may be ignored, and may initiate output to sockets.

data files[9]. Those files can be read by subsequent transformations via `xsl:document()`. Furthermore, XSL files can be rewritten the same way and used by subsequent transformations via `xsl:include` or `xsl:import`. This provides a mechanism for preserving stateful data, and also for modifying the transformation itself while the virtual machine is running.

Around this time our proof of concept implementation began to seg-fault Perl. The issue seemed to have something to do with conflicts between Perl and LibXML memory management when performing multiple transformations on the same data structures, but the precise nature of the problem eluded us. For this and other reasons we therefore decided to recode in C, and to take advantage of the richness of the C interface to LibXSLT to optimise and extend the virtual machine in various ways.

Table 1. Project and subproject names

Name	Role	Language
Xcruciate	Overall project, daemon control scripts	Perl
Xacerbate	Virtual machine based on LibXSLT	C
Xteriorize	HTTP gateway for Xacerbate	Perl
Xiguous	Library for handling HTTP requests	XSLT 1.0
Xcathedra	Library for manipulating virtual worlds	XSLT 1.0
Xtravagate	Test client for Xiguous	TCL/TK

2. Inside Xacerbate, an XSLT virtual machine

Xacerbate is a general purpose server that speaks XML. Exactly what XML the client should send and exactly how the server will respond depends on the application the server is running. Inasmuch as the client requests are XML and the server responses are XML, it follows that an Xacerbate application is an XSLT stylesheet, is XML markup that compiles to XSLT transforms, or is a combination of both.

2.1. Implementational decisions

Xacerbate, like all Xcruciate projects, is open source and is released under the BSD licence. Xacerbate is targeted for use on Un*x operating systems. It is being developed and deployed on various flavours of Ubuntu, CentOS, and Red Hat operating systems.

[9]We considered using `exsl:document` to produce secondary documents but ultimately opted for keeping all output within one document, which simplifies sandboxing of write operations.

Xacerbate uses LibXSLT, which is a XSLT 1.0 processor, because LibXSLT is already installed on the operating systems of interest, is fast, documents how to implement extension functions, and implements the most useful of the EXSLT extension functions. Xacerbate is implemented in C because LibXSLT is implemented in C.

Man pages for the executable programs and configuration files are written in DocBook XML and transformed into 'man' format using xsltproc and the DocBook 'manpage' stylesheet. Additionally, structured comments in the C source code (similar to Javadoc comments) are processed using the GTK-Doc utilities to produce DocBook 'refentry' XML, are combined with narrative chapters also written in DocBook XML, and processed to produce a reference manual in HTML.

2.2. Architecture

Xacerbate can be viewed as a virtual machine that runs one XSL transformation per cycle. The Xacerbate server is an executable C program, but applications are written in XSLT, and there may be one or more layer of XSLT libraries between the application and the server. The XSL transformation has no access to OS-type functionality other than via the XML API provided by Xacerbate.

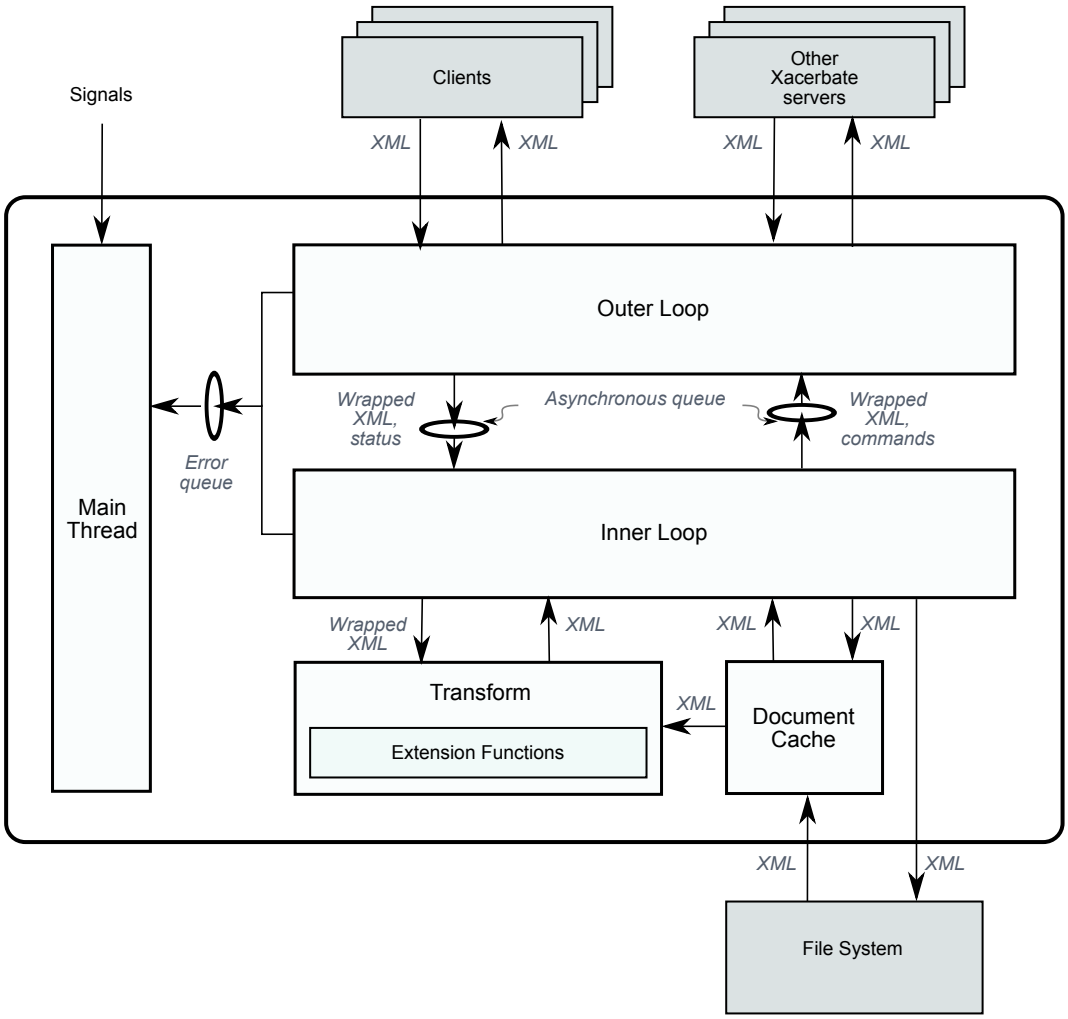
The Xacerbate virtual machine runs on XML. It receives XML from clients and other Xacerbate servers, passes XML to the application stylesheet, and acts on commands emitted as the result of the XSL transformation. It also reads its configuration files as XML. The functions of the Xacerbate virtual machine include:

- Manage socket connections and disconnections by clients and other servers
- Accept XML documents from clients and other servers
- Send XML documents to clients and other servers
- Pass XML documents to the transformation
- Cache stylesheets and data documents
- Modify data files under control of the transform
- Connect to other servers
- Log connections and other activity

Other than accepting incoming socket connections, receiving documents, and logging, everything is under the control of the transformation.

Non-XML input and output can be handled via gateways. For example, Xteriorize accepts HTTP requests, translates them into XML which is passed to Xacerbate, and the corresponding output from Xacerbate is converted into an HTTP response. From an architectural point of view, Xacerbate treats a gateway like any other client - decisions about what end users can do via the gateway are dealt with at the XSLT level.

2.2.1. Block Diagram



2.2.2. All XML, all the time

2.2.2.1. Client <--> Xacerbate (and Xacerbate <--> Xacerbate)

For the purposes of the Xacerbate engine, the format of the XML received from the client is immaterial, provided there is a `0x0` character separating the documents. Xacerbate is in use in chat-rooms, serving XHTML and generating Flash applets: the specific functionality is provided by the application that is written in a combination of XSLT and higher-level markup.

The XML text received from the client is parsed both to check that it is well-formed and to convert it into an in-memory representation as a LibXML2 xmlDoc.

2.2.2.2. Outer loop --> inner loop

XML from the client is wrapped in an element with attributes recording its origin and then queued for the inner loop without being reserialised as markup. Notifications of socket connections and disconnections and periodic timer events also generate XML documents (as xmlDoc) that are also queued for the inner loop.

Example 1. XML from client wrapped by Xacerbate element

```
<xacw:input xmlns:xacw="http://xacerbate.co.uk/xacw"
  source="connection" socket_id="-1">
  <gateway_auth>LittlePigLittlePigLetMeComeIn</gateway_auth>
</xacw:input>
```

2.2.2.3. Inner loop --> transform

XML from the outer loop is placed on the command stack without modification (and without reserialisation). `stack` output events also cause documents to be placed on the stack, and they pre-empt the documents received from the outer loop. For example, the stack is used when compiling scripts into XSLT.[10]

2.2.2.4. Transform --> inner loop

The transform result is also an XML document (as a xmlDoc). The children of the document element are each a command to be acted on by the Xacerbate engine. The child element's name and attributes specify the action, and the element's content, if any, serves as data.

2.2.2.5. Inner loop --> outer loop

Five actions in the transform result make the inner loop queue an XML document for the outer loop:

- Send an XML document to one or more sockets, i.e., to one or more clients or other servers
- Disconnect a socket
- Open a socket to connect to another server
- Cause the server to shut itself down

[10]Explicit manipulation of the command stack partially addresses one curious aspect of Xacerbate programming, namely that changes to data structures requested during a transformation have no effect on those data structures within the lifetime of the current transformation. A subsequent command, posted on the stack, can see those changes, which facilitates certain multi-step modifications. However, doing as much as possible per transformation maximises efficiency, so this technique should be used with caution.

- Log errors from the transform

2.2.2.6. Document cache --> transform

Xacerbate overrides LibXSLT's document loader function so the main stylesheet, stylesheets accessed using `xsl:include` or `xsl:import`, and documents accessed using `document()` are fetched from the document cache.

When the requested document is not already in the cache, Xacerbate opens the file on the filesystem and parses it into a xmlDoc in the cache.

2.2.2.7. Inner loop <--> document cache

In response to a command in the transform result to modify a file, the inner loop fetches the file from the document cache. It both reinserts the modified file into the document cache (as a xmlDoc) and writes the file to the filesystem (as XML).

Whenever the inner loop reloads the stylesheet, it flushes the document cache before reloading to avoid problems with stale references in the dictionaries of the libxslt data structures for the stylesheet modules.

2.3. XSLT extension functions

Xacerbate implements several extension functions that are available to the application. The functions are registered in the `http://xacerbate.co.uk/xacf` namespace and conventionally use the `xacf` prefix.

2.3.1. ID functions

These functions provide identifiers that are unique across multiple transformations.

- `xacf:declare_id(id)` attempts to declare a unique id.
- `xacf:generate_id()` registers a new identifier.
- `xacf:last_id()` returns the last id generated using `xacf:generate_id()`.

2.3.2. Error functions

These functions provide a simple flag-based mechanism for detecting if errors have occurred during the execution of library functions. The application code can use or ignore information provided via this mechanism. It should be noted in passing that these functions could be hijacked to implement modifiable variables in XSLT.

- `xacf:clear_errors()` clears all set error flags and returns `true` if any flags were cleared.
- `xacf:set_error(name)` sets an error flag for (the lowercase form of) `name`.

- `xacf:clear_error(name)` clears an error flag for (the lowercase form of) `name`.
- `xacf:check_error(name)` checks whether an error flag is set for (the lowercase form of) `name`.
- `xacf:check_errors()` returns `true` if any flags are set. Returns `false` if no flags are set or if an error occurred.
- `xacf:errors()` returns a space-separated list of the (lowercase form of) names of flags that are set.

2.3.3. Authentication-related functions

These functions provide functionality for authentication-related tasks that are either impossible or computationally expensive to implement in XSLT.

- `xacf:crypt(key,salt)` provides one-way encryption for storing and testing passwords.
- `xacf:hex(number,width?)` returns the `number` as a hexadecimal string. If `width` is provided, pads the string to at least that many digits.
- `xacf:random_seed(seed1,seed2,seed3)` sets the random number seed.
- `xacf:random()` returns a random number in the range 0.0 to 1.0 (exclusive).
- `xacf:nrandom()` returns a random integer in the range 0 to `integer` - 1.

2.3.4. Miscellaneous functions

- `xacf:epoch_time()` returns the number of seconds since the Epoch (00:00:00 UTC, January 1, 1970), which allows cheap time calculations, eg for authentication timeouts.
- `xacf:test(string,regex,flags?)` provides regex testing via the PCRE library. It is used to implement an application-level type-checking system.
- `xacf:parse(string)` parses *`string`* as XML. This is useful for handling user-generated XML, eg markup submitted via a web form.
- `xacf:version()` returns the extension function library version.
- `xacf:conf(param)` returns the value of a configuration parameter.

2.3.5. Accessing extension functions

It is a quirk of LibXSLT that prefixes for extension *functions* must be declared as prefixes of extension *elements*, even when there are no extension elements defined in that namespace.

Since it has been the defensive practice to declare all known namespaces in every stylesheet whether or not they are used in that stylesheet, some of the prefixes also

end up being explictly excluded, so a typical `xsl:stylesheet` start-tag is as shown below.

Example 2. `xsl:stylesheet` start-tag

```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:date="http://exslt.org/dates-and-times"
    ...
    xmlns:anon="http://xcruciate.co.uk/anon"
    extension-element-prefixes="exsl func xacf xigf xcaf app anon"
    exclude-result-prefixes =
       "date xsl xac xcr xacw xacf xig exsl str func xigf xtef">
```

2.3.6. Testing extension functions

Extension functions are tested both at the C level (using CUnit) and at the XSLT level.

xacdproc, a cut-down **xsltproc** variant that also loads the extension functions' C library, runs tests written for Eric van der Vlist's XSLTunit testing framework to test the extension functions.

XSLTunit is used since it requires only XSLT 1.0, whereas most other frameworks require XSLT 2.0 or Java (or .Net) or both.

3. Example 1: Handling chat-room and HTTP requests within the same transformation

For testing purposes we have developed a minimal application that handles both HTTP requests via Xteriorize and chat-type persistent connections. For your own non-trivial applications you would use the various XSLT libraries to simplify the application code, but this minimal application has the merit of showing all the work required by Xacerbate of the XSLT layer, inline.

Example 3. Minimal Xacerbate XSLT: declarations and default template

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xacw="http://xacerbate.co.uk/xacw">

  <xsl:output omit-xml-declaration="yes" method="xml"
      encoding="UTF-8" indent="yes"/>

  <xsl:template match="/*"/>
```

The first template makes ignoring events the default. This is particularly useful if timer events are not used by the application.

Example 4. Minimal Xacerbate XSLT: client connections

```
<xsl:template ▶
match="/xacw:input[@xacw:source='self'][@xacw:event_type='socket']">
  <xsl:if test="xacw:event_input[@xacw:event='client_connected']">
    <xacw:outputs>
      <xsl:call-template name="to_socket">
     <xsl:with-param name="socket"
          select="xacw:event_input/@xacw:socket_id"/>
     <xsl:with-param name="content">
       <hello you_are="world"/>
     </xsl:with-param>
      </xsl:call-template>
    </xacw:outputs>
  </xsl:if>
</xsl:template>
```

The second template matches events generated on socket connection and sends a short welcome message to that socket. Most useful applications would keep track of open sockets and attempt some form of authentication.

Example 5. Minimal Xacerbate XSLT: socket input

```
<xsl:template match="/xacw:input[@xacw:source='connection']">
  <xacw:outputs>
    <xsl:call-template name="to_socket">
   <xsl:with-param name="socket" select="@xacw:socket_id"/>
   <xsl:with-param name="content">
     <xsl:choose>
       <xsl:when test="http_request">
         <http_outputs
              id="{http_request/http_info[@name='xteriorize_request_id']/.}">

        <http_output type="response" response_code="200"
                  response_code_text="OK" ▶
http_version="{http_request/http_version/.}">
          <http_document document_type="text/html" uuencoded="no"
                    doctype="xhtml1-strict">
            <html xmlns="http://www.w3.org/1999/xhtml">
              <head>
             <title>Minimal response to input</title>
              </head>
              <body>
           <h1>Thanks for your HTTP request</h1>
```

```
            <h2>
                      You asked for
                      <b><xsl:value-of select="http_request/http_url/."/></b>
                    </h2>
                </body>
              </html>
            </http_document>
          </http_output>
        </http_outputs>
      </xsl:when>
      <xsl:otherwise>
        <message>Thanks for your zero-terminated input element called
                  <xsl:value-of select="local-name(*[1])"/></message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:with-param>
    </xsl:call-template>
  </xacw:outputs>
</xsl:template>
```

The third template tests incoming XML to see if it looks like an HTTP request from Xteriorize (the next example describes such requests in more detail). If so, it returns an HTTP response containing a simple HTML web page. For all other socket input the template sends back a message element that echos the name of the incoming element.

Example 6. Minimal Xacerbate XSLT: generating socket output

```
<xsl:template name="to_socket">
  <xsl:param name="socket"/>
  <xsl:param name="content"/>
  <xacw:output xacw:destination="connection" xacw:socket_id="{$socket}">
     <xsl:copy-of select="$content"/>
  </xacw:output>
</xsl:template>

</xsl:stylesheet>
```

The final, named template produces an Xacerbate output document, and is used for both forms of output above.

Many frameworks allow XSLT to produce HTML output, but few can control chat-room interaction from the same stylesheet, let alone the same template. One of the aims of the Xcruciate project is to allow tight integration between website and virtual world media.

4. Example 2: From HTTP request to Shockwave file creation

In this example the HTTP is received by Xteriorize, the Xcruciate project's HTTP gateway. Xcruciate turns the request into XML, adds some useful information about the connection itself and leaves Xacerbate to process the request.

Example 7. Input event generated by XML-encoded HTTP request for /makeaflash2 from Xteriorize

```
<xacw:input xmlns:xacw="http://xacerbate.co.uk/xacw"
       source="connection" socket_id="-1">
  <http_request>
    <http_info name="xteriorize_request_id">0</http_info>
    <http_info name="xteriorize_serveraddr">127.0.0.1</http_info>
    <http_info name="xteriorize_serverport">8080</http_info>
    <http_info name="xteriorize_peeraddr">127.0.0.1</http_info>
    <http_info name="xteriorize_peerport">50222</http_info>
    <http_info name="xteriorize_epoch">1233900564</http_info>
    <http_info name="xteriorize_datetime">
       2009-02-06T06:09:24+00:00
    </http_info>
    <http_info name="xteriorize_request_length">431</http_info>
    <http_method>GET</http_method>
    <http_url>/makeaflash2?screenname=banana</http_url>
    <http_version>HTTP/1.1</http_version>
    <http_header_line name="accept-charset">
       ISO-8859-1,utf-8;q=0.7,*;q=0.7
    </http_header_line>

  <!-- Potentially long list of browser headers here -->

  <http_body/>
  </http_request>
</xacw:input>
```

In this case the code that processes the request has been specifed in a data file using non-XSLT markup. The script definition consists of various tests on the request, plus the algorithm to be executed if all the tests are passed. In this case the algorithm is defined by wrapping basic control structures around XPaths, but Xacerbate can potentially use any meta-stylesheet that compiles to valid XSLT.

Example 8. Script definition for /makeaflash2

