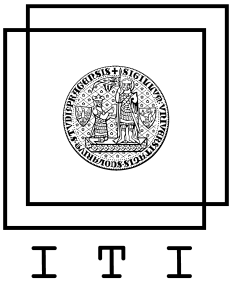
ITI Series

Institut Teoretické Informatiky
Institute for Theoretical Computer Science



2010-488

xmlprague

XML Prague 2010

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 2010 – Conference Proceedings

ISBN 978-80-7378-115-6

XML Prague 2010

Conference Proceedings

Lesser Town Campus
Prague, Czech Republic

March 13–14, 2010

a•gile / ă′•juhl
adj. marked by ready ability to move with quick easy grace

MARK LOGIC

marklogic.com/agility

Accelerate the creation of information products.

Syntea software group a.s.

(www.syntea.cz)

Syntea is developer and supplier of IT solutions of large process-oriented systems. Syntea has designed, implemented and successfully uses the technology of **Xdefinition**.

Xdefinition

- facilitate validation and processing of XML documents
- simplify and make more precise the communication between users, analysts, architects and programmers
- accelerate implementation of IT solutions and increase the robustness of the applications in routine use

Xdefinition is the tool for description of structure of XML documents in very readable and understandable form. The structural description can be decomposed to number of logical units. Technology of Xdefinition allows both the validation and construction of XML data. Together with structural description of XML data it can be added also an executive code. The Xdefinition **allows to describe different events and actions to be executed**. Due to possibility to invoke external methods it is possible to provide complex error handling/error recovery and data processing.

Xdefinition makes possible **processing of large amount of XML data**. The size of processed data is practically unlimited. Xdefinition technology can process very large data (size of source XML files can exceed the size of local memory).

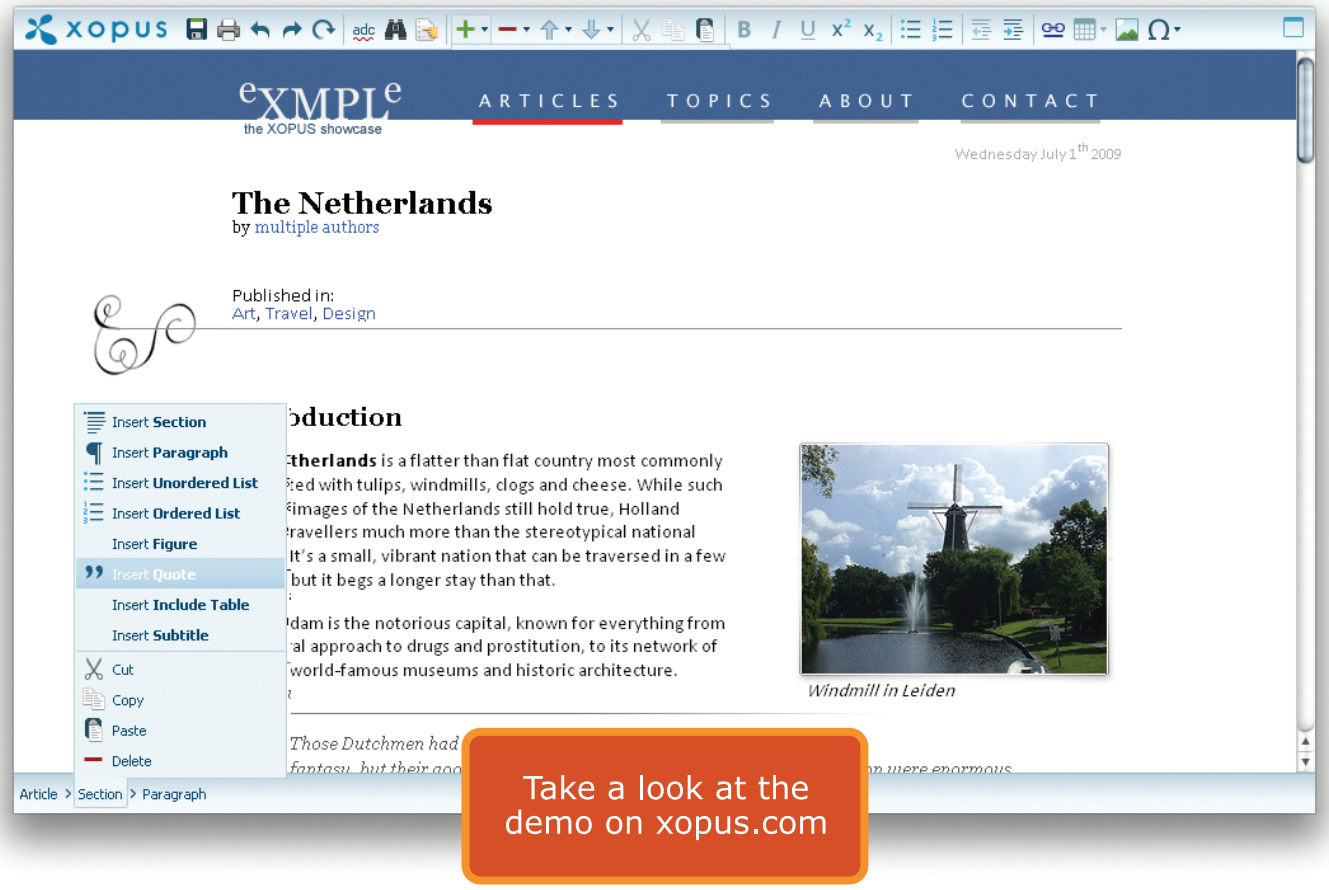
Xdefinition is accessible from JAVA. There are available configurations for different Java versions. It is possible to generate a special version **supporting XPATH 2.0 and XQUERY** (in connection with Saxon library).

Trial version of Xdefinition is available on www.syntea.cz.
E-mail: xdef@syntea.cz

XOPUS The web-based WYSIWYG* XML Editor

Xopus is the software for environments with large groups of authors and a controlled information structure. Authors can get started without training or installation.

Xopus improves productivity, increases the value of knowledge workers' content, reduces the costs for reusing material for multiple media, protects corporate identity and reduces legal and translation costs.



Take a look at the demo on xopus.com

Xopus works in modern browsers and is thereby excellently suited for writing and editing web content, documentation and manuals, educational content and contracts & legal and policy texts.

*WYSIWYG: What You See Is What You Get

TEAM data<2>type

<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 199 USD.
- **<oXygen/> XML Editor**, for developers, containing the complete development environment starting from 64 USD Academic / 349 USD Professional.

Both editions can run as a standalone application or as an Eclipse IDE plugin, on Windows 7, Vista, XP, 2000, Mac OS X, Linux and Solaris.

XML

RECRUITMENT SPECIALISTS

Looking for XML staff?
Mercator IT Solutions needs to be your first point of contact.

Mercator IT Solutions are specialists in providing XML related staff. Having worked with leading organisations, our consultants have many years of experience in finding the right candidates for your roles. We pride ourselves in matching our candidates to clients' technical and soft skill specifications. Our proven track record of developing long-term relationships with both our clients and candidates enables us to work to tight deadlines.

Looking for XML Staff?

MERCATOR I.T. SOLUTIONS

info@mercatorit.com Tel: +44 (0) 1892 611161

For further information please call or email us

Table of Contents

General Information

Date

Saturday, March 13th, 2010
Sunday, March 14th, 2010

Location

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

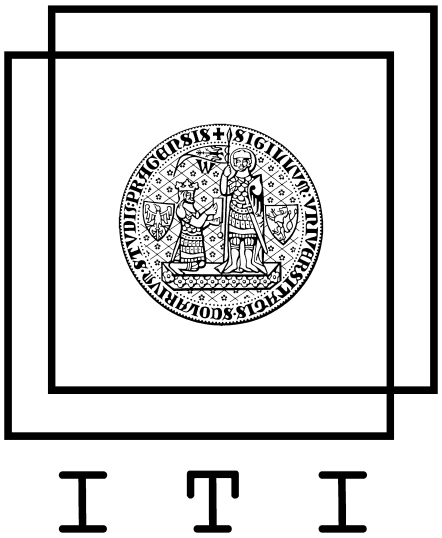
Organizing Committee

Petr Cimprich, *Ubiqway*
James Fuller, *Webcomposite*
Vít Janota
Tomáš Kaiser, *University of West Bohemia, Pilsen*
Jirka Kosek, *xmlguru.cz & University of Economics, Prague*
Pavel Kroh
Petr Pajas, *Charles University, Prague*
Mohamed Zergaoui, *Innovimax*
Martin Žák

Produced By

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

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 2010–2011
- 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`)

Sponsors

Gold Sponsors

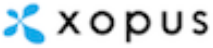
Mark Logic Corporation (http://www.marklogic.com)
The FLWOR Foundation (http://www.flworfound.org)

Silver Sponsors

Xopus (http://xopus.com)
oXygen (http://www.oxygenxml.com)
River Valley Technologies (http://www.river-valley.com)
28msec (http://www.28msec.com)

Bronze Sponsors

data2type GmbH (http://www.data2type.de)
Jalfrezi Software Limited (http://www.jalfrezisoftware.co.uk)
Mercator IT Solutions Ltd (http://www.mercatorit.com)
Syntea software group a.s. (http://syntea.cz)



Preface

This publication contains papers presented at XML Prague 2010.

XML Prague is a conference on XML for developers, markup geeks, information managers, and students. In its 5th year, XML Prague focuses on emerging trends in core XML technologies and their application in the real world, with stress on temporal aspects of XML. The conference provides an overview of successful XML technologies, with the focus being more towards real world application versus theoretical exposition.

XML Prague conference takes place 13–14 March 2010 at the Lesser Town Campus of the Faculty of Mathematics and Physics, Charles University, Prague. XML Prague 2010 is jointly organized by the XML Prague Organizing Committee and by the Institute for Theoretical Computer Science.[1]

The full program of the conference is broadcasted over the Internet (see http://xmlprague.cz) – XML fans from around the world are encouraged to take part on-line. Remote and local participants are visible to each other and all have got a chance to interact with speakers.

This is the fifth year we have organized this event. Information about XML Prague 2005, 2006, 2007 and 2009 was published in ITI Series 2005-254, 2006-294, 2007-353 and 2009-428 (see http://iti.mff.cuni.cz/series/).

— *Petr Cimprich & Mohamed Zergaoui*
XML Prague Organizing Committee

[1]The Institute for Theoretical Computer Science is supported by project 1M0545 of the Czech Ministry of Education.

Papers and invited talks

Streaming in XSLT 2.1

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

The XSL Working Group has been studying how to extend the language to support streaming, that is, the ability to transform source documents into result documents without holding either in memory. This paper describes the current state of the work, as represented in the first Working Draft of XSLT 2.1.

This paper discusses a W3C Working Draft which at the time of writing has not been published, and which is therefore not referenced in the paper. When it is published, it will be announced on the home page for the XSL Working Group at `http://www.w3.org/Style/XSL/`.

1. Introduction

XSLT 1.0 [1] was published in 1999 and proved a great success, with perhaps a dozen implementations for users to choose from, all achieving a high level of interoperability. XSLT 2.0 [2] followed in 2007: take-up has been less rapid, with evidence that vendors need to recover the substantial cost of developing products, but after three years there are now four implementations on the market (Altova[3], IBM[4], Intel[5], and Saxonica[6]) and suggestions of more in the pipeline: and while vendors might have been slow to implement the standard, users have embraced it with enthusiasm because of the greatly increased functionality and usability on offer.

An XSLT stylesheet describes a transformation from one or more source trees to one or more result trees. One of the limitations of all the popular implementations is that they require enough memory to represent the source trees in memory, typically for the entire duration of the transformation. This requirement follows naturally from the language semantics, which describe access to the source tree in terms of the XPath axes such as child, descendant, preceding-sibling and following-sibling: the stylesheet can use these axes to navigate freely around the source tree, which makes it difficult or impossible for a processor to determine when parts of the tree can be discarded. By contrast, result trees are written sequentially (that is, the "natural" order of execution of instructions in the stylesheet is aligned with the order of elements in the result tree), and therefore it is easy for processors to avoid holding the result tree physically in memory.

This has meant that in practice, XSLT processors cannot handle source documents that are larger than the amount of physical memory available. In practice, the amount of memory needed is typically 3 to 10 times the size of the raw lexical XML, because of the need to maintain data structures for efficient navigation. Of course, larger memories help: in 1999 a typical desktop machine might have had 64MiB [1] of physical memory, while today 4GiB is more usual: this increases the limits on XSLT processing from say 15MB source documents to 1GB documents. But however much memory grows, there will still be people whose files are too big. Indeed, some people want to transform infinite data streams, such as a stream of telemetry data coming from a continuous process or from a satellite.

Expanding into virtual memory rather than real memory might be seen as one way forward. But unless a high locality of reference can be achieved, paging into virtual memory causes an order-of-magnitude decline in performance, and in practice the techniques needed to achieve high locality of reference are very similar to those needed to achieve pure streaming.

XPath and XSLT are declarative languages, and optimizers therefore have a great deal of freedom in rearranging the execution of a stylesheet to reduce its resource usage. In theory this should enable them to devise a streamed execution plan (one which avoids reading the whole source document into memory) when this is required. A number of research projects have attempted to achieve this: see for example Dvořáková[7], Dvořáková and Zavoral[8]. Generally the academic work simplifies the problem, for example by allowing only a single input document, and by ignoring the more complex parts of the language such as the `xsl:number` instruction. In the XSL Working Group we do not have this luxury. There has been more work on streaming processors for XPath and XQuery, which are inherently more tractable: a significant difficulty with XSLT is the fact that the rule-based mechanism for dynamic template despatch, from which the language derives so much of its power, is unamenable to static analysis.

Commercial vendors such as Datapower (now part of IBM)[9] and Intel[10]have issued press releases and white papers claiming to perform streamed transformation of XSLT subsets, but have been noticeably reticent in providing detailed technical information to back this up: in particular, no information is published as to the actual subset of XSLT that they handle in streaming mode. Nevertheless, the marketing activity points to a real recognition that there is a user requirement for these features.

My own company, Saxonica, has not been idle either. For several years the commercial version of the Saxon product has supported a simple XSLT extension that allows what one might call "burst-mode" streaming: the idea here is to handle the transformation of a large document as a sequence of transformations of small sub-documents. The full power of the XSLT language is available for processing each sub-document, but these mini-transformations must be independent of each

[1] K = 1000, Ki = 1024, etc.

other, and their results must be output in the sequence that the sub-documents appear in the input. While rather restrictive, this facility has proved useful for a large class of problems. More recently, Saxon-EE 9.2[11] includes prototype implementations of many of the features appearing in the first draft of the XSLT 2.1 specification, discussed in this paper.

The belief that full streaming of XSLT was too difficult led a group of individuals to define an alternative language, STX[12], that adopted many ideas from XSLT but left out all features that had no natural streaming implementation. A key difference from XSLT is that the language is procedural and has a defined order of execution: it uses mutable variables, though in a restrained way. The thinking here is that if you can only visit nodes in the source document once, then you are going to have to remember what you have seen so you can use the information later; the requirement for navigation in arbitrary directions in XPath exists only because of the decision to disallow mutable memory. STX was a significant influence on the XSLT 2.1 specification, with many of the same people participating, but the working group decided not to depart from the the foundations of XSLT as a declarative, functional, stateless language. Instead, as we will discuss later, the requirement to remember data that was seen earlier has been satisfied by the new `xsl:iterate` instruction, which is in essence syntactic sugaring of head-tail recursion.

2. XSLT 2.1: Design Approach

Once the initial decision was made to address the streaming problem as the major focus of XSLT 2.1, the working group embarked on a project to collect and document use cases. Some of these were "made up", some based on real user requirements known to members of the group; some of them were straightforward, others deliberately chosen to be taxing. All had the characteristic that they represented transformations that were intuitively streamable in that the order of information in the output corresponded to the order of information in the input.

These use cases were then implemented as transformations coded in XSLT and STX, which enabled the working group to ask itself: How could a compiler, with only a modest amount of intelligence, devise a streamed execution plan for executing this code?

In some cases it was clear that no optimizer could generate such an execution plan, because it had insufficient information about the nature of the source document. In some cases, the information available in a schema for the source document would help. For example, a schema might reveal that the children of an element always come in a particular order, and that some elements are never recursive - that is, they never contain themselves as descendants. But sometimes the intuitive recognition of streamability relied on other knowledge that cannot even be captured in a schema, for example the knowledge that the entries in a log file are sorted in order of date/time.

The working group never took the view that streaming should be entirely the responsibility of the optimizer. A precedent here is the XSLT `key()` function, which (by a convention that can never be completely mandated in a language specification) gives users a high degree of confidence that they can search for data with `log(n)` performance. The working group aims for interoperability not just at the formal level where different processors produce the same result, but also at the practical level where users can have a reasonable level of confidence that what performs well on one processor is also likely to perform well on others. So the design approach is that if your application depends on streaming, then you should say so, and we should define a subset of the language that is streamable on all processors that claim to support the facility. (This leads to legalistic problems in defining what exactly counts as streaming: but it doesn't actually matter if the definition is a little fuzzy at the edges. The important thing is that if a user asks for streaming, and satisfies all the rules for streamable code, and if the processor then runs out of memory despite claiming in its specification that it supports streaming, then the user can reasonably complain to the vendor and say unfriendly things on Twitter.)

Many of the use cases required streamed processing to be mixed with fully-navigational processing. For example, a large log file might be accessed in streaming mode, and transformed by reference to a small look-up document that can readily be held in memory. For this reason, the working group decided against defining a syntactic subset of the language that authors of streamable stylesheets have to work within. Instead, the rules for streamability apply only to those constructs that access the streamed input document. Nevertheless, the decision was made that this analysis should always be done statically.

Many of the use cases turned out to be soluble in what I have called "burst-streaming" mode, where the transformation essentially takes place a subtree at a time. In the simplest cases, the stylesheet can iterate over the roots of these subtrees, for example by means of a construct such as `<xsl:for-each select="//employee">`, then use the instruction `<xsl:copy-of select="."/>` to materialize the subtree within a local variable, and then perform arbitrary transformations (using full XPath navigation) on the subtree held in this variable before moving on to the next subtree root. The only requirement on the processor here is to support streamed selection of the root nodes; and the only requirement on the language is to define a subset of XPath that allows the root nodes to be selected in streaming mode without excessive contortions on the part of the processor, and without placing too heavy restrictions on the user. In fact, however, XSLT 2.1 does not define a streamable subset of XPath directly; instead, the analysis of which XPath expressions are streamable falls out of the much deeper analysis applied to entire XSLT template rules.

In the following paragraphs I will give an overview of the new language features defined in the draft specification to assist in the writing of streamable code, together with a discussion of the rules that are applied to determine whether constructs in the language (in particular, template rules) are in fact streamable. While detailed

conformance rules have yet to be worked out, the basic principle is that when users ask for the transformation to be streamed, then (a) if they write code that conforms to all the streamability rules in the spec, any processor that claims to be a streaming processor must honour that request, and (b) if they write code that falls outside the guidelines, the processor must tell them at compile time whether it can be streamed or not.

3. Streamed Templates

Consider a common design pattern, the modified identity transform, that consists of an identity template to copy most elements unchanged, plus an overriding template rule to apply some modification to selected elements. For example, here is one that deletes all the `NOTE` elements from the source document:

```
<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

<xsl:template match="NOTE"/>
```

Intuitively, this is streamable, because the sequence of events coming from the parser of the source document is isomorphic with the sequence of events going to the serializer of the result document. But how do we know it is streamable, especially if the code is a small part of a stylesheet that also does many other things?

The Working Group made several decisions. Firstly, users should say that they want this to be streamed, it shouldn't be left to processors to consider this strategy among many others. Secondly, if they want to be certain it can be streamed, they should have to follow a set of rules defined in the language specification. Finally, the request to make it streamed should not be at the level of an individual template rule (too fine grained), nor at the level of the stylesheet (too coarse), but at the level of a mode.

Modes exist in XSLT 1.0 and 2.0, but not really as first class objects: they exist merely as labels attached to template rules to indicate whether they are candidates for matching by a particular apply-templates call. This decision elevates the status of modes to first-class named objects with their own properties, one of which is streamability. The assumption is that when processing a particular document during a particular phase of the processing pipeline, a family of template rules will be used, and these will all share the same mode: a mode now becomes a set of template rules, rather than merely a label attached to them. So we introduce:

```
<xsl:mode name="m" streamable="yes"/>
```

to say that the mode `m` is streamable, or

```
<xsl:mode streamable="yes"/>
```

to say that the default (unnamed) mode is streamable. The introduction of this element also gives us the opportunity to define other processing options at this level, for example whether an ambiguous rule match should be treated as an error.

The rules that a template must satisfy to guarantee streamability are essentially these:

1. No navigation from the context item is allowed other than (a) to its attributes, its ancestors, and their attributes, or (b) to its descendants.
2. The descendants of the context item can be visited at most once, and in document order.

The first rule carries an assumption that a streaming processor will retain in memory the ancestors of the node at the current parsing position, together with their attributes. In fact, the specification also implies that for each of these ancestors, the processor will also retain summary information about the preceding siblings of the element, specifically a count of preceding siblings broken down by node kind and name. This is sufficient information to support matching of nodes using patterns such as `para[1]`, and simple numbering using `xsl:number`.

The simple phrase *navigation from the context item* accounts for most of the complexity in the specification. To establish what navigation routes are followed from the context item, it is necessary to do static analysis to establish which expressions set the context item used by other expressions. For example, the following template is not streamable:

```
<xsl:template match="para">
  <xsl:for-each select="ancestor::section">
    <heading><xsl:value-of select="head"/></heading>
  </xsl:for-each>
</xsl:template>
```

This is because there is a navigation path from the `para` element to its ancestors, and thence to their `head` children: navigation may go upwards or downwards from the context node, but it cannot go up and then down.

The analysis used to establish streamability is heavily influenced by the analysis done by Marian and Simeon[13] as the basis of document projection, which determines the parts of a document reachable by a query so that the unreachable parts can be discarded during parsing without affecting the results of the query. However, there are differences. In particular, Marian and Simeon are not concerned with how

often the descendants of a node are visited, or whether they are visited in the right order. Also, their analysis is of XQuery, which is much more amenable to static analysis than XSLT.

The navigation performed by a template needs to be considered in conjunction with the navigation performed by the templates that it invokes. Since we cannot know statically which templates will be activated as a result of an `xsl:apply-templates` instruction, we have to make worst-case assumptions. One thing we do know is the mode. If the `xsl:apply-templates` instruction specifies a non-streamable mode (and selects nodes in the streamed input document), then there are no constraints on what the called template does, so streamability cannot be guaranteed - the analysis fails. Conversely, if the mode is streamable, then we know that the called template is only allowed to navigate within its subtree.

Now we encounter a tricky problem. If two navigation steps A and B both select downwards in the tree, we know that when we combine the two steps by way of a nested loop, we will always remain within the subtree of the original context node; but we don't know that the nodes will be accessed in document order. For example, if the first step is `descendant::section` and the second is `child::number`, then a glance at the following structure shows that the result of the expression `<xsl:for-each select=".//section"><xsl:value-of select="number"/></xsl:for-each>` is ("1", "1.1", "1.2"), which cannot be derived by visiting the input nodes in document order.

```
<section>
  <section>
    <number>1.1</number>
  </section>
  <section>
    <number>1.2</number>
  </section>
  <number>1</number>
</section>
```

This problem is serious because it means that many problems that are intuitively streamable with knowledge of the input data cannot be proved to be streamable by static analysis: they are defeated by pathological cases like the one in this example.

The current XSLT 2.1 working draft takes a conservative approach and states that a path is non-streamable if it contains a descendant step followed by any other downward step. It is recognized that this is very restrictive. Other possibile rules being considered include:

1. Allowing such a sequence of steps if they form part of a single path expression, so we know the results will be delivered in document order

2. Making the optimistic assumption that the results of the nested loop will actually be in document order, and failing at run-time if they are not
3. Analyzing the construct statically as streamable, and requiring the processor to perform buffering or look-ahead it it finds at run-time that the results are not actually in document order.