```
<script id="makeaflash2" secure="0">
   <method>get</method>
   <post_max>4096</post_max>
```

```
    <parameter name="screenname" presence_optional="0" value_optional="0">
       <valid_type>screenname</valid_type>
       <invalid_error>invalid_screen_name</invalid_error>
       <missing_error>you_must_supply_a_screenname</missing_error>
    </parameter>
    <algorithm>
       <variable name="fileid">
          <value-of>xacf:generate_id()</value-of>
       </variable>
    <copy-of>
       xtef:write_file(
          'flash',
          concat('hello',$fileid,'.swf'),
          'mxmlc',
          app:hello_flash(xtef:param($REQUEST,'screenname'))
       )
    </copy-of>
    <copy-of>
       xcaf:response_html(
          $REQUEST,
          xtef:i18n('makeaflash_return_page'),
          xcaf:link(concat('/flash/hello',$fileid,'.swf'),'Click here')
       )
     </copy-of>
  </algorithm>
</script>
```

At startup, scripts are compiled into XSLT which is linked into the transformation used by Xacerbate, resulting in the following func:function definition. At present, parameter and other checks are handled by a wrapper, but these could be compiled into the function inline to increase performance by reducing the run-time search space.

Example 9. Compiled function definition for /makeaflash2

```
<func:function name="anon:makeaflash2" xsl:extension-element-prefixes="func">
   <xsl:param name="REQUEST"/>
   <xsl:param name="SCRIPT"/>
   <func:result>
      <xsl:variable name="fileid">
         <xsl:value-of select="xacf:generate_id()"/>
      </xsl:variable>
      <xsl:copy-of
         select="xtef:write_file(
            'flash',
            concat('hello',$fileid,'.swf'),
```

```
            'mxmlc',
            app:hello_flash(xtef:param($REQUEST,'screenname'))
          )"/>
       <xsl:copy-of
      select="xcaf:response_html(
         $REQUEST,
         xtef:i18n('makeaflash_return_page'),
         xcaf:link(concat('/flash/hello',$fileid,'.swf'),'Click here')
      )"/>
    </func:result>
</func:function>
```

In Xcathedra, the Xcruciate project's XSLT layer for handling HTTP requests, it is assumed that script definitions are represented as `func:function`s, because XPath function calls are easy to build and evaluate dynamically:

Example 10. XSLT snippet that builds and evaluates an XPath expression to process a script request

```
<xsl:when test="not(xacf:check_errors())">
   <xsl:copy-of select=
      "dyn:evaluate(concat('anon:',$script/@id,'($request,$script)'))"/>
    </xsl:when>
```

The result of calling the meta-stylesheet-generated function on the HTTP request is one Xacerbate output document that will be passed back to Xteriorize. The payload document contains two HTTP output elements. The first instructs Xteriorize to compile the enclosed MXML and write the resulting Shockwave file within the docroot. The second generates an HTTP response containing an HTML document with a link to the freshly-generated Shockwave file.[11]

Example 11. Output document produced by Xacerbate in response to HTTP request

```
<xacw:output xmlns:xacw="http://xacerbate.co.uk/xacw"
  destination="connection" socket_id="-1">
  <http_outputs id="0">
    <http_output type="write_doc_to_file" directory="flash"
      file="helloa3.xml" url_path="flash"
      url_name="helloa3.xml" filter="mxmlc">
      <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
         layout="absolute">
```

[11] It would be entirely possible to serve the Shockwave file directly, but the MXML compiler is too slow to form the basis of a scalable on-demand Shockwave production system. A more plausible scenario involves using the MXML compiler in admin mode to produce resources to be served statically to other users.)

```
        <mx:Button label="Hello banana"/>
      </mx:Application>
    </http_output>
    <http_output type="response" response_code="200"
       response_code_text="OK" http_version="HTTP/1.1">
       <http_document document_type="text/html"
          uuencoded="no" doctype="xhtml1-strict">
          <html xmlns="http://www.w3.org/1999/xhtml">

             <!-- Lots of html here -->

             <h1>Flash animation published!</h1>
             <p>
                <a href="/flash/helloa3.xml" target="_self">
                   Click here
                </a>
             </p>

             <!-- Lots more html here -->

          </html>
       </http_document>
    </http_output>
  </http_outputs>
</xacw:output>
```

5. Future development

More work needs to be done, and some of it is expected to be done between the writing of this paper and its presentation at XML Prague 2009.

5.1. Security

Security is crucial for any server technology. Xacerbate has some inherent advantages in terms of security, by virtue of running application code within a virtual machine and coercing all I/O into well-formed XML. Nevertheless, some potential issues have been identified and are being addressed.

5.1.1. Flood control

Configuration file parameters for controlling the volume and frequency of documents received from a client need to be implemented.

5.1.2. Controls on file reading

The initial strategy was to impose a chroot-like directory for all file reading. However, this proved to be incompatible with the use of shared XSLT libraries. The proposed solution is to specify a list of readable directories in the configuration file.

Xacerbate also uses the LibXSLT security framework to stop stylesheets from reading from the network and to prohibit writing files directly from the stylesheet.

5.1.3. Controls on file writing

From the beginning Xacerbate has restricted file writing to a directory declared in the configuration file. However, `exsl:document` provides a potential mechanism for writing to arbitrary locations. The solution is to disable the `exsl:document` functionality since, in any case, Xacerbate provides its own mechanism for multiple output documents.

5.1.4. Controls on configuration file access

At present, XSLT can access all configuration file entries. There are occasions when this may not be desirable. The solution is to implement a server-wide blacklist. (The list will be server-wide as the risks depend largely on the type of applications being run, the deployment strategy for different Xacerbate units and the trustworthiness of the users maintaining those units.)

5.2. Data integrity

Server applications should be as robust as possible in the area of data integrity. Three issues have been identified for future work.

5.2.1. Validation

Relax NG validation is to be implemented for incoming and outgoing XML, as well as for modifiable data and transform files. The extent of validation will be configurable.

5.2.2. Persistent ids across sessions

While the xacerbate-specific id functions described above are a step forward on the native XSLT ones in terms of persistence, there is still a potentially non-trivial problem when the server is restarted using data files modified during a previous session. The solution is for Xacerbate to store id information, and to provide the option to retreive this information on restart.

5.2.3. File backup strategy

At present modifiable files are modified constantly, which makes taking any form of backup difficult. The solution is to provide support for writing copies of the files to an alternative location at a predefined frequency, possibly with log-type rotating names.

5.3. Development environment

These developments are intended to facilitate the lives of application developers.

5.3.1. JavaScript

To make it even easier for people who are not XSLT experts to write applications, we intend to implement a script extension element using the Seed library, since Seed already has introspection information for LibXML2 object types. Once the implementation matches the definition of EXSLT's `func:script`, the code will be submitted for inclusion in LibEXSLT.

5.3.2. Debugger

LibXSLT has its own debugger utility. The same debugging callback mechanism could be implemented in Xacerbate so that an Xacerbate debugger could debug both the Xacerbate virtual machine and the transform from the one session.

5.4. Improved and new virtual machines

5.4.1. Increasing throughput

The speed of the Xacerbate virtual machine has yet to be a bottleneck, but there are several ideas for how to speed it up. This may involve not writing every state change to disk or adding more threads, e.g., for writing files to disk. It has so far been prudent to get Xacerbate working properly and to measure its performance before attempting any speedup.

5.4.2. Proof of concept implementations in other languages

There is nothing that ties Xacerbate to being implemented in C or ties the transforms to libxslt provided that the Xacerbate implementations behaves correctly or that the XSLT processor implements the extension functions correctly.

For example, Xacerbate could be implemented in Erlang – the asynchronous queues in Xacerbate have a strong similarity to messages in Erlang – once there is an Erlang driver for LibXSLT.

Xacerbate could be implemented in Java once the extension functions were implemented for a Java XSLT processor.

5.4.3. XSLT 2.0?

XSLT 2.0 offers many features that would be of potential interest to Xcruciate application writers, including native support for functions and the capacity to manipulate generated XML without recourse to `exsl:node-set()`. However, it would still be necessary to write extension functions and to take control of aspects of the document cacheing process. Also, garbage collection is not ideal behaviour in a long-running server application for virtual worlds.

6. Conclusion

Xcruciate is very much work in progress - there is more work to do on the Xacerbate virtual machine, and experimentation with different programming idioms for the XSLT layer has hardly begun. Nevertheless, progress to date suggests that the basic model provides a viable basis from which to develop cross-media applications within an all-XML environment.

Bibliography

[1] Richard A Bartle. *Designing virtual worlds*. 2004. New Riders Publishing.

[2] *Cocoon*. http://cocoon.apache.org.

[3] *Community initiative to provide extensions to XSLT*. http://www.exslt.org/.

[4] Dimitre Novatchev. *The Functional Programming Language XSLT - A proof through examples*. http://fxsl.sourceforge.net/articles/FuncProg/Functional%20Programming.html.

[5] *System for documenting C code*. http://library.gnome.org/devel/gtk-doc-manual/stable/.

[6] Michael Kay. *MetaStylesheets: On how, and why, XSLT can be used to transform XSLT (or XML Schema, or XQuery)*. http://2006.xmlconference.org/proceedings/26/slides.pdf.

[7] Sal Mangano. *XSLT Cookbook*. 2003. O'Reilly Media, Inc.

[8] *GObject JavaScriptCore bridge*. http://live.gnome.org/Seed.

[9] *The Extensible Stylesheet Language Family (XSL)*. http://www.w3.org/Style/XSL/.

[10] *Xcruciate: real-time XSLT for cross-media social networking*. http://www.xcruciate.co.uk/.

Advanced Automated Authoring with XML

Petr Nálevka
University of Economics, Prague
<petr@nalevka.com>

Abstract

This article proposes a set of powerful XML technologies to automate authoring of large, detailed and highly visual documentation which would be difficult and error prone to reproduce manually. The author further proposes best-practices for XML authoring and introduces a simple yet powerful framework which supports tasks typically related to document publishing and integration of information from various sources.

Rather than building a complex theoretical background this article focuses on being very practical. It demonstrates the use of various technologies on a case study taken from the networking industry.

Keywords: Authoring, Publishing, XML, DocBook, SVG, Saxon, Ant, Visio, Excel

1. Introduction

There are many reasons why to use semantic automated authoring tools rather than using presentational visual word processors. As there are loads of articles on this topic, there is no reason to go into details. The following text summarizes only the most important pros and cons.

Why doing automated authoring

Content separated from style	this allows pluggable styles; the same document is produced with a completely different styling or suited for a different media without doing any changes to it
Auto-generation	many document elements such as table of contents, numbering, contextual headings etc... are auto-generated for the author
Professional looking outputs	follows typesetting standards, uses hyphenation, advanced kerning etc...
Highly customizable output	only the medium is the limit

Modularity	documents may be split into smaller chunks and than combined in variable ways, this helps to avoid any duplicities which are hard to maintain

Why NOT doing automated authoring

Visual editing	to the authors knowledge, there are no tools which would allow semantic editing of documents with the same level of visual user experience as presentational word processors

This article demonstrates that publishing automation does not end with auto generated ToCs or references. There is a far greater potential in specific applications. It introduces a specific problem domain and shows how DocBook [1] and other excellent XML technologies has been utilised and extended to automate the authoring process as much as possible.

Publishing automation is achieved through a framework which is basically a pure XML (some would say POX) application. Data are described in XML domain specific language, transformations are defined in XSLT [6] and the procedural aspect is expressed using ANT [14] build files, again in XML format.

This article is basically a celebration of XML. It shows what immense flexibility is gained when semantics is attached to data.

The key benefits of what is being proposed in this article

- no redundancy in data
- highly specific domain model perfectly suited and intended for instant changes
- professional looking typesetting
- 100% control over the output
- potential applications beyond publishing

2. The case study

The author of this article works for a company which implements large scale networking projects like nation wide backbones or inter-bank networks.

Even the projects differ a lot from each other they share a significant portion of the domain model.

Common elements of the domain model

- Each project has one or more sites in one or more cities. Each site has a geographical location, contact information and other properties.
- Sites are usually interconnected using networking protocols with specific configuration.
- There is a set of hardware devices at each site, some of them installed in racks.

- Network with certain criteria is modelled. This includes different communication technologies and protocols at each network layer. For example IP address plan is modelled at layer 3.

The aim of an automated authoring system is to support the project from the very beginning — the proposal stage — until the project is hand over to the customer with detailed documentation. The same domain model is used through different stages of the project life cycle and different kind of documents are being generated out of it to support the individual stages.

Documents auto-generated from the same data during the project life-cycle phases

1. Proposal
   - Commercial Proposal
   - Technical Proposal
2. Design
   - Network design
   - Per-site documentation
3. Implementation
   - Time and progress planning
   - Installation guides
   - Compliance testing How-Tos
   - Detailed per-site documentation
4. Support
   - Inventory registry
   - Network status

3. Domain modelling

DocBook [1] is a perfectly suitable grammar for describing documents. But event the desired output are in fact various documents, it makes a good sense to model the domain first in a domain specific language and than transform it automatically into DocBook in the next stage.

Expressing the data in a well designed and highly specific language will always be beneficial over the use of a generic purpose grammar. Most of all, such data are easier to express, understand, change, reuse and validate.

Modelling domains in XML has several specific characteristics over object oriented programming or UML modelling. In addition, the grammar designer needs to find answers to the following design questions:

- Use rather attributes or elements to model a certain aspect of the domain

- Use ID references or rather prefer tree structures
- Direction of ID references between entities

Those decisions significantly influence the ease of entering new data, maintaining them and understanding them. It also determines how difficult it is to work with such data in terms of expressing transformations or validation rules. In many cases those two concerns go against each other.

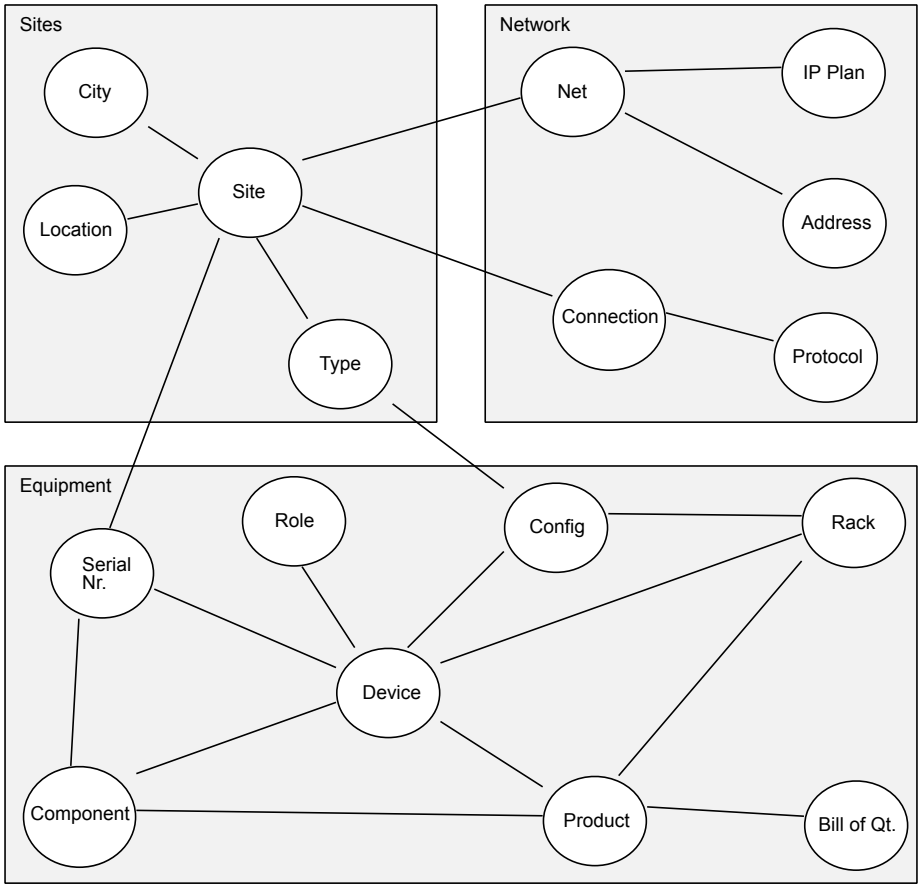


Figure 1. High level overview of the case study domain model

Obviously it is far more important to design an elegant DSL where expressing data is straightforward. Complexity in processing may always be reduced by applying simplification transformations to the data first, before further processing[1].

3.1. Schema and validation

Usually the domain model is the most rapidly changing part of any IT application or system. Therefore it is important to decide how loosely or strictly we like to define

[1]This approach is commonly used in XML grammars (e. g. Relax NG or NVDL simplification).

the model. If every little change needs to be propagated into the processing XSLT stylesheets and the schema, such change requires a lot of time and resources.

It took several projects until the domain model for networking projects described in Section 2 settled down in some more stable form. Thinking that you can model a perfect language from the beginning, create a perfect schema for it and use it unchanged is a pure fiction.

Having an XML Schema defined for your grammar from the beginning may be a maintenance overhead. Every time there is a change in the model, not only current data needs to be migrated, but also the schema and stylesheets need adjustments.

On the other hand, validation of data is important. There has to be some way to know whether changing a certain aspect of the structure of the input data will cause some XSLT stylesheets malfunction. Moreover, even a perfectly designed grammar cannot guarantee data consistency[2]. In the networking domain, there are many potential data inconsistencies. For example an IP address is assigned to a network where it does not belong to because of the very nature of the IP protocol.

This article proposes validating data loosely[3] as an alternative to the traditional strict validation approach. A perfect language to define loose schemas is Schematron. In Schematron everything is allowed by default, unless a rule exists which says otherwise. Moreover, using the full power of XPath, Schematron is able to express even very complicated rules which operate over multiple contexts in the source document. Probably the biggest advantage of Schematron over grammar-based schema languages is the ability to output domain specific highly descriptive diagnostics [4]. Find more about Schematron and its unique features in [8], [9] and [10].

The loose Schematron schema for the networking domain checks only those aspects of the XML structure which are necessary for the underlying stylesheets to work properly. In addition it contains high level consistency checks with verbose domain specific diagnostic messages.

Also XSLT stylesheets themselves may be written in a way to make them as loosely couplet with the domain model as possible. The aim is to minimize the impact of refactoring in the model to the actual stylesheets.

The loose approach to the domain model helps to keep it open to extensions and flexible when being changed.

[2] A perfect example of data inconsistency is the HTML tables model, where it is possible to define overlapping cells.

[3] The aim of this article is not to discourage people from writing schemas. It just proposes an pragmatic lightweight alternative approach in case flexibility and fast change management is more important than precise binding.

[4] For example a diagnostic message may tell us that according to the target rack configuration there is a missing device in rack X on site Y.

3.2. Namespaces and specific accents

Having a common domain specific language shared across multiple projects allows to reuse XSLT stylesheets for common tasks. Duplicating similar stylesheets in various projects would be error prone and would cause maintenance difficulties.

On the other hand, projects differ from each other, and we can hardly expect that in the real word a common language would describe all their specifics. Specific grammars based in a different namespace (assigned to a particular project) are used to describe such specific properties of certain projects.

Specific stylesheets may than handle the specific constructs. It is easy to recognize stylesheets which are specific to a particular project, as they contain the `xmlns` declaration for the particular namespace.

4. The authoring framework

This section introduces an authoring framework which helps to auto-generate documents out of data described in a domain specific XML grammar. It shows what is the framework composed of, what tools are involved and the overall architecture of the system.

4.1. The framework in a nutshell

The framework itself is not tight to the networking domain. It is a generic purpose set of tools which automates the tasks involved in publishing documents. It consists of several components, responsible for passing the source domain specific XMLs through a transformation chain to generate the requested document out of it.

First the DSL data get simplified and than they are transformed by applying a set of XSLT stylesheets on them (see Figure 2). Common XSLT templates are shared among all projects which use the same DSL. This allows to write a stylesheet once and reuse it in several other projects. Specific functionality may be implemented by importing a common stylesheet and extending it on a per project bases.

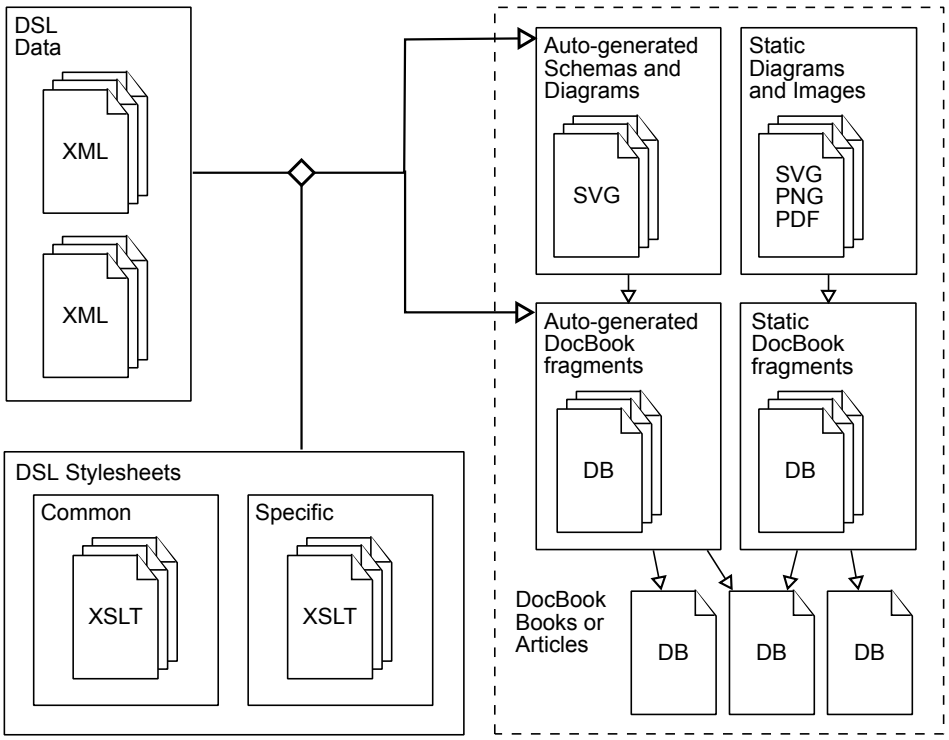


Figure 2. Document generation — Phase 1

The aim of the framework is to generate a document describing the source data. The result of the transformations usually are DocBook fragments such as chapters, sections, tables and figures or diagrams in Scalable Vector Graphics (SVG) format [4][5].

The generated document fragments are combined together with static fragments (usually descriptive texts, diagrams and images created directly by content authors) into chapters using XInclude [11].

For each resulting document there is a DocBook article or book skeleton which contains meta-data about the document (the DocBook `<info>` element with authors, organization, disclaimer, copyright, titles and so on). In addition, it includes a set of static or generated chapters or sections.

In the next phase, the DocBook sources are transformed into PDF (or eventually HTML) format (see Figure 3). This is done through a DocBook stylesheet customization layer [2]. The layer is composed of several styles one for each corporate identity involved.

[5] A typical result of such transformations for the networking project domain is for example: a table listing all sites, a table with wireless connections between sites, per-site networking schemas etc...

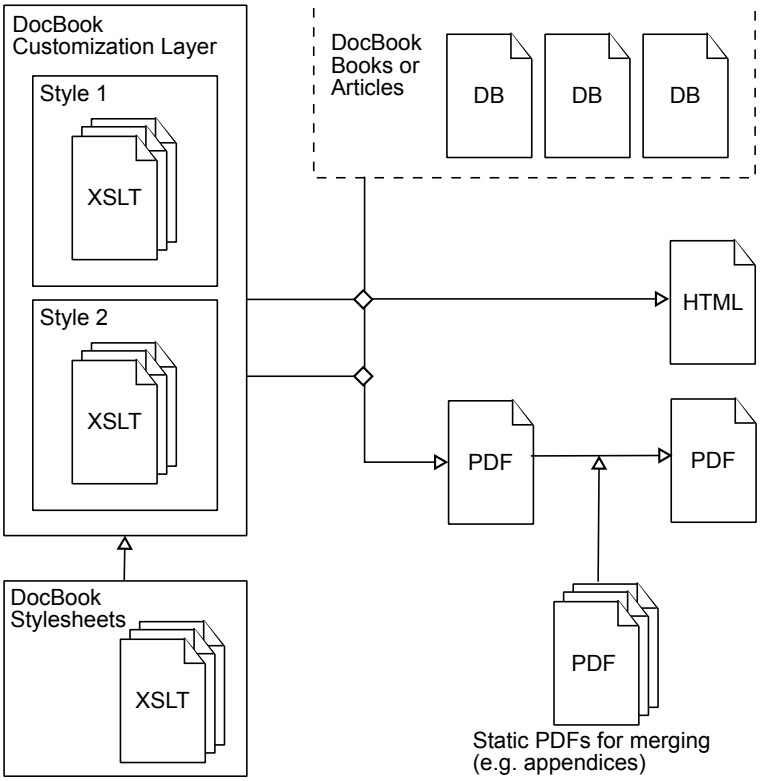


Figure 3. Document generation — Phase 2

There is a three level hierarchy in customization stylesheets. In the first level, there are customizations which are shared among all styles. Those declare common typesetting best practices shared by all produced documents.

Than there are styles which define appearance for each corporate identity involved. For example each subsidiary, product or department may have different requirements on styling of documents (for example different title pages, logos, colors, fonts). DocBook is very flexible in the way the output may be customized, even in a pixel precise manner. Even a very creative corporate identity design may be easily implemented in the DocBook customization XSLT if it adheres at least to some extend to general typesetting conventions.

The last layer composes of specific modifications to different styles. For example the header of the document may slightly differ in case the output is a company official letter, a proposal or a detailed design document. For the proposal we like to highlight the company name or maybe even logo on every page in the header but for detailed design documents with hundreds of pages this is not desirable. It rather makes sense to use the header to show the current context (chapter / section) of the document.

Finally, if required, the resulting PDF document can be automatically split and merged with static PDF blocks using a PDF manipulation library. In some cases it makes no sense to convert large presentational appendices into DocBook and

maintain them. For example PDF data sheets for equipment involved in a certain project. Merging such blocks into the resulting document needs to be automated. Doing manual merges every time the document changes is cumbersome and error-prone.

4.2. XInclude

XInclude [11] is a perfect tool to make XML data modular and to reduce duplicities. It helps to keep a single source of information. Changing a certain information in one place will automatically result in changes wherever the information is included.

XInclude is used in the DSL data to modularize it for easier maintenance, but duplicate data are avoided already in the DSL design [6].

XInclude plays a more important role when interweaving individual document fragments to compose the resulting document. In this case XInclude allows to pick arbitrary sets of elements from a set of XML documents. Moreover, includes may be embedded inside other includes. For example an included section may have further figure includes inside. Correct relative URI resolution of `hrefs` inside the included fragments is done through `xml:base` attributes.

One of the common issues which needs to be coped with is the limitation of XML that each well-formed document needs to have a single root element. Imagine one of the XSLT stylesheets generates a sibling set of sections. Such sections can't be stored in a well-formed XML document without having a common root element. A correct DocBook ancestor for a set of sibling sections is for example a chapter. In case, the described set of sections needs to be included into another chapter, simple XInclude would produce a chapter in chapter situation, which is an invalid DocBook. In this case the `xpointer` [12] construct has to be used to specify the range of elements to be included[7].

Although it brings flexibility and easier maintenance, the use of XInclude is also problematic. Even being a W3C recommendation for several years now the tool support in Java (a mainstream programming platforms) is quite buggy. Several workarounds need to be done to get correct XInclude behaviour with the latest Xerces (2.9)[8] and Saxon (9.1).

First of all Xerces has to be patched to produce correct `xml:base` attribute for embedded includes[9]. Than still Xerces supports only the XPointer element() scheme

[6] DSL data can be normalized for example using ID references.

[7] There are much more use-cases where the use of `xpointer` is necessary. For example a main table which has all source data and several other tables which include only certain rows from the main table. Or a certain figure has to be included into another context from a DocBook chapter and so on.

[8] Note that Xerces is the default XML parser in the latest Java JVMs

[9] This is a known bug since 2005. `xml:base` for embedded includes are not relative to the parent include base URI. The patch (bug 1102) has been released just recently (middle 2008) and it still did not make it into the latest release.

[13]. This means only individual elements may be referenced. Moreover, only DTD-determined shorthand IDs are supported[10] and addressing element by position is error-prone.

Unfortunately the only way how to make a reasonable subset of XPointer working with latest Xerces is to associate a DTD with the DocBook documents using DOCTYPE. To make offline generation of documents possible and to allow DocBook fragments on any arbitrary place in the file system an XML Catalog needs to be configured in the transformer.

Still with all the patches and workarounds listed above, only a very basic XInclude/XPointer subset is supported. Either whole XML files or individual elements marked with IDs can be included. Advanced constructs as for example ranges or XPath are not functional.

Even in this crippled form, XInclude is extremely useful and the authoring framework would suffer significant drawbacks without it.

4.3. Automate with Ant

Ant [14] is a multi-platform Java build tool. The build process is described using XML. It consists of a hierarchical set of targets dependent on each other. Each target than consists of a sequence of tasks. Each task is described by a certain XML element with certain attributes and child elements to define the tasks execution parameters.

In the authoring framework, where all other components are mostly declarative, Ant plays the procedural role. Its build file specifies the whole document generation process from the DSL data transformations to the resulting document generation.

There are generic build definitions common to all projects within the framework and specific (per project) build definitions which may override the generic behavior. In simple scenario, the generic definition may be used as it is to generate documents from DSL data. Placing stylesheets and input data into a certain directories within the project directory structure will result in applying all XSLT automatically to the source data.

For complex projects with several different target documents the default behaviour needs to be overridden. In such case, the specific build definition is basically a serie of transformation tasks defining inputs, outputs and stylesheets.

The Ant's default XSLT transformation task is not sufficient for the needs of the authoring framework which requires XInclude (as discussed in Section 4.2) and XSLT 2.0 and XPath 2.0 support (for more powerful and easier XSLT templates and Schematron validation). That's why the authoring framework implements it's own Ant macro for XSLT transformations.

[10] Without an DTD no attribute is considered to be an ID, even the `xml:id` attribute.

Example 1. The <saxon> task usage

The simplest usage[11].