The current Saxon implementation adopts the last of these strategies. In the example cited, the result of processing the two nested `section` elements will be held in memory until the result of processing the outer `section` is complete. The input will thus be processed in a single pass, but there will be some buffering of output results. This requires an unpredictable amount of memory, so under some definitions it is not pure streaming; but this seems a more user-friendly solution than rejecting the stylesheet at compile-time based on pessimistic assumptions.

In order to make the static analysis feasible, the specification does not allow nodes from the streamed input document to be passed as parameters to other templates. It does however allow them to be passed to stylesheet functions, since such calls can be analyzed statically. It does not allow streamed nodes to be returned as the result of a template, or to be bound to global variables, but they can be bound to local variables, since the references to such variables can again be analyzed.

There are many complications in the path analysis rules, and there would be little benefit in describing them all in this paper. Many of the rules are designed to handle all the various quirks of the XSLT and XPath specifications (instructions such as `xsl:number` and functions such as `last()`). The rule that descendants can only be visited once means that conditional expressions need to be analyzed (each branch is allowed to access the descendants, since they are mutually exclusive), as does any iteration over anything other than nodes in the streamed input (`for $i in 1 to 10 return .//x`).

4. The `xsl:stream` instruction

The working draft introduces a new instruction, `xsl:stream`, whose effect is to read an external document and process it in streaming mode using the contained child instructions. For example, the following code computes the average salary of the employees in a document:

```
<xsl:stream href="{$employees-doc}">
  <xsl:value-of select="avg(employees/employee/salary)"
</xsl:stream>
```

Like the contents of a template in a streamable mode, the instructions within `xsl:stream` are analyzed for streamability. This example satisfies the rules because its only navigation is a sequence of downwards navigation steps. It could, of course, contain a call on `xsl:apply-templates` in a streaming mode.

Very often, transformations will only have a single streamed input document, and the transformation can be initiated by supplying this as the input, and with a streaming mode as the initial processing mode. The `xsl:stream` instruction is needed when more than one input document needs to be processed in streaming mode.

5. The `xsl:merge` instruction

A number of the use cases considered by the working group involved merging multiple input file: for example, merging the log files from a number of web servers. This process typically depends on knowledge that all the files are sorted by comparable key values (in this case a date/time stamp). This use case is sufficiently specialized that the working group came to the conclusion it needed custom syntax to support it: the potential performance advantage of a strategy that uses knowledge of the input sort order is too great to ignore.

So an `xsl:merge` instruction has been added to the language. This identifies the input sequences to be merged. It describes the sort keys for the input files using a declaration similar to `xsl:sort`, but unlike `xsl:sort` this does not cause the sequence to be reordered, it merely checks the order and signals an error if the input is not already properly ordered.

The instruction can handle both homogenous and heterogenous input files, for example a master file in one format and an indefinite number of transaction files in a different format.

As with the `xsl:for-each-group` instruction, `xsl:merge` invokes user-supplied code (the `xsl:merge-action`) to handle a group of input nodes with the same merge keys. The duplicates may come from different input sequences, or from the same input. In fact, `xsl:for-each-group` with `group-adjacent` can be seen as a special case of merging with a single input sequence.

6. The `xsl:iterate` instruction

Some of the hardest use cases to tackle in streaming mode are those where the processing of one input node depends in some way on the processing of previous nodes. Classically, XSLT stylesheets will handle this by navigating back to the earlier nodes each time their value is required.

Some common cases are handled by retaining the attributes of ancestors in the source tree. This doesn't work, however, if the source document uses child elements instead of attributes. For example, if processing an XHTML `<p>` element as the input, it isn't possible to navigate backwards to see if the most recent heading was an `<h1>` element.

Other use cases are traditionally handled using sibling recursion. For example, given a sequence of transactions on a bank account, the problem of adding the running balance to the transaction elements is typically handled in XSLT by a re-

cursive function or template: this makes a recursive call to process each successive transaction element, passing the current balance as a parameter. We decided that it was not reasonable to expect a processor to translate such a recursive program into a streaming execution plan, even though it might be theoretically possible. In any case, experience with the user community shows that many XSLT users have great difficulty writing and debugging recursive code to handle tasks that they think of more naturally in terms of iteration.

So we devised the `xsl:iterate` instruction. To the user this feels very much like `xsl:for-each`: it iterates over an input sequence. The difference is that it really does iterate, whereas `xsl:for-each` is technically a functional mapping expression in which the same processing is applied to each item in the input sequence independently. The processing performed by `xsl:iterate` is guaranteed to process the sequence in order, and at the end of processing one item, it can set parameters which are then visible to the next iteration. Because the processing is sequential, it is possible to break out of the loop at any time. The construct has the familiar usability of procedural programming, while having formal semantics that are entirely declarative, and that can be defined directly in terms of head-tail recursion.

Here's the code to handle the "cumulative balance" problem:

```
<xsl:iterate select="transaction">
  <xsl:param name="running-balance" select="0"/>
  <xsl:variable name="new-balance" select="$running-balance + @amount"/>
  <transaction amount="{@amount}" balance="{$new-balance}"/>
  <xsl:next-iteration>
    <xsl:with-param name="running-balance" select="$new-balance"/>
  </xsl:next-iteration>
</xsl:iterate>
```

So `xsl:iterate` has benefits quite independent of streaming; but it was also designed quite deliberately to provide a way of expressing solutions to problems like this one that are relatively straightforward to translate into a streaming execution plan - no harder, at any rate, than a simple `xsl:for-each` loop.

7. The `xsl:fork` instruction

Some of the use cases that caused the most difficulty were those that required delivering multiple results from a single pass of the input data. Two examples:

1. Given an input document containing `employee` elements, copy the full-time employees to one output document, and the part-time employees to another.
2. Delete all the `NOTE` elements in the document, and count how many there were.

These are clearly streamable in principle, at least in the sense that they can be done with a single pass over the input (they may require some buffering of output). But

when they are expressed in XSLT code in the normal way (that is, with multiple passes over the input), one can see that implementing the transformation with a streamed execution plan would be an extreme challenge for an optimizer.

After many false starts, the solution adopted in the draft specification is a new instruction, `xsl:fork`. This has multiple child instructions which conceptually are evaluated in parallel, with the results being assembled in sequence on completion. The requirement that only one downward selection is allowed is removed if the selections occur in different branches of an `xsl:fork` instruction. So the user continues to write the stylesheet as if it is scanning the input more than once.

A possible implementation might be to fire off one thread for each independent child of the `xsl:fork` instruction, and then to send copies of the parsing events for the input document to each ot these threads independently. A parallel implementation is not the only one possible; though as multi-core processors become more and more common, it becomes increasingly attractive.

8. Conclusions

Streamed processing of XSLT has always been a goal, and one of the aims of making the language design purely declarative was to ensure that compilers had maximum freedom to devise optimum execution strategies. However, experience over ten years has shown that the goal remains elusive, and that researchers are still struggling to identify streamable subsets.

Rather than rely on innovation in compiler technology, the XSL Working Group has taken a pragmatic approach, enhancing the language with features that can be used to write code in a way that can be streamed without excessive internal rewriting.

The most complex part of the approach is the path analysis, which examines the code of templates to determine whether the navigation they perform is consistent with a streamed implementation.

The work is still at an early draft stage, and all input is welcome.

References

[1] James Clark. *XSL Transformations (XSLT) Version 1.0. W3C Recommendation.* 16 November 1999. W3C. `http://www.w3.org/TR/xslt`

[2] Michael Kay. *XSL Transformations (XSLT) Version 1.0. W3C Recommendation.* 23 January 2007. W3C. `http://www.w3.org/TR/xslt20`

[3] *AltovaXML.* `http://www.altova.com/altovaxml.html`

[4] *IBM Websphere Application Server Feature Pack for XML.* `http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/xml/features/index.html`

[5] *Intel SOA Expressway XSLT 2.0 Processor*. http://software.intel.com/en-us/articles/intel-soa-expressway-xslt-20-processor/

[6] Michael H. Kay. *Saxon: The XSLT and XQuery Processor*. http://saxon.sf.net/

[7] Jana Dvořáková. *Automatic Streaming Processing of XSLT Transformations Based on Tree Transducers*. 2008. *Informatica*. 32. 373-382. Comenius University. http://www.informatica.si/PDF/32-4/13_Dvorakova%20-%20Automatic%20Streaming%20Processing%20of%20XSLT...pdf

[8] Jana Dvořáková and Filip Zavoral. *Using Input Buffers for Streaming XSLT Processing*. Charles University in Prague, Czech Republic. http://www.ksi.mff.cuni.cz/xord/papers/db2009paper.pdf

[9] *DataPower Streaming XML Processing Breakthrough for XML Documents of Unlimited Size*. Datapower. 2005-03-29. ftp://ftp.software.ibm.com/software/websphere/integration/datapower/news/pr_032505_xpath.pdf

[10] Intel. *Boosting XSLT Transformation Performance with Intel XSLT Accelerator 1.1*. 2007. http://cache-www.intel.com/cd/00/00/34/42/344227_344227.pdf

[11] Saxonica. *Streaming of Large Documents*. 2008. http://www.saxonica.com/documentation/sourcedocs/serial.html

[12] *Streaming Transformations for XML (STX)*. http://stx.sourceforge.net/

[13] Amelie Marian and Jerome Simeon. *Projecting XML Documents*. VLDB. 29. Berlin, Germany. 2003.

What XSL 2.0 means for implementers and users

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

Abstract

The block diagram for XSL 1.0 and XSL 1.1 was fairly straightforward: source XML transformed to FO markup then creating pages. The official block diagram for XSL 2.0 doesn't yet exist, but indications are that it will be closer to an octopus than to the arrowed straight line of the current diagram.

The requirements for XSL 2.0 includes features such as animations, feedback from the pagination stage, formatting of document collections, variable-sized regions, and positioning objects relative to each other.

A recitation of the features selected for XSL 2.0 may roll over you with the occasional spark of interest for some of the features, but an implementer has to consider the full gamut of features when designing a formatter -- irrespective of whether the features will be in the first releases, they all still have to be considered. This paper discusses the changes that will have to take place under the hood of any XSL formatter that supports XSL 2.0 and what those additional capabilities can bring to your stylesheets.

1. Overview

The block diagram for XSL 1.0 and XSL 1.1 [9] was fairly straightforward: source XML transformed to FO markup then creating pages.

Even the summary diagram for what goes on inside the formatting stage is comparatively straightforward: the FO markup becomes a tree of formatting objects that have properties; refinement expands shorthands, resolves expressions, and handles inheritance so the properties map into trait values that can be used with areas; then the generated area tree expresses the paginated result.

The official block diagram for XSL 2.0 doesn't yet exist, but indications are that it will be closer to an octopus (or even to a Klein Bottle [4]) than to the straight lines of the current diagram.

The requirements for XSL 2.0 includes features such as animations, feedback from the pagination stage, formatting of document collections, variable-sized regions, and positioning objects relative to each other.

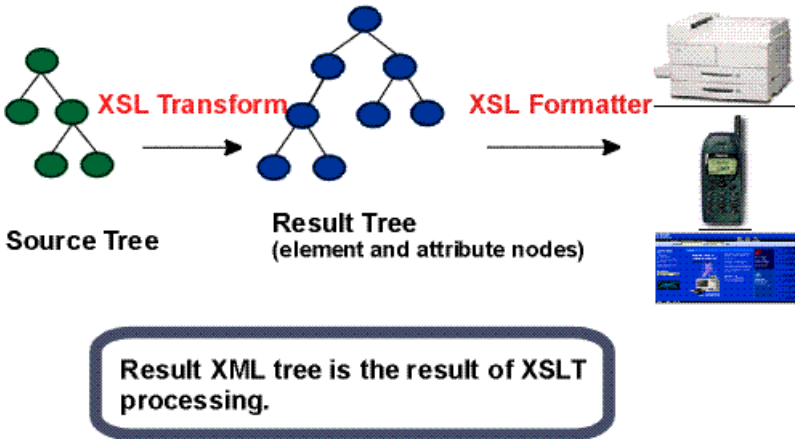


Figure 1. XSL 1.1 block diagram

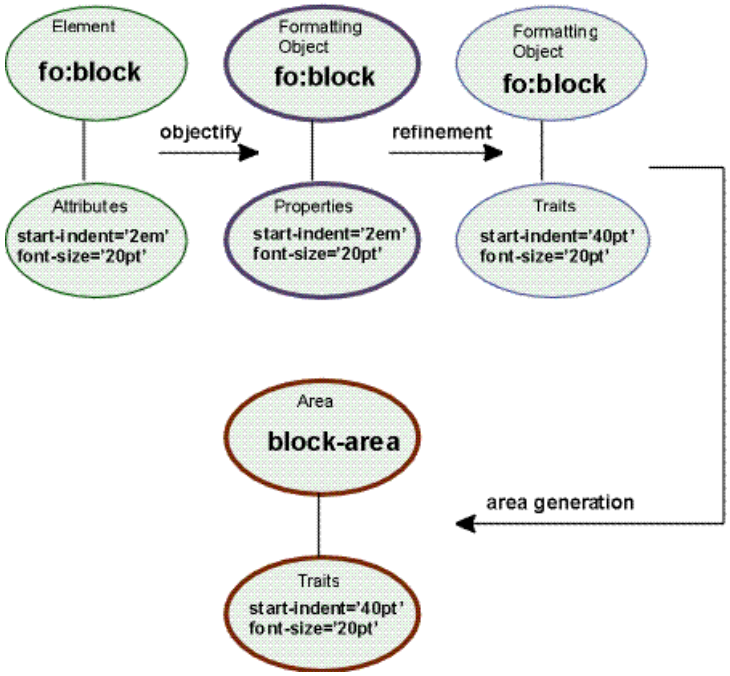


Figure 2. XSL 1.1 formatting process summary

A recitation of the features selected for XSL 2.0 may roll over you with the occasional spark of interest for some of the features, but an implementer has to consider the full gamut of features when designing a formatter — irrespective of whether the features will be in the first releases, they all still have to be considered.

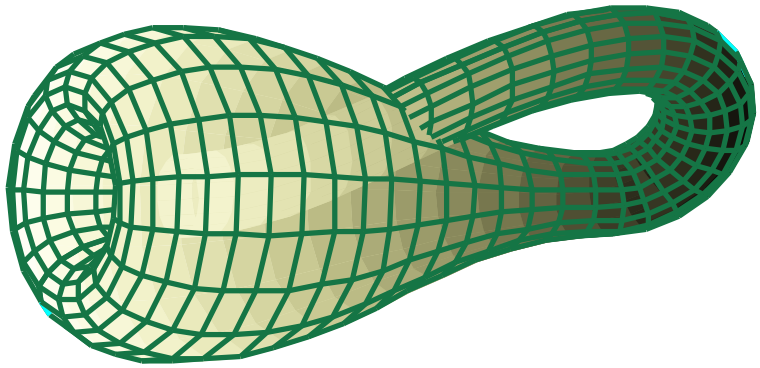


Figure 3. Klein bottle

1.1. Disclaimer

At the time of this writing in January 2010, the first public Working Draft [10] of XSL 2.0 has been published, and a second Working Draft is nearing publication. The W3C XSL FO Subgroup (SG) is actively working on XSL 2.0, and there may be another Working Draft published around the time of XML Prague 2010. The status of the Working Draft (and of all W3C working drafts) is that it "is a draft document and may be updated, replaced or obsoleted by other documents at any time." You've been warned!

This paper discusses some aspects of the XSL 2.0 that have yet to be decided upon by the XSL FO SG, and in some cases have yet to be considered by the SG. Everything that is not in a published working draft is the personal opinion of the author and, as noted above, even when it is in a published working draft, it should not be considered stable.

2. XSL 1.1

2.1. Time line

An associated stylesheet language was part of the original vision for XML, similarly to how DSSSL [2] is associated with SGML. The initial, informal proposal by Jon Bosak, who referred to it as "XML Part 3" or "XS" [7], was based on a subset of DSSSL known as "dsssl-o". The August 1997 proposal to the W3C for a stylesheet language, by then known as XSL [5], incorporated aspects of DSSSL and CSS but used XML syntax. The W3C XSL Working Group was formed in January 1998. The previously monolithic XSL working drafts were split into separate transformation and formatting working drafts in April 1999. XSL 1.0, the first XSL FO specification, was published in October 2001, and XSL 1.1, which added additional formatting objects for indexes, change bars, and other features and which incorporated the XSL 1.0 errata, was published in December 2006.

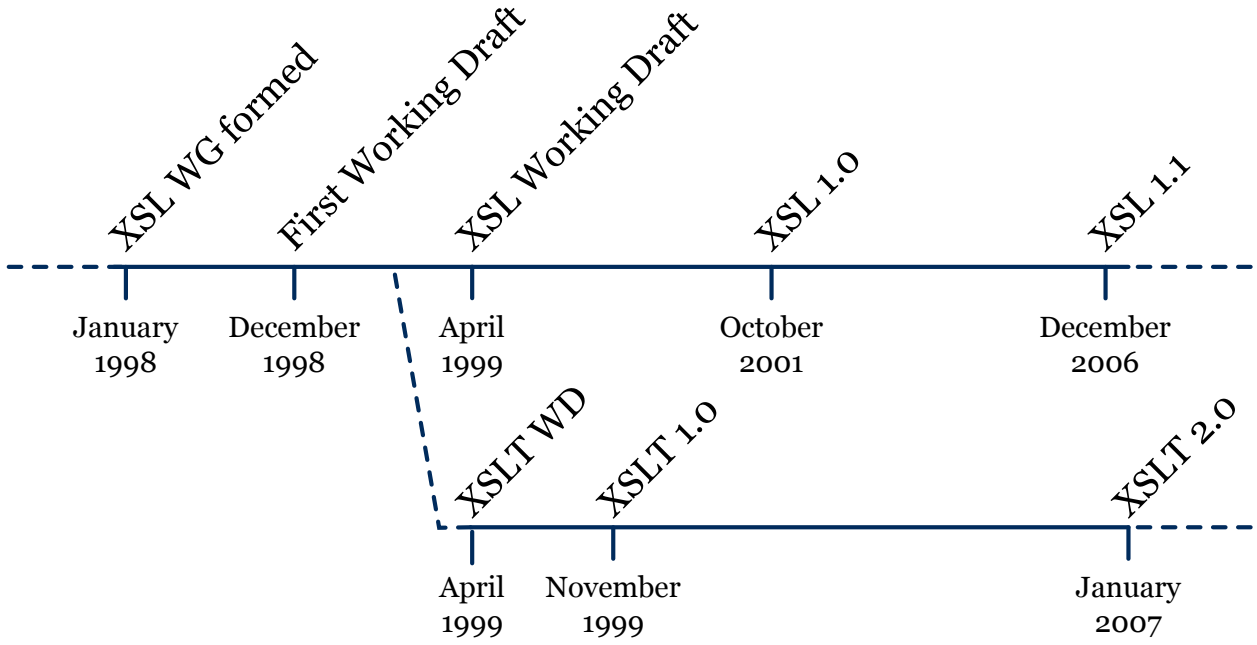


Figure 4. XSL 1.1 time line

2.2. XSL 1.1 Processing Model

The straight-through, one-pass model for formatting XSL follows from the similar processing model in DSSSL [2]. The absence of feedback from the pagination stage isn't because that's all that could be done back in the Dark Ages of computing: it was a deliberate decision to keep things simple so there could be more interoperable implementations. The strategy worked — for example, FO support was added to the existing Arbortext Epic Editor — but it has proven to not be sufficient for all users' needs.

2.2.1. Conceptual procedure

Conceptually, a formatting object (FO) typically creates areas and returns them to its parent to be placed in the area tree. Some FOs don't generate areas of their own and just return, or arrange and return, the areas produced by their child FOs. Other FOs, such as fo:page-sequence, may continue generating areas as long as its children return new areas.

2.2.2. Accessing properties in the FO tree

The FO tree could be generated (using XSLT) such that every non-default property value is present on every FO where it will be needed. In practice, this doesn't happen, since there are multiple mechanisms for accessing properties from “higher up” (or elsewhere) in the FO tree:

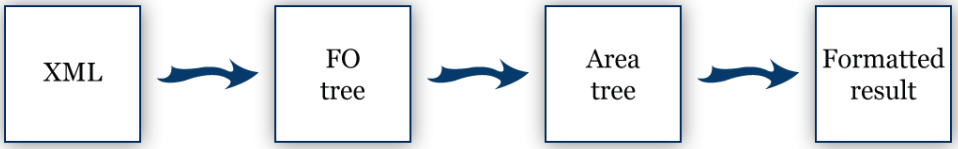


Figure 5. XSL 1.1 block diagram

- Inherited properties — Many properties are inherited, such that the value defined on one FO applies to all its descendant FOs (unless they also specify the property). Since any property can be declared on any FO, inherited properties can be declared on the fo:root, even when they don't apply to fo:root, and their values will be used throughout the document.
- "inherit" keyword — Even on non-inherited properties, using the "inherit" keyword as the property value resolves to using the corresponding property value from the parent FO.
- "from-parent()" — Unlike the "inherit" keyword, this can be used as part of an expression so that you could compute a new value based on the parent FO's value. It's also possible to specify a different property name, so you can base one property on another property's value.
- "from-nearest-specified-value()" — This is similar to from-parent() but uses the value from the closest ancestor FO where the property is defined.
- "from-page-master-region()" — New with XSL 1.1, this allows accessing the "writing-mode" and "reference-orientation" properties from the FO being used to generate the current region. XSL 2.0 may extend this to more properties since it has requirements for, for example, retrieving the "font-size" property for use when formatting footnotes.
- "from-table-column()" — Used only within tables for retrieving property values from column definitions.

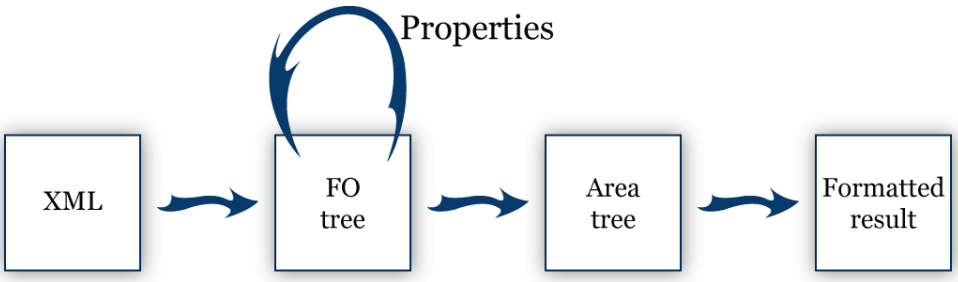


Figure 6. Accessing properties in the FO tree

2.2.3. Percentages of current width, height

The one-way model of FOs producing areas in isolation breaks down when an FO's property is expressed as a percentage of the available width or height (more properly, as a percentage of the inline-progression-dimension or block-progression-dimension of the containing reference area). So there is obviously and necessarily feedback from the area tree to the FOs in the FO tree.

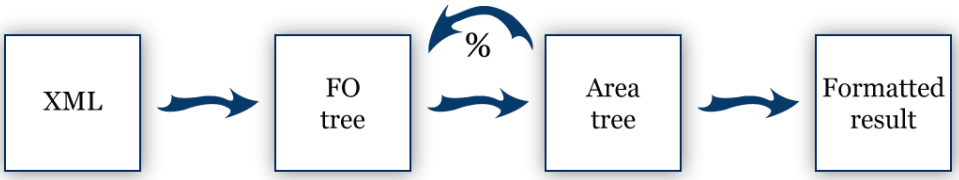


Figure 7. Accessing area tree for percentages

2.2.4. XSL Formatters' Extensions to the Processing Model

Although it is not part of the XSL 1.1 Recommendation, many XSL 1.1 formatters provide access to, or can output a representation of, the area tree — i.e., to structured information about the layout of the finished pages — after the document has been paginated. Some users massage the area tree to achieve effects currently not possible with XSL 1.1 or use information in the area tree to modify the source XML for a second pass of XSL formatting. One example from private email to the author:

> *We've been using our XSL:FO output for a couple of years now and it works really well. It is based around XEP and makes use of the intermediate format along with some XSLT analysis of that to iterate the output until the algorithm is happy with the layout. So it kind of makes up for a lack of some of the page layout stuff that will hopefully be in XSL:FO 2.0 by iterating round the process. Not ideal but I think we pushed XSL:FO 1.x pretty hard!*

Other users have worked around limitations in XSL 1.0 and XSL 1.1 by extracting information from the PDF output by the XSL formatter and using that data to modify the input. For example, Crane Softwrights has a technique for creating back of book indexes [1] using XSL 1.0 and data extracted from formatted PDF files. (Note that index-related FOs were added in XSL 1.1.)

As another example, the Antenna House formatter supports two-pass processing for handling long documents.

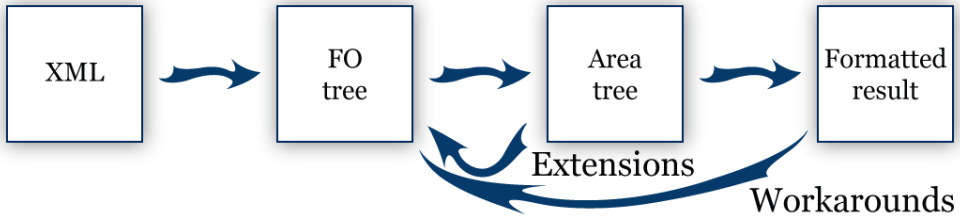


Figure 8. Vendor extensions and workarounds

3. XSL 2.0

XSL 2.0 work kicked off with a requirements workshop in Heidelberg, Germany, in October 2006 following the W3C Print Symposium (and shortly before XSL 1.1 was finalised), but in reality, many of the requirements — such as drop caps, text along a curve, marginalia, masks, transparency, and maths — were carried over from the requirements for XSL 1.0 [8].

The XSL 2.0 requirements [11] were published in March 2008, and the first public working draft of XSL 2.0 [10] was published in September 2009.

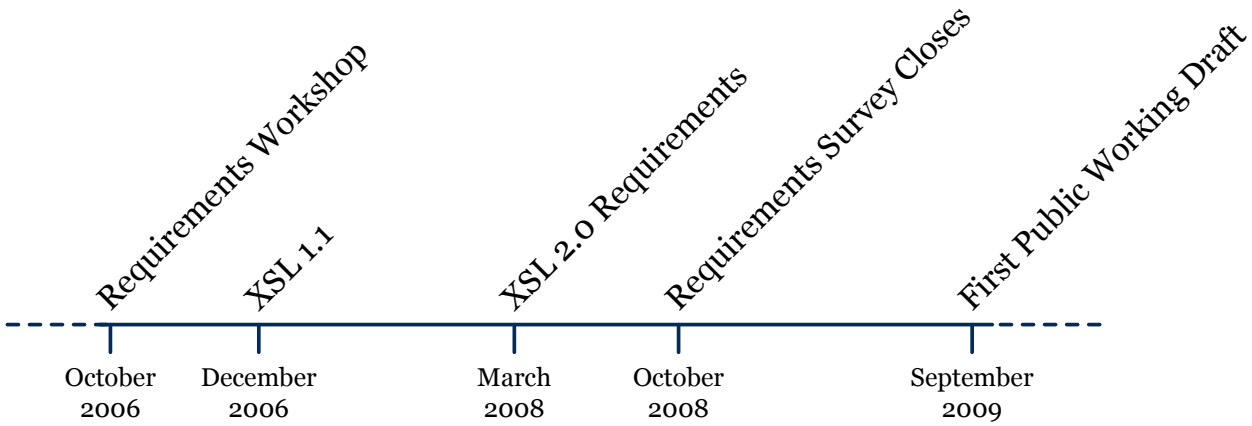


Figure 9. XSL 2.0 time line (so far)

3.1. Highlights

Highlights of the XSL 2.0 requirements include:

- Non-rectangular areas
- Columns
- Footnotes
- Floats
- Marginalia
- Vertical positioning

- Tables and lists
- XSL FO inside other languages
- Improved non-Western language support
- Numbering
- Cross-references
- Markers
- Text before/after break
- Columns
- Spreads
- Bleeds
- Feedback from pagination stage
- Document collections
- Fonts
- Hanging punctuation
- Tabs and tab stops
- Rotated images
- Color/Colour
- Metadata
- Schema for XSL FO
- Closer collaboration with SVG

  Some of these items are explained in more detail in the following sections.

3.1.1. Non-rectangular areas

With the new fo:page-master formatting object, regions may have an irregular shape defined using SVG. The z-index of the region determines its priority with respect to other regions.

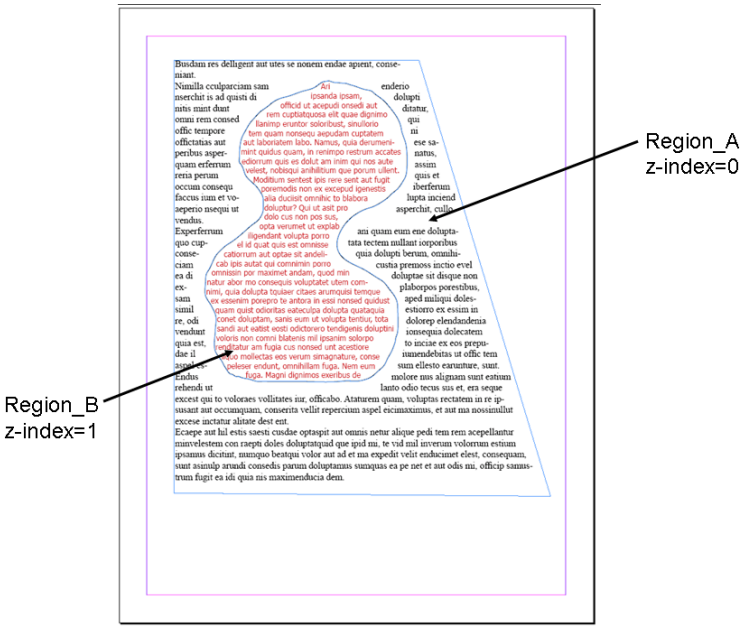


Figure 10. fo:page-master and irregularly shaped regions

The complete region definition may include SVG shapes for: background image; shape of the region's content; the borders; and the shape to be used when handling overlap with other regions.

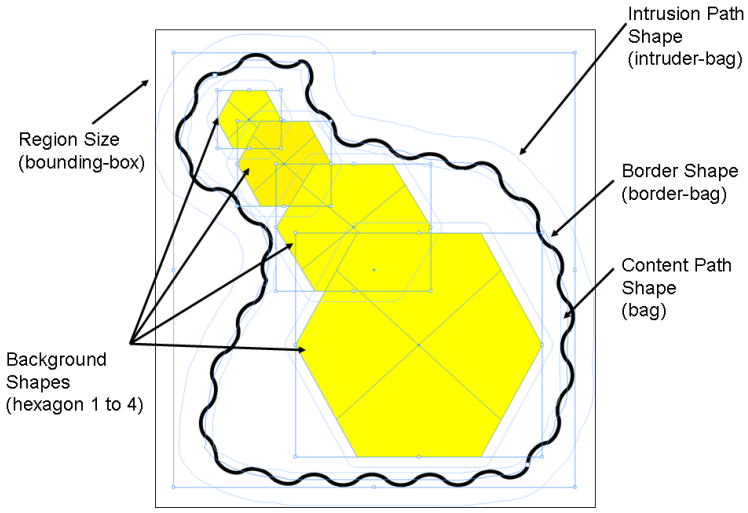


Figure 11. FO inside SVG inside FO

Regions are required to be able to be placed relative to each other (instead of only being absolutely positioned on the page as in XSL 1.1) and also to be able to grow or shrink to fit their content.

3.1.2. Closer collaboration with SVG

Similarities between SVG and XSL FO have been recognised by both groups: the diagram in the "Collaboration with SVG" section of the XSL 2.0 requirements originated in SVG Print 1.2 [6]. Many XSL formatters already support SVG as a graphic format with fo:instream-foreign-object and fo:external-graphic elements, and many also support SVG as an output format, but there is no interplay between the two formats.

Opportunities for closer (less "foreign") collaboration with SVG could include:

- Being able to embed XSL FO inside SVG (or other languages), e.g., to put a fo:block into a SVG object.

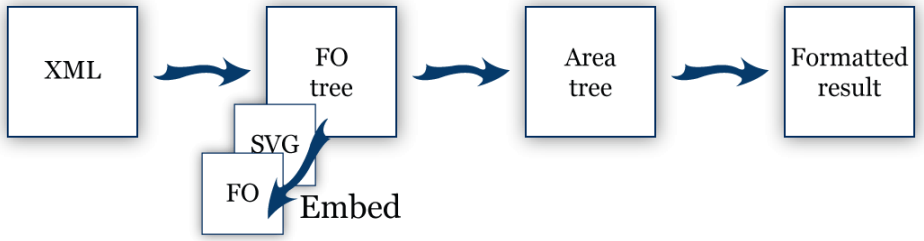


Figure 12. FO inside SVG inside FO

- Defining how to pass property values from FO to SVG (and SVG to FO) so, for example, an embedded SVG diagram uses the same font as the main text
- Using SVG for the shapes of irregularly shaped regions, masks, etc.
- Sharing a common colour model between FO and SVG
- Using SVG for stylised borders and rounded corners on regions
- Applying SVG transformations (shear, rotate, etc.) to XSL areas

3.1.3. Document collections

There is a requirement for "master indexes" and other types of document that summarise or refer to the formatted results of other processing runs, e.g., to get the page numbers for index hits in multiple other documents. The solution was not in the first Working Draft, but it will likely require a mechanism for recording or extracting IDs and page numbers, etc., from a formatter run as well as a convention for looking up the IDs to get their associated data so that it can be reused in the current formatter run.

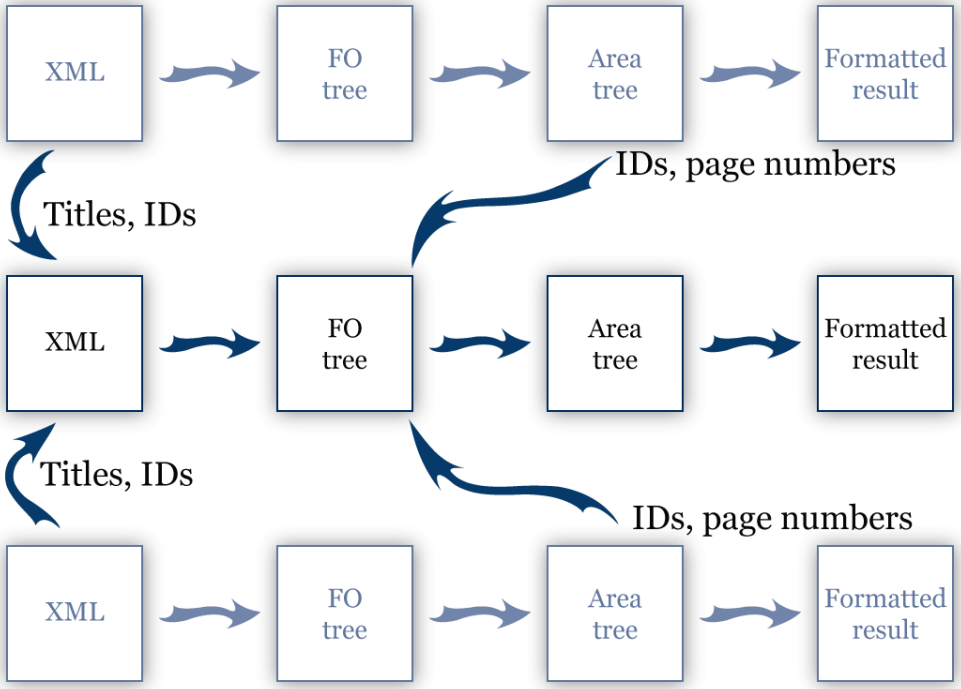


Figure 13. Multiple documents

3.1.4. Interleaving layout masters and producing multiple results

XSL 1.1 requires the definitions of the page geometries — the "layout masters" — to be present in the FO markup before the fo:flow FOs containing the content to be formatted. This ordinarily is convenient, but when, for example, the FO markup is produced from a database report and the formatted result runs to millions of pages, having to know all the layout masters in advance isn't such a convenience. Two requirements address this and similar problems: it will be possible to interleave layout masters and content (provided the layout masters are defined before they are used), and it will be possible for one formatter run to write to multiple output documents.

3.1.5. Improved non-Western language support

XSL 1.1 is good at the "big picture" of internationalisation since it supports multiple writing modes and text directions, but it does not have script-specific or layout-tradition specific formatting objects. Some XSL formatters already do what they can with layout of languages such as Arabic, Hebrew, Japanese, Chinese, Korean, and Thai and supporting hyphenation of many languages in addition to English, but current implementations similarly do not have script-specific extension formatting objects.

The corresponding emphasis in XSL 2.0 is on the "small picture" of the specific needs of at least some non-Western scripts and layout traditions. The first non-

Western script to have specific support is Japanese, since the W3C Japanese Layout Task Force (comprising members of four W3C Working Groups and some Japanese experts) recently produced the "Requirements for Japanese Text Layout" Working Group Note [3]. Japanese support is, or should be, just the start, and the XSL FO SG would be happy to hear from anyone willing to work on XSL support for their script.

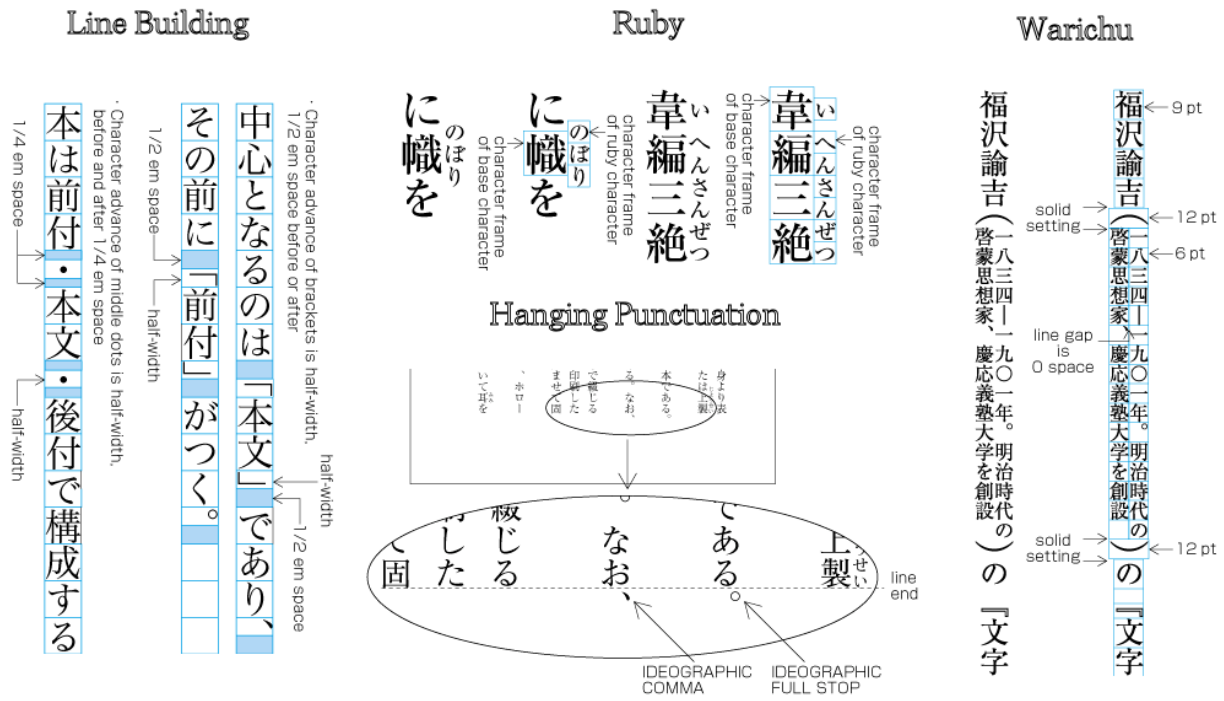


Figure 14. Japanese layout examples

3.1.6. Feedback from pagination stage

There are multiple requirements involving feedback from the pagination stage or inheritance from the containing area or region, including: footnotes inheriting font size from the region; numbering lines on a page; restarting footnote numbering on each page and using the correct markers in footnote references; and calculating subtotals for numbers on a page.

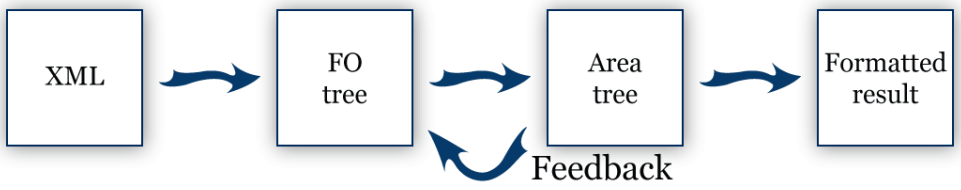


Figure 15. Feedback from the pagination stage

3.1.7. Schema for XSL FO

XSL FO presents a challenge to any schema or validation technology since, for example: properties may appear on any formatting object; properties may inherit (either by design or by using the "inherit" keyword or one of the property-related functions); and there are irregular rules about which FOs may not contain certain other FOs (either directly or as a descendant) and other rules about which FOs may not have certain other FOs as an ancestor.

A XSL FO SG member is expecting to produce a schema for XSL 2.0 when it becomes a Recommendation. We wish him luck!

3.2. Effect on XSL Formatters

The many, diverse requirements for XSL 2.0 [11] will affect any XSL 1.1 formatter as it is reengineered to handle XSL 2.0:

- The closer integration with SVG — ranging from the addition of SVG-like effects such as masks, transparency, rotations, and transformations through to common colour support and the use of SVG to define irregularly-shaped page regions and possible SVG support for embedded FOs — pushes the XSL formatter towards using a SVG DOM for the area tree.
- The requirements for feedback from the pagination stage, for placing objects relative to each other, and for sizing objects to fit their content go far beyond the capabilities implied by the one-way arrow in the XSL 1.1 block diagram and are a further incentive to use an active DOM rather than passive in-memory structures for expressing the area tree.
- The recent Japanese Layout Task Force initiative by the W3C and its effect on Japanese support in relevant W3C specifications will raise the bar on people's expectations of localised output from XSL. Improved Japanese support could be only the beginning of improved non-Western script support, but adding a new script requires work from experts in that script.
- The requirements for handling document collections and for producing master indexes from multiple documents will require XSL formatters to produce intermediate data that can be consumed in a later formatting run.The ability to interleave page layout masters between the FOs for the content of sequences of pages will make it easier to produce a streaming XSL formatter that can be coupled to a streaming XSLT processor to produce documents of enormous length (provided the capability of feedback from the processing stage is used only within limited scopes).

3.3. Issues affecting compatibility between XSL 1.1 and XSL 2.0

There is no requirement for how XSL 1.1 formatters should handle XSL 2.0 documents, but not surprisingly, there is a requirements that XSL 2.0 remains compatible with XSL 1.1 such that "Existing XSL-FO documents should still be supported unchanged by XSL 2.0 processors.". It remains to be seen whether compatibility is possible in light of the following potential issues.

3.3.1. No version indicator

XSL 1.1 (and XSL 1.0 before it) does not have a mechanism to indicate the version of the FO markup being submitted to a XSL formatter. This hasn't been a problem for XSL 1.0, when it was the only XSL version, or for XSL 1.1, since it extended XSL 1.0 with minimal changes to existing functionality.

This won't be a problem for XSL 2.0 formatters since XSL 2.0 is extremely unlikely to drop any XSL 1.1 formatting objects or properties.

This will be a problem for XSL 1.1 processors encountering XSL 2.0 markup, since they will generate errors for the new FOs and properties. In the absence of a version indication, there is nothing to distinguish an unrecognised XSL 2.0 FO or property from a simple typo in a XML element or attribute name. It may be that individual XSL formatter products can be updated to recognise the unsupported XSL 2.0 markup and report it as a version mismatch, but that doesn't help existing, installed XSL formatters or their users.

However, the same problem would be experienced by XSL 1.0 formatters encountering XSL 1.1 markup but, to date, that has not been widely reported as a problem. The XSL formatting objects vocabulary is not meant as an interchange format[2], and people do not send each other FO markup to be processed on an arbitrary XSL formatter: XSL is meant to be produced, and in practice usually is produced, in a tightly coupled system where an XSLT stylesheet transforms source XML into FO markup (which might not even be serialised to disk as an XML file) that is immediately processed to produce the formatted output.

3.3.2. Requirements forcing incompatibilities

The current XSL 2.0 Working Draft does not introduce any incompatibilities with XSL 1.1, but some incompatibilities may be necessary if XSL 2.0 is to fulfill its requirements. One potential incompatibility is with the "border-collapse" property:

- In XSL 1.1, "border-collapse" applies only to fo:table and has effect only on borders of fo:table, fo:table-row, fo:table-cell, etc. The property is inherited so,

[2]XSL FO markup is possibly one of the worst possible interchange formats, since it's lost the meaning inherent in the source XML yet still needs to be processed to make something that can be rendered for viewing.

for example, it could be specified on the fo:root and apply to all fo:table in the document.

- One XSL 2.0 requirement is:

  > *When one formatting object is immediately preceding another in block-progression-dimension, be able to specify what to do with their adjacent borders.*

- *If* this requirement is handled by letting "border-collapse" apply to block-level FOs *and* a fo:table's "border-collapse" value is inherited from higher in the FO tree, then the formatted result *may* be different because borders on other, descendant, non-table FOs may be rendered differently. It *may* also change the formatted result of FOs within a table.

3.3.3. Superseded formatting objects

XSL 1.1 (originally, XSL 1.0) defines page geometries using fo:simple-page-master. The name includes "simple" because it was always known that it did not handle everything that people required, yet it was all that could be standardised and implemented in the XSL 1.0 time frame. The definition of fo:simple-page-master includes this note:

> *The fo:simple-page-master is intended for systems that wish to provide a simple page layout facility. Future versions of this Recommendation may support more complex page layouts constructed using the fo:page-master formatting object.*

fo:simple-page-master defines a main, rectangular "body" region of the page (one or more in XSL 1.1) and four possible rectangular "outer" regions at fixed positions at the page edges.

The XSL 2.0 requirements include requirements for non-rectangular areas, for attaching marginalia to regions, for two-page spreads, and for positioning regions relative to each other. Some of these are in the published working draft, some are expected to be in the next working draft, and others can be expected in future working drafts. These can all be expected to be written in terms of fo:page-master and to not apply to fo:simple-page-master and its rectangular regions, yet XSL 2.0 formatters will still be required to format fo:simple-page-master as if they were XSL 1.1 formatters.

It's likely that XSL 2.0 formatters will implement fo:simple-page-master as if it is an instance of the more flexible fo:page-master but one with less than fully-featured regions and limited properties and property values. It is also possible that the fo:simple-page-master definition in the XSL 2.0 Recommendation (and those of its regions) could be rewritten in terms of the newer formatting objects, which could help users transition to using the new FOs and help implementers implement the old in terms of the new.