```
<saxon source="${domain}" output="${out}" stylesheet="${xslt}"/>
```

Advanced usage with up-to-date checks and stylesheet parameters. Up-to-date checks allow to regenerate the target only in case the sources did change. The default source is the file in the `source` attribute, but in case of XInclude use, all the included files need to be specified manually using `<uptodat-source>` element.

```
<saxon source="${domain}"
       output="${out}"
       stylesheet="${xslt}">
<uptodate-source><fileset dir="${domain.dir}">
<includes name="**/*"/>
</fileset></uptodate-source>
<parameters><arg value="param1=1"/></parameters>
</saxon>
```

Example 2. Running Saxon with catalog and XInclude support

This `<java>` task definition, is the core of the `<saxon>` macro defined in the generic Ant build files.

```
<java classname="net.sf.saxon.Transform" fork="true" ▶
failonerror="@{failonerror}">
 <arg value="-xi"/>
 <arg value="-x"/><arg value=" ...ResolvingXMLReader"/>
 <arg value="-y"/><arg value=" ...ResolvingXMLReader"/>
 <arg value="-r"/><arg value=" ...CatalogResolver"/>
 <arg value="@{source}"/><arg value="@{stylesheet}"/>
 <parameters/><classpath>
  <path refid="transform.classpath"/>
  <pathelement location="... resolver${os-env}.jar"/>
 </classpath>
 <jvmarg value="-client"/>
 <jvmarg value="-D... DocumentBuilderFactory= ...DocumentBuilderFactoryImpl"/>
 <jvmarg value="-D... SAXParserFactory= ...SAXParserFactoryImpl"/>
</java>
```

In addition to the `<saxon>` tasks the authoring framework common build definition contains several other useful XML manipulation tasks. All of them are defined with the use of common Ant tasks as Ant Macros. No Java programming was required to define them. They are pure XML.

[11] `output` and `stylesheet` attributes are optional. If `output` gets omitted standard output is used, if `stylesheet` is missing the `xml-stylesheet` processing instruction is used to determine the stylesheet.

Further custom Ant tasks

schematron	Schematron validation task, uses the `<saxon>` task.
xmlconcat	Concatenates several XML files into one big XML file with an artificial root element. This is useful in case the a stylesheet needs to operate over several such XML files at once and the use of XInclude is not possible because it is no known in advance what files shall be concatenated[12].
csv2xml	Converts a comma separated values file into XML tabular format which later may be processed using an XSLT stylesheet and thus turning the CSV file into the DSL format. This is a way how to turn for example purely presentational data in Excel into a semantic XML by adhering to some agreed structure of the input Excel document.

5. SVG

Modern documents need to get visual in order to succeed in competition. Visual data are significantly more understandable to humans than paragraphs of texts. Having a more human understandable proposal may help win a tender. Having a better understandable documentation may help to sell a product or decrease requirements on stuff knowledge.

This trend leads to an increasing ratio of visual data in documents but it may also increase maintenance requirements. Textual data are usually easier to maintain especially when doing frequent changes. This section introduces several techniques how to increase maintainability of visual data through the use of XML technologies.

Scalable Vector Graphics [4] is the technology which can help to maintain visual data in output documents. The language is XML-based which allows to utilize the very same XML-based techniques described in the previous sections. Today's FO processors as well as browsers are very well able to handle SVG data which means SVG may be used directly without any conversions to produce visual PDF or HTML outputs.

5.1. Image callouts

The DocBook documentation [1] describes callouts as — " a visual device for associating annotations with an image, program listing, or similar figure. Each location is identified with a mark, and the annotation is identified with the same mark."

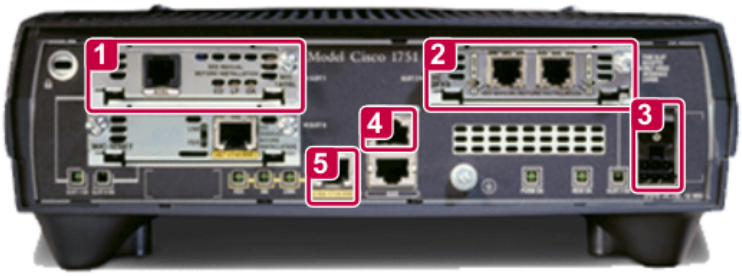
Keeping annotations separate from the actual annotated content makes it much easier to apply changes to that content. This is especially true for image callouts. Visually annotating images (diagrams, photographs...) is a frequent use-case but

[12] Alternatively `collection()` may be used.

creating and maintaining them manually is a nightmare. Even DocBook stylesheets do not implement image callouts by default, the common DocBook customization layer of the authoring framework described in this article does.

The authoring framework uses purely XML technologies (XSLT and SVG) to create annotation regions and marks within annotated images.

First the DocBook sources are being preprocessed by XSLT and for each annotated `<mediaobject>` with an image reference an SVG file is created using the XSLT `<result-document>`. The generated SVG size is based on the size of the original image, and the image gets included in the background using `<svg:image>`. Than the stylesheet reads coordinates of the individual annotated regions and generates SVG rectangles including numbering for each annotation. The `fileref` of `<imagedata>` is altered to point to the newly generated SVG file. Figure 4 shows the generated image — an annotated back of a router device with ports and buttons explained. The DocBook example source fragment follows.



1. WIC/VIC Slot 1
2. VIC Slot 2
3. Power switch and power socket
4. Console port
5. 10/100 Ethernet port

Figure 4. Image callouts result

Example 3. DocBook source for the annotated image

```
<imageobjectco>
<areaspec>
 <area xml:id="p1"
  coords="45,20 111,36"
  units="px"/> ...
</areaspec>
<imageobject>
 <imagedata
  width="383" height="140"
```