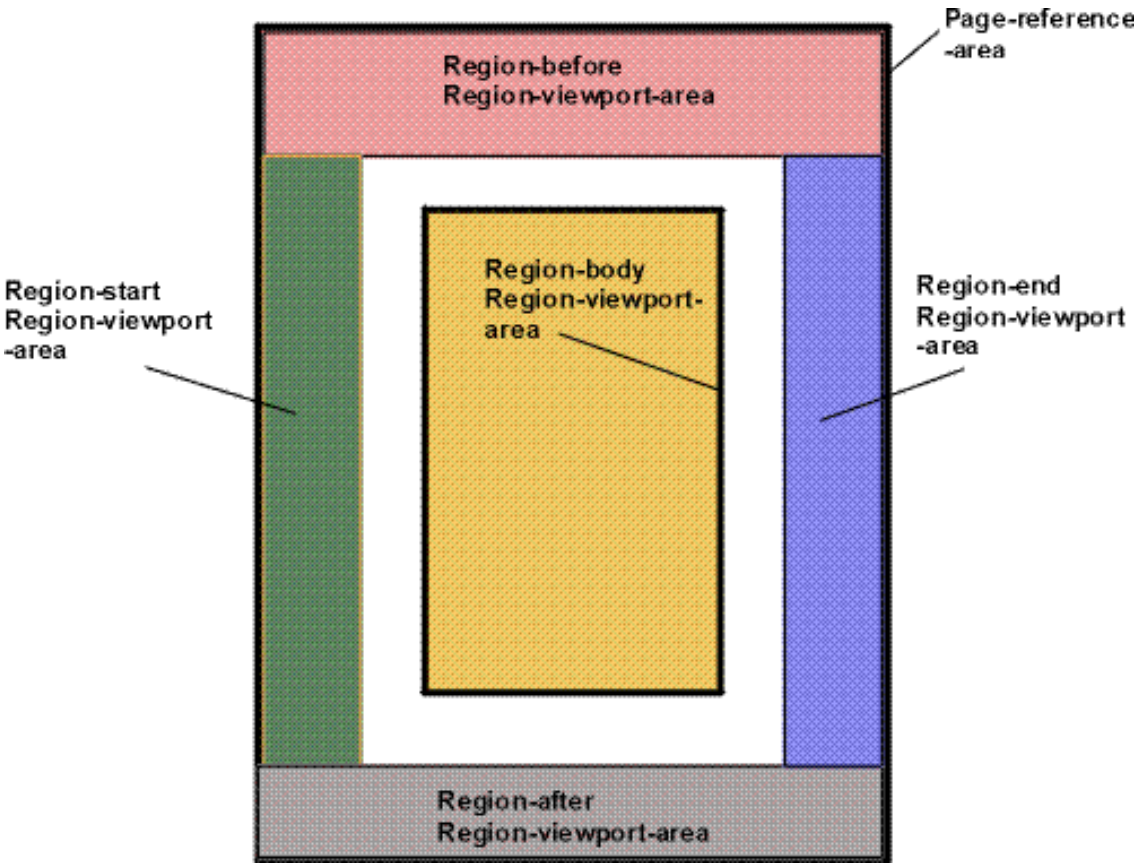


Figure 16. fo:simple-page-master regions

3.3.4. Changes that couldn't be XSL 1.1 errata

Some of the comments to xsl-editors@w3.org about XSL 1.1 issues could not be resolved by issuing erratum since their resolutions would have changed XSL 1.1 conformance requirements. Incorporating those deferred changes into XSL 2.0 will, by definition, potentially change the formatted result of conforming XSL 1.1 documents.

4. Conclusion

The conceptually simple XSL 1.1 block diagram (Figure 1) doesn't tell the whole story for XSL 1.1, and certainly won't tell the whole story for XSL 2.0. A block diagram that attempts to show all that can happen would certainly look less like an arrow and more like an octopus.

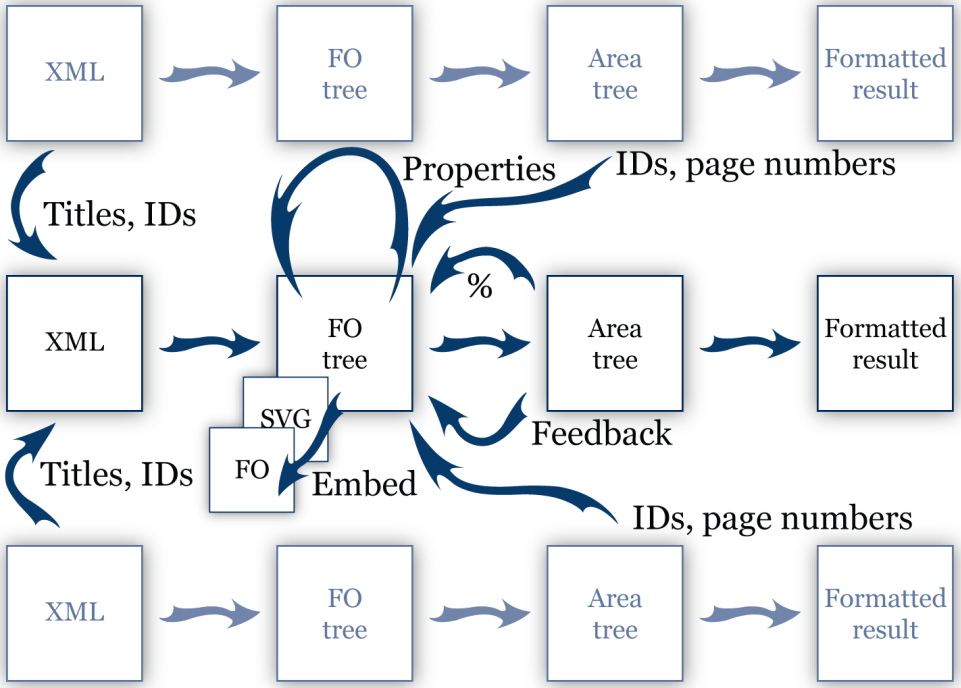


Figure 17. Octopus?

Of course, the other conclusion that can be drawn from the new features being added to XSL 2.0 is that XSL 2.0 will is even more fun than XSL 1.1.

Bibliography

[1] Back of the Book Indexes using XSL-FO 1.0, Crane Softwrights. (See http://www.cranesoftwrights.com/resources/bbi/index.htm.)

[2] International Organization for Standardization, International Electrotechnical Commission. ISO/IEC 10179:1996. Document Style Semantics and Specification Language (DSSSL). International Standard.

[3] World Wide Web Consortium. Requirements for Japanese Text Layout W3C Working Group Note. (See http://www.w3.org/TR/2008/WD-xslfo20-req-20080326/.)

[4] Klein bottle (See http://en.wikipedia.org/wiki/Klein_bottle)

[5] Microsoft, Inso, and ArborText. A Proposal for XSL. (See http://www.w3.org/TR/NOTE-XSL.)

[6] World Wide Web Consortium. SVG Print 1.2, Part 1: Primer. W3C Working Draft. (See http://www.w3.org/TR/2007/WD-SVGPrintPrimer12-20071221/.)

Figure 18. XSL 2.0 is even more fun!

[7] Jon Bosak. XS discussion begins: xml-style (Part 3 of the XML specification suite). (See http://xml.coverpages.org/xs-970524.html.)

[8] World Wide Web Consortium. XSL Requirements Summary W3C Working Draft. (See http://www.w3.org/TR/1998/WD-XSLReq-19980511.)

[9] World Wide Web Consortium. Extensible Stylesheet Language (XSL) 1.1. W3C Recommendation. (See http://www.w3.org/TR/xsl.)

[10] World Wide Web Consortium. XSL-FO 2.0 Working Draft. (See http://www.w3.org/TR/xslfo20/.)

[11] World Wide Web Consortium. XSL-FO 2.0 Requirements W3C Note. (See http://www.w3.org/TR/2008/WD-xslfo20-req-20080326/.)

Automating Document Assembly in DocBook

Norman Walsh
Mark Logic Corporation
<norman.walsh@marklogic.com>

Abstract

While XML is an enabling technology for efficient, granular reuse of information, using angle brackets alone won't get the whole job done. In addition to markup, authors need tools and technologies for finding, extracting, and dynamically assembling new documents from reusable parts.

Dynamic assembly, in this context, refers to the ability to pull together a collection of resources and compose them into a new document which can then be further transformed into any number of presentations: a web site, an online help system, an eBook, or even a traditional, printed book format.

This paper examines ongoing work to provide facilities in DocBook for dynamic assembly. That work includes the DocBook Technical Committee's progress in defining a declarative specification for dynamically assembled documents as well as the author's work to materialize such assemblies using modern XML technologies like XSLT and XProc.

One modern school of thought on technical documentation stresses the development of independent units of documentation, often called topics, rather than a single narrative. Instead of writing something that DocBook users would easily recognize as a book consisting of a preface, several consecutive chapters, and perhaps a few appendixes, the author (or authors) write a set of discrete topics covering aspects of the system as if they were wholly independent.

In a typical online presentation system, for example the world wide web or online help, each topic is a page that stands alone. Except, of course, that just as no man is an island, no topic is completely unrelated to the other topics that are available. From any given topic, there may be topics of obviously related interest. The nature of the relationships may vary.

Some topics are related by physical proximity (if you're interested in the ink cartridges in a printer, you may also be interested in the print head), others by their procedural nature (adding or replacing memory, adding or replacing a hard drive, or even changing the CPU are all topics that might logically follow a topic that describes how to open the computer case).

In a single narrative, it is the responsibility of the author to manage these relationships. He or she can reasonably assume that anyone reading chapter 4 has read chapters 1, 2, and 3. If the reader needs to be directed elsewhere, a cross reference can be used (for example, "for more information on paper jams, see Section 3.5, *The Paper Path*").

In a topic-oriented system, authors are explicitly instructed to write independent units. No linear order can be assumed and many forms of explicit cross-reference are discouraged because the author cannot know what topics will actually be present in any given publication.

Documentation managers treat the library of available topics very much as programmers treat libraries of available functions. Just as any given program can pick and choose from the available libraries, the documentation for any given system can pick and choose from the available topics.

If you imagine a large documentation group managing the documentation for several related systems (different models of printer, different configurations of a software system, automobile engines assembled from different components, etc.) it's easy to see the appeal of topic-oriented authoring.

In a successful deployment, you might find a library of say 1,000 topics which, taken together, document five or six related systems, each of which uses 700-800 topics. Some topics are used in every system, many are used in several systems, and a small number of topics are unique to a specific system.

In order to make such a documentation platform functional, you need not only the individual topics, but also some sort of "map" or "assembly" file that describes which topics from the library are used, what relationships exist between them and, at least for print presentation, what linear order should be imposed upon them.

Historically, DocBook has not addressed this writing methodology. Of course, it has always been possible to write this way. Topics written in articles, sections, or even chapters could be combined into different products. However, the argument can be made that the element names in DocBook encourage authors to think in linear terms.

Recently, the DocBook Technical Commitee[1] decided to attempt to address "topic oriented authoring" more directly. Current plans include the introduction of a `topic` element and the development of a sophisticated "assembly" mechanism for describing which topics are part of a project and how the are organized.

This paper describes the author's understanding of the current state of that work and some of the author's personal explorations into how assemblies might be processed using commodity XML technologies. This paper does not necessarily reflect the consensus of the DocBook Technical Committee.

[1] http://www.oasis-open.org/committees/docbook/

1. Physical structure of an assembly

The notion of a DocBook `assembly` is predicated on a few assumptions about the physical layout of the topics that are to be combined together into units of documentation. We call the units of documentation "structures" and the topics from which they are composed "resources".

For the most part, we assume that resources exist as stand alone documents accessible via URIs. The structures that result from assembling the resources together may be a single file (as in a single PDF book) or a collection of files (as in a web site or help system).

Other arrangements are possible, but for simplicity in this document we assume that the resources are accessible via URIs and the resulting structures can be written to the local filesystem as one or more files.

2. Logical structure of an assembly

Many features of an assembly allow the assembly to update metadata associated with a document, for example changing the title or removing the metadata altogether.

Throughout the description of assemblies, we assume that all metadata always occurs inside an `info` wrapper. In other words, although the following is perfectly legal:

```
<chapter>
  <title>Chapter Title</title>
  <info>
    <pubdate>2009-11-23</pubdate>
  </info>
  <para>Some chapter content.</para>
</chapter>
```

we always assume that this is instead represented with a single `info` element:

```
<chapter>
  <info>
    <title>Chapter Title</title>
    <pubdate>2009-11-23</pubdate>
  </info>
  <para>Some chapter content.</para>
</chapter>
```

Even in cases where there is no `info` element:

```
<section>
  <title>Section Title</title>
  <para>Some section content.</para>
</section>
```

we assume one is present:

```
<section>
  <info>
    <title>Section Title</title>
  </info>
  <para>Some section content.</para>
</section>
```

Authors *are not* required to author in this way in order to use assemblies. Processing systems are to behave *as if* they had.

The assumption that all metadata is always present in a single, explicit `info` wrapper *greatly* simplifies the exposition of assemblies without introducing any actual limitations.

3. Assembly Files

An `assembly` has four major parts:

`resources`	Identifies a collection of topics. An assembly may identifer one or more collections.
`structure`	Identifies a single document. A document in this case is the particular collection of topics that forms the documentation for a product or system. An assembly may identify one or more structures.
`relationships`	Identifies relationships between resources. These relationships may be manifested in any number of structures during assembly. An assembly may identify any number of relationships.
`transformations`	Identifies transformations that can be applied during assembly. An assembly may identify any number of transformations.

3.1. Resources

Individual resources are identified through a `resource` element:

```
<resource xml:id="topicA" fileref="uri/for/topicA.xml"/>
```

Here the resource "`topicA`" is associated with the URI `uri/for/topicA.xml`. Relative URIs are made absolute with respect to the element on which they appear. An optional `description` attribute may also be used to provide a description of the resource.

A collection of resources may appear in a `resources` wrapper. The primary motivation for the resources wrapper is to group physically and logically colocated

resources together. The `resources` wrapper is a convenient place, for example, to specify a common `xml:base` or `xml:lang` and `description`.

If the `resource` element does not have a `fileref` attribute, then the content of the `resource` element itself is the resource. This is often used to create modules which trigger the automatic generation of content, for example Tables of Contents or Indexes:

```
<resource xml:id="toc">
  <toc/>
  <toc role="procedures"/>
</resource>
```

It is an error if the `fileref` attribute is specified and the element is not empty.

3.2. Structures

Structures are where the real work is performed. There is considerable tension between the goal of providing `structure` elements that are simple for the simple cases and simultaneously allowing them to be complex enough to describe the full richness of possibility.

This inherent tension is compounded by the fact that the nature of `structure` outputs is naturally open ended. The goal is that DocBook assemblies should be able to handle both the richness of the structures that we produce today as well as the structures we will want to produce tomorrow.

The principle task of the `structure` element is to identify the resources that are to be combined for that structure. The processing expectation of a structure is that after combining the resources together, the result will be a single document, the *realized structure*. The realized structure is then processed to produce the desired output. (Whether or not a processor actually realizes the structure as a single document in memory or on disk is an implementation detail; logically, that's what happens.)

It is important to observe that the realized structure is often, but not necessarily, a valid DocBook document. What is important is not it's validity from an authoring perspective, but rather it's validity with respect to what the downstream processor is expecting.

For example, if the `structure` is defining a book, then the realized structure should be a valid `book`. If, on the other hand, what is defined is a help system, then the realized structure may be a collection of nested `topic` elements. Even though authors are not allowed to nest topics (per the currently proposed `topic` element), the assembly process may be allowed to nest them.

A `structure` consists mostly of `module` elements.

3.3. Relationships

A relationship, in the context of an assembly, is an assertion that a particular collection of resources are related. The nature of the relationship is specified with a `type` attribute. For example:

```
<relationship type="seealso">
  <instance resourceref="tut1"/>
  <instance resourceref="tut2"/>
  <instance resourceref="task1"/>
</relationship>
```

This asserts that there is a "seealso" relationship between these resources. The processor might use this information to automatically generate a "See Also" section at the end of any of these topics.

Relationships are often, but not necessarily, unordered. This relationship establishes a path through a set of resources:

```
<relationship type="path">
  <info>
    <title>New User Introduction</title>
  </info>
  <instance resourceref="over1"/>
  <instance resourceref="over2"/>
  <instance resourceref="task3"/>
  <instance resourceref="cleanup"/>
</relationship>
```

A sophisticated help system could use this path to guide a new user through a sequence of resources.

3.4. Transformations

An assembly can identify a collection of transformations that can be used during the assembly process. A transformation can be associated with a resource (for example, to translate from some other format into DocBook), or with a module (to address requirements beyond the limited transformation capabilities of the assembly).

```
<transformations>
  <transform name="dita2docbook" type="text/xsl" fileref="dita2db.xsl"/>
  <transform name="tutorial" type="text/xsl" fileref="db2tutorial.xsl"/>
  <transform name="art2pi" type="text/xsl" fileref="art2pi.xsl"/>
  <transform name="office" type="application/xproc+xml" fileref="office2db.xpl"/>
  <transform name="office" type="text/xsl" fileref="extractoffice.xsl"/>
</transformations>
```

If there are several ways to provide a transformation, they may all be listed provided that they have different types. In the example above, it may be that the XProc transformation from office documents to DocBook is superior to the XSLT-only transformation, but the XSLT-only transformation is better than nothing. If no type is specified, the default is implementation dependent.

Note
Not all systems may support arbitrary transformations, nor can all systems support all possible transformation languages. To maximize interoperability it is best to use as few explicit transformations as possible, ideally none.

Given the transformations above, a DITA resource might be included in the assembly:

```
<resource xml:id="overview" fileref="dita/over.xml"
          transform="dita2docbook"/>
```

Whenever a module refers to this resource, it will receive the transformed, DocBook result.

If a module needs to perform a transformation to get from one DocBook format to another, it can name a transform as well:

```
<module resourceref="overview">
  <transform name="art2pi"/>
  <output type="book" renderas="partintro"/>
</module>
```

In this case, two transformations will occur. This can be generalized to an arbitrary number by listing more than one `transform` in the `module`. The transforms are applied in the order specified.

If the `output` specifies a `renderas`, it is applied to the result of the last transformation.

4. Example: Assembling a Printed Book

For the purposes of this section, let's assume that we have the following resources available:

```
<resource xml:id="full-toc">
  <toc/>
  <toc role="figures"/>
  <toc role="tables"/>
  <toc role="procedures"/>
</resource>

<resource xml:id="index">
  <index/>
</resource>
```

```
<resources xml:base="tutorial/">
  <resource xml:id="tut1" fileref="tut1.xml"/>
  <resource xml:id="tut2" fileref="tut2.xml"/>
  <resource xml:id="tut3" fileref="tut3.xml"/>
  <resource xml:id="tut4" fileref="tut4.xml"/>
  <resource xml:id="tut5" fileref="tut5.xml"/>
</resources>

<resources xml:base="tasks/">
  <resource xml:id="task1" fileref="task1.xml"/>
  <resource xml:id="task2" fileref="task2.xml"/>
  <resource xml:id="task3" fileref="task3.xml"/>
  <resource xml:id="task4" fileref="task4.xml"/>
</resources>
```

Further, let's assume that each of the tutorials and tasks is authored as a standalone `topic`.

Our first challenge is to create a PDF user guide. A subject matter expert has informed us that the proper linear presentation for the user guide is tutorials 1 and 2, task 1, tutorial 3, task 4, and an index.

Our first attempt at a structure might look like this:

```
<structure xml:id="user-guide">
  <output renderas="book"/>
  <module resourceref="full-toc"/>
  <module resourceref="tut1"/>
  <module resourceref="tut2"/>
  <module resourceref="task1"/>
  <module resourceref="tut3"/>
  <module resourceref="task4"/>
  <module resourceref="index"/>
</structure>
```

The `output` element identifies the kind of output to be generated. There may be more than one type. The `renderas` attribute on the `output` tells us that the realized structure should be a book.

The `module` elements identify the resources to be included. That's fine as far as it goes. Processing this `structure` will create a realized structure that is a `book` (because that's what the `renderas` on the `output` tells us) consisting of the tables of contents and the correct six topics:

```
<book xmlns="http://docbook.org/ns/docbook">
  <toc/>
  <toc role="figures"/>
  <toc role="tables"/>
  <toc role="procedures"/>
```

```
  <topic xml:base="tutorial/tut1.xml">
    <title>Introduction</title>
    ...
  </topic>
  <topic xml:base="tutorial/tut2.xml">
    <title>Getting Started</title>
    ...
  </topic>
  <topic xml:base="tasks/task1.xml">
    <title>Engaging the spindle</title>
    ...
  </topic>
  <topic xml:base="tutorial/tut3.xml">
    <title>Troubleshooting</title>
    ...
  </topic>
  <topic xml:base="tasks/task4.xml">
    <title>Diagnosing spindle problems</title>
    ...
  </topic>
  <index/>
</book>
```

The trouble is, that's not a valid `book`. Not only does it consist of topics instead of the expected chapters and such, but it's lacking necessary metadata like a title.

We can address the first problem by allowing `output` elements inside modules.

Most high-level elements (`chapter`, `appendix`, `book`, `section`, `topic`, etc.) in DocBook have the same general structure: they contain an (optional) `info` element followed by a collection of block elements.

The semantic of `renderas` in a `module` that points to a resource is that it renames the root element of that resource. This simple transformation will allow us to turn all those topics into elements more appropriate for `book` content. (We'll consider how to address the situation where more complex transformation is necessary later.)

We can address the problem of missing metadata by creating a new `resource` that contains an `info` element containing the appropriate metadata and then pointing to it. But this case is so common, that we simply allow the `info` element to be placed in the `structure`.

With these amendments, here's our new structure:

```
<structure xml:id="user-guide">
  <output renderas="book"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc"/>
  <module resourceref="tut1">
```

```
    <output renderas="chapter"/>
  </module>
  <module resourceref="tut2"/>
  <module resourceref="task1"/>
  <module resourceref="tut3">
    <output renderas="appendix"/>
  </module>
  <module resourceref="task4"/>
  <module resourceref="index">
    <output renderas="index"/>
  </module>
</structure>
```

By default, the `renderas` value is inherited from a module's preceding sibling. Even though the index is an `index`, because there is an explicit rendering on a preceding module, it must be explicit on the index.

The realized structure that this produces is:

```
<book xmlns="http://docbook.org/ns/docbook">
  <info>
    <title>Widget User Guide</title>
  </info>
  <toc/>
  <toc role="figures"/>
  <toc role="tables"/>
  <toc role="procedures"/>
  <chapter xml:base="tutorial/tut1.xml">
    <title>Introduction</title>
    ...
  </chapter>
  <chapter xml:base="tutorial/tut2.xml">
    <title>Getting Started</title>
    ...
  </chapter>
  <chapter xml:base="tasks/task1.xml">
    <title>Engaging the spindle</title>
    ...
  </chapter>
  <appendix xml:base="tutorial/tut3.xml">
    <title>Troubleshooting</title>
    ...
  </appendix>
  <appendix xml:base="tasks/task4.xml">
    <title>Diagnosing spindle problems</title>
    ...
  </appendix>
```

```
    <index/>
  </book>
```

That realized structure is a valid `book` so you would expect the processing system to produce the correct results.

Except that it's not the right book. Upon further review by our subject matter expert, it's been decided that tutorial 3 and task 4 shouldn't be separate appendixes, they should instead both be sections in a single appendix called “Troubleshooting”.

We can address this issue with nested modules. Instead of having separate top-level modules for the tutorial and task, we'll make them subordinate modules in a new top-level module:

```
<structure xml:id="user-guide">
  <output renderas="book"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc"/>
  <module resourceref="tut1">
    <output renderas="chapter"/>
  </module>
  <module resourceref="tut2"/>
  <module resourceref="task1"/>
  <module>
    <output renderas="appendix">
    <info>
      <title>Troubleshooting</title>
    </info>
    <module resourceref="tut3">
      <output renderas="section"/>
    </module>
    <module resourceref="task4"/>
  </module>
  <module resourceref="index"/>
</structure>
```

The semantics of `renderas` on a module that does not have a `resourceref` is that it simply generates that element.

The realized document is now:

```
<book xmlns="http://docbook.org/ns/docbook">
  <info>
    <title>Widget User Guide</title>
  </info>
  <toc/>
  <toc role="figures"/>
  <toc role="tables"/>
```

```
  <toc role="procedures"/>
  <chapter xml:base="tutorial/tut1.xml">
    <title>Introduction</title>
    ...
  </chapter>
  <chapter xml:base="tutorial/tut2.xml">
    <title>Getting Started</title>
    ...
  </chapter>
  <chapter xml:base="tasks/task1.xml">
    <title>Engaging the spindle</title>
    ...
  </chapter>
  <appendix>
    <info>
      <title>Troubleshooting</title>
    </info>
    <section xml:base="tutorial/tut3.xml">
      <title>Troubleshooting</title>
      ...
    </section>
    <section xml:base="tasks/task4.xml">
      <title>Diagnosing spindle problems</title>
      ...
    </section>
  </appendix>
  <index/>
</book>
```

That's probably not quite what was intended. Given that tutorial 3 has the title "Troubleshooting", what we probably wanted was to make that tutorial the `appendix` and make the task a subordinate section of that appendix.

We can achieve that by nesting the modules differently:

```
<module resourceref="tut3">
  <output renderas="appendix"/>
  <module resourceref="task4">
    <output renderas="section"/>
  </module>
</module>
```

Now the `appendix` in the realized structure has the form that we want:

```
<appendix xml:base="tutorial/tut3.xml">
  <title>Troubleshooting</title>
  ...
  <section xml:base="tasks/task4.xml">
    <title>Diagnosing spindle problems</title>
```

```
    ...
  </section>
</appendix>
```

If a module “A” is nested within a module “B” that refers to a resource, the result of processing “A” is inserted into “B” as the last child of “B”. (in the case of several nested modules, they are inserted as the last children in the order specified.)

After further review of the content, the subject matter expert decides that tutorial 5 and task 3 are also relevant to trouble shooting. Tutorial 5 also has the title “Troubleshooting” and naturally follows tutorial 3. Task 3 is about diagnosing bearing issues.

It's easy to combine them the new content into the appendix:

```
<module resourceref="tut3">
  <output renderas="appendix"/>
  <module resourceref="tut5">
    <output renderas="section"/>
  </module>
  <module resourceref="task4"/>
  <module resourceref="task3"/>
</module>
```

but that doesn't produce a pleasing result:

```
<appendix xml:base="tutorial/tut3.xml">
  <title>Troubleshooting</title>
  ...
  <section xml:base="tutorial/tut5.xml">
    <title>Troubleshooting</title>
    ...
  </section>
  <section xml:base="tasks/task4.xml">
    <title>Diagnosing spindle problems</title>
    ...
  </section>
  <section xml:base="tasks/task3.xml">
    <title>Diagnosing bearing problems</title>
    ...
  </section>
</appendix>
```

What we really want to do is combine tutorials 3 and 5, not make 5 a subordinate section of 3. While we're at it, let's change the title of the appendix to “Troubleshooting spindle and bearing problems” since that's what it's really about.

We can solve the first problem with the `contentonly` attribute on `module`:

```
<module resourceref="tut3">
  <output renderas="appendix"/>
```

```
  <module resourceref="tut5" contentonly="true"/>
  <module resourceref="task4">
    <output renderas="section"/>
  </module>
  <module resourceref="task3"/>
</module>
```

If `contentonly` is `true`, then only the content of the document element, and not the document element itself, is included. Any metadata associated with the document element is also discarded. (If the referenced resource has several top-level element children, then this processing is applied to each, in turn.)

The realized structure for this appendix is now:

```
<appendix xml:base="tutorial/tut3.xml">
  <title>Troubleshooting</title>
  ...
  ...content (only) of tutorial 5...
  <section xml:base="tasks/task4.xml">
    <title>Diagnosing spindle problems</title>
    ...
  </section>
  <section xml:base="tasks/task3.xml">
    <title>Diagnosing bearing problems</title>
    ...
  </section>
</appendix>
```

There are two ways that we can change the title. The first uses only semantics we've already encountered:

```
<module>
  <output renderas="appendix"/>
  <info>
    <title>Troubleshooting spindle and bearing problems</title>
  </info>
  <module resourceref="tut3" contentonly="true"/>
  <module resourceref="tut5" contentonly="true"/>
  <module resourceref="task4">
    <output renderas="section"/>
  </module>
  <module resourceref="task3"/>
</module>
```

The only disadvantage of this approach is that we lose all of the metadata associated with tutorials 3 and 5. This might include publication dates, copyright information, etc. We could add those fields to the `info` wrapper for the `appendix` in the assembly, but then it may have to be maintained in two places.

Instead, we introduce one more convention: if a `module` refers to another resource *and* contains an `info` element, then the elements within that `info` replace any elements of the same name in the referenced resource's metadata. (If the referenced resource has no metadata, then the specified `info` becomes the first child of the referenced resource.)

The second approach uses this convention:

```
<module resourceref="tut3">
  <output renderas="appendix"/>
  <info>
    <title>Troubleshooting spindle and bearing problems</title>
  </info>
  <module resourceref="tut5" contentonly="true"/>
  <module resourceref="task4">
    <output renderas="section"/>
  </module>
  <module resourceref="task3"/>
</module>
```

In the realized structure for this example, the appendix metadata includes everything in tutorial 3's metadata, with the title changed as indicated. Of course, all of the metadata for tutorial 5 is still lost, but there's nothing we can do about that because there's no where to put it.

5. Example: Assembling an Online Book

This example extends the book example, see Section 4. In addition to the printed user guide, we want to produce an online user guide. The online guide is, by design, a straightforward, linear rendering of the book. We'll look at a more intricately structured example later.

We could create a separate assembly for the online guide, but it will be better in the long run if we can manage the structure of the user guide in one place.

The first step is to create a new output type:

```
<structure xml:id="user-guide">
  <output type="book" renderas="book"/>
  <output type="web" renderas="book"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc"/>
  <module resourceref="tut1">
    <output renderas="chapter"/>
  </module>
  <module resourceref="tut2"/>
  <module resourceref="task1"/>
```

```
  <module resourceref="tut3">
    <output renderas="appendix"/>
    <info>
      <title>Troubleshooting spindle and bearing problems</title>
    </info>
    <module resourceref="tut5" contentonly="true"/>
    <module resourceref="task4">
      <output renderas="section"/>
    </module>
    <module resourceref="task3"/>
  </module>
  <module resourceref="index"/>
</structure>
```

Now we can process this structure either as a "book" type structure or a "web" type structure. At the moment, there's no difference between them. Let's consider some of the changes we need to make for the online site:

1. We want to control the names of the HTML documents produced.
2. We want to control how chunking is performed.
3. We want to suppress some "print only" content.

The first two of these we can address with `output` elements:

```
<structure xml:id="user-guide">
  <output type="book" renderas="book"/>
  <output type="web" renderas="book" file="user-guide.html"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc">
    <output type="web" chunk="false"/>
  </module>
  <module resourceref="tut1">
    <output renderas="chapter"/>
  </module>
  <module resourceref="tut2"/>
  <module resourceref="task1"/>
  <module resourceref="tut3">
    <output renderas="appendix"/>
    <info>
      <title>Troubleshooting spindle and bearing problems</title>
    </info>
    <module resourceref="tut5" contentonly="true"/>
    <module resourceref="task4">
      <output renderas="section"/>
      <output type="web" chunk="false"/>
    </module>
```

```
    <module resourceref="task3"/>
  </module>
  <module resourceref="index">
    <output type="web" file="book-index.html"/>
  </module>
</structure>
```

Specifying a `file` on an `output` tells the assembly where to write the chunk. (If it occurs on a module which is not chunked, it is simply ignored.) The `chunk` attribute can be used to specify which modules should (or should not) be placed in separate chunks. The implied value on modules where it is not specified depends on the name of the rendered element and the output type.

When processing a module, all of the relevant output directives are combined. An `output` is relevant if it does not specify a type or if the type specified matches the type of processing that is being performed. If conflicting values are specified for any attribute, the last value specified (in document order) is used.

To exclude (or include) content, we add the `filterout` (and `filterin`) elements to the assembly. Each may have a `type` and specifies some number of effectivity values.

If no conditions are specified on `filterout`, the module is unconditionally excluded. (Specifying no conditions on a `filterin` has no effect.)

If filters are used at the `structure` level, they apply to the entire document. If they are used at the `module` level, they apply only to that module.

With filters, our `structure` might look like this:

```
<structure xml:id="user-guide">
  <output type="book" renderas="book"/>
  <output type="web" renderas="book" file="user-guide.html"/>
  <filterout type="web" condition="print"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc">
    <output type="web" chunk="false"/>
  </module>
  <module resourceref="tut1">
    <output renderas="chapter"/>
  </module>
  <module resourceref="tut2">
    <filterin type="web" condition="print"/>
    <filterout type="web" userlevel="advanced"/>
  </module>
  <module resourceref="task1"/>
  <module resourceref="tut3">
    <output renderas="appendix"/>
```

```
    <info>
      <title>Troubleshooting spindle and bearing problems</title>
    </info>
    <module resourceref="tut5" contentonly="true"/>
    <module resourceref="task4">
      <output renderas="section"/>
      <output type="web" chunk="false"/>
    </module>
    <module resourceref="task3"/>
  </module>
  <module resourceref="index">
    <filterout type="web"/>
  </module>
</structure>
```

This `structure` applies the following filters when rendering for the web: print only content is excluded globally; in tutorial 2, print only content *is* included, but advanced material is excluded; and the entire index is excluded.

6. Processing an assembly

The key observation in processing an assembly is that it is really a sequence of transformations, some implicit and some explicit. Nested modules can be seen as rules for performing inclusion processing; alternate `info` wrappers imply a merge, output elements may specify a renaming, etc. In addition to these implicit transformations, an assembly author may request explicit ones as well, for example to transform DITA content into DocBook.

If an assembly can be viewed as declaring a sequence of transformations over content, what we need is a language for describing a sequence of transformations, we need a pipeline language.

Not surprisingly, I have one of those: XProc.

The initial approach then seems clear:

1. View a particular structure in a particular assembly as declaring a sequence of transformations.
2. Transform that declaration into an XProc pipeline that makes the implicit transformations explicit.
3. Feed this “derived” or “compiled” pipeline into an XProc processor.
4. And get our assembled document out the other end.

In fact, this approach works quite well. Consider the example assembly in Example 1. If we apply the transformation, [2] the resulting pipeline, Example 2, can be used to construct the assembled book.

Example 1. An example assembly

```
<assembly xmlns="http://docbook.org/ns/docbook" version="5.1">

<resource xml:id="full-toc">
  <toc/>
  <toc role="figures"/>
  <toc role="tables"/>
  <toc role="procedures"/>
</resource>

<resource xml:id="index">
  <index/>
</resource>

<resources xml:base="src/tutorials/">
  <resource xml:id="tut1" fileref="tut1.xml"/>
  <resource xml:id="tut2" fileref="tut2.xml"/>
  <resource xml:id="tut3" fileref="tut3.xml"/>
  <resource xml:id="tut4" fileref="tut4.xml"/>
  <resource xml:id="tut5" fileref="tut5.xml"/>
</resources>

<resources xml:base="src/tasks/">
  <resource xml:id="task1" fileref="task1.xml"/>
  <resource xml:id="task2" fileref="task2.xml"/>
  <resource xml:id="task3" fileref="task3.xml"/>
  <resource xml:id="task4" fileref="task4.xml"/>
</resources>

<structure xml:id="user-guide">
  <output renderas="book"/>
  <info>
    <title>Widget User Guide</title>
  </info>
  <module resourceref="full-toc"/>
  <module resourceref="tut1">
    <output renderas="chapter"/>
  </module>
```

[2] Ed. note: for space reasons source code of the transformation was removed from the paper. It will be available for download from the conference site.

```
  <module resourceref="tut2">
    <output renderas="chapter"/>
  </module>
  <module resourceref="task1">
    <output renderas="chapter"/>
  </module>
  <module>
    <output renderas="appendix"/>
    <info>
      <title>Troubleshooting</title>
    </info>
    <module resourceref="tut3">
      <output renderas="section"/>
      <info>
        <title>Troubleshooting spindle and bearing problems</title>
      </info>
      <module resourceref="tut5" contentonly="true"/>
      <module resourceref="task4">
        <output renderas="section"/>
      </module>
      <module resourceref="task3"/>
    </module>
  </module>
  <module resourceref="index">
    <output renderas="index"/>
  </module>
</structure>

</assembly>
```

Example 2. The "compiled" pipeline

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:db="http://docbook.org/ns/docbook"
                name="user-guide"
                version="1.0">
   <p:input port="source"/>
   <p:output port="result"/>
   <p:import href="asmutils.xpl"/>
   <p:group name="user-guide.3">
      <p:output port="result" sequence="true"/>
      <p:identity name="rsrc-user-guide.3">
         <p:input port="source">
            <p:inline>
               <toc xmlns="http://docbook.org/ns/docbook"/>
            </p:inline>
            <p:inline>
```

```
                <toc xmlns="http://docbook.org/ns/docbook" role="figures"/>
             </p:inline>
             <p:inline>
                <toc xmlns="http://docbook.org/ns/docbook" role="tables"/>
             </p:inline>
             <p:inline>
                <toc xmlns="http://docbook.org/ns/docbook" role="procedures"/>
             </p:inline>
          </p:input>
       </p:identity>
    </p:group>

    …

    <p:wrap-sequence wrapper="db:book">
       <p:input port="source">
          <p:pipe step="user-guide.3" port="result"/>
          <p:pipe step="user-guide.4" port="result"/>
          <p:pipe step="user-guide.5" port="result"/>
          <p:pipe step="user-guide.6" port="result"/>
          <p:pipe step="user-guide.7" port="result"/>
          <p:pipe step="user-guide.8" port="result"/>
       </p:input>
    </p:wrap-sequence>
    <p:insert match="/*" position="first-child">
       <p:input port="insertion">
          <p:inline>
             <info xmlns="http://docbook.org/ns/docbook">
                <title>Widget User Guide</title>
             </info>
          </p:inline>
       </p:input>
    </p:insert>
 </p:declare-step>
```

Writing the XProc pipeline by hand would be tedious and error prone, but it can be generated quite easily from the relatively simpler and easier to maintain assembly file.

7. Open questions

The first open question is how will the final assembly specification approved by the Technical Committee compare to this one? It seems very likely, to me, that the approach outlined in this paper will be applicable, but it's impossible to say for certain.

A more interesting open question is whether or not this approach will actually work for all the possible assemblies. The model taken here is that an assembly document takes as input a set of documents and a declaration of the desired result and produces a single, presumably valid, output document.

In environments where assemblies are used to create complex hypertext help systems, what must be produced is a collection of files that bear a well-understood relationship to each other.

I'm optimistic that it will be possible to view construction of those files as a separate phase: first assemble the result document (perhaps with embedded markup or processing instructions to inform the next phase of processing), then process that document to produce the collection of files.

It's possible, however, that such an approach will not be practical. It's possible that the assembly processor will need to produce the resulting files directly. If this is the case, it may turn out to be impractical to simply compile an assembly to an XProc pipeline. But I hope not.

A final question is how best to handle the various sorts of relationships asserted in the assembly. At present, the transformation ignores those.

EXPath: Packaging, and Web applications

A packaging system for XML libraries and a portable web application framework

Florent Georges
H2O Consulting
<fgeorges@fgeorges.org>

Abstract

EXPath is a project to define portable extensions to core XML technologies. The main means to doing so is the specification of libraries of XPath functions, usable within any XPath expressions evaluator, including XSLT, XQuery and XProc. But EXPath is also open to other kinds of specifications, like the Packaging System and the framework to write Web Applications, both described in this paper.

Keywords: EXPath, extension, XPath, XSLT, XQuery

1. Introduce EXPath

1.1. History

Exactly one year ago, in this same room during XML Prague 2009, the idea of EXPath was born. EXQuery was just launched and with some people we were trying to launch a new version of EXSLT, for XSLT 2.0. EXProc welcomed its first extensions, too. While all those projects make perfect sense regarding their own technology, they also overlap in providing some identical functionalities to their target languages. So the idea emerged here: acting at the XPath level to touch a broader audience and to define a same extension only once.

So one year after it was born, this paper show the current state of the project and introduces some new ideas and modules.

1.2. Processes & goals

The project is fundamentally a collaborative project. It is driven by the community (essentially a mix of the XSLT, XQuery and XProc communities). When people want to introduce a new module, for instance because they have to use such an extension within their own project and think there is an interest to share their work and benefit from the community feedback, then the first step is just to send an email to the EX-

Path mailing list. They can see if the idea is well received and if someone else is also interested in the same extension.

Once the discussion is launched, the next step is to write a draft specification for this extension, and ask for feedback. The main project delivery unit is the *module*. A module is a loosely-defined concept, gathering everything related to the same functionality. For instance, all functions to deal with ZIP files (reading, writing, etc.) are grouped into the same module. Most of the time, a module is a library of XPath functions, but it could be anything related to core XML technologies. For instance the packaging system introduced below is a module. Common sense is our friend.

The first objective of a module is to define an implementation-independent specification. It has always a maintainer, who is in charge of coordinating the efforts around the module. The tasks involved are leading the discussions, finding consensus when appropriate, writing and maintaining the spec(s) and the test suite(s), promoting the module and coordinating implementation and documentation efforts. His/her tools are the mailing list, the website, the community itself, the technical tools (editorial schemas, test suite tools, etc.), Subversion repositories, the wiki, and any relevant tool we could provide as a community project, supported by inpendent individuals.

The utimate goal is to have the specifications endorsed by vendors and implemented directly by existing processors. But every vendors are not interested in providing the same extensions, so it sometimes makes sense to implement them as third-party extensions, and EXPath provides facilities to anyone willing to contribute such an implementation. It is worth noting that such implementations are also a good way to experiment and get feedback while in the process of writing the module specification.

2. Packaging System

2.1. Context

If you work with XSLT or XQuery for a few years, I guess you already tried to install a third-party stylesheet or query, because it provided an out-of-the-box bar code formating facility or some utility functions to deal with a particular XML language. So you went to the library website, looked for the download page, downloaded the component somewhere on your computer, and read the installation instruction: *put the stylesheet (or the query) in an appropriate place, please see your processor documentation*. So you put the component *somewhere*, sometimes in the same directory than your stylesheets, sometimes you tried to create a kind of central repository on your system. And you finally succeeded, after you figured out the correct import statements, to run your transform from the command line. Then you tried to configure your new XML IDE to be able to run the same transforms. Then you integrated your compon-

ents, and their dependencies, into the enterprise application that uses then. Or even worse, you tried to find a way to deliver your stylesheets or queries as a library, and had to take care about their third-party dependencies.

Then you needed the new version of that third-party library...

Why couldn't you just ask your tools: *could you use this and that libraries, please*? After all, every XML technologies use the same tool to identify dependencies: a URI. You want to use an XQuery library module? You have to give its namespace URI. You want to import a stylesheet module? You have to give its URI. You want to use an XProc pipeline? Well, guess what...

Actually URIs are very good at that game. There can be any level of indirections: the processor itself, either if it knows the resource built-in, or if it resolves it through its own documented mechanism, but also some configuration options, or networking redirections. There is even a standard format to express URI redirections as an XML document, or set of documents: XML Catalogs, from OASIS. (Most of) the existing processors support (some of) those and other redirecting technologies, (more or less) out of the box.

So why is it still so difficult to install an XSLT or XQuery library? In fact, while the installation is not always easy, that's much the delivery of the components that is difficult. For now, you cannot really do nothing much than putting your components into a ZIP file or any other archive format, and say *please see you processor documentation*.

It is interesting to note that even though XQuery has a module system that enforces associating a URI to a library module, the situation is not much better than for XSLT. It seems sometimes easier to find a place where to put the queries on an XML database server, because it controls its own data space, as opposed to a standalone processor. But some of those XML databases do not allow one to choose his/her own resolution mechanism, requiring importing queries to use an implementation-defined "at hint" in its import statements. Furthermore, publishing a library does not mean only providing its components and the associated URIs. Some components are private and are just library's internals. Or another component is an extension developed in another language. Or yet another component is specific to a given processor.

The key is a common, portable packaging format.

A format that allows library authors to package their libraries without having to worry about it. A common format understood by most of the existing processors. A format that allows you to package a consistent library, not regarding the technologies it uses, but its purpose: a format that allows you to deliver XSLT stylesheets, XQuery modules, XML Schemas and XProc pipelines all together.

2.2. Architecture

As noted above, all XML technologies use URIs to identify dependencies. So the first thing to do in order to resolve a URI to a resource is to get this URI. As explained, XQuery has a first-class module system, and enforces the fact that a library module has an associated URI: its target namespace. Other technologies, like XSLT and XProc, do not have such a built-in module system. You have to use a URI to refer to an XSLT stylesheet, but nothing force a user to use the same URI. Nothing is provided to the library author to say: *this is the public URI associated to this stylesheet*.

The EXPath Packaging System associates a **public URI** to every public components. A public component is any XSLT stylesheet, XQuery module, XProc pipeline, XML Schema or any other supported technology, aimed to be used directly by the user. For instance, a top-level stylesheet is a public component, but not some of the other stylesheets it imports, providing only private tools intern to the library itself. The library author chooses the public URIs for his/her components. The user can use those public URIs in other components. For instance in a stylesheet importing your stylesheet. Because the public URIs are defined once by the library author, there is no more the problem of modifying import statements in a third-party library just because its author does not use the same processor than you.

When you provide a processor supporting the packaging system with a package, it will install its content in some implementation-defined way. The way to provide the processor with the package is implementation-defined itself. It can be by using a web-based admin console, by putting the package in a predefined directory, or by using an XML IDE graphical interface to manage a local repository on the disk, shared by several processors. What is important is that the processor finds the correct component when you use its public URI in, say, an `xsl:import` statement:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

   <!-- there is nothing at that URL, the module must have been installed -->
   <xsl:import href="http://example.org/project/something.xsl"/>

   ...
```

This matches the usual way to deal with dependencies with other languages: you write on the one hand the code that uses a library in a canonical way, especially regarding how it references the imported library, and you maintain the list of installed packaged on the other hand, in an implementation-dependent way.

2.3. The package format

A package is a simple ZIP file containing the components (i.e. the source files). In order to achieve independence on the processor, the package format must provide

enough information to map the public URIs to their respective components. This information, as well as a few others like the name of the package, are provided within the package descriptor. The package descriptor is a file named `expath-pkg.xml` at the root of the package ZIP file.

If we take Priscilla Walmsley's famous FunctX library as an example, we could have the following package descriptor:

```
<package xmlns="http://expath.org/ns/pkg"
         name="http://functx.com/"
         version="1.0">
   <title>FunctX library</title>
   <xquery>
      <uri>http://functx.com</uri>
      <resource>functx.xql</resource>
   </xquery>
   <xslt>
      <uri>http://functx.com/functx.xsl</uri>
      <resource>functx.xsl</resource>
   </xslt>
</package>
```

That is, the descriptor declares two public components: an XQuery library module and an XSLT stylesheet. For each it gives its public URI and the source file containing the component. The structure of the package ZIP file itself is very simple:

```
expath-pkg.xml
content/
   functx.xql
   functx.xsl
```

2.4. A "system", you said?

Ok, so you have a package format. But why do you call it a package "system"?

This packaging format supports various XML technologies: XSLT, XQuery, XProc, XML Schema, RelaxNG, Schematron and NVDL. Besides these standard component types, the package format has been designed to be extensible in two ways. First, vendors can extend it in a controled way to describe components specific to their processor. That could be as simple as having standard components using vendor-specific extensions, or as complex as extensions written in another language, like Java or C++. Second, another specification can use this packaging format as a basis, and add specific informations either in the package descriptor, or in additional files within the ZIP structure. This is for instance the approach taken by the Webapp module, introduced later in this paper.

Besides these extensibility mechanisms, this format allows the creation of repository management softwares, support within XML IDEs, or even the CXAN system.