```
    fileref="images/cisco1750.png"/>
  </imageobject>
  <calloutlist>
   <callout arearefs="p1">
    <para>WIC/VIC Slot 1</para>
   </callout>...
```

Rather than pre-processing the DocBook document, an alternative option for image callouts is to place the SVG generation directly into the DocBook customization stylesheet where it naturally belongs. But in this case we need to use an XSLT 2.0 construct (`<result-document>`) within an XSLT 1.0 DocBook stylesheets[13] or we could use processor specific extensions for that.

5.2. Graphs

Another frequently used visual element in documents are graph-like diagrams. Although it would be possible to enable automatic graph layouting within the authoring framework, this is out of scope of this article which rather demonstrates a different approach.

Auto-generated graphs may safe a lot of time and resources especially when considering a huge amount of diagrams to maintain. But even advanced layouting algorithms are involved, the result can never compete in terms of beauty or human readability with manually layouted diagrams.

A typical example of graphs in the networking domain are network diagrams. Those are usually undirected graphs, where nodes represent some kind of a network device (router, switch...) and edges are labeled with interfaces, IP addresses and protocols.

Networking experts love to use Microsoft Visio for drawing network diagrams as it has nice layouting features and a huge clipart gallery of various networking devices. From the XML tool chain perspective Vision has quite good SVG export which makes it possible to post-process the diagrams automatically.

The networking projects domain model groups individual sites into sets of a certain type. Those sets share same hardware configuration and same networking schema. Only the individual IP addresses, interfaces and device names differ according to the IP plan for each site. This means the diagram authors only draws a schema per site type rather than maintaining schemas for each individual site. Correct per-site IP addresses, interface and device names are obtained by the authoring framework stylesheets from the DSL data when automatically generating detailed per-site diagrams.

Mapping between Visio entities (usually labels) and the domain specific data is achieved through Visio custom properties. Visio features special property sets which

[13] XSLT 2.0 stylesheets for DocBook is currently work in progress

may be assigned to the diagram as a whole or to individual entities or groups of entities. When exporting to SVG, elements in the SVG namespace are wrapped in Visio-specific elements which preserve many (although not all) Visio-specific information[14].

During generation of the resulting document, each Visio diagram is processed by an XSLT stylesheet. Each label with a certain set of custom properties is mapped to an XPath expression in the stylesheet which is than evaluated against the DSL data and the text of the label is replaced with the retrieved text nodes.

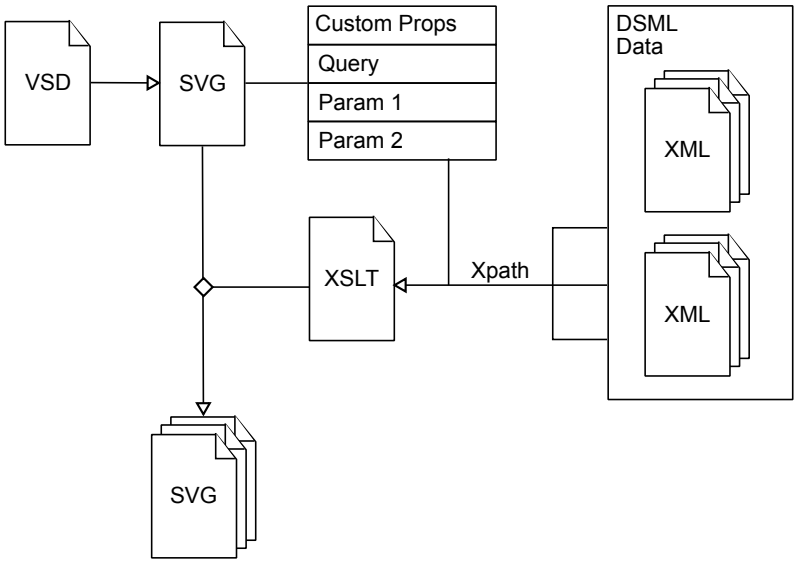


Figure 5. Processing Visio diagrams

When exporting SVG, Visio makes several mistakes which needs to be corrected by the processing stylesheet.

Some auto-corrected Visio SVG mistakes

- styles need to be adjusted to display arrow ending correctly
- only align left works out of the box for labels, right and center needs to implemented in the stylesheet by adjusting the SVG output according to Visio-specific align information

[14]This means that the SVG diagram may still be opened in Visio and edited. This helps the content authors to edit the resulting appearance of the diagrams using Visio directly. Changing the position of some diagram entity in Visio will result in a changed position also in the generated PDF output. Unfortunately the exported SVG does not equal to the original native Visio file and some information may be lost during export. Therefore it is anyway a good idea to keep original Visio sources along with the exported SVG.

5.3. Fully auto-generated SVG diagrams

Layouting graphs automatically is a complex tasks, but some simpler diagrams can be fully auto-generated easily. For example for the networking documentation it is important to visualize rack layouts with devices on a per-site bases.

Fully visual rack layouts may help the maintenance personnel recognize the individual devices at a particular site, see how many interfaces they have or what is their position within racks.

Domain specific stylesheets in the authoring framework are able to generate rack layout diagrams automatically by composing different SVG fragments into one resulting SVG diagram. Rack configurations for individual site types are described in the DSL data. Each rack has a number of slots defined which may be occupied by devices. Each device may occupy one or more slots. An SVG clipart may be associated with a certain group of devices and thus defining their appearance. A shared SVG clipart gallery is used to maintain them. If a certain device group does not have a clipart in the gallery a default device appearance is used.

XSLT stylesheets are used to generate SVG representation of each rack, gather the different SVG fragments for each device in the rack from the gallery, scale the graphics to occupy the right amount of slots in the rack and filter everything redundant to obtain a single valid SVG output[15].

Device cliparts are set in place using SVG `matrix()` transformation which translates the embedded SVG to the appropriate position and scales it up or down to fill the requires space. This requires to do some mathematical calculations within the stylesheet including unit conversions.

The aim is to simplify creation of device cliparts in the shared clipart library as much as possible. All manipulations to the SVG clipart are done automatically by the stylesheet. Creating a new clipart may be as easy as opening Visio, choosing a certain device from the palette and exporting it into SVG[16].

[15] Such filtering includes for example merging `<style>` element contents from various SVG fragments into a single `<style>`. Or filtering any embedded `<svg:svg>`.

[16] Some FO processors support only a subset of SVG, e. g. XEP does not support gradients. Automatic processing needs to be done in this case in the stylesheet. For example for gradients, the background color may be set to a color in the middle of the first and last stop.

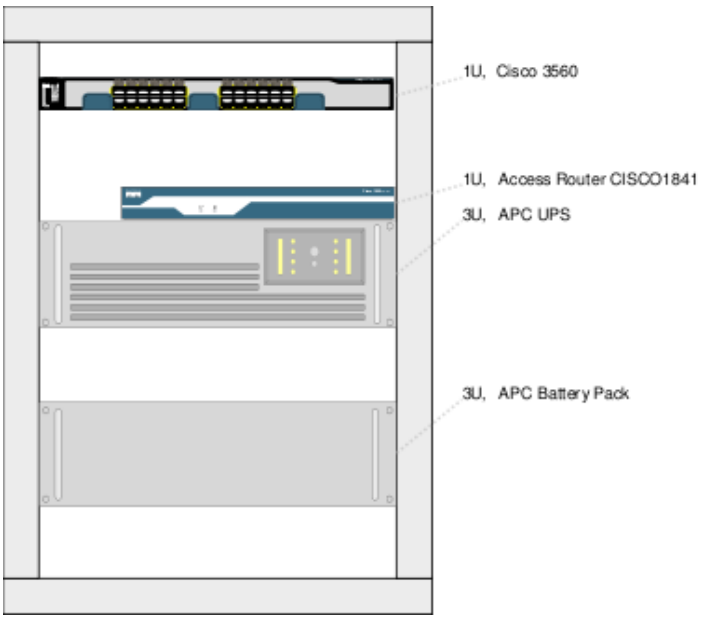


Figure 6. Auto-generated rack layout

5.4. Other generated graphics

Expressing information in a visual form is very powerful. Why sticking to figures? DocBook can be enriched by graphics in a more fine grained manner. SVG may be embedded for example directly into DocBook table cells.

Let's again demonstrate this on the networking domain example which we use across the whole article. Consider some sites in the project needs to be connected wirelessly with each other. From their geographical location we can calculate their distance and azimuth[17].

An azimuth is typically expressed in degrees. A nice visual representation of the azimuth is a circle with a pointer. Such visual documentation may help engineers installing the wireless directional antennas to point them in the right direction to gain signal strength. Figure 7 shows visual data being mixed with traditional textual data in a table. The screenshot is taken from the resulting PDF file. The small SVG clips are generated directly in the DocBook customization layer and injected into FO using `<instream-foreign-object>`.

[17]This is no issue as XSLT is well equipped for mathematics. For example there are stylesheets in the authoring framework for the networking domain which do distance and bearing calculations from geographical coordinates for individual sites. This involves calculations with trigonometric functions and requires high precision.

			Primary Connection		
BS/Sec	Azimuth [°]	Distance [m]	Frequency [MHz] MAC	BS/Sec	Azimuth [°]
3 / 1	153° 	2,708	3573.75 00:00:00:00:00:01	1 /4	73° 

Figure 7. SVG inside table cells

6. When semantics is missing

Sooner or later any XML authoring framework will need to address the issue how to gather data from different non-semantic sources. Section 5.2 describes how to automatically process Visio diagrams. In Section 4.3 there is a little note on how to export data from Excel/CSV and convert them first into tabular XML and later into domain specific XML.

There are basically two options how to cope with this issue. Either force users (even non-technical) to output semantic XML directly or leave the users use the tools their are used to and attach semantics to data later automatically or at least semi-automatically.

6.1. Produce semantic XML visually

The stumbling block of wider adoption of XML for authoring is the lack and limitations of visual editing tools. Non-technical people are afraid of XML, they consider it some sort of black magic.

Developing a good visual tool for XML editing resembles a quadrature of the circle. Tools are trying to shield the user from all kinds of complexities of the XML tree and thus getting the users into a position where they are not aware of what they are actually doing. Such users are unable to solve issues which the editor cannot solve for them when the scenario gets just a little more complicated.

From the experience of evaluating visual XML editors such as Epic, XMetaL, Oxygen and XML Mind, it is obvious that those tools are still after years of development very far from being perfect[18].

Of course the quality of the editors differs very much and each editor is more or less suitable for a certain set of tasks, but the author of this article has so far best experiences with the XML Mind editor for it's flexible extensibility, very good level

[18] A simple test will reveal the immaturity of the tools. Open a DocBook document in Oxygen Author, click to a paragraph level and insert a new section. No problem, the editor inserts the section right inside the paragraph and thus producing an invalid DocBook document. As the next step the editor will underwave it's own mistake with red.

of visual user experience and also for the licensing policy. Anyway, it is still very difficult to get non-technical users use such tools.

6.2. Non-XML tools

Traditional presentational tools are getting more and more XML enabled recently. Not only Microsoft Visio has SVG export. MS Word is able to present XML documents visually and even allow very primitive editing of such documents. MS Excel has a limited capability to bind tabular data to simple XML schemas and thus producing semantic XML out of spreadsheets.

There are also semi-automatic approaches to convert purely presentational documents from MS Word into DocBook. In theory, Open XML could be transformed to some form of DocBook which may be later enhanced with additional semantics manually. Another approach is to open an MS Word document in Open Office and use it's DocBook export feature. This produces a very ugly DocBook source which may be partially automatically enhanced (XSLT).

7. Beyond authoring

Authoring is a relatively narrow domain which can be very well handled with XML tools. But the potential is far greater. There is no need to limit the output to a traditional document.

Structuring data correctly and assigning semantics to them brings immense flexibility. With XSLT we are free to transform the data into all sorts of very different formats to make the best use of them for specific applications. To demonstrate the potential applications lets show few interesting use-cases specific to the networking project domain.

7.1. Google Earth

Google Earth uses the Keyhole Markup Language (KML), an XML-based language for expressing geographic annotation. Having longitude and latitude specified for each site in the domain model allows to mark and annotate the sites on the Google Earth surface. Lines connecting individual sites may represent wireless or other connections, different icons may represent different site types. Clicking on a site will reveal further information about a particular site.

Having projects visualized in Google Earth is not only imposing, but it also helps to very well visualize certain aspects of the project for effective project planning, resource management etc...

7.2. Network monitoring

Network monitoring is very important during the support phases of every networking project.

There are already all necessary information in the DSL data to setup a monitoring system. There are all devices, their IP addresses and interfaces and how are they connected with each other. Such data may be transformed to create configuration files for a monitoring system such as HP Open View or Nagios/Nagvis.

With help of the monitoring tools the static domain data may be turned into a dynamic view of the current network displaying current status of all devices and connections with flashy green/red colours.

7.3. Device configuration files

The DSL data may be used to automatically generate configuration files for different type of devices in the network. Specific XSLT stylesheets can be used to configure routers, switches, firewalls and other device.

Automatic generation will guarantee consistency and may avoid errors when configuring devices manually.

8. Conclusion

This articles has shown how to automate authoring for detailed and visual documentation to support all phases of networking projects using purely XML-based tools. But the networking field was more or less used only as an example use-case. The principles, approaches and best-practices described in this article are widely applicable for many different domains.

Moreover the reader witnessed how XML was utilized with advantage to model a domain specific language, validate data consistency, describe modularity of data, define data transformations and transformation chains, describe document structures, style documents and visualize data.

Bibliography

[1] Walsh, N.: DocBook 5.0: The Definitive Guide. 2008. URL: http://www.docbook.org/tdg5/en/html/docbook.html

[2] Stayton, B.: DocBook XSL: The Complete Guide. Sagehill Enterprises, 2008. URL: http://www.sagehill.net/docbookxsl/index.html

[3] Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.0 (Second Edition). W3C, 2006. URL: http://www.w3.org/TR/2006/REC-xml-names-20060816

[4] Ferraiolo, J., Fujisawa, S., Jackson, J.: Scalable Vector Graphics (SVG) 1.1 Specification. W3C, 2003. URL: http://www.w3.org/TR/2003/REC-SVG11-20030114

[5] Berglund, A.: Extensible Stylesheet Language (XSL) Version 1.1. W3C, 2006. URL: http://www.w3.org/TR/2006/REC-xsl11-20061205

[6] Clark, J.: XSL Transformations (XSLT) Version 2.0. W3C, 2007. URL: http://www.w3.org/TR/2007/REC-xslt20-20070123

[7] Berglund, A., Boag, S., Chamberlin, D., Fernández, M., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C, 2007. URL: http://www.w3.org/TR/2007/REC-xpath20-20070123

[8] Nálevka, P., Kosek, J.: Advanced approaches to XML document validation. University of Economics, Prague, 2007. URL: http://www.idealliance.org/papers/extreme/proceedings/html/2007/Nalevka01/EML2007Nalevka01.html

[9] Nálevka, P.: Grammar vs. rules. University of Economics, Prague, 2007. URL: http://nalevka.com/content/Home/regular_grammar-all.en.html

[10] Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron. ISO/IEC 19757-3, 2006. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c040833_ISO_IEC_19757-3_2006(E).zip

[11] Marsh, J., Orchard, D., Veillard, D.: XML Inclusions (XInclude) Version 1.0 (Second Edition). W3C, 2006. URL: http://www.w3.org/TR/2006/REC-xinclude-20061115

[12] Grosso, P., Maler, E., Marsh, J., Walsh, N.: XPointer Framework. W3C, 2003. URL: http://www.w3.org/TR/2003/REC-xptr-framework-20030325

[13] Grosso, P., Maler, E., Marsh, J., Walsh, N.: XPointer element() Scheme. W3C, 2003. URL: http://www.w3.org/TR/2003/REC-xptr-element-20030325

[14] Loughran, S., Hatcher, E.: Ant in Action. Manning, 2007. ISBN: 1-932394-80-X

[15] Walsh, N.: Image Callouts. 2006. URL: http://norman.walsh.name/2006/06/10/imageobjectco

Xdefinition 2.1

Václav Trojan
Syntea software group a.s.
<vaclav.trojan@xmlprague.cz>

Abstract

The paper describes Xdefinition 2.1 as an integral instrument for the design and implementation of projects with XML objects. In our concept of Xdefinitions we respected as much as possible the requirement of application of XML documents in the whole process throughout the life cycle of the information system development. The principal requirements were comprehensibility for all participants of the individual stages of IS formation as well as the binding character of the description and its modularity. By comprehensibility we understand that the description must be short and clear not only for the IT specialists but also for the wider range of other experts participating in the project, particularly in the analytical stages. By the binding character we understand exactness of the description and applicability of the description for the machine processing: description of data may include also the directives for data processing or data generation. The modularity allows decomposition of objects descriptions and configuration of the versions of their structure in the complicated and permanently changing environment of the distributed system. Xdefinitions 2.1 also allow defining bindings of the objects to different situations in the course of their processing.

Xdefinitions itself are XML objects that may be used also as a meta-language to describe generally languages above XML.

Keywords: XML, validation, data modeling

Παντα ρει. Everything is in a state of flux
— *Herakleitos of Ephesos 535–475 B.C.*

1. Why Xdefinitions 2.1?

XML data may be validated with DTD, RELAX NG or XML schema or Schematron. XML objects can be generated by using for example XSLT transformation. Moreover, in the data processing we cannot then avoid working with XML objects also with a universal programming language. In communication with analysts and external participants we must describe the objects structures in a language they understand. Thus the project inevitably involves several different forms of the description of

identical objects. The problem is that such a system is difficult to maintain and its integrity might be at risk. It is not easy to make sure that the changes in one form of description are reflected in relevant sites and in different languages. That is why in the version 2.1 of Xdefinitions we emphasised maximum comprehensibility of the description, simple maintenance of objects and possibility of automatic transformation of data to the language of Xdefinitions. We respected also easiness of implementation.

In the continuously changing world of real projects which involves work with a large number of XML objects and their generation, processing and modification or transformation at many often very different sites and in various situations it is useful to minimise the number of the forms of description. Such tool should be intuitively comprehensible and able to:

1. Describe the valid structure as much as simple and exactly
2. Describe commands for processing of objects in various sites (to minimize dependence of program code on the data structure)
3. Describe generation of objects and guarantee validity of a generated objects
4. Decompose the object description into more simple segments which then may be used to compose and newly configure more complicated objects
5. Ensure easy implementation to different platforms and environments.

To illustrate the philosophy of Xdefinitions concept we will now try to describe the work with a simple XML object. Let us take a simple XML object containing information about a book:

```
<book category="children">
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <IBSN>9780439887458</IBSN>
  <price>39.99</price>
</book>
```

First we will try to design a formula for generating such an object. Let us imagine we have a method which for example retrieves from the database a value according to the name of a column in the line of a table. The name of the column will be the parameter of this method. For example invoking the method getItem("cover") will return the string "children". We can simply create the formula for the generation of our object by describing how the values should be filled in the XML object. So instead of values in our object we shall simply specify the methods with relevant parameters that will return the relevant values:

```
<book category="create getItem('category')" >
  <title> create getItem('title') </title>
  <author> create getItem('author') </author>
  <IBSN> create getItem('IBSN') </IBSN>
```

```
  <price> create getItem('price') </price>
</book>
```

The processor of this description will process the above formula and compose the subsequent XML objects by creating relevant elements and filling in the values obtained by invoking the functions specified after the key word "create". The result of the invocation will then be XML object composed in accordance with the formula above. The advantage of such description will be independence of the program code on the structure of the data. Note that the method may be an XPath function or an external method (however, it can be a built-in method of the processor of Xdefinitions). The external program doesn't know anything about the structure of data.

And now let us imagine a different situation when we have an XML object we want to validate. Let us presume we have methods parsing data in the validated object and returning a boolean value "true" if the result is correct or value "false" if the result is incorrect. The formula for validation may then look like:

```
<book category="optional enumeration('children','adult','unknown')">
  <title> required string() </title>
  <author> required string() </author>
  <IBSN> optional numeric(8,15) </IBSN>
  <price> required decimal(4,2) </price>
</book>
```

As we can see instead of values in this case we have described specification whether the value is compulsory or optional and we have described the validation of values by methods verifying the formal correctness of values. The verification of the *type* of values may thus be understood again as a specification of a method that will return a boolean value depending on whether the value is valid. In this case the processor will proceed in the reverse manner than in our first example. It will process (parse) the input data and it will check up the format of values by invoking relevant methods according to the above description. In case the compulsory value is missing or if the validation method returns false value, the processor will report an error and the object will not be recognised as valid.

Both above mentioned formulas can be merged into one:

```
<book category="optional enumeration('children','adult','unknown');
                create getItem('category')" >
  <title> required string(); create getItem('title') </title>
  <author> optional string(); create getItem('author') </author>
  <IBSN> optional numeric(8,15); create getItem('IBSN') </IBSN>
  <price> required decimal(4,2); create getItem('price') </price>
</book>
```

In the validation regime we shall ignore the part describing the generation of XML object.

When processing XML objects, it may be useful to define instructions determining what should happen in various situations that could occur in the course of processing and insert them in the description. In such a case we may for example require certain actions such as e.g. error notification, deposition of values to the database, etc. So let us now insert in our description instructions for actions to be executed in certain situations. For example:

```
<book category="optional enumeration('children','adult','unknown');
                default setValue('unknown')">
   <title> required string(); onAbsence error('Missing title') </title>
   <author> optional string() </author>
   <IBSN> optional numeric(8,15); onError error('Incorrect IBSN');
          onAbsence error('Missing IBSN')
   </IBSN>
   <price> required decimal(4,2); onError error('Incorrect price');
           onAbsence error('Unknown price')
   </price>
</book>
```

And bellow we have the formula containing all the above information:

```
<book category="optional enumeration('children','adult','unknown');
                default setValue('unknown');
                create getItem('category')">
   <title> required string();
           onAbsence error('Missing title');
           create getItem('title')
   </title>
   <author> optional string();
            create getItem('author')
   </author>
   <IBSN> optional numeric(8,15);
          onError error('Incorrect IBSN');
          create getItem('category')
   </IBSN>
   <price> required decimal(4,2);
           onError error('Incorrect price');
           onAbsence error('Unknown price');
           create getItem('category')
   </price>
</book>
```

It is clear this formula may be used in the mode for XML object generation (create) or the mode for its processing (and/or validation). In the processing mode we shall ignore the formula parts related to “create” and in the “create” mode we shall ignore the instructions concerning processing.

Such a formula describing methods to be executed in different situations enables us to describe not only the structure but also the processing of XML data and this way to minimize number of forms of description. Note that it also minimizes the dependence of programming code on the structure of data.

2. Models of elements

As we have demonstrated, the description of an element in Xdefinition contains in the sites of data values the data about their occurrence, generation instructions and potentially also directions for data processing. The complete element description in Xdefinitions is called **model of element**. For the description we use a special language: **script** of Xdefinitions. In an element model the script describes values or text nodes in sites where they may occur. Element model is thus intuitively comprehensible because it has a similar structure as the data it describes.

The validation of the types of values in Xdefinitions is performed by so called validation methods the result of which is a boolean value containing information about the validity of the parsed or created object. Since these methods may have parameters which enable further specification of the required features of values, these parameters can be understood as restrictions for a certain type. For example formula “numeric(8, 15)” means sequence of minimum 8 and maximum 15 digits. The whole part of the script describing request for object occurrence and validation method is called **validation section** of the script.

It is not sufficient for the description of element features to describe attribute values and text nodes and that is why Xdefinitions include a special attribute “script” (from Xdefinitions’ namespace). The script enables us to describe element features e.g. its occurrence, invocation of methods in various situations, etc. It also describes occurrences of child elements in the element model. The following example illustrates the application of the script in case we want to describe that a certain child element can be omitted or that it can occur more than once:

```
<author xd:script="occurs +"> required string </author>
<IBSN   xd:script="occurs ?"> optional decimal(8,15) </IBSN>
```

The occurrence description may explicitly state the minimum and maximum number (unbound limit we can specify as “*”):

```
<author xd:script="occurs 1,*"> required string </author>
<IBSN   xd:script="occurs 0,1"> optional numeric(8,15) </IBSN>
<price  xd:script="occurs 1"> required decimal(4, 2) </price>
```

Default occurrence if the specification is omitted is equal to "occurs: 1" for elements or “required for attributes”.

To minimize the specification we can omit keywords "occurs", "required" and "optional". Specification of the occurrence may be reduced to characters ‘?’, ‘+” and

'*'. Also, if a method has the empty parameter list it is possible to omit brackets. The example or reduced form will be:

```
<book category="? enumeration('children','adult','unknown')" >
   <title> string </title>
   <author xd:script="+"> string </author>
   <IBSN   xd:script="?"> ? numeric(8,15) </IBSN>
   <price> decimal(4,2) </price>
</book>
```

3. Groups of objects

Description of more complicated structures requires description of the groups of objects and potentially their variations. For this purpose we use auxiliary elements from Xdefinitions' namespace into which we insert the group items that is being described. The elements inside the group form description of the nodes that belong to the group (e.g. elements, text values, processing instructions, comments or groups again). Xdefinitions enable us to describe the following types of groups:

1. "**xd:sequence**" groups describes the set of items the occurrence of which corresponds with the sequence in the group description.
2. "**xd:choice**" group describes the set of elements from which a certain variation has been selected.
3. "**xd:mixed**" group are useful if we do not care about the sequence of nodes inside the group – all node permutations are valid (an analogy to "interleave" in RELAX NG)

Script can be also used to describe occurrence inside a group:

```
<foo>
   <xd:choice>
      <a/>
      <b xd:script="occurs 3"/>
      optional string()
   </xd:choice>
</foo>
```

The following versions of element "foo" correspond to the above description:

```
<foo><a/></root>
<foo><b/><b/><b/></foo>
<foo>text</foo>
<foo/>
```

Similarly as in case of elements, the attribute "script" may be specified also in the description of groups. For example:

```
<root>
   <xd:sequence script="occurs *">
      <a/>
      <b/>
   </xd:sequence>
</root>
```

To the above description corresponds to any number of element sequences of "foo" a "bar".

4. Events and actions

As we have demonstrated, in Xdefinitions we can specify invocation of methods to be executed in various situations or events in course of object processing. For example in course of data processing it may be important to report detail errors (not only formal validity of an object). It may be important also to describe object processing (storing to the database, etc.). Thus in Xdefinitions we enter the names of **events** and specification of the method to be executed. This method is called **action**. The relevant actions are specified after the name of the event.

Xdefinitions recognise a number of different events. In some cases an action connected to an event must return a value of a certain type, in other cases no value needs to be returned. (In the Xdefinition 2.1 the user may also define aliases for the predefined events to be able to specify which actions are executed or ignored on different sites.)

Various events may occur in the course of XML objects processing. Bellow we have listed the principal ones.

4.1. Events defined for all the objects

init – occurs at the start of further processing of an object; the action does not return a value

create – occurs only in the “create” mode, action for attributes and text nodes returns value string; for elements it is the iterator and we shall discuss “create” mode later in greater detail

finally – occurs only after the whole object has been processed (allows to execute the action following the prior processing of an object, when the whole object, the errors that occurred etc. are already known; the action does not return a value

onAbsence – object in data is missing; the action does not return a value

4.2. Events defined only for elements

match – invoked prior to the model application (before "init" event). The action must return boolean value and if the value is “true”, the model is applied, otherwise

the model is not applied to the given element (we shall return to this situation in the discussion)

onStart – occurs when processing of element starts (after "init" and after the attributes were validated, before processing of child nodes starts); the action does not return value

onExcess – occurs when the number of objects exceeds permitted number of occurrences; the action does not return value

forget – invoked after all events, even after finally. Specification of the event does not expect any other action. It just releases all data of the processed element from RAM of the computer. Specification of this event enables to process very large XML objects.

4.3. Events defined only for text values (attributes and text nodes)

passed – the action is executed only if the validation method returns value “true” ; the action does not return a value

onError – the action is executed only if the validation method returns value “false”; the action does not return a value

default – occurs when the value is optional and it is missing; the action must return value of the string type

4.4. Event "match"

Xdefinitions allow - in the element or group script - to describe a situation when the application of an element model depends on yet another condition. This situation can be described by the means of "match" event. In this case the result of the action must be a boolean value. This action is executed prior to the other actions (even prior to "init" action). It is in fact a filter determining whether the relevant model will be applied. If the result of the action is "false", the application of relevant model is skipped (to the given object) and if the result is "true", the model is accepted. The event "match" allow us to describe different variants of models. The following example describes various structures of elements that have the same name but whose structure differs depending on the value of the "type" attribute:

```
<xd:choice>
   <Shape type="string" xd:script="match xpath(@type='circle')">
      <r> float </r>
   </Shape>
   <Shape type="string" xd:script="match xpath(@type='rectangle')">
      <a> float </a>
      <b> float </b>
   </Shape>
</xd:choice>
```

4.5. Event "create"

In the opening section we have mentioned the “create” event. As we have already said, all actions connected to “create” events are invoked only when we are generating new XML documents (this mode is called **create mode**). Otherwise those actions are ignored. And on the contrary the other actions are ignored in the create mode (except for the validation section the execution of which can be set by a parameter). The actions for create event are used to describe how to generate an XML object. An action for create must return a value which allows creation of the relevant object. In the case of attributes or text values the result of the action must be convertible to a character data. In the case of element or groups the result of the action may be an object convertible to the iterator type which is able to return one or more then one items – child nodes (or none item). This iterator has two methods: hasNext() and getNext() and it is from some types of objects generated automatically:

- element, text node (iterator returns just one item)
- nodeList (iterator returns items from the list)

In the case no “create” action is described, a default action depends on the context in which it is invoked (e.g. the element of a relevant name is automatically created). Context is generally a part of the XML object tree. Two situations may occur: no context is available or a context from the previous action is available.

One possible variant of “create” mode execution is when we add as the input parameter a XML object according to which the result is generated. In such case “create” mode may be perceived as a transformation of the input XML object (similarly as XSLT). In this case create actions can work with a context which may be set by invoking XPath on the current context. For this purpose we use implemented method “from” which executes XPath on the current input context.

Unlike XSLT the Xdefinitions guarantee that the generated object is valid XML (it corresponds to the relevant Xdefinition). To understand following example we should know that if the result of the operation in action create” is a NodeList, the create action creates the iterator, which gradually applies the items of this list to the element description. Bellow we can see the “create” action with XML object on the input. Let us have an input XML object:

```
<monarchs>
   <monarch>
      <name>George I</name>
      <reigned>
         <start>1714</from>
         <end>1727</to>
      </reigned>
   </monarch>
   <monarch>
      <name>George II</name>
      <reigned>
```

```
         <start>1727</from>
         <end>1760</to>
      </reigned>
   </monarch>
</monarchs>
```

Xdefinition:

```
<governors>
   <king xd:script="*; create from('//monarchs/monarch')"
         name="string; create from('name/text()')"
         from="int; create from('reigned/start/text()')"
         to  ="? int(); create from('reigned/end/text()')" />
</governors>
```

The result of create action will then be:

```
<governors>
   <king name="George I" from="1714" to="1727"/>
   <king name="George II" from="1727" to="1760"/>
</governors>
```

4.6. Alias events

Users may define for the events in Xdefinitions their own alias names. These user defined events then mean that actions attached to them can be under certain conditions invoked and under different conditions ignored. These conditions can be specified with the activation of Xdefinitions processor. In this way we can modify Xdefinitions behaviour in various situations or at various sites.

5. Declaration of the types of values

In Xdefinitions types of values can be also declared separately from the model. For this purpose we use an auxiliary element “type” from the namespace of Xdefinitions:

```
<xd:type name = "currency">
   <validate> enumeration('GBP','USD','EUR') </validate>
   <onError> error('Incorrect currency')</onError>
</xd:type>
```

The reference to the declared type is formally similar to the specification of validation method:

```
<price> currency <price>
```

6. Namespaces in Xdefinitions

Xdefinitions – similarly as RELAX NG – treat namespaces in the intentions of namespace specification in XML 1.0 as with strings to distinguish them from local names. In Xdefinitions it is possible to define an arbitrary number of namespaces. The following example shows an Xdefinition describing a model with two different namespaces (prefixes "a" and "b"):

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/2.1"
        xmlns:a ="http://www.a.com"
        xmlns:b ="http://www.b.com">
   <a:foo>
      <b:bar a:id="string" />
   <a:foo>
</xd:def>
```

7. References

Within an Xdefinition mutual references to models can be made using "ref" construction. This case is illustrated by the following example:

```
<xd:def xmlns:xd = "http://cz.syntea.xdef/2.1">
  <person>
     <firstName> string </firstName>
     <lastName> string </lastName>
  </person>

  <family>
     <mother      xd:script="ref person"/>
     <father      xd:script="?; ref person"/>
     <xd:mixed>
        <daughter xd:script="*; ref person"/>
        <son      xd:script="*; ref person"/>
     </xd:mixed>
  </family>
</xd:def>
```

If we want to enable also references to groups, it is necessary to perceive such groups similarly as the element models, i.e. to record them as direct child nodes of Xdefinitions root and to give them a name. We call such kind of groups the **group models**. Therefore an attribute "name" which serves for a reference must be specified in the group model. Specification to the referred group is written in the script attribute on the site of reference:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:choice name="foobar">
      <foo/>
```

```
      <bar/>
   </xd:choice>

   <root>
      <xd:choice script="ref foobar"/>
   </root>
</xd:def>
```

Xdefinitions also enable making references to the description of attributes:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:attr name="myAttr"
            script="? int(1, 999)" />

   <root attr1="ref myAttr" />
</xd:def>
```

8. Model extension and model modification

You can add to the referred model the attributes and/or we can to add to it the child nodes simply by declaring them:

```
<shape>
   <x> float </x>
   <y> float </y>
</shape>

<rectangle xd:script="ref shape" a="float" b="float" />

<circle xd:script="ref shape">
   <diameter> float </diameter>
</circle>
```

The element „rectangle“ is extended by the attributes „a“ and „b“ and the element „circle“ is extended by the child element „diameter“.

The properties of referred model can be also redefined (including actions). Sometimes it we need to skip or omit some object. For this purpose we can redefine the specification of the occurrence by the keyword „ignore“ (the occurrence of the object is ignored) or illegal“ (the occurrence of the object is forbidden):

```
<square xd:script="ref rectangle" b="illegal" />
```

Attribute „b“ declared in model „rectangle“ is illegal in model „square“.

9. Element "any", otherElement, otherAttribute

To enable description of a situation when any element can occur at any site, Xdefinitions introduce a special element “any” from the namespace of Xdefinitions.

We can describe in the script of this element what should happen. Description of a case when we want to say that also other than specified elements or attributes may occur in the element model will be described in attributes "otherElement" and "otherAttribute" from the Xdefinitions namespace.

10. Processing instructions and XML document

In Xdefinitions it is possible also to describe occurrence of the processing instructions on specified position:

```
<xd:processingInstruction script="onAbsence genProcessingInstruction('myInstr', ▶
'myValue')"
    name="checkName"
    value="checkValue"/>
```

When processing XML objects the Xdefinitions processor must be instructed which model is a document root. The document is described in Xdefinitions by means of an auxiliary element from the Xdefinitions namespace.

```
<xd:document>
  <root>
    ...
  </root>
</xd:document>
```

A choice group enables specification of more variants of root elements in the document:

```
<xd:document>
  <xd:choice>
    <foo/>
    <bar/>
  </xd:choice>
</xd:document>
```

11. Recursion in Xdefinitions

References in Xdefinitions can be recursive and we can thus describe complex languages generally above XML. Simple example of recursion in the model:

```
<foo>
   <bar>
      <foo xd:script="?; ref foo" />
   </bar>
</foo>
```

Note that the inner „foo" element has occurrence „?", otherwise recursion would be infinite! Valid structures are:

```
<foo><bar/></foo>
<foo><bar><foo><bar/></foo></bar></foo>
<foo><bar><foo><bar><foo><bar/></foo></bar></foo></bar></foo>
...
```

The recursive references we can specify also in the groups:

```
<xd:choice name="command">
   <if>
      <condition xd:script="ref expr"/>
      <xd:choice script="ref command"/>
      <else xd:script="?">
         <xd:choice script="ref command"/>
      </else>
   </if>

   <assign ...>
   ...
</xd:choice>
```

12. Collections of Xdefinitions and mutual references

An Xdefinition can contain more element models and/or groups. However, it is also possible to compose a pool of Xdefinitions, in which in the individual Xdefinitions mutual references can be made to objects they contain. The mutual references are composed from the names of Xdefinitions and it is separated from the names of referred objects with the character "#". Such a set of Xdefinitions may be perceived as a **collection**. In this way it is possible to describe projects working with a large number of XML objects. The collection item can be an Xdefinition or another collection.

For example:

```
<xd:collection xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:def name="site1">
      <foo xd:script="ref site2#bar" />
   </xd:def>

   <xd:def name="site2">
      <bar> string </bar>
   </xd:def>
</xd:collection>
```

In Xdefinition situated in „site1" is a reference to the model in Xdefinition in „site2". Xdefinitions may be stored in different files in different sites and from these building stones the collection can be composed. The parts can be specified by attribute "include" in the collection.

Let us have two Xdefinitions with the description of an object "Person" stored in different files: in the first case describes data with the name and address and the second version contains only the numerical identifier to the database:

1) file at site "http://site1.com/person.xd":

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="person">
  <Person name="string" address="string" />
</xd:def>
```

2) file at site „http://site2.com/person.xd":

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="person">
  <Person id="integer" />
</xd:def>
```

At local file let us have an Xdefinition describing an object "Family" in a file „/myfolder/family.xd":

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="family">
   <family>
      <mother   xd:script="ref person#Person" />
      <father   xd:script="?; person#Person" />
      <xd:mixed>
         <daughter xd:script="*; ref person#Person" />
         <son      xd:script="*; ref person#Person" />
      </xd:mixed>
   </family>
</xd:def>
```

The structure of the individual family members will depend on whether we compose the collection from Xdefinitions from file described in paragraphs 1) or 2). We can specify collection composed with:

```
<xd:collection xmlns:xd="http://cz.syntea.xdef/2.1"
    include="file://folder/family.xd, http://site1.com/person.xd" />
```

or with:

```
<xd:collection xmlns:xd = "http://cz.syntea.xdef/2.1"
    include="file://folder/family.xd, http://site2.com/person.xd" />
```

The fact that the Xdefinitions collection may be composed from different parts makes it possible to describe in detail the variations of objects and their behaviour at different sites of the project.

13. Macros

The flexibility of description is further extended by the possibility of macro declarations and macro specifications. The expansion of macros is performed in a preprocessor prior to further compiling and processing of Xdefinitions. Macro reference

can be anywhere in the script. Macros may have parameters and they can be nested. The possibility of placing macros in a separate Xdefinition further extends the possibility of object modification.

14. Finally...

In this presentation we have managed to describe only some of the basic features of Xdefinitions. We did not go into details of types descriptions, for example the possibilities of working with values as with keys, of descriptions of canonical forms, declarations of script variables and their applications, etc. We have introduced Xdefinitions as an example of an integrated tool for the work with XML objects in real projects which can be easily implemented into various environments and platforms. The important feature of Xdefinitions for the purpose of specification of the interface with XML objects is their comprehensibility and ease of design. Xdefinitions enable not only the description of the structure of objects and their validation but also programming of these objects including detailed control of error situations. The possibility to describe behaviour of the process on the place where the values occurs enables a special way of programming. The code of such programs is highly independent on the structure of data. These features allow using Xdefinitions as the powerful tool for data validation. Xdefinifions are proved to be able to describe complex processing of XML data in distributed IT systems.

For more information see www.syntea.cz/xdef/2.1/info

Bibliography

[1] Clark, James: Do we need new kind of schema language, 2007, http://blog.jclark.com/2007/04/do-we-need-new-kind-of-schema-language.html

[2] Clark, James - Murata, Makoko: RELAX NG Specification, 2001, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[3] Fallside, David C. - Walmsley Priscilla: XML schema, 2004, W3C Recomendation

[4] Kamenicky, Jiří - Měska, Jiří - Trojan Václav: Why All of Humanity Does Not Speak Esperanto, 2007, http://xdef.syntea.cz/xdweb/userdoc/XMLPrague2007en.pdf

[5] W3C recomendation: Extensible Markup Language (XML) 1.0, 2008, http://www.w3.org/TR/REC-xml

Cool mobile apps with SVG and other Web technologies

Robin Berjon
Robineko
<robin@berjon.com>

Abstract

The capabilities of mobile devices increase ceaselessly, and on occasion they are even useful. That is the case of Web technologies that have been becoming mature and gradually more important in mobile devices.

This talk will look at the state of current implementations, at where mobile Web technology stands today notably concerning the recent release of SVG Tiny 1.2 and the improvement in support for WICD documents, and will show demos to give an idea of what can be done.

Keywords: SVG, XHTML, WICD, Mobile, Web, applications

1. Introduction

A few months ago SVG's new mobile version made it to its final release, which makes it a good time to sum up where open Web technologies stand in the mobile area today, and where they can be expected to go next.

That having been said, the best way to give a feel for what can be done is by showing it. Therefore, the talk that is associated with this paper is intended to be heavy in demos — which doesn't map well to proceedings. I will nevertheless do my best to outline the bigger improvements in this paper.

2. Open Web Standards Now

The situation that we have today is far from ideal, but not truly dire either. After several confusing years in mobile markup that have seen several different versions of mobile HTML, not to mention WML, the situation is clearing up. The fights that took place when a small group of people working in MPEG tried to take control of this segment are over and productive work can resume. On feature phones, users are happy with Opera Mini, whereas in more advanced phones browsers are increasingly interoperable, and can process much of the Web at large.

SVG is also growing healthily, even if it always seems to take more time than one would want. Hard numbers are difficult to come by but simple maths based

on reliable sales numbers show that at least one billion SVG-capable phone terminals were sold last year. When added to the fact that 30% of Web browsers being used support SVG natively, it is clear that progress is being made.

There are however limitations to be seen in this picture. First, it is often very difficult to know if a given handset supports SVG. And even when it does, it is often used for the on-device portal, the user interface, and things such as themes, but not always available to the user.

More importantly, the vast majority of the deployed SVG players conform to SVG Tiny 1.1, occasionally with a few extra features. And 1.1 is pretty limited, bringing nowhere near the interactive power and rich multimedia support that SVG Mobile 1.2 supports.

That being said, even if this state of affairs is imperfect, it does have the advantage that it provides a large existing and well-tested install base from which to grow, and existing usage on which to build a better system. Furthermore, interoperability between mobile SVG implementations is generally good, with issues mostly appearing when some of the more advanced non-graphical features are used in conjunction — issues that 1.2 also helps with.

3. What's New Today

Compared to SVG Tiny 1.1, 1.2 brings a number of new features that constitute a major jump forward. Since many of these features are self-descriptive and far more impressive as demos, I will only discuss them briefly here.

- *Better graphics.* New graphical features appear in this version, most notably the ability to use gradients and non-group opacity (which is to say that it is possible to specify the opacity of a fill, stroke, or gradient stop, but not to apply it to an entire group of shapes). In practice these features were already available in the unofficial SVG profile known as "SVG Tiny 1.1+" but having them standardised means they are to become available across the board. To be honest, I don't fully understand how we managed to live without gradients so long, as they can truly make an interface stand out.
- *Improved multimedia support.* SVG was always designed to work in conjunction with rich multimedia features, but so long as such features were not mandated by the specification their usage was not reliable. This version adds `audio` and `video` elements that can be embedded seamlessly into the content. Where the `video` element is concerned, there are some limitations with lower-end devices whereby it may not be possible to scale, rotate, or overlay the video; but despite those unavoidable issues, embedding video directly is still a major improvement. These features have notably been put to use to create music and TV players for mobile phones.

- *Scripting.* There simply was no way to script an SVG document until now. In practice, a lot of the implementations that shipped were able to support it (which is how phone some phone application user interfaces have been built) but there was no agreed-upon subset of the DOM that was known to be usable everywhere. This now changes with the introduction of the MicroDOM (or μDOM) which defines a subset that works well on the mobile, and adds a few features specific to SVG. Amongst other things, it notably supports the recently released Element-Traversal API which makes handling a DOM tree much simpler than it is with the regular DOM (it is expected that desktop browsers will release that too, in fact some already have).
- *General smaller improvements.* There were a number of issues with some parts of the 1.1 specification being poorly defined that are now thoroughly cleaned up — in fact it is this part that has taken the greatest effort, and while it is not necessarily the most impressive it provides a much stronger foundation on which to grow SVG further. A number of smaller features have also been added, such as better integration of events, more powerful 2D transformations, some level of network access akin to XMLHttpRequest, the ability to react to events in an audio or video stream in order to provide a user interface to control it, and more.

But SVG is not on its own, and belongs to an ecosystem of other open standards. One standard that is being deployed today but has largely flown under the radar is WICD Mobile. From the point of view of end-users, it does not add much. But for content developers, it finally resolves the many issues that appear when using HTML and SVG together. It is a boring specification to read as it concerns mostly small details that browsers need to get right, but having it available means that content can now mix HTML and SVG, and get interoperable results.

4. Sexy Stuff — Not Just For Mobile

It is all fine and well to discuss the mobile Web, but in an ideal world content should only be authored once and work everywhere. While there are problematic constraints to make that a reality, at the very least we can hope that new features in SVG, and in the integration of SVG with HTML, would make it to the desktop as well.

One of the first benefits of this convergence is the ability to develop mobile content on a desktop without needing to use a tedious emulator and ceaseless copying of content over to the phone. Of course, one will always need to test on the eventual device, but any progress that can be made without going to the terminal or to some specific (and often ghastly) emulation program is a boon to a developer's productivity. A good example of this unfolding is Opera. One can develop WICD content using Opera on the desktop and be reasonably confident that it will work the same on the device. Certainly, some of the more intensive features will not be

possible, and one has to be cautious not to become overly optimistic with performance, but it globally works.

Another benefit is more powerful graphics coming to the browsers. Both Firefox and Opera have recently shown that the latest or upcoming versions of their browsers support a host of truly exciting features.

One of the most exciting aspects is simply the ability to use SVG in many places in which images can be used, and to apply SVG features to non-SVG content. It starts with simple things such as using SVG for CSS backgrounds, in such a way that the SVG can be stretched to fill the box while still having (for instance) rounded corners; or using SVG gradients in CSS so that one can avoid having to create an image every time a little bit of spice is needed.

But it gets better as some of the more impressive advances involve applying SVG to HTML content. For instance, one can use an arbitrary SVG shape to crop HTML content. Better, SVG filters such as blurring, edge finding, or displacement maps can be applied to any HTML content — in fact there are several demos showing SVG filters being applied in real time to video. The examples are often somewhat useless even if cool, but experience shows that when such features fall in the hands of Web developers, after a little while spent tinkering they invent brand new ways of providing a better experience.

Of course, until Internet Explorer catches up with the rest of the browsers it is going to be difficult to make use of these features, but at least the way forward is being shown and innovation is happening.

Another related segment in which the conjunction of HTML and SVG is helping is widgets. There is currently a lot of momentum behind specifying widgets that interoperate across platforms, and a large part of the interest there is mobile-driven since there is little difference between a widget and most mobile applications.

Work in that area isn't finished but the basics are within reach, and already some parts of the industry are getting ready to ship widget support. This will enable people to create small mobile applications that work equally on a large number of mobile devices — something that has been extremely difficult up to now. And of course these make use of the usual suspects: HTML, SVG, CSS, the DOM (all wrapped in a zip archive with some metadata).

5. Things Needed & Things to Come

With all these nice things happening, one has to also point out the bad parts. The biggest issue remains that authoring tools are still very much lacking for SVG on its own, and it is naturally even worse for mixed content. There have been some improvements from some companies, including Adobe which surprisingly still improves its SVG export with new versions of its Creative Suite, and Ikivo for instance has made very interesting progress with their Animator and IDE tools, but there is no equivalent to Flash, Flex, or the Silverlight tools for WICD and SVG at

this time. Hopefully the market will evolve enough that someone will address this issue, or developers won't care much (as they don't seem to when it comes to "Web 2.0" content).

And of course, while all this unfolds there is still work to be done on preparing for the next version in this ceaseless kludge of technologies that we have come to love as the Web.

Current Support of XML by the "Big Three"

Irena Mlýnková
Department of Software Engineering, Charles University in Prague, Czech Republic
<mlynkova@ksi.mff.cuni.cz>

Martin Nečaský
Department of Software Engineering, Charles University in Prague, Czech Republic
<necasky@ksi.mff.cuni.cz>

Abstract

XML technologies have undoubtedly become a standard for data representation and manipulation. Thus it is inevitable to propose and implement efficient techniques for managing and processing XML data. A natural alternative is to exploit tools and functions offered by relational database systems. Even though the native XML databases are undoubtedly more efficient, relational databases are still more popular among XML users due to their long history, maturity and reliability.

In this paper we provide an overview of XML-processing functions that are currently supported by the so-called "Big Three", i.e. Oracle 11g, IBM DB2 9, and Microsoft SQL Server 2008. We firstly show what are the key aspects a user may require from an XML-enabled database. Then, we provide an overview of their support in the respective systems. And, finally, we compare and contrast the findings so that advantages and disadvantages of the particular systems are apparent.

Keywords: XML support, Oracle, IBM DB2, Microsoft SQL Server

1. Introduction

Without any doubt the eXtensible Markup Language (XML) [1] is currently one of the most popular formats for data representation. The wide popularity naturally invoked an enormous endeavor to propose faster and more efficient methods and tools for managing and processing XML data. Soon it was possible to distinguish several different directions based on various storage strategies. The four most popular ones are methods which store XML data in a file system, methods which store and process XML data using an (object-)relational database management system ((O)RDBMSs), methods which exploit a pure object-oriented approach, and native methods that use special indices, numbering schemas, and/or structures suitable particularly for tree structure of XML data.

Naturally, each of these approaches has both keen advocates and objectors who emphasize its particular advantages or disadvantages. The situation is not good especially for file system-based and pure object-oriented methods. The former ones suffer from inability of querying without any additional preprocessing of the data, whereas the latter approach fails especially in finding a corresponding efficient and comprehensive tool. Expectably, the highest-performance techniques are the native ones, since they are proposed particularly for XML processing and do not need to artificially adapt existing structures to a new purpose. Nevertheless, the most practically used ones are undoubtedly methods which exploit features of (O)RDBMSs. We speak about so-called *XML-enabled databases*. The reason for their popularity is that (O)RDBMSs are still regarded as universal and powerful data processing tools which can guarantee a reasonable level of reliability and efficiency.

The key aim of this paper is to provide an analysis of XML support that is offered by the three leading database vendors and their systems, i.e. *Oracle 11g*, *IBM DB2 9*, and *Microsoft SQL Server 2008*, sometimes denoted as the "Big Three". We firstly show what are the key aspects a user may require from an XML-enabled database management system. Then, we provide an overview of their support in the respective systems including examples that depict their functionality. And, finally, we compare the key findings so that advantages and disadvantages of the particular systems are more apparent.

The paper is structured as follows: Section 2 introduces the problems and issues related to XML processing in general. Section 3 describes how these issues are faced in the Oracle 11g database, Section 4 does the same for IBM DB2 9 database and Section 5 for Microsoft SQL Server 2008. Section 6 overviews the key findings and provides the general comparison of the three systems. And, finally, Section 7 provides conclusions.

2. General Requirements for XML Processing

In general the basic idea of XML processing based on an (O)RDBMS is relatively simple. The XML data are firstly stored into relations - we speak about so-called *XML-to-relational mapping* or *shredding XML data into tables*. Then, each XML query posed over the data stored in the database is *translated* to a set of SQL queries (which is usually a singleton). And, finally, the resulting set of tuples is transformed to an XML document. We speak about *reconstruction* of XML fragments.

Consequently, the primary concern of the database-based XML techniques is the choice of the way XML data are stored into relations. On the basis of exploitation or omitting information from XML schema we can distinguish so-called *generic* [2],[3] and *schema-driven* [4],[5] methods. From the point of view of the input data we can distinguish so-called *fixed* methods [2],[3],[4],[5] which store the data purely on the basis of their model and *adaptive* methods [6],[7],[8],[9], where also sample XML documents and XML queries are taken into account to find more efficient

storage strategy. And there are also techniques based on user involvement which can be divided to *user-defined* [10] and *user-driven* [11],[12],[13], where in the former case a user is expected to define both the relational schema and the required mapping, whereas in the latter case a user specifies just local mapping changes of a default storage strategy.

Approaching the aim from another point of view, the SQL standard has been extended by a new part *SQL/XML* [14] which introduces new XML data type and operations for both XML and relational data manipulation within SQL queries. It involves functions such as, e.g. `XMLELEMENT` for creating elements from relational data, `XMLATTRIBUTES` for creating attributes, `XMLDOCUMENT` or `XMLFOREST` for creating more complex structures, `XMLNAMESPACES`, `XMLCOMMENT` or `XMLPI` for creating more advanced parts of XML data, `XMLQUERY`, `XMLTABLE` or `XMLEXISTS` for querying over XML data using XPath [15],[16] or XQuery [17], etc.

As we have mentioned in the introduction, the native XML databases differ from the XML-enabled ones in the fact that they do not adapt an existing technology to XML, but exploit techniques suitable for XML tree structure. Most of them use a kind of *numbering schema*, i.e. an index that captures the XML structure. Examples of such schemas are *Dietz's encoding* [18], *interval encoding* [21], *prefix encoding* [22], *ORDPATHS* [19] or *APEX* [20]. And, naturally, such indices can be also exploited in relational databases to optimize query processing.

Last but not least, not only storing, querying and indexing are the key operations with XML data. A natural requirement is also the ability to check data validity or to apply XSL [24] transformations. However, such features do not need to be incorporated within the database, but they can be ensured using a kind of middleware. On the other hand, one of the key problems of each application is that it is usually dynamic, i.e. the data change. From a short-time period it means that we need to be able to update the data (in XML technologies it is currently covered by the *XQuery Update Facility* [25]). On the other hand, from a long-time period also the problem of *XML data evolution* occurs [23], i.e. the situation when the modifications of the data are more significant, they usually violate validity and new storage techniques need to be established.

In the following sections we show how these issues are faced in Oracle, IBM DB2 and Microsoft SQL Server.

3. Oracle 11g

The Oracle corporation[1] calls its ORDBMS Oracle as well. Recently, *Oracle 11g Release 1 (11.1)* [26],[27] has been released.

[1] `http://www.oracle.com`

3.1. Storing XML Data

Oracle supports two types of storage strategies - XML data type and a user-defined XML-to-relational mapping. The XML data type is called `XMLType` and its basic usage is very similar to classical atomic SQL data types such as `INTEGER` or `VARCHAR` as depicted by the following example.

Example 1. Oracle: Basic usage of `XMLType`

```
CREATE TABLE person (id NUMBER, desc XMLTYPE);

INSERT INTO person (id, desc) VALUES (1, XMLTYPE(
  '<record>
     <name>Irena Mlynkova</name>
     <date>13/1/2003</date>
     <email>irena.mlynkova@mff.cuni.cz</email>
   </record>'));

INSERT INTO person VALUES (2,
   XMLTYPE(bfilename('mydirectory', 'person.xml'),
           nls_charset_id('AL32UTF8')));

SELECT p.desc.getCLOBVal() FROM person p;
```

The `XMLType` can be stored in three possible ways. If we choose the *structured* storage, the XML data are stored as a set of objects in a set of respective relations. Apparently, this strategy is suitable for data-centric, highly structured XML documents. In case of *binary* storage the data are stored in a binary format optimized for XML. This approach is suitable for semi-structured XML data which cannot be fully shredded into tables. And, finally, in case of the *non-structured* storage the XML data are stored in the form of `CLOB`. This approach ensures the highest level of *round-tripping*, however at the cost of low efficiency of more complex operations than retrieval of a whole document.

The storage strategy is specified in the `CREATE TABLE` command using `STORE AS` clause. With regard to the three strategies we have three options - nothing, `CLOB` or `BINARY XML`. In addition, we can add characteristics determining the (un)necessity of data validity, e.g. `XMLSCHEMA` *schema_name*, `ALLOW ANYSCHEMA` or `ALLOW NONSCHEMA`, and the root element using `ELEMENT` *element_name* clause.

Example 2. Oracle: Determining storage strategies

```
CREATE TABLE person (id NUMBER, desc XMLTYPE)
XMLTYPE desc STORE AS CLOB
   XMLSCHEMA "http://www.example.com/personschema.xsd" ELEMENT "record";
```

If we do not specify anything, the default storage strategy is structured. In this case Oracle supports two options - each XML collection, i.e. an element with `maxOccurs > 1` can be stored either into a `VARRAY` or into `LOB`. By default all XML collections are stored into `LOB`s, however the user-required modifications can be specified within the `CREATE TABLE` command using clauses `VARRAY` and `LOB`. In addition, in both the cases we can also specify respective storage characteristics.

Example 3. Oracle: User-defined mapping in `CREATE TABLE`

```
CREATE TABLE person (id NUMBER, desc XMLTYPE)
XMLTYPE desc
   XMLSCHEMA "http://www.example.com/personschema.xsd" ELEMENT "record"
VARRAY desc."XMLDATA"."email"
   STORE AS TABLE tableOfEmails (
      (PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
LOB (desc."XMLDATA"."date")
   STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
             STORAGE(INITIAL 4K NEXT 32K));
```

Apparently, this approach can be used only in case of simple XML data, since in more complex cases it is quite user-unfriendly. Hence, Oracle supports also another option - annotating the XML schema of the XML data. The XML schema must be first associated with two namespaces `http://xmlns.oracle.com/xdb` and `http://xmlns.oracle.com/2004/CSX`. Then, the elements, attributes or complex types can be annotated using attributes such as `SQLName`, i.e. the name of respective SQL attribute for an element or an attribute stored into a single column, `SQLType`, i.e. name of object type for complex types or SQL type for simple types, `storeVarrayAsTable`, i.e. requirement for storing all collections into `VARRAY` type, etc.

3.2. Indexing XML Data

Similarly to indexing of relational data, also indexing of XML data is denoted for the purpose of higher efficiency of respective operations. The indexing approaches in Oracle naturally depend on the selected storage strategy. In case of structural storage, we can exploit classical relational indices. In case of binary or non-structured storage we can exploit XML indices similar to numbering schemas used in native XML databases - so-called `XMLIndex`.

Example 4. Oracle: Indexing an `XMLType`

```
CREATE INDEX myXMLindex ON person.desc INDEXTYPE IS XDB.XMLIndex;
```

The `XMLIndex` consists of three parts - *path index* that indexes all paths of the XML tree, *order index* that indexes relations parent-child, ancestor-descendant and sibling and *value index* that indexes all values. The position of each node is preserved using a variant of the ORDPATHS numbering schema.

3.3. Querying XML Data

For the purpose of XML querying Oracle supports two options - XQuery and SQL/XML. Evaluation of the queries can be optimized by the indices as described before.

Example 5. Oracle: SQL/XML and XQuery querying