CXAN is a central repository of packages, and its associated tool to manage a local repository, automatically downloading and installing packages from the internet and resolving dependencies. It means Comprehensive XML Archive Network, and is named after CTAN and CPAN, the same kind of websites and tools for resp. TeX and Perl.

The packaging system can be the basis to spread core XML technologies around, by giving them a way to build a strong basis of libraries.

3. Web Applications

3.1. Context

These days, more and more applications are developed as *web applications*. I do not want to even try to define what's a web application, but let's say that is an application running on a server, and which interacts with the rest of the world (usually end users) by responding to HTTP requests. XML technologies are not an exception, and most processors offer a way to be accessed via HTTP, from the simple REST API to access the processor features themselves, to frameworks to build entire web applications.

In such systems, an HTTP request arrives to the server, which looks at the URI to identify how to respond to this request. The URI will resolve either in a static page, an internal server routine or in a user component, for instance an XQuery module. The module can then use functions provided by the server to access information from the HTTP request, like its headers, and to set properties of the response, like the HTTP response code. The XQuery module will usually access a database (in read and/or write) and extract some data, then it will transform this data to the format the user expect it (for instance in HTML, or in a standard XML vocabulary).

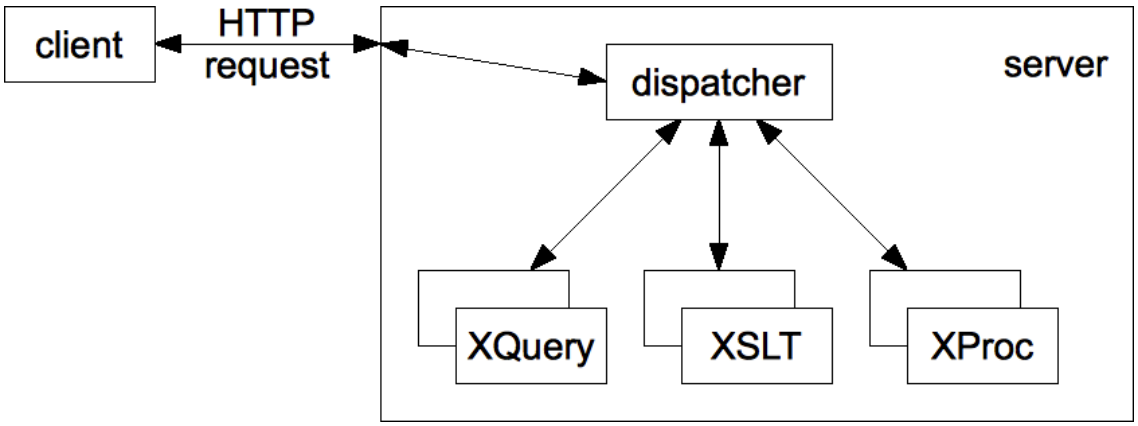
As usual, the way to plug your components in such a system, as well as the way it can interact with it (accessing information from the request and providing the response) is implementation-defined. Even though most of those informations are exactly the same on all those systems: they are defined by the HTTP request-response model.

Besides the benefit of writing end-user web applications in a processor-independent way, such a standard mechanism to deploy XML-based web applications would allow writing frameworks for web applications directly in XQuery, XProc or XSLT (or more likely a mix of them). Most popular frameworks for Java web applications for instance, like Struts or Spring MVC, could be developed because the Java Servlet specification existed. Having a mechanism similar to servlets would allow to write MVC frameworks in an implementation-independent way.

3.2. Architecture

A web application is a set of entry points. An entry point can be an XPath function (provided by an XQuery library module, a stylesheet, or any other means), an XSLT named template, an XQuery main module, an XSLT stylesheet or an XProc pipeline. Each kind of entry point must follow different rules regarding how it accesses the request data and how it provides the response content (for instance a function will get the request as parameter, while an XQuery main module will get it as an external variable), but besides these slightly different interfaces, they can do the same thing.

Each entry point is associated a regular expression to match the URI a request is sent to. When a request is sent to server, the web container try to match the accessed URI to an entry point. A representation of the request is build as an element node (gathering informations like the HTTP headers, the request body content, etc.). This request element is passed to the entry point component (for instance an XPath function). The return value of this component is used to create the response to send back to the user.



A request element looks like the following:

```
<request servlet="srv-1" path="/some/page" method="post">
   <uri>http://www.host.org:8000/myapp/some/page</uri>
   <context-root>/myapp</context-root>
   <path>
      <part>/some/</part>
      <match>page</match>
   </path>
   <auth method="basic" username="user">
      <basic sesame="YWRtaW46YWRtaW5hZG1pbg==" password="..."/>
   </auth>
   <header name="host" value="www.host.org"/>
   <header name="connection" value="keep-alive"/>
   <multipart content-type="multipart/alternative"
              boundary="=-=-=-=-=-=">
      <body content-type="text/plain"/>
      <body content-type="text/html"/>
```

```
      <body content-type="application/xml"/>
   </multipart>
</request>
```

The evaluation of the component (for instance calling the XPath function or running the query module) will give the result to send back to the user. The result is also represented as an element node, but representing an HTTP response instead of a request. It looks like the following:

```
<response status="200" message="Ok">
   <header name="x-myheader" value="My own header value"/>
   <body content-type="text/plain">Hello, world!</body>
</response>
```

In order to deploy an application, the web container needs to know the match patterns and their associated URI patterns. The user gives those informations as an XML document: the deployment descriptor. It lists the entry points with their components and URI patterns, as well as a few other configuration infos:

```
<webapp xmlns="http://expath.org/ns/webapp"
        xmlns:my1="http://mycorp.com/proj/my-app/servlet-1"
        xmlns:my2="http://mycorp.com/proj/my-app/servlet-2"
        name="http://fgeorges.org/example/my-webapp"
        abbrev="my-webapp"
        version="0.1">

   <title>My simple web application</title>

   <context>
      <param name="webmaster" value="webmaster@mycorp.com"/>
      <param name="collection" value="http://mycorp.com/dataset"/>
   </context>
   <page>
      <url pattern="/(.+)\.html"/>
   </page>

   <servlet name="srv-xq-1">
      <xquery function="my1:servlet"/>
      <url pattern="/catalog/(.+)"/>
   </servlet>

   <servlet name="srv-xq-2">
      <xquery resource="servlet-2.xq"/>
      <url pattern="/xq2/(.+)"/>
   </servlet>

   <servlet name="srv-xsl-1">
      <xslt uri="http://www.mycorp.com/proj/my-app/servlet-2.xsl"
```

```
            template="my2:servlet"/>
        <url pattern="/(.+)-xsl/1/"/>
    </servlet>

</webapp>
```

After a few infos about the application itself, like its title and some configuration parameters, the descriptor declares the various end points with the element `servlet`. Each of them references a component (like an XSLT named template or an XQuery main module) and a URI pattern. Based on this deployment descriptor, the web container is able to dispatch the HTTP requests to the correct end point.

The last part of the architecture is how to provide the web container with the data. This one is easy: it is built upon the packaging *system*. A web archive has exactly the same structure as a regular package. A ZIP file with all components, and the package descriptor `expath-pkg.xml` at the root of the package. For a web archive, one just has to add the deployment descriptor, called `expath-web.xml`, next to the package descriptor.

3.3. ServleX

ServleX is an early implementation of this specification, using Java Servlet technologies for the integration with HTTP, and Saxon and Calabash for its X* processors (to actually evaluate the XSLT, XQuery and XProc components). But the main target of this specification is of course existing XML databases like eXist and MarkLogic, and more specialized solutions like Sausalito. Because it uses Java Servlet, it can be deployed in any servlet container, like Tomcat, Jetty, GlassFish, WebLogic, etc.

Let's look at a real example. This example is verry simple. The webapp contains only one component, a function defined in an XQuery library module:

```
module namespace my = "http://my-corp.com/";
declare namespace srv = "http://expath.org/ns/webapp";

declare function my:servlet($req as element(srv:request), $bodies as item()*)
  as element(srv:response)
{
  <srv:response status="200" message="Ok">
    <srv:body content-type="text/plain"> {
      string-join(('Path: ', $req/srv:path/*), '')
    }
    </srv:body>
  </srv:response>
};
```

The deployment descriptor looks like:

```
<webapp xmlns="http://expath.org/ns/webapp"
         xmlns:my="http://my-corp.com/"
         name="http://my-corp.com/example"
         abbrev="my-webapp"
         version="1.0">

   <title>A sample web application</title>

   <!-- catch all requests -->
   <servlet name="my-servlet">
      <xquery function="my:servlet"/>
      <url pattern="/(.+)"/>
   </servlet>

</webapp>
```

All requests will be dispatched to the only one servlet, which will return a text/plain body with the string `Path: /some/path`, with the path requested by the client.

3.4. Future

Like Java Servlet, this is really a low-level technology in the web application development process. But because it is aimed to be independent on any specific implementation, it will allow writing separate frameworks to provide more high-level features, like general-purpose web MVC frameworks handling mapping from requests to components by using their file names for instance.

This specification is in its early stage but the current state already allows to write working, simple web applications entirely in core XML technologies. If you have any idea and want to contribute, just have a look at http://expath.org/ and join the mailing list!

4. Conclusion

The EXPath project is just one year, and it seems a lot of people are already looking forward to useful modules, following the feedback received, both from users and vendors. Its goal is not to be comprehensive nor to try to write specifications for everything possible, but instead to be a central place to coordinate the efforts to write practical extensions in a standard and collaborative way.

If you want to get more information, and maybe to join the mailing list, have a look at the website[1].

[1] http://expath.org/

“Full Impact” Schema Differencing

Anthony B. Coates
Londata Limited
<abcoates@londata.com>

Daniel Dui
Independent
<daniel.dui@gmail.com>

Abstract

For "enterprise" or other large XML schema sets, managing the changes between releases is of major importance. Normal XML differencing tools can be applied to many XML schemas, like W3C XML Schemas, but these do not give users an indication of the complete ("full") impact of a change. A change to a W3C XML Schema might be to only a single location in that Schema, but the knock-on effect of that change might affect thousands of locations. It is important to be able to report the "full impact" of any schema changes to users. This presentation discusses how XML schema differences can be calculated using an "extended XPath" approach, and will demonstrate some software for calculating these differences.

1. Introduction

Differencing tools, or "diff" tools, have become a standard part of any software developers toolbox. In particular, version control systems typically come with software to compare versions, highlighting the differences between versions. Comparing the differences between successive versions of software source code is a standard part of release testing.

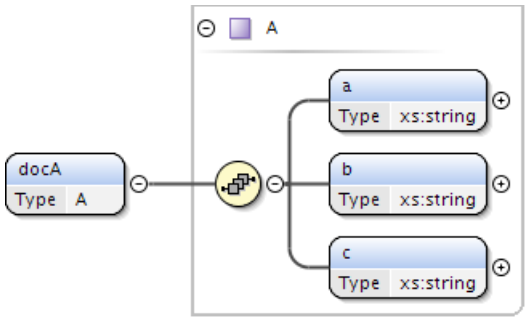
Most differencing tools are text-based, but can be used with XML if the XML has been formatted in a standard way ("pretty printed"). Additionally, there are XML-specific differencing tools. Both text and XML diff tools are used to compare (W3C) XML Schemas between versions. For large sets of XML Schemas, differencing is as important a part of release testing as it is for programming language source code, especially if there are multiple authors working on the Schemas. However, it can be difficult to judge the full impact a change to an XML Schema, e.g. a change to a complex type, by only looking at the differences between the Schema files.

This paper discusses how an "extended XPath" approach can be used to expose the full impact of changes to XML Schemas, and introduces an open source application called "XML Zebra" which can generate these paths.

2. Example

To demonstrate what pitfalls exist when looking at diffs of XML Schemas, consider the following example. Here is version 1 of a Schema:

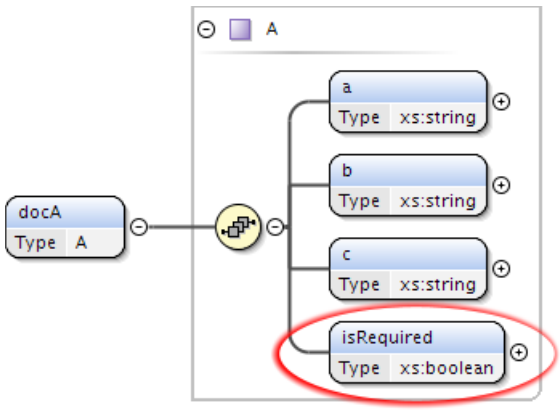
Example 1. Schema A.xsd, version 1




```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:a="http://xml.example.com/ns/A"
 xmlns="http://xml.example.com/ns/A"
        targetNamespace="http://xml.example.com/ns/A"
 elementFormDefault="qualified">
 <xs:element name="docA" type="A"/>
 <xs:complexType name="A">
  <xs:sequence>
   <xs:element name="a" type="xs:string"/>
   <xs:element name="b" type="xs:string"/>
   <xs:element name="c" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Now, here is the next version, version 2:

Example 2. Schema A.xsd, version 2



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:a="http://xml.example.com/ns/A"
 xmlns="http://xml.example.com/ns/A"
        targetNamespace="http://xml.example.com/ns/A"
 elementFormDefault="qualified">
 <xs:element name="docA" type="A"/>
 <xs:complexType name="A">
  <xs:sequence>
   <xs:element name="a" type="xs:string"/>
   <xs:element name="b" type="xs:string"/>
   <xs:element name="c" type="xs:string"/>
   <xs:element name="isRequired" type="xs:boolean"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

The difference between versions 1 and 2 of the Schema is that in version 2 the complex type "A" has had a mandatory boolean element "isRequired" appended to its content. Applying the command line "diff" command (Linux/Unix/Mac) to compare the two Schemas gives

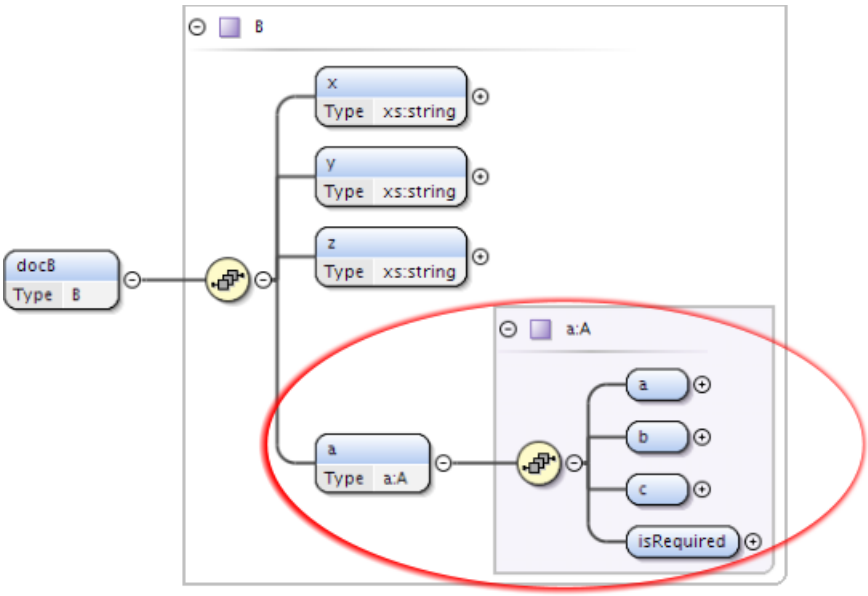
Example 3. Schema A diff result

```
10a11
>   <xs:element name="isRequired" type="xs:boolean"/>
```

This indicates that the difference from version 1 to version 2 is that a new line 11 has been added in version 2, and the new line is the `<xs:element>` definition for "isRequired". Windows users can use "fc" to do the same kind of comparison.

At this stage, you might decide that the text diff has done its job — it has correctly identified the change, and given you the information that you need. For a small enough set of XML Schemas, such simple diffs might be sufficient for comparing Schemas between releases. However, things are more complex with a larger set of Schemas. Suppose, for example, that there is a second Schema `"B.xsd"`, defined as follows:

Example 4. Schema B.xsd




```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:a="http://xml.example.com/ns/A"
 xmlns="http://xml.example.com/ns/B"
        targetNamespace="http://xml.example.com/ns/B"
 elementFormDefault="qualified" xmlns:b="http://xml.example.com/ns/B">
 <xs:import namespace="http://xml.example.com/ns/A" schemaLocation="A.xsd"/>
 <xs:element name="docB" type="B"/>
 <xs:complexType name="B">
  <xs:sequence>
   <xs:element name="x" type="xs:string"/>
   <xs:element name="y" type="xs:string"/>
   <xs:element name="z" type="xs:string"/>
   <xs:element name="a" type="a:A"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

The important thing to note is that this Schema "B.xsd" has an element "a" which uses the type "A" from the Schema "A.xsd". So the change to "A.xsd", the addition of the element "isRequired", has **changed the content model** of the type "B" in "B.xsd", even though there was **no change** to the "B.xsd" file itself.

Now, with only two small files, it's not so hard to check for something like this. However, imagine that these are just two Schemas from a large set, maybe hundreds of Schemas. Imagine that one author "A" is responsible for "A.xsd", and a different author "B" is responsible for "B.xsd". In practice, a common circumstance is that author "A" has a requirement to add an element like "isRequired" to "A.xsd" for the

direct users of "A.xsd". Author "A" may do some checking of what the impact of the change is, but doesn't check every single Schema in the set, and in particular doesn't check "B.xsd". When the Schemas are diff'd before the release, the implicit change to the content model of complex type "B" in Schema "B.xsd" goes unnoticed until some users start complaining about a new mandatory element that they didn't ask for.

3. XPath Differencing

One way to spot the unplanned change in "B.xsd" would have been to track the XPaths. Originally, the XPaths that were possible using "B.xsd" were

Example 5. Schema B.xsd - XPaths - Before

```
/b:docB
/b:docB/b:x
/b:docB/b:y
/b:docB/b:z
/b:docB/b:a
/b:docB/b:a/a:a
/b:docB/b:a/a:b
/b:docB/b:a/a:c
```

After the change to "A.xsd", the XPaths that were possible using "B.xsd" were

Example 6. Schema B.xsd - XPaths - After

```
/b:docB
/b:docB/b:x
/b:docB/b:y
/b:docB/b:z
/b:docB/b:a
/b:docB/b:a/a:a
/b:docB/b:a/a:b
/b:docB/b:a/a:c
/b:docB/b:a/a:isRequired
```

Using the Linux command line "diff" command again, the difference between the two sets of XPaths is

Example 7. Schema B.xsd - XPaths - Diff

```
8a9
> /b:docB/b:a/a:isRequired
```

Comparing the possible XPaths immediately shows that the content model of the "B" complex type changed (i.e. the content model of <b:docB> changed), even though the file "B.xsd" was unchanged.

XPath differencing can be useful, but it only tells you about certain changes. For example, it doesn't tell you anything about how the multiplicity of elements or attributes has changed. It doesn't tell you anything about how simple types or facets have changed. It doesn't tell you anything about how default or fixed values have changed. So XPaths are useful, but they aren't enough to do "full impact" differencing, i.e. to show you the full potential impact on users of a Schema change.

4. Extended XPaths

The approach that the author has used in a number of projects is to create "extended" XPaths that add the extra information needed for calculating differences between Schema versions. These paths are not strict XPaths; they take inspiration from the XPath syntax, and from here on will be referred to as "paths". The best way to demonstrate the path syntax is by looking at some examples. These examples are taken from the XML Schemas "testA.xsd" and "testB.xsd" that are listed in Appendix A.

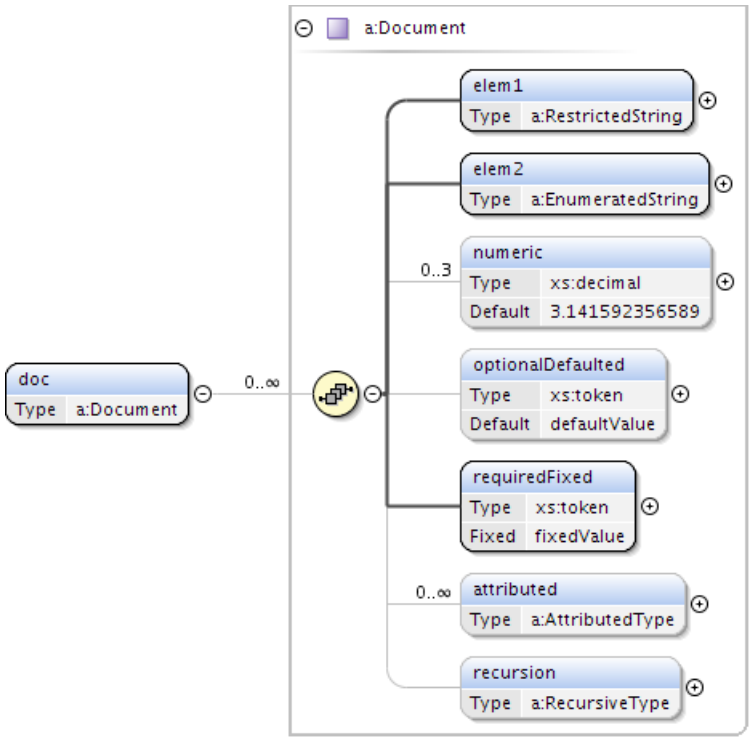
The paths that will shown have been generated using an open source tool written by the author, called "XML Zebra". It is implemented in some hundreds of lines of Scala code (http://www.scala-lang.org/), and can be run just using Java. The difficult work of parsing and combining the XML Schemas is done by using the XMLBeans "*Schema Object Model*" API (http://bit.ly/xb240sts).

Example 8. Scala Code Snippet

```
...
while (argIndex < args.length) {
  val file = new File(args(argIndex))
  if (file exists) {
    loadFile(file) // add XmlObject for parsed file to 'schemaFiles'
  }
  argIndex += 1
}
...
schemaTypeSystem = XmlBeans compileXsd (
  schemaFiles.keySet.toArray, XmlBeans.getBuiltinTypeSystem(), null
);
...
```

Now to the XML Schema examples, and the paths generated from them.

Example 9. /a:doc element from testA.xsd



`/a:doc`
: Top-level (root) element. This is the usual XPath.

`/a:doc/xs:sequence*`
: Path that indicates that `<a:doc>` contains an optional, repeatable (unbounded) sequence ("*" means "0..unbounded"). It is important that sequences/choices/etc. are tracked in the paths, as a change to a sequence/choice/etc. (or to the multiplicity) is a significant change for a user.

`/a:doc/xs:sequence*/a:elem1[base=xs:string]`
: Path for the element `<a:elem1>` in the diagram. Note that the name of the complex type for `<a:doc>` ("`a:Document`") does not appear in the path, as it is not significant for users. Changing the name of a complex type does not change whether an XML document is valid or not with respect to the Schema (in general). The name of the simple type for <a:elem1> also does not appear. However, the name of the XML Schema base type "xs:string" appears, since a change of data type is a significant change for a user.

`/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:minLength/@value=3`
`/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:maxLength/@value=5`
: These paths indicate that "`minLength`" and "`maxLength`" facets (restrictions) apply to `<a:elem1>`. The simple type name does not appear in the path, as it is not significant for users. Changes to facets are significant, because they affect what values will or will not be valid.

Note

<a:elem1> should also have a "pattern" facet, but there is an unresolved issue on how to extract pattern information using the XMLBeans API.

/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum1
/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum2
/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum3

These paths for <a:elem2> indicate the enumeration facets (allowed values) that apply to it. Again, these are significant because they affect what values will or will not be valid.

/a:doc/xs:sequence*/a:numeric[base=xs:decimal][default=3.141592356589][0..3]

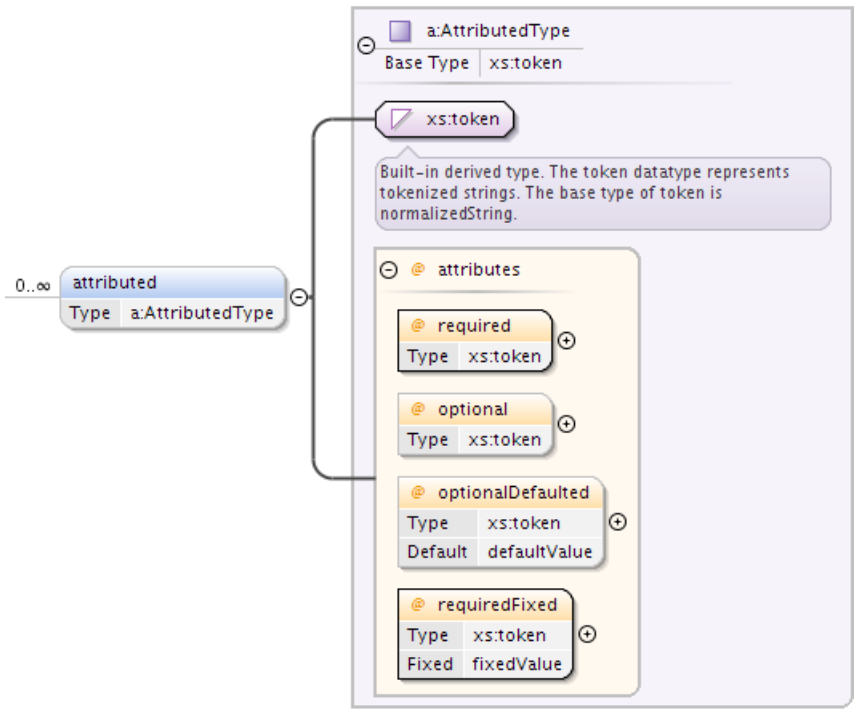
This path shows that <a:numeric> has "xs:decimal" as its base type, and has a default value. It also shows that it can occur zero to three times.

/a:doc/xs:sequence*/
a:optionalDefaulted[base=xs:string][default=defaultValue]?
/a:doc/xs:sequence*/a:optionalDefaulted[xsi:type=a:AttributedType]?
/a:doc/xs:sequence*/a:requiredFixed[base=xs:string][fixed=fixedValue]
/a:doc/xs:sequence*/a:requiredFixed[xsi:type=a:AttributedType]

<a:optionalDefaulted> is an optional element ("?" indicates optional, equivalent to "0..1") with a default value of "defaultValue". Note that there are 2 paths for <a:optionalDefaulted>. This is because it is possible to use an "xsi:type" extension with it, using "a:AttributedType" as the type instead of "xs:token". Users who are **not** using "xsi:type" extension can filter out such paths.

<a:requiredFixed> is a mandatory element with a fixed value of "fixedValue". The default multiplicity in paths is "1", so the multiplicity is never shown for a mandatory element. Again, a possible "xsi:type" extension path exists.

Example 10. /a:doc/a:attributed element from testA.xsd



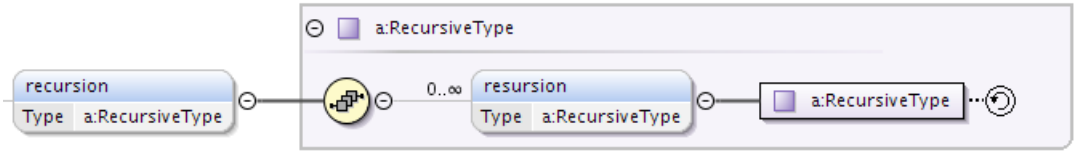

```
/a:doc/xs:sequence*/a:attributed[base=xs:string]*
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@optional[base=xs:string]?
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/
@optionalDefaulted[base=xs:string][default=defaultValue]?
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@required[base=xs:string]
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/
@requiredFixed[base=xs:string][fixed=fixedValue]
```

These paths indicate that <a:attributed> has "xs:string" as its base type, and it is optional and repeatable (unbounded). It also has attributes, which have their own paths.

"optional" is an attribute with a base type of "xs:string", and it is optional (indicated by "?"). "optionalDefaulted" is similar to "optional", but it has a default value of "defaultValue".

"required" is a mandatory (required) attribute. The default multiplicity in paths is "1", so the multiplicity is never shown for a mandatory attribute. "requiredFixed" is similar to "required", but it has a fixed value of "fixedValue".

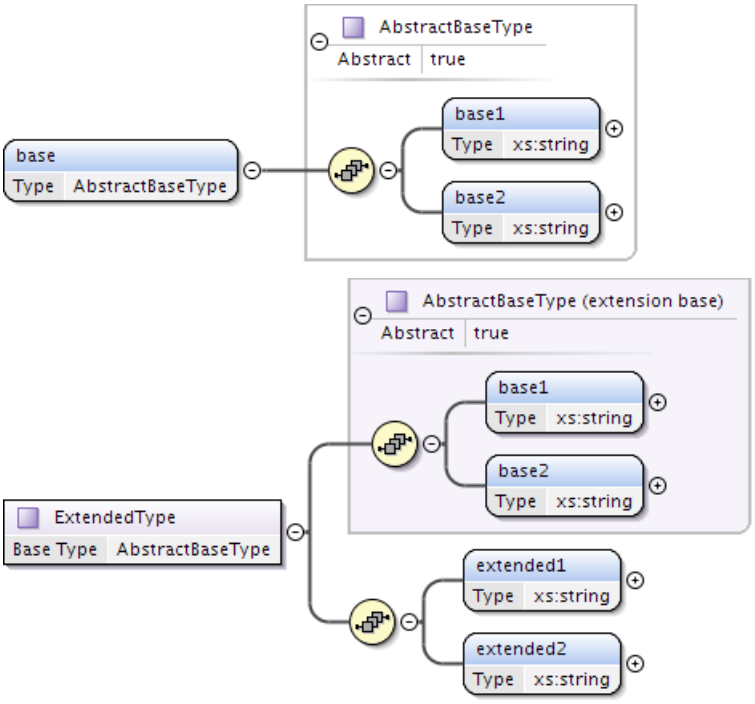
Example 11. /a:doc/a:recursion element from testA.xsd



`/a:doc/xs:sequence*/a:recursion?`
`/a:doc/xs:sequence*/a:recursion?/a:resursion*`
`/a:doc/xs:sequence*/a:recursion?/a:resursion*/...[recursion=a:RecursiveType]`
`<a:recursion>` has a recursive complex type "`a:RecursiveType`". A path is shown for `<a:recursion>`, and a 2nd path is shown for `<a:recursion>` as a child of itself. At that point, the path generator terminates the recursion, and adds an extra "..." path to indicate the recursion. So that the user can establish which part of the path is recursive, the name of the recursive complex type is included in the path (indirect recursions are handled as well as direct recursions).

Now to the second example Schema:

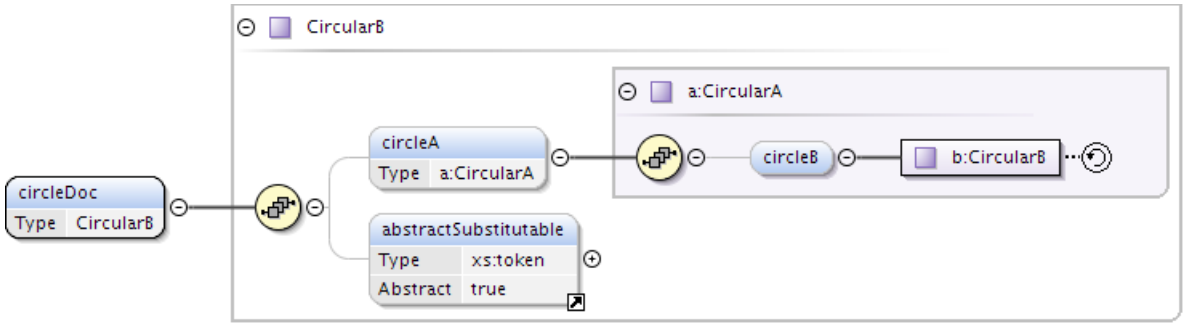
Example 12. /b:base element and b:ExtendedType complex type from testB.xsd



```
/b:base[xsi:type=b:ExtendedType]
/b:base[xsi:type=b:ExtendedType]/b:base1[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:base2[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:extended1[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:extended2[base=xs:string]
```

The <b:base> element has an abstract type "b:AbstractBaseType", so it can only be used when its type is extended using "xsi:type". The path generator does not generate paths for abstract types, only for concrete types. However, the path generator includes all possible "xsi:type" extensions in the paths. "b:AbstracBaseType" is extended by the context type "b:ExtendedType", and only by that type, so all of the <b:base> paths have "[xsi:type=b:ExtendedType]" in them. When extended, <b:base> has 4 possible child elements, as shown in the paths; 2 from "b:AbstractBaseType" and 2 from "b:ExtendedType".

Example 13. /b:circleDoc element from testB.xsd




```
/b:circleDoc
/b:circleDoc/b:circleA?
/b:circleDoc/b:circleA?/a:circleB?
/b:circleDoc/b:circleA?/a:circleB?/...[recursion=b:CircularB]
```

In this final path example, <b:circleDoc> has the type "b:CircularB" which contains the element <b:circleA> which has the type "a:CircularA" which contains the element <a:circleB> which has the type "b:CircularB". That is to say, there is a circular dependency between the complex types "b:CircularB" and "a:CircularA". These types are in different Schemas with different namespaces. As with the previous recursion example, the path generator detects the indirect circular recursion and generates a path which shows the user the complex type where the circular recursion was first detected ("b:CircularB").

Note that while the example Schema includes a substitution group example, substitution groups are not handled at the time of writing; support is planned for a future update.

5. Using Extended XPaths in Practice

In practice, the XML Zebra tool is used to generate paths for each version of a set of XML Schemas. The tool generates an XML format, show in Appendix B. This XML file is a "*fingerprint*" of the contents of the Schemas, one that can compared between versions. Some tools which can compare XML files, in an XML-aware way, are Altova's "XML Spy" and "DiffDog", and the "oXygen" XML editor. Additionally, the IBM Rational "ClearCase" version control system comes with an XML differencing tool.

However, these XML fingerprint files can be large. For one enterprise set of Schemas that was tested (but cannot be identified here), the XML path file was some **200M**. This can be difficult for diff tools to work with, especially XML diff tools which need to load both files entirely in order to compare them. For that reason, it is sometimes easier to work with a text file of the paths. The XQuery in same appendix can be used to extract the paths as a text file, and the result is also shown in the same appendix. This textual format can be compared using any standard text diff tool, and compares nearly all of the information that the XML format contains. Typically, the text file is 1/3 to 1/2 the size of the XML file.

Doing a diff between 2 such large text files can produce a huge number of results, if there are more than a few differences. One way to reduce the number of differences that are presented to the user is to programmatically remove all of the difference paths that start with another difference path, so that only "*top-level*" difference paths remain. For example, if you change the name of a root element, all of the paths under it will change. In that case, it is only important to show the change in the root element, the other changes are only change to the paths, not changes to the content under the root document.

However, reducing the paths in this way can have its drawbacks too, since you might hide a change at a lower-level in the document. If you have change not only the name of the root element, but the names of many other elements in the document, then showing only the root element path will hide important changes from the user. The point to note is that while there may be ways you can reduce the number of paths that are displayed to a user, there is no generic way that applies in all cases. It is a matter of knowing your own project and your own needs and priorities, and then deciding how best to filter your difference paths.

A method that can be helpful is to generate a file of the differences between the before and after path files, with the lines prefixed somehow to indicate what is an addition and what is a deletion. That file of differences can be presented as a tree in a GUI; in the author's experience the code to do so is comparatively simple to write (but hasn't been implemented as yet for "XML Zebra").

6. Conclusion

This paper has discussed how "extended XPaths", path expressions which borrow from XPath syntax but extend it to include extra XML Schema information, can be used to "*fingerprint*" versions of a set of XML Schemas. Those "fingerprints", when compared between versions, expose the changes between those versions more clearly than simple file-level comparisons can. Using the path format presented in this paper, path-based comparisons can capture all changes to content models, multiplicities, types and facets, both direct changes and indirect changes. File-based comparisons tend to highlight only the direct changes. In large sets of Schemas with multiple authors and shared definitions, indirect and unintended changes can have a show-stopping impact on end users. The approach in this paper was designed from the outset to be able to catch such "*hidden*" changes during release testing, protecting end users.

The "XML Zebra" application is currently available as a pre-release from http://www.xmlzebra.com/. Writing such an application is greatly simplified by using the XMLBeans Schema Object Model API. "XML Zebra" is already usable in its current form, covering all of the cases discussed in this paper. If anyone would like to contribute to its further development or testing, please get in touch with the author. Equally, if you would just like to ask questions about the tool itself, you can send an e-mail to "devteam@londata.com[1]".

A. Example XML Schemas

Example A.1. testA.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:a="http://xml.ecxample.com/ns/testA"
 xmlns:b="http://xml.ecxample.com/ns/testB"
        targetNamespace="http://xml.ecxample.com/ns/testA"
 elementFormDefault="qualified">
 <xs:import namespace="http://xml.ecxample.com/ns/testB"
                   schemaLocation="testB.xsd"/>
 <xs:complexType name="Document">
  <xs:sequence maxOccurs="unbounded" minOccurs="0">
   <xs:element name="elem1" type="a:RestrictedString"/>
   <xs:element name="elem2" type="a:EnumeratedString"/>
   <xs:element maxOccurs="3" minOccurs="0" name="numeric" type="xs:decimal"
    default="3.141592356589"/>
   <xs:element default="defaultValue" minOccurs="0" name="optionalDefaulted"
                               type="xs:token"/>
```

[1] mailto:devteam@londata.com

```
   <xs:element fixed="fixedValue" name="requiredFixed" type="xs:token"/>
   <xs:element maxOccurs="unbounded" minOccurs="0" name="attributed" ▶
type="a:AttributedType"/>
   <xs:element minOccurs="0" name="recursion" type="a:RecursiveType"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="doc" type="a:Document"/>
 <xs:simpleType name="RestrictedString">
  <xs:restriction base="xs:string">
   <xs:minLength value="3"/>
   <xs:maxLength value="5"/>
   <xs:pattern value="[A-Z]{3}[A-Z0-9]{0,2}"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="EnumeratedString">
  <xs:restriction base="xs:string">
   <xs:enumeration value="enum1"/>
   <xs:enumeration value="enum2"/>
   <xs:enumeration value="enum3"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:complexType name="RecursiveType">
  <xs:sequence>
   <xs:element maxOccurs="unbounded" minOccurs="0" name="resursion" ▶
type="a:RecursiveType"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="CircularA">
  <xs:sequence>
   <xs:element minOccurs="0" name="circleB" type="b:CircularB"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="AttributedType">
  <xs:simpleContent>
   <xs:extension base="xs:token">
    <xs:attribute name="required" type="xs:token" use="required"/>
    <xs:attribute name="optional" type="xs:token"/>
    <xs:attribute default="defaultValue" name="optionalDefaulted" ▶
type="xs:token"/>
    <xs:attribute fixed="fixedValue" name="requiredFixed" type="xs:token" ▶
use="required"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:schema>
```

Example A.2. testB.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://xml.ecxample.com/ns/testB"
 targetNamespace="http://xml.ecxample.com/ns/testB"
        xmlns="http://xml.ecxample.com/ns/testB"
 elementFormDefault="qualified" xmlns:a="http://xml.ecxample.com/ns/testA">
 <xs:import namespace="http://xml.ecxample.com/ns/testA" ▶
schemaLocation="testA.xsd"/>
 <xs:complexType name="CircularB">
  <xs:sequence>
   <xs:element minOccurs="0" name="circleA" type="a:CircularA"/>
   <xs:element minOccurs="0" ref="abstractSubstitutable"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="circleDoc" type="CircularB"/>
 <xs:complexType name="AbstractBaseType" abstract="true">
  <xs:sequence>
   <xs:element name="base1" type="xs:string"/>
   <xs:element name="base2" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="ExtendedType">
  <xs:complexContent>
   <xs:extension base="AbstractBaseType">
    <xs:sequence>
     <xs:element name="extended1" type="xs:string"/>
     <xs:element name="extended2" type="xs:string"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:element name="base" type="AbstractBaseType"/>
 <xs:element name="abstractSubstitutable" type="xs:token" abstract="true"/>
 <xs:element name="substitutionA" substitutionGroup="abstractSubstitutable" ▶
type="xs:token"/>
 <xs:element name="substitutionB" substitutionGroup="abstractSubstitutable" ▶
type="xs:token"/>
</xs:schema>
```

B. Example Path Results

Example B.1. Combined Paths for testA.xsd and testB.xsd (XML format)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsp:schemaPaths xmlns:xsp="http://xml.londata.com/2009/ns/schemaPath"
 xmlns:xs="http://www.w3.org/2001/XMLSchema" ▶
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <xsp:ns prefix="xs">http://www.w3.org/2001/XMLSchema</xsp:ns>
 <xsp:ns prefix="xsi">http://www.w3.org/2001/XMLSchema-instance</xsp:ns>
 <xsp:ns prefix="a">http://xml.ecxample.com/ns/testA</xsp:ns>
 <xsp:ns prefix="b">http://xml.ecxample.com/ns/testB</xsp:ns>
 <xsp:element name="a:doc" path="/a:doc">
  <xsp:sequence minOccurs="0" maxOccurs="unbounded" path="/a:doc/xs:sequence*">

   <xsp:element name="a:elem1" base="xs:string"
    path="/a:doc/xs:sequence*/a:elem1[base=xs:string]">
    <xsp:minLength value="3"
    path="/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:minLength/@value=3"/>

    <xsp:maxLength value="5"
    path="/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:maxLength/@value=5"/>

   </xsp:element>
   <xsp:element name="a:elem2" base="xs:string"
    path="/a:doc/xs:sequence*/a:elem2[base=xs:string]">
    <xsp:enumeration value="enum1"

path="/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum1"/>

    <xsp:enumeration value="enum2"

path="/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum2"/>

    <xsp:enumeration value="enum3"

path="/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum3"/>

   </xsp:element>
   <xsp:element name="a:numeric" minOccurs="0" maxOccurs="3" ▶
default="3.141592356589"
    base="xs:decimal"

path="/a:doc/xs:sequence*/a:numeric[base=xs:decimal][default=3.141592356589][0..3]"/>

   <xsp:element name="a:optionalDefaulted" minOccurs="0" default="defaultValue" ▶
```

```
base="xs:string"

path="/a:doc/xs:sequence*/a:optionalDefaulted[base=xs:string][default=defaultValue]?"/>

   <xsp:element name="a:optionalDefaulted" xsi:type="a:AttributedType" ▶
minOccurs="0"
    path="/a:doc/xs:sequence*/a:optionalDefaulted[xsi:type=a:AttributedType]?"/>

   <xsp:element name="a:requiredFixed" fixed="fixedValue" base="xs:string"

path="/a:doc/xs:sequence*/a:requiredFixed[base=xs:string][fixed=fixedValue]"/>

   <xsp:element name="a:requiredFixed" xsi:type="a:AttributedType"
    path="/a:doc/xs:sequence*/a:requiredFixed[xsi:type=a:AttributedType]"/>
   <xsp:element name="a:attributed" minOccurs="0" maxOccurs="unbounded" ▶
base="xs:string"
    path="/a:doc/xs:sequence*/a:attributed[base=xs:string]*">
    <xsp:attribute name="optional" use="optional" base="xs:string"

path="/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@optional[base=xs:string]?"/>

    <xsp:attribute name="optionalDefaulted" use="optional" default="defaultValue"

     base="xs:string"

path="/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@optionalDefaulted[base=xs:string][default=defaultValue]?"/>

    <xsp:attribute name="required" use="required" base="xs:string"

path="/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@required[base=xs:string]"/>

    <xsp:attribute name="requiredFixed" use="required" fixed="fixedValue" ▶
base="xs:string"

path="/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@requiredFixed[base=xs:string][fixed=fixedValue]"

    />
   </xsp:element>
   <xsp:element name="a:recursion" minOccurs="0" ▶
path="/a:doc/xs:sequence*/a:recursion?">
    <xsp:element name="a:resursion" minOccurs="0" maxOccurs="unbounded"
     path="/a:doc/xs:sequence*/a:recursion?/a:resursion*">
     <xsp:recursion type="a:RecursiveType"

path="/a:doc/xs:sequence*/a:recursion?/a:resursion*/...[recursion=a:RecursiveType]"/>
```

```
    </xsp:element>
   </xsp:element>
  </xsp:sequence>
 </xsp:element>
 <xsp:element name="b:base" xsi:type="b:ExtendedType" ▶
path="/b:base[xsi:type=b:ExtendedType]">
  <xsp:sequence path="/b:base[xsi:type=b:ExtendedType]">
   <xsp:element name="b:base1" base="xs:string"
    path="/b:base[xsi:type=b:ExtendedType]/b:base1[base=xs:string]"/>
   <xsp:element name="b:base1" xsi:type="a:AttributedType"
   path="/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:AttributedType]"/>

   <xsp:element name="b:base1" xsi:type="a:EnumeratedString"

path="/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:EnumeratedString]"/>

   <xsp:element name="b:base1" xsi:type="a:RestrictedString"

path="/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:RestrictedString]"/>

   <xsp:element name="b:base2" base="xs:string"
    path="/b:base[xsi:type=b:ExtendedType]/b:base2[base=xs:string]"/>
   <xsp:element name="b:base2" xsi:type="a:AttributedType"
   path="/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:AttributedType]"/>

   <xsp:element name="b:base2" xsi:type="a:EnumeratedString"

path="/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:EnumeratedString]"/>

   <xsp:element name="b:base2" xsi:type="a:RestrictedString"

path="/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:RestrictedString]"/>

   <xsp:element name="b:extended1" base="xs:string"
    path="/b:base[xsi:type=b:ExtendedType]/b:extended1[base=xs:string]"/>
   <xsp:element name="b:extended1" xsi:type="a:AttributedType"

path="/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:AttributedType]"/>

   <xsp:element name="b:extended1" xsi:type="a:EnumeratedString"

path="/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:EnumeratedString]"/>

   <xsp:element name="b:extended1" xsi:type="a:RestrictedString"

path="/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:RestrictedString]"/>
```

```
   <xsp:element name="b:extended2" base="xs:string"
    path="/b:base[xsi:type=b:ExtendedType]/b:extended2[base=xs:string]"/>
   <xsp:element name="b:extended2" xsi:type="a:AttributedType"

path="/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:AttributedType]"/>

   <xsp:element name="b:extended2" xsi:type="a:EnumeratedString"

path="/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:EnumeratedString]"/>

   <xsp:element name="b:extended2" xsi:type="a:RestrictedString"

path="/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:RestrictedString]"/>

  </xsp:sequence>
 </xsp:element>
 <xsp:element name="b:circleDoc" path="/b:circleDoc">
  <xsp:sequence path="/b:circleDoc">
   <xsp:element name="b:circleA" minOccurs="0" path="/b:circleDoc/b:circleA?">

    <xsp:element name="a:circleB" minOccurs="0" ▶
path="/b:circleDoc/b:circleA?/a:circleB?">
     <xsp:recursion type="b:CircularB"
      path="/b:circleDoc/b:circleA?/a:circleB?/...[recursion=b:CircularB]"/>
    </xsp:element>
   </xsp:element>
  </xsp:sequence>
 </xsp:element>
 <xsp:element name="b:substitutionA" base="xs:string" ▶
path="/b:substitutionA[base=xs:string]"/>
 <xsp:element name="b:substitutionA" xsi:type="a:AttributedType"
  path="/b:substitutionA[xsi:type=a:AttributedType]"/>
 <xsp:element name="b:substitutionB" base="xs:string" ▶
path="/b:substitutionB[base=xs:string]"/>
 <xsp:element name="b:substitutionB" xsi:type="a:AttributedType"
  path="/b:substitutionB[xsi:type=a:AttributedType]"/>
</xsp:schemaPaths>
```

Example B.2. XQuery script to extract paths from XML format into (unsorted) text format