```
SELECT id, EXTRACTVALUE(desc, '/record/name') "person_name"
FROM person
WHERE id > 10;

SELECT id, XMLQUERY('for $i in /record
                     where $i/name != "Irena Mlynkova"
                     order by $i/name
                     return $i/name'
                     PASSING BY VALUE desc RETURNING CONTENT) XMLData
FROM person;
```

3.4. Other Operations

Apart from storing and querying XML data, Oracle supports also other XML-related operations, in particular validity checking and XSL transformations. Validity can be checked in two ways - either within the `CHECK` clause of `CREATE TABLE` command or using a built-in function of `XMLType`.

Example 6. Oracle: Validity checking

```
CREATE TABLE person (id NUMBER, desc XMLTYPE)
CHECK (XMLIsValid(desc) = 1))
XMLTYPE desc
   XMLSCHEMA "http://www.example.com/personschema.xsd" ELEMENT "record";

SELECT p.desc.isSchemaValid(
   'http://www.example.com/personschema.xsd','record')
FROM person p;
```

XSL transformations can be applied using the `XMLTRANSFORM` function on any `XMLType` column. It returns the result as `XMLType` as well.

Example 7. Oracle: XSL transformations

```
CREATE TABLE tableXSL (xsl XMLTYPE);

SELECT XMLTRANSFORM(p.desc, x.xsl).GetClobVal()
FROM person p, tableXSL x
WHERE p.id = 2;
```

3.5. Updating XML Data

If we apply the classical `UPDATE` operation on an `XMLType` column of a relation, it causes replacement of the whole XML document stored in it. Naturally, this kind of update operation is required only in very special cases. On the other hand, neither XQuery nor SQL/XML involves update operations, whereas the XQuery Update Facility is not supported in Oracle so far. Nevertheless, Oracle supports own set of SQL functions that enable to replace, insert and delete various XML nodes of XML data. Examples of these functions are `updateXML`, `insertXMLbefore`, `insertXMLafter`, `appendChildXML` or `deleteXML`.

Example 8. Oracle: Update operations

```
UPDATE person
SET desc = APPENDCHILDXML(desc, 'record/name', XMLType('<dg>PhD</dg>'))
WHERE id = 1;
```

3.6. XML Schema Evolution

Oracle is able to cope with the problem of XML schema evolution, i.e. a situation when XML schema of the stored data is modified which can cause violation of validity or changes in the storage strategy.

Oracle supports two kinds of schema evolution. In *copy-based* schema evolution all instance documents that conform to an old schema are copied to a temporary location in the database, the old schema is deleted, the evolved schema is registered, and the instance documents are inserted into their new locations from the temporary area. Procedure `DBMS_XMLSCHEMA.copyEvolve` is defined for this purpose and its main parameters involve the old and the evolved schema and an XSL script that re-validates the old XML data if necessary.

On the other hand, the *in-place* evolution does not require copying, deleting, and inserting existing data and thus it is much faster. However, it can be applied only when the *backward compatibility* is ensured, i.e. if no changes to the storage strategy

are required and the evolution does not invalidate existing documents. `DBMS_XMLSCHEMA.inPlaceEvolve` is the procedure defined for this purpose.

4. IBM DB2 Version 9

DB2 version 9 [28],[29] is the latest release of database system provided by the IBM corporation[2].

4.1. Storing XML Data

DB2 supports similar storage strategies as Oracle, i.e. XML data type called `XML` and shredding into tables. `XML` is stored in a native structure optimized for XML which is similar to binary storage in Oracle. An XML document inserted into an `XML` column is stored separately from the base table and the column itself contains only a document ID. The shredding strategy exploits a user-defined strategy, where the required mapping is specified using XML schema annotations from namespace `http://www.ibm.com/xmlns/prod/db2/xdb1`. They involve elements/attributes such as `rowSet` for specifying target table name, `column` for specifying column name, `condition` that determines if decomposition inserts a row into a table, `defaultSQLSchema` that specifies the default SQL schema, `expression` that specifies a customized expression whose result is inserted into the respective table, etc.

4.2. Indexing XML Data

DB2 supports three XML indices. The *XML region index* stores the locations of each XML document stored in the database. An XML document is stored in one or more regions and the XML region index provides a logical mapping of these regions to retrieve document data. DB2 creates an XML region index for each table with `XML` column automatically. The *XML column path* index is also created automatically for each `XML` column and provides mappings of unique XML paths to their IDs. The last XML index called *XML index* allows enhancing the query performance by indexing XPath expressions. It is not created automatically but must be specified explicitly for each particular XPath expression. A sample XML index is depicted in the following example.

[2] http://www.ibm.com

Example 9. DB2: XML index

```
CREATE INDEX person_name_idx ON person(desc) GENERATE KEY USING
XMLPATTERN '/record/name' AS SQL VARCHAR;
```

4.3. Querying XML Data

Similarly to Oracle, DB2 supports XQuery for querying XML data and SQL/XML for exporting relational data to XML and embedding XQuery to SQL queries. In addition, it supports a special non-standard feature - embedding SQL queries in XQuery.

Example 10. DB2: Embedding SQL in XQuery

```
XQUERY
for $person in db2-fn:sqlquery('SELECT desc
                                FROM person WHERE id > 1000')/record
return $person/name;
```

4.4. Other Operations

Naturally, DB2 allows validating XML documents against XML schemas. In particular, `XMLVALIDATE` function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values and type annotations.

Example 11. DB2: XML schema validity checking

```
INSERT INTO person (id, desc)
VALUES (3, XMLVALIDATE('<record>...</record>'
     ACCORDING TO XMLSCHEMA URI 'http://www.example.com/personschema.xsd'
     LOCATION '/mydirectory/person.xml'));
```

On the other hand, XSLT transformations are supported by a built-in engine in DB2 version 9.5 and higher. For transformations, DB2 introduces the `XSLTRANSFORM` SQL function similar to the Oracle one.

4.5. Updating XML Data

DB2 allows updating whole XML documents in `XML` columns using standard `UPDATE` SQL command. Moreover, since version 9.5, DB2 supports the XQuery Update Facility as well. It allows changing values of specific XML nodes, replacing nodes as well as inserting, deleting and renaming particular nodes.

Example 12. DB2: XQuery Update Facility

```
UPDATE person
SET desc = xmlquery('
            copy $new := $desc
            modify do replace value of $new/record/email
            with "mlynkova@ksi.mff.cuni.cz"
            return $new')
WHERE id = 1;
```

4.6. XML Data Evolution

Lat but not least, DB2 also considers evolution of XML schemas. However, only the case when a new version of an XML schema is backwardly compatible with an old version is considered. Then, a stored procedure `XSR_UPDATE` replaces an existing schema with a new one.

5. Microsoft SQL Server

The last but not least database vendor is naturally the Microsoft corporation[3]. It produces an object-relational database system whose latest version is called *SQL Server 2008* [30],[31],[32].

5.1. Storing XML Data

XML data can be stored in SQL Server in `LOB` data type, native XML data type called `XML` or shredded into tables. The `LOB` option is directed for applications that require the exact copy of the stored XML data including insignificant white-spaces, order of attributes, etc. In case of XML data type the XML data are stored in a native XML repository which preserves all the XML-relevant items. And in the last case a classical user-defined storage strategy is supported, where an annotated XSD determines the particular schema.

Both the `LOB` and `XML` types are used in the same way as in the previous cases, i.e. as classical atomic SQL data types. The set of mapping annotations is defined in namespace `urn:schemas-microsoft-com:mapping-schema` and involves attributes such as `relation` for specifying the relation to store the XML node into, `field` for specification of a particular column, `key-fields` for specifying the columns that uniquely identify the relation, `relationship` for specifying key-foreign key relationship between relations, etc.

[3] `http://www.microsoft.com`

5.2. Indexing XML Data

An XML index can be created on XML data type columns. The so-called *primary XML index* indexes all tags, values and paths over the XML instances in the column (exploiting the ORDPATHS schema). Having a primary index, we can create any of the allowed *secondary indices* - `PATH`, `PROPERTY` or `VALUE`. The `PATH` index builds a B+ tree on (path, value) pair of each XML node in document order over all XML instances. The `PROPERTY` index creates a B+ tree clustered on the (PK, path, value) tuple within each XML instance, where PK is the primary key of the base table. And, finally, the `VALUE` index creates a B+ tree on (value, path) pair of each node in document order across all XML instances.

Example 13. SQL Server: Indices

```
CREATE TABLE person (id INT PRIMARY KEY, desc XML)

CREATE PRIMARY XML INDEX idx_desc on person (desc)

CREATE XML INDEX idx_desc_path on person (desc)
   USING XML INDEX idx_desc FOR PATH

CREATE XML INDEX idx_desc_property on person (desc)
   USING XML INDEX idx_desc FOR PROPERTY
```

In addition, SQL Server enables to create a full-text index (that can be used also for other SQL data types) over the XML data type column. Then, the SQL function `CONTAINS` enables to check whether the XML instance contains the given string anywhere in the document.

Example 14. SQL Server: Full-text indices

```
CREATE FULLTEXT CATALOG ft AS DEFAULT

CREATE FULLTEXT INDEX ON person (desc) KEY INDEX PK_docs_023D5A04

SELECT * FROM person
WHERE CONTAINS (desc, 'mff.cuni.cz')
```

5.3. Querying XML Data

Firstly, the XML data type can be queried using XQuery. For this purpose SQL Server supports various built-in functions having the XQuery query as a parameter, such as `exist` checking existence of nodes, `value` returning an SQL value and `query` returning XML data type result. In case of shredded XML data, function `nodes` returns one row for each node that matches the query.

To ease usage of both SQL and XML data in queries, SQL Server exploits the idea of data binding, i.e. mapping SQL values to XML values. For this purpose it provides two functions - `sql:variable` to use the value of an SQL variable and `sql:column` to use values from a relation column in XML query.

Example 15. SQL Server: Data binding in SQL queries

```
DECLARE @date varchar(20)
SET     @date = '13/1/2003'
SELECT  desc
FROM    person
WHERE   desc.exist ('/reord[date = sql:variable("@date")]') = 1
```

Contrary to the previous two cases the SQL Server's *SQLXML* has nothing in common with the SQL/XML standard. The idea is similar - to bridge the gap between SQL and XML data - however the syntax and usage is different. The `OPENXML` construct provides an SQL view of XML data using the user specified mapping, that can be specified either explicitly (as in the following example) or using a parameter that denotes a general mapping strategy.

Example 16. SQL Server: `OPENXML`

```
SELECT *
FROM OPENXML (@docHandle, '/record')
WITH (PersonName  varchar(10)   'name',
      PersonID    int           '@id',
      PersonEmail varchar(10)   'email')
```

Conversely, the `FOR XML` construct enables to create an XML view of SQL relations. The way the data are mapped can be influenced using four modes. The `RAW` mode generates a single `<row>` element per each row in the rowset that is returned by the `SELECT` statement; its columns are mapped either to attributes or subelements depending on other parameters. The `AUTO` mode generates nesting in the resulting XML using heuristics based on the way the `SELECT` statement is specified. The `EXPLICIT` mode allows more control over the shape of the XML view using a set of special directives. And, finally, the `PATH` mode together with the nested `FOR XML` query capability provides the flexibility of the `EXPLICIT` mode in a simpler manner - using paths and nested queries.

Example 17. SQL Server: `FOR XML` query and result

```
SELECT ProductModelID as "@id", Name
FROM Production.ProductModel
WHERE ProductModelID=122 or ProductModelID=119
FOR XML PATH ('ProductModelData')
```

```
<ProductModelData id="122">
  <Name>All-Purpose Bike Stand</Name>
</ProductModelData>
<ProductModelData id="119">
  <Name>Bike Wash</Name>
</ProductModelData>
```

5.4. Other Operations

Similarly to the previous cases, validity checking is supported in SQL Server. For this purpose it introduces a so-called `SCHEMA COLLECTION` which enables to store one or more XML schemas. An `XML` column or variable can be then associated with such collection and, hence, validity checking is ensured.

Example 18. SQL Server: `SCHEMA COLLECTION` and its usage

```
CREATE XML SCHEMA COLLECTION MyColl AS '
   <schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/personschema.xsd">
     <!- ... ->
   </schema>'

CREATE TABLE person (id INT PRIMARY KEY, desc XML (MyColl))
```

On the other hand, the XSL processing is not a direct part of SQL Server, e.g., in the form of a built-in function. However, it can be easily extended with such functionality using a stored procedure implemented using an external tool.

5.5. Updating XML Data

The XML data type supports a built-in function `modify`, that enables to update the respective XML data. As a parameter it gets the required operation. In particular, subtrees can be inserted before or after a specified node, or as the leftmost or rightmost child. Attribute, element, and text node insertions are all supported as well. Deletion of subtrees is supported and scalar values can be replaced with new scalar values.

Example 19. SQL Server: Updating XML data

```
UPDATE person SET desc.modify('
  insert <dg>PhD</dg>
  after (/record/name[.="Irena Mlynkova"])')
```

5.6. XML Data Evolution

SQL Server also deals with evolution of XML schemas, even though only in a very special way and only for `XML` columns. In fact, it is easily ensured using `SCHEMA COLLECTION`s and their ability to be extended with new schemas.

Example 20. SQL Server: Adding a schema to `SCHEMA COLLECTION`

```
ALTER XML SCHEMA COLLECTION MyColl ADD '
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.com/personschema2.xsd">
     <!- ... ->
  </schema>'
```

Consequently, the respective `XML` column or variable can contain data valid against both old and new XML schemas in the collection.

6. Overview and Comparison

For better lucidity the following table provides and overview of the XML-related functions that are (not) supported in the three systems and their key characteristics.

Table 1. Overview and comparison of key XML features

Feature	Oracle	DB2	SQL Server
XML data type	`XMLType`	`XML`	`XML`
	Structured, binary, non-structured	Binary, structured	`LOB`, native, structured
Mapping	User-defined	User-defined	User-defined
	Names, data types and storage strategies (`VARRAY` vs. `LOB`)	Relations, columns, conditions, expressions	Relations, columns, keys, relationships
Indexing	`XMLIndex`	Region index, column path index, XML index	Primary, secondary (`PATH`, `PROPERTY`, `VALUE`), full-text index
	ORDPATHS, path index, axes index	Indexing particular XPath expressions	ORDPATHS
Querying	XQuery, SQL/XML	XQuery, SQL/XML, SQL embedded to XQuery	XQuery, SQLXML (`OPENXML`, `FOR XML`)

Feature	Oracle	DB2	SQL Server
Other operations	Validity checking, XSL transformations	Validity checking, XSL transformations	Validity checking, XSL transformations only via an external tool
Updating	Own functions for inserting, replacing, deleting nodes	XML Update Facility	Own function with parameter for inserting, replacing, deleting nodes
Evolution	With/without backward compatibility	Backward compatibility must be ensured	Only for `XML` columns using `SCHEMA COLLECTION`s

As we can see, in general, all the three vendors follow the same pattern and try to support as much XML functionality as possible. The most advanced and, at the same time, standard-conforming support has Oracle, whereas the SQL Server traditionally ignores the proposed standards the most.

Under a closer investigation we can see that there are some significant differences in respective areas of XML support. Firstly, while all three systems support a kind of XML data type as well as shredding into relations, the storage strategies do not follow any kind of common standards. In addition, in case of user-defined mapping, all the systems require quite a highly skilled user, i.e. user-driven strategies are not supported.

As for the query capabilities, all the three systems support several kinds of indices that enable to speed up XML query evaluation. Naturally, they are highly related to the selected storage strategy. All the systems support the XQuery language and its embedding in SQL to enable working with both XML and SQL data at the same time. To further increase this ability, both Oracle and DB2 support the SQL/XML standard, while SQL Server provides own set of functions called SQLXML. Surprisingly, DB2 also supports a new feature - embedding SQL queries into XQuery.

Considering other operations with XML data, all systems naturally support validity checking and XSL transformations. On the other hand, considering the update operations, each of them has its own approach. Oracle provides a set of update functions, SQL Server provides a single function with multiple parameters and only DB2 already supports the XQuery Update Facility, i.e. a standard approach.

Last but not least, the important aspect of XML data evolution is being considered by all the systems, but only Oracle enables to deal with significant structural changes using XSL transformations. However, in this case the user responsibility for data correction is full, there are no automatic options.

7. Conclusion

The aim of this paper was to provide an overview of XML support in the currently most popular (object-)relational database management systems - Oracle 11g, IBM DB2 9, and Microsoft SQL Server 2008 - the so-called "Big Three". Firstly, we discussed the main topics related to XML processing. Then, we described how these issues are faced in particular systems. And, finally, we provided their mutual comparison. Our aim was to show which of the solutions already proposed in the scientific world are being exploited in the industry and to what extent. This text should serve as a good source of information for developers who search for a system supporting particular functions, as well as for researchers looking for an up-to-date topic with practical exploitation.

8. Acknowledgement

This work was supported in part by the Czech Science Foundation (GAČR), grant number 201/09/P364.

Bibliography

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau: Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C, 2008.

[2] D. Florescu and D. Kossmann: Storing and Querying XML Data Using an RDMBS. IEEE Data Eng. Bull., 22(3):27–34, 1999.

[3] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura: XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM Trans. Inter. Tech., 1(1):110–141, 2001.

[4] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In VLDB'99, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[5] K. Runapongsa and J. M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. In EDBT'02, pages 266–285, London, UK, 2002. Springer.

[6] M. Klettke and H. Meyer. XML and Object-Relational Database Systems – Enhancing Structural Mappings Based on Statistics. In Selected papers from the 3rd Int. Workshop WebDB'00 on The World Wide Web and Databases, pages 151–170, London, UK, 2001. Springer.

[7] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In ICDE'02, pages 64–75, Washington, DC, USA, 2002. IEEE.

[8] W. Xiao-ling, L. Jin-feng, and D. Yi-sheng. An Adaptable and Adjustable Mapping from XML Data to Tables in RDB. In VLDB'02 Workshop EEXTT and CAiSE'02 Workshop DTWeb, pages 117–130, London, UK, 2003. Springer.

[9] S. Zheng, J. Wen, and H. Lu. Cost-Driven Storage Schema Selection for XML. In DASFAA'03, pages 337–344, Kyoto, Japan, 2003. IEEE.

[10] S. Amer-Yahia. Storage Techniques and Mapping Schemas for XML. Technical Report TD-5P4L7B, AT&T Labs-Research, 2003.

[11] A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. The VLDB Journal, 14(1):30–49, 2005.

[12] S. Amer-Yahia, F. Du, and J. Freire. A Comprehensive Solution to the XML-to-Relational Mapping Problem. In WIDM'04, pages 31–38, New York, NY, USA, 2004. ACM.

[13] I. Mlynkova. A Journey towards More Efficient Processing of XML Data in (O)RDBMS. In CIT'07, pages 23–28, Los Alamitos, CA, USA, 2007. IEEE.

[14] ISO/IEC 9075-14:2003. Part 14: XML-Related Specifications (SQL/XML). Int. Organization for Standardization, 2006.

[15] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C, November 1999. http://www.w3.org/TR/xpath.

[16] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. W3C, January 2007. http://www.w3.org/TR/xpath20/.

[17] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C, January 2007. http://www.w3.org/TR/xquery/.

[18] P. F. Dietz. Maintaining Order in a Linked List. In STOC'82, pages 122–127, New York, NY, USA, 1982. ACM.

[19] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In SIGMOD'04, pages 903–908, New York, NY, USA, 2004. ACM.

[20] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an Adaptive Path Index for XML Data. In SIGMOD'02, pages 121–132, New York, NY, USA, 2002. ACM.

[21] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In VLDB'01, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers, Inc.

[22] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In PODS'02, pages 271–281, New York, NY, USA, 2002. ACM.

[23] M. Mesiti, R. Celle, M. A. Sorrenti, and G. Guerrini. X-Evolution: A System for XML Schema Evolution and Document Adaptation. In EDBT'06, LNCS, pages 1143–1146. Springer, 2006.

[24] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C, November 1999. http://www.w3.org/TR/xslt.

[25] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Simon. XQuery Update Facility 1.0. W3C, srpen 2007. http://www.w3.org/TR/xquery-update-10/.

[26] Oracle Database 11g. Oracle Corporation. http://www.oracle.com/technology/products/database/oracle11g/.

[27] Oracle XML DB Developer's Guide 11g Release 1 (11.1). Oracle Corporation. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28369/ toc.htm.

[28] DB2 Product Family. IBM. http://www-01.ibm.com/software/data/db2/.

[29] IBM DB2 for Linux, UNIX, and Windows: Managing XML Data - Best Practices. IBM. http://download.boulder.ibm.com/ibmdl/pub/software/dw/dm/db2/bestpractices/DB2BP_XML_0508I.pdf.

[30] Microsoft SQL Server 2008. Microsoft Corporation. http://www.microsoft.com/sqlserver/2008/.

[31] XML Best Practices for Microsoft SQL Server 2005. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/ms345115.aspx.

[32] White Paper: What's New for XML in SQL Server 2008. Microsoft Corporation. http://www.microsoft.com/sqlserver/2008/en/us/wp-sql-2008-whats-new-xml.aspx.

Solving problem of XML data orchestration in large and distributed information systems

Jiří Měska
Syntea software group a.s.
<jiri.meska@syntea.cz>

Abstract

This paper is an attempt to orchestrate definitions of XML data objects and their processing in a large and distributed information system. In developing and managing IT systems the main challenge is to maintain the analytical and processing consistency of data object descriptions in the context of the system implementation as well as in the context of the permanently changing world and its impact on the business individual features of the information system.

To achieve the goal we introduce the concept of ***Data Orchestration Project*** *which is an analogy to the role of BPEL in solving problem of the orchestration of the web services. Through* ***Data Control Points*** *we can interconnect the management or control activities of some particular program with the data definitions.*

The objective of this paper is to present a possibility of creating IS Data Project of fictitious IS - Traffic Accidents Register (IS TAR). Implementation of the Data Control Points is based on ***Xdefinition*** *technology.*

Keywords: XML, validation, data modeling

1. Problems of consistency of data descriptions in large information systems

For the purpose of this paper the **Information system** means a set of hardware and software modules and applications that serve business intent. Such information systems are on one hand distributed in space and heterogeneous, on the other hand from business point of view strictly interlinked by data flows that link the individual components and applications. These data flows are represented by data objects that are shared across the IS.

As an example we have decided to use a fictitious IS TAR - Traffic accidents register. This system includes applications for the online/offline composition and

editing of traffic accidents, central register of traffic accidents, the register interface to other public administration registers, an interface to insurance companies, statistical applications, etc.

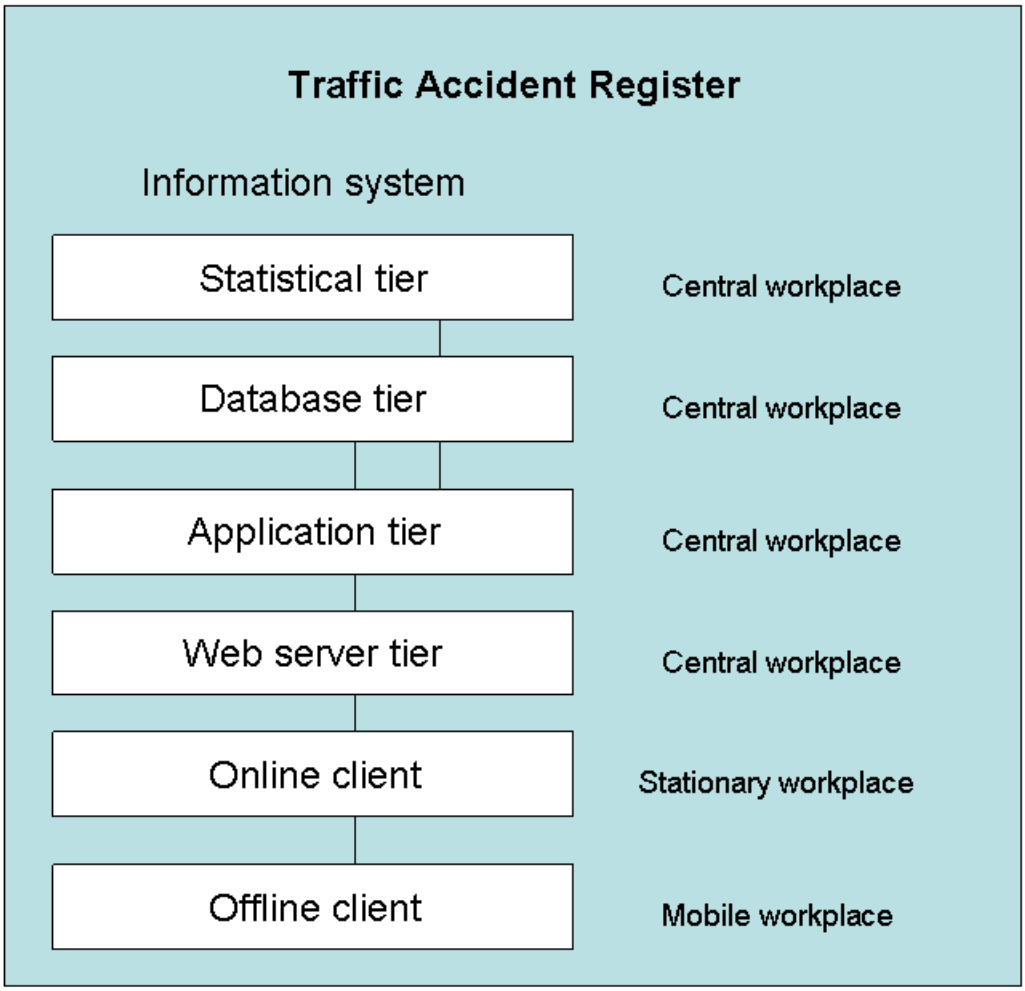


Figure 1. The following picture is simplification of the described task

2. Data Orchestration Project

In developing and managing large and distributed IT systems the main challenge is to maintain the analytical and processing consistency of data objects in the context of the system implementation as well as in the context of the permanently changing world and its impact on the business individual features of the information system.

The **Data Orchestration Project** means a set of mutually interlinked and mutually referring data descriptions which analytically and program-wise render the data descriptions in data control points of the whole information system and which are at the same time embedded in the business intent that guarantees they are meaningful and justified.

The **Data Control Point** with regard to the data model means a specific place in the IS, in which we can interconnect the management or control activities of the program with the data descriptions. By data control point we mean e.g. definitions of data validation conditions, interfaces, database model, data object creation etc.

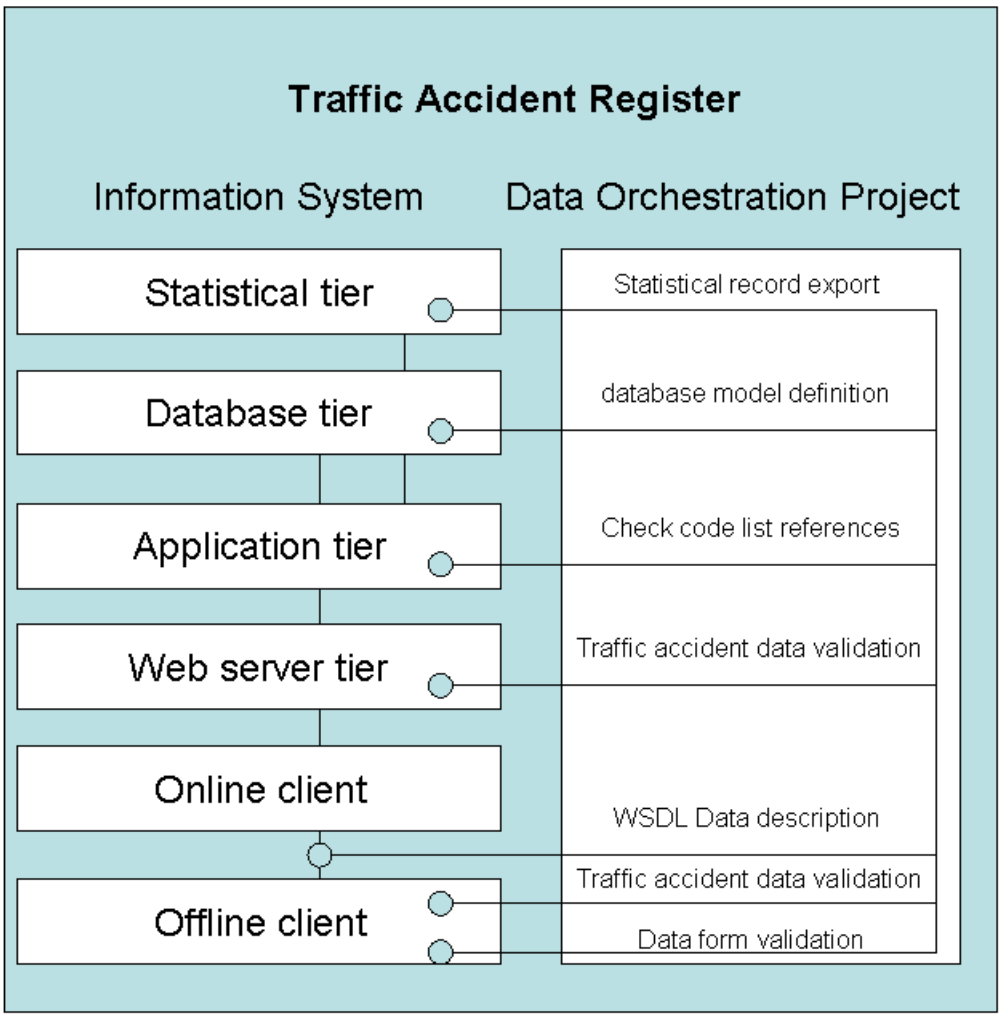


Figure 2. Trafic Accident Register

If we are able to create a data model comprehensible on the analytical level, the Data Orchestration Project enables us to understand - on analytical level - the behavior of the system in the Data Control Points and at the same time to guarantee this behavior in these points. For example through the logging on analytical level we can even manifest the required behavior. On the other hand, if all data definitions are mutually interlinked and if they run under single administration, any modification in any data definition must comply with the data administration of the whole system.

1. XML structure and its validation at a data control point
2. XML structure and its creation at a data control point
3. In both above cases the description of activities or dynamic invocation of external functions related to the processing of XML data in a context of Data Control Point

Data Orchestration Project is set of Xdefinitions that describes XML data objects and their behavior on different Data Control Points.

Table 1. Overview of the IS - TAR Data Orchestration Project used in our example

File	Purpose
TAR.xd	XML model definitions
FormPerson.xd	Simplified description of input data form validation
CheckCodeList.xd	Check of external code list references (alternative keys)
Statistic.xc	Statistical export - creation

```
Purpose: XML model definitions
File: TAR.xd

<xd:documentation>Traffic accident general description</xd:documentation>
<TrafficAccident
    ReferenceNumber   ="optional string(0,26)"
    DateDnFrom        ="required datetime('d.M.y H:mm')"
    DateDnTo          ="optional datetime('d.M.y H:mm')"
    CodeDistrict      ="required string()"
    CodeRoad          ="required string()"
    CodeCrossRoad     ="required string()"
>
   <xd:mixed>
      <Vehicle      xd:script= "occurs 0..; ref Vehicle"/>
      <Person       xd:script= "occurs 0..; ref Persen"/>
      <Inspection   xd:script= "occurs 0..; ref Inspection"/>
```

```
      <Survey       xd:script= "occurs 0..; ref Survey"/>
   </xd:mixed>
</TrafficAccident>
<xd:documentation>
      Description of vehicle taking part on traffic accident
</xd:documentation>
<Vehicle
      VehicleSequenceNumber  ="required int()"
      CodeTypeVehicle        ="required string()"
      Damage                 ="required int(0,99999999)"
      DamageDescription      ="required string()"
      DamageCargo            ="required int(0,99999999)"
      DamageCargoDescription ="required string() "
      VIN                    ="optional string(0,26) "
      CodeInsuranceCompany   ="required string()"
      InsuranceNumber        ="required string()"
>
   <Position xd:script= "occurs 0..;"/>
</Vehicle>
<xd:documentation>
      Person/subject taking part on traffic accident
</xd:documentation>
<Person
      RefVehicleSequenceNumber  ="optional int() "
      Title                 ="optional string(1,24)"
      Name                  ="required string(1,24)"
      Surname               ="required string(1,26)"
      Birthday              ="optional datetime('d.M.y')"
      InsurenceNumber       ="optional string()"
      CodeSex               ="required string()"
      CodeInjury            ="required string()"
      InjuryDescription     ="optional string(1,4000)"
      Behavior              ="optional string(1,4000)"
      CodeDriverLicence     ="required string()"
      Alcohol               =" required boolean()"
/>
   <Address xd:script= "occurs 0..;"/>
</Person>
<xd:documentation>
      Inspection of the place of traffic accident </xd:documentation>
<Inspection
      CodeVisibility   ="required string()"
      CodeWeather      ="required string()"
/>
<xd:documentation> Survey of the traffic accident </xd:documentation>
<Survey
```

```
        Sanction            ="optional int() "
        CodeCause           ="required string()"
/>
```

Purpose: Simplified description of input data form validation
Comment: Input form for inserting data about traffic accident participant
File: FormPerson.xd
Data Control Point: Offline Client/Person form

```
<PersonForm
      xd:ref             = "Person"
      xd:reftype         = "implements"
      xd:controlpoint    = "OfflineClient/FormPerson"
      xd:processing      = "Validation"

      RefVehicleSequenceNumber =""
      Title             ="onError error('Title must not exceed 24 character')"
      Name               ="onAbsence error('Name is obligatory');
                          onError error('Title must not exceed 24 character')"
      Surname           ="onAbsence error('Name is obligatory');
                         onError error('Title must not exceed 26 character')"
      Birthday          ="onError error('Birthday is of the form d.M.y')"
      InsurenceNumber   ="onAbsence error('insurance number is obligatory')"
      CodeSex           =""
      CodeInjury        =""
      InjuryDescription =""
      Behavior          =""
      CodeDriverLicence =""
      Alcohol           =""
/>
```

Explanation: xd:reftype= "implements"
This condition express that this definiton must be one to one to the ref model.

Purpose: Check of external code list references (alternative keys)
Comment: Check of data integrity
File: CheckCodeList.xd
Data Control Point: ApplicationTier/CheckCodeListReferences

```
<TrafficAccident
   xd:ref="TrafficAccident"
   xd:reftype= "implements"
   xd:controlpoint="ApplicationTier/CheckCodeListReferences"
   xd:processing="Validation"

   ReferenceNumber =""
```

```
   DateDnFrom  =""
   DateDnTo    =""
   CodeDistrict="required CheckCodeList(@CodeDistrict, 'CC_ListDistrict')"
   CodeRoad  ="required CheckCodeList(@CodeRoad, 'CC_ListRoad')"
   CodeCrossRoad="required CheckCodeList(@CodeCrossRoad,
                                         'CC_ListCrossRoad')"
>
   <xd:mixed>
      <Vehicle        xd:script= "occurs 0..; ref Vehicle"/>
      <Person       xd:script= "occurs 0..; ref Persen"/>
      <Inspection   xd:script= "occurs 0..; ref Inspection"/>
      <Survey       xd:script= "occurs 0..; ref Survey"/>
   </xd:mixed>
</TrafficAccident>
…
<PersonForm
      RefVehicleSequenceNumber=""
      Title     =""
      Name      =""
      Surname   =""
      Birthday  =""
      InsurenceNumber=""
      CodeSex   =" required CheckCodeList(@CodeSex, 'CC_ListSex')"
      CodeInjury=" required CheckCodeList(@CodeInjury, 'CC_ListInjury')"
      InjuryDescription =""
      Behavior  =""
      CodeDriverLicence="required CheckCodeList(@CodeDriveLicence,
                                         'CC_ListCrossRoad')"
      Alcohol   =""
/>
```

Purpose: Statistical export for year 2008 - creation
File: Statistic.xd
Data Control Point: StatisticalTier/ListAccidents

```
<xd:DataPointInterface>
      <xd:function xd:definiton="prepare('string', 'string')"/>
      <xd:function xd:definiton=" getCollumn('string', 'string')"/>
</xd:DataPointInterface>
<TrafficAccident
      xd:ref="TrafficAccident"
      xd:reftype= "reduction"
      xd:controlpoint="StatisticalTier/ListAccidents"
      xd:processing="Create"
      xd:oninit="prepare('select* from TrafficAccident
                          where DateDnTo in interval(?, ?)',
                          '1.1.2008','31.12.2008')"
```

```
      ReferenceNumber ="create  getCollumn('ReferenceNumber')"
      DateDnFrom      ="create  getCollumn('DateDnFrom')"
      DateDnTo        ="create  getCollumn('DateDnTo')"
      CodeDistrict    ="create  getCollumn('CodeDistrict')"
      CodeRoad        ="create  getCollumn('CodeRoad')"
      CodeCrossRoad   ="create  getCollumn('CodeCrossRoad')"
>
</TrafficAccident>

Explanation: xd:reftype= "reduction"
This condition express that that original data model is reduced to elements ▶
and atributes described here
```

3. Conclusion

If we follow IS architecture on SOA principles we need tools for orchestration of web services. To achieve this goal we can use WS-BPEL.

In this presentation we have tried to postulate requirements on comprehensible orchestration of data object descriptions. We have achieved this goal on an example of IS - TAR through the Data Project Orchestration based on technology of Xdefinition.

Jiří Kosek a Vít Janota (ed.)

XML Prague 2009
Conference Proceedings

Vydal
MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 255. publikaci

Obálku navrhl prof. Nešetřil

Z předloh připravených v systému DocBook
a vysázených pomocí XSL-FO a programu XEP
vytisklo Reprostředisko UK MFF
Sokolovská 83, 186 75 Praha 8

1. vydání

Praha 2009

ISBN 978-80-7378-061-6