```
xquery version "1.0";

declare variable $eol := "&#xA;";

let $paths := for $path in //*/@path return string($path)
```

```
for $path in distinct-values($paths)
(:order by $path:)
return concat($eol, $path)
```

Example B.3. Combined Paths for testA.xsd and testB.xsd (unsorted, text format)

```
/a:doc
/a:doc/xs:sequence*
/a:doc/xs:sequence*/a:elem1[base=xs:string]
/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:minLength/@value=3
/a:doc/xs:sequence*/a:elem1[base=xs:string]/xs:maxLength/@value=5
/a:doc/xs:sequence*/a:elem2[base=xs:string]
/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum1
/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum2
/a:doc/xs:sequence*/a:elem2[base=xs:string]/xs:enumeration/@value=enum3
/a:doc/xs:sequence*/a:numeric[base=xs:decimal][default=3.141592356589][0..3]
/a:doc/xs:sequence*/a:optionalDefaulted[base=xs:string][default=defaultValue]?
/a:doc/xs:sequence*/a:optionalDefaulted[xsi:type=a:AttributedType]?
/a:doc/xs:sequence*/a:requiredFixed[base=xs:string][fixed=fixedValue]
/a:doc/xs:sequence*/a:requiredFixed[xsi:type=a:AttributedType]
/a:doc/xs:sequence*/a:attributed[base=xs:string]*
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@optional[base=xs:string]?
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@optionalDefaulted[base=xs:string][default=defaultValue]?
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@required[base=xs:string]
/a:doc/xs:sequence*/a:attributed[base=xs:string]*/@requiredFixed[base=xs:string][fixed=fixedValue]
/a:doc/xs:sequence*/a:recursion?
/a:doc/xs:sequence*/a:recursion?/a:resursion*
/a:doc/xs:sequence*/a:recursion?/a:resursion*/...[recursion=a:RecursiveType]
/b:base[xsi:type=b:ExtendedType]
/b:base[xsi:type=b:ExtendedType]/b:base1[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:AttributedType]
/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:EnumeratedString]
/b:base[xsi:type=b:ExtendedType]/b:base1[xsi:type=a:RestrictedString]
/b:base[xsi:type=b:ExtendedType]/b:base2[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:AttributedType]
/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:EnumeratedString]
/b:base[xsi:type=b:ExtendedType]/b:base2[xsi:type=a:RestrictedString]
/b:base[xsi:type=b:ExtendedType]/b:extended1[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:AttributedType]
/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:EnumeratedString]
/b:base[xsi:type=b:ExtendedType]/b:extended1[xsi:type=a:RestrictedString]
/b:base[xsi:type=b:ExtendedType]/b:extended2[base=xs:string]
/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:AttributedType]
/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:EnumeratedString]
/b:base[xsi:type=b:ExtendedType]/b:extended2[xsi:type=a:RestrictedString]
/b:circleDoc
/b:circleDoc/b:circleA?
```

```
/b:circleDoc/b:circleA?/a:circleB?
/b:circleDoc/b:circleA?/a:circleB?/...[recursion=b:CircularB]
/b:substitutionA[base=xs:string]
/b:substitutionA[xsi:type=a:AttributedType]
/b:substitutionB[base=xs:string]
/b:substitutionB[xsi:type=a:AttributedType]
```

Tracking Changes: Technical and UX Challenges

Laurens Van den Oever
Xopus BV
<laurens@xopus.com>

1. Intro

As XML editor vendor, we're thoroughly familiar with the intricate details of tracking of changes in an XML life cycle. In this session we will show how change tracking differs from comparing versions of a document. We will discuss the complex issues we have to deal with when change tracking is combined with structure and real time XML validation. Finally we will discuss the more exotic requirements that some our customers have.

2. Background

There are roughly three reasons why people want to know the changes made to a document between two stages in its life cycle:

- Collaborating while editing
- Reviewing changes before approval
- Auditing changes

These options each have their own technical and user experience challenges.

2.1. Collaborating

When multiple authors need to collaborate on one piece of information, showing the changes is a communication mechanism. The authors need to comment on changes made and maybe suggest alternatives.

2.2. Reviewing

A lot of information needs to be approved before it can be published. Showing changes is a tool to make the approval process more efficient. The editor needs to have an overview of the changes made rather than very detailed insight into the changes.

The original author may not need to see the changes made. The editor can communicate to the author if the changes are not approved, but typically not as fine grained as when multiple authors are collaborating.

2.3. Auditing

In some environments there is a legal requirement to be able to recover detailed information about who wrote which text and when. Maybe even to the extend that all deleted text needs to be retained and the complete process that lead to a particular text including intermediate versions needs to be stored.

3. Tracking vs Comparing

There are two approaches to get the changes between two versions of a document:

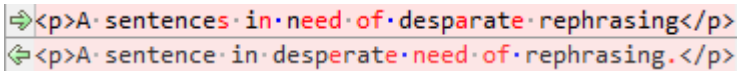
- Track changes while editing
- Compare two documents

Change tracking is the method of choice when there is the requirement to see the changes made by a larger number of people (in many consecutive document versions).

Comparing allows for a version tree, whereas tracking of changes will only work in a linear work flow.

The compare option is less granular and can result in very confusing results. Better algorithms may yield better results, but it is (still?) impossible for a diff algorithm to understand the logic behind a change and present it in a way that makes sense for a human in all cases.

Example 1. Diffed text change



Example 2. Tracked text change

A sentences in desperate need of ~~desparate~~ rephrasing.

In this example the author moved the word “desperate” in the sentence. Technically that is equivalent with moving “need of” in the opposite direction. People would rarely change the sentence in that way because they feel “desperate” is in the incorrect position. Presenting the change as a move of “need of” will be confusing.

Comparing may not work well when the goal is collaborating, for an audit trail it may be too coarse.

3.1. Applying changes

Both track changes and document compare have a mechanism to combine changes into a final document. For document compare there exist merge tool which allow the user to pick from 2 or 3 versions of a changed fragment.

In an environment with tracked changes, the interface typically offers accept and reject functionality to apply or remove each individual change.

4. Tracking vs Validation

When tracking changes while editing a lot of information is gathered. That meta information needs to be stored in the document in some way. The two main approaches are:

- Store changes in processing instructions
- Store changes in elements

In both cases, changes to attributes are typically stored in attributes.

The usage of processing instructions will not break validation as the processing instructions are treated as whitespace by the XML validator. The downside of processing instructions is that marking an insert can only be done with two matching processing instructions. There is no guarantee that that relation will not be broken by other XML processing tools.

Example 3. Inserted text marked with processing instructions

```
<p>A<?begin?> new<?end?> sentence</p>
```

The usage of elements will use the inherent structure of XML to maintain integrity. But at a cost; introducing new elements will interfere with XML validation.

Example 4. Inserted text marked with element

```
<p>A<ct:add> new</ct:add> sentence</p>
```

In this example the validator will encounter an unknown `ct:add` element. Validation will be even harder when you consider fixed sequence content models.

Example 5. Insert in sequence

```
<meta>
  <creationDate/>
  <ct:add><description/></ct:add>
  <keywords/>
</meta>
```

Here the validator needs to confirm the position of the `description` element within the `meta` element even though it is wrapped in a `ct:add` element.

5. Real vs Perceived Changes

Not all changes are atomic. Some operations may be a single click or key press for the author, but may result in a number of changes in the XML. The simplest example of that is splitting an element:

Example 6. Splitting an element

```
<p>Some sentence on no particular subject. |Followed by another random ▶
sentence.</p>

<p>Some sentence on no particular subject. <ct:split1/></p>
<p><ct:split2/>Followed by another random sentence.</p>
```

In this example a single user action resulted in multiple change tracking markup elements. In reality these need to have an id as well in order to differentiate between multiple split actions.

Advanced XML editors have a DOM API which can be used to update the XML following a change. Consider splitting a list item in a contract where item numbers must be part of the content.

Example 7. Splitting a list item

```
<list>
  <item>
    <nr>1.</nr>
    <para>An item</para></item>
  <item>
    <nr>2.</nr>
    <para>Another item|</para></item>
  <item>
    <nr>3.</nr>
    <para>Last item</para></item>
</list>

<list>
  <item>
    <nr>1.</nr>
    <para>An item</para></item>
  <item>
    <nr>2.</nr>
    <para>Another item<ct:split1 id="a"/></para>
```

```
      <ct:split1 id="b"/></item>
    <item>
      <ct:split2 id="b"/>
      <ct:add><nr>3.</nr></ct:add>
      <para><ct:split2 id="a"/>|</para></item>
    <item>
      <nr><ct:add>4.</ct:add><ct:del>3.</ct:del></nr>
      <para>Last item</para></item>
  </list>
```

In this example simply pressing enter resulted in no less than 7 change tracking elements. Xopus has a `makeValid` DOM function which can result in even more markup.

This markup has to be represented to the author in a logical user interface. This can either be done as multiple consecutive atomic changes which is can be very confusing. Or grouped per user action.

In either case multiple changes can result in changes that are interdependent and can only be accepted/rejected in a certain order. No one seems to have solved that problem.

Another point of interest is changing of deleted nodes. This is considered user friendly as the author can add text in a position they consider logical, but that may be red lined. MS Word will allow this.

Example 8. Deleted text

This is the ~~original~~ sentence.

Example 9. Added text in deleted text

This is the ~~orignewinal~~ sentence.

Accept and reject becomes ambiguous in this case and even Word can get confused and throw away the new text when a delete is accepted. So this behavior should be prevented at all times.

6. Changes & WYSIWYG

A key challenge with change tracking is representing the change in a WYSIWYG environment. Just as you don't want to change the DTD/XSD to accommodate for change tracking markup, you also don't want to change and maintain the CSS/XSL style sheets to render the change tracking markup in the WYSIWYG view of the editor.

A workaround to that problem is comparing the styled output of the XML publishing engine. That has a few problems.

- Changes to hidden content (meta data) can not be displayed
- Output from the style sheet (auto numbers, static headers) will be treated as content
- Operations like sorting and multiple rendering (table of contents) interfere

Other environments require changes to the style sheets. That may not be a big deal for simple text changes, but in a complex WYSIWYG environment it is very hard to handle all variations of changes. Also rendering the changes in a way that doesn't interfere with the possibly complex output of an XSL style sheet is very challenging even for die hard HTML/CSS experts.

Xopus will maintain the change tracking information independently of the XML content. After the XSL is applied, Xopus will render change tracking annotations in a layer over the published content. That way the output is still WYSIWYG and there are no changes needed in the style sheets.

The downside is that this method will only allow rendering of what Word calls: the final view without markup. This means that deleted content can not be rendered (red lined).

7. Exotic Requirements

On top of everything discussed earlier, some clients have additional requirements. Usually these depend on the type of markup they have.

In some cases every detailed change need to be reviewed, including changes in whitespace and punctuation. With standard annotation methods, those changes are very hard to see. In the traditional paper review cycle there is specific markup for changes like that. It would be very helpful to introduce that in a change tracking environment.

Some tools offer the ability to add suggestions for changes. They typically have a richer set of annotations are they are not limited by most of the challenges involved with tracking of changes.

In environments where the audit trail of the content is important there sometimes is the requirement to track the rejection of a change with a reason. This is particularly complex to show in a user interface.

8. Conclusion

Tracking and representing changes is very complex and there is no one size fits all solution. Looking at particular requirements it is possible to offer a working implementation but each has its weaknesses.

We continue our research and welcome all feedback and recommendations.

Schema-aware editing

George Bina
Syncro Soft / oXygen XML Editor
<george@oxygenxml.com>

Abstract

Find how schema information can be used to improve the XML editing process. The presentation will cover techniques like content completion, automatic insertion of content, different strategies that can be applied for operations like delete, type, paste, drag and drop in trying to keep the document schema-valid and how schema information can be used for better XML formatting. Multiple schema languages will be analyzed both from a theoretical and a practical point of view.

Keywords: XML, authoring, editing, schema

1. Introduction

Parallel between Java development and XML editing.

Most Java developers when working with a Java library rely on their IDEs to present them the available classes and methods and to help them as they write their code. For some it seems impossible to work without this help as that means either looking up all the time in that library documentation or to memorize a very large amount of information that may be very time consuming or even impossible taking into account the large number of libraries one needs to use.

Similarly, for XML editing, the schema can be seen as a library containing the definitions of the available elements and attributes that can be used in an XML document. The XML editors should be able to offer similar support for the XML authors as Java IDEs do for Java developers, helping them to easily create valid XML documents.

Using schema information provides a generic mechanism that reacts automatically to schema changes.

All the main XML frameworks, DocBook, DITA and TEI do not have a fixed or closed vocabulary. They allow some degree of customization by removing some elements or adding new elements. Using the schema information to provide editing

support, as opposed to manually configuring a tool to work with a specific language, allows to automatically handle customizations/specializations and in general to be able to offer support for any language as long as its schema is known.

Schema-aware editing use cases.

Schema information can clearly help when the user is entering markup. That is not however the only use case, it can also help when formatting the document by providing information that otherwise will require an immense configuration effort and that is essential in preserving correctly the whitespaces in the document. When editing visually, without seeing the tags, the schema information becomes even more useful as it allows to provide intelligent actions that help the user keep the document valid.

2. Determine the schema

The first step in providing schema-aware editing support is to determine the schema. That is not always easy! Some schema languages specify such an association between the instance documents and schemas, others do not.

DTDs and XML Schemas specify a how they can be associated with a document.

DTDs provide means to associate a DTD with an instance document though the DOCTYPE declaration.

Example 1. DocBook 4 DTD association

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN"
    "http://www.docbook.org/xml/4.4/docbookx.dtd">
```

A special attention should be paid here to the internal DTD subset that can provide declarations in-line, inside the edited document.

W3C XML Schemas specifies that an instance document can provide hints using two special attributes from the schema instance namespace, the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes.

Example 2. Sample XML Schema associations for no namespace and for a specific namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<personnel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="personal.xsd">
...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ipo:purchaseOrder xmlns:ipo="http://www.example.com/IPO"
    orderDate="1999-12-01"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.com/IPO ipo.xsd">
...
```

Other schema languages, Relax NG, Schematron and NVDL do not specify how they can be associated with a document.

Relax NG, Schematron and NVDL do not specify any way of associating a schema with an instance document. Most applications that support them use either processing instructions or application settings to perform the association between documents and schemas, or both.

Example 3. Associating a Relax NG schema using a product specific (oXygen XML Editor) PI

```
<?xml version="1.0" encoding="UTF-8"?>
<?oxygen RNGSchema="personal.rnc" type="compact"?>
<personnel>
...
```

The associations made at application level are in general based on analyzing the header of the XML file or external information such as the file name or the extension and decide based on that what schema to use. One such example is James Clark "locating rules" configuration files used by nxml mode for Emacs to edit XML files using Relax NG schemas. Another example is the "document type associations" options from oXygen XML Editor.

Both inside document associations and external associations are useful.

Depending on the use cases both specifying the association inside the instance documents and specify that externally, at application level are useful. For example, when receiving a document that has the root element in the DocBook 5 namespace it is great if the editor uses automatically the DocBook 5 schema without requiring the user to make a specific association. However, if someone creates a DocBook document for a customized DocBook schema it is useful to be able to refer to his custom DocBook schema in the instance document so that any tool will correctly validate that against the right schema.

W3C "xml-model" processing instruction for associating schemas to XML instances will enable portability.

W3C started recently an initiative to standardize a schema association processing instruction that will allow to associate schemas to instance documents. That will enable a way to associate schemas that will be portable between different applications.

3. Content completion

This feature is present in the majority of the XML tools. It consists in offering to the users different proposals for entering elements, attributes or values by looking into the associated schema. This information can be used in different contexts, mainly at entering new data but also for renaming an element, for inserting in other locations (inside, before, after), etc.

Most tools support DTDs and XML Schemas, some support Relax NG and NVDL and as far as I know there is no tool supporting Schematron.

The two main techniques for obtaining proposals.

There are two main techniques that can be used to determine proposals. One is to validate each possible proposal in the context it will be inserted and the other is to explore the schema model to determine what it specifies. The second case also requires to determine all possible elements and attributes in order to be able to handle wildcards like *any element from a specified namespace* or *any attribute except some attributes*.

3.1. Schema languages

Each schema language is different when it comes to the context information needed to find the next elements or the attributes that can appear at a given location in the document. The complexity of the work needed to determine the proposals increases with the power of the schema language.

3.1.1. DTD

DTDs have all the elements declared globally so there cannot be two elements with the same name in the document and with different content models. That means that if we determine the name of an element then we know what elements and attributes can be inserted inside that element.

3.1.2. XML Schema

The W3C XML Schemas are a little more complicated. They allow declaring local elements that can specify a different content model for an element when used in a specific context. Also the use of xsi:type in the instance document can change the content model of an element by pointing to a derived type. Thus for XML Schema it is not sufficient to determine the parent element to know what that can contain. In this case the context information needed to get the content completion proposals is formed by all the ancestor elements up to the root together with their xsi:type attributes.

3.1.3. Relax NG

The Relax NG schemas allow maximum flexibility but in the same time it also requires more work to provide content completion support. The context in case of Relax NG schemas it the whole document up to the insertion point. For practical purposes we found that it is a good trade-off to take as context the previous elements from the insert position, the parent element and all the ancestors up to the root element, all with all their attributes and attribute values.

3.1.4. NVDL

NVDL is a special case. It is able to specify how different languages can be combined and it can use any schema type for each language. Thus its main function is to delegate to other schemas for specific sub-contexts.

3.1.5. Schematron

Schematron specifies a set of assertions and reports that are checked in given contexts, all expressed with XPath expressions. One possibility to provide content completion will be to recognize some patterns in schemas and infer from those what elements or attributes can be inserted in the document. For instance if a rule asserts the existence of a `title` element in the context of a `section` element then the content completion can propose a `title` as a possible element inside `section`.

3.2. Elements and attributes

The main information proposed by the content completion is the list of available elements and attributes.

Required and optional content generation.

Along with inserting the chosen element or attribute the editor should also be able to insert the required content. For example if an element has a required name attribute then it is really helpful if the editor inserts that along with the element. In general it should be possible to also insert optional content or follow options in choice particles but these are useful only in a limited number of use cases so they should be enabled only on user request.

Handle undeclared prefixes.

There are cases when the available elements or attributes are in a namespace that is not yet declared in the instance document. The editor should be able to handle such cases by automatically inserting the missing namespace declaration.

3.3. Values

Most editors offer value proposals for attributes. However, both XML Schema and Relax NG can define elements with simple types and thus they can have the same values as attributes.

Enumerations

DTDs, XML Schema and Relax NG allow to specify an enumeration of possible values. An editor can pick those values and present them as proposal when the user specifies an attribute or element value.

An edge case here is when the values type is QName as that needs to be obtained in the namespace context from the schema and when presented the namespace needs to be correctly mapped to the corresponding prefix from the instance document.

Default and fixed values

Default values can be offered by the content completion but in general for DTD and XML Schema they are not necessary as if that attribute is not specified then the default is used anyway, so in the majority of cases the user wants to enter a different value. Fixed values on the other hand are clearly helpful as they are the only those values can appear for that attribute or element.

Single values and list values.

The schemas can specify also that a value is a space separated list of values. In this case the editor should be able to assist the insertion of additional values.

Typed values and ID values

In case of enumerations and default or fixed values the schema specifies those values. But the schema also can specify a type for the value and that information can be also used to provide proposals to the user. For example for a `boolean` type the editor can propose `true` and `false` as possible values.

For `IDREF` and `IDREFS` types the possible values can be determined from the edited document itself. There are some issues though, because when edited the document goes though many invalid and even not well-formed states. Also, the document may link to external files through external entities or XInclude so the ID values need to be collected also from those external files.

3.4. Entities

Entities are coupled with DTDs because none of the other schema languages specifies entities or something equivalent. The user can be assisted when entering entities with the list of possible entity names and also with the expanded values of the entities.

Although entities are specific to DTDs they can be used also together with other schemas, in that case the DTD may be used only for entity resolution.

3.5. Schema annotations/comments as documentation

Coming back to the initial analogy between Java development and XML editing - a very important part of the proposals offered by a Java IDE is formed by the classes and methods documentation that is presented next to the offered proposal. Without this the content completion realizes only half of the support it can provide.

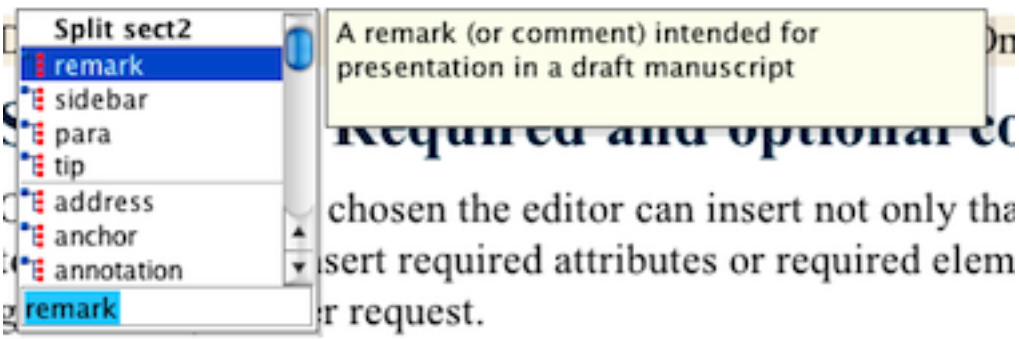


Figure 1. The documentation of the `remark` element from DocBook 5

XML Schema and Relax NG have built in annotation support. This allows to have a standard mechanism to identify the documentation for an element or attribute.

Example 4. The `remark` element definition in the DocBook 5 Relax NG schema

```
<element name="remark">
  <a:documentation>A remark (or comment) intended for presentation
    in a draft manuscript</a:documentation>
  <ref name="db.remark.attlist"/>
  <ref name="db._text"/>
</element>
```

DTDs on the other hand do not have such a mechanism. One possibility is to use the comments as documentation. A problem here is that existing DTDs have comments that do not always represent documentation, or some have the comments before the element declaration and others below. A possible solution is to specify a format for the comments that represent documentation and thus only the correct ones will be used.

4. Visual editing

The schema information is even more important in helping the visual editing, when the tags are no longer visible. It allows to make the actions more intelligent as they can use the schema information to avoid creating invalid content and to apply different recovery strategies.

4.1. Text in element only content

Accidentally typing text in empty or element only content will make the document invalid. A schema aware editor can determine that the content model of the current element is empty or element only and block that action, informing the user why the action cannot be performed. Even more, it can try to recover from that situation.

For example, let's consider a case when a `section` element has a content model like `(title, para+)` where title and para contain text. If the user types text inside an empty `section` element then a schema aware editor can automatically surround that text in a title element. Further, if the user types text after the `title` element the editor can put that automatically inside a `para` element, and so on.

4.2. Restrict editing actions

There are many actions that result in different structural changes to the document. As these may result in invalid content a schema aware editor can act before that and either present the user an explanation for why the action cannot be performed

or, similar with the case for inserting text, apply or propose different strategies to obtain a valid, or feasibly valid document. A few examples of actions that can be restricted are: split the current element, remove element tags, insert or rename element.

4.3. Smart delete

The schema information can be used to check that when deleting a content the result is invalid and in that case apply different strategies to try to make that valid. For example if there are two `para` elements containing text one after the other in a `section` element that does not allow text content and the user presses **backspace** at the beginning of the second paragraph the default action would be to delete the paragraph tags, that is have its content unwrapped. As the `section` element does not allow text that will result in a validation error. A schema-aware editor can propose to move that content in the preceding `para` element that accepts that content. This is just an example of a simple strategy but one can imagine many strategies and either present a list of choices to the user or apply the first strategy that succeeds.

4.4. Smart paste and drag and drop

The paste and drag and drop action make new content appear at some position in the document. This represents a generalization of the typing text case because these action contain a document fragment. Also the strategies that can be applied to recover after a detection of an invalid state have more freedom as the content may be moved also to adjacent positions in the document or they can decide to split the current element, etc.

Let us consider again a structure when a `section` element can contain a `title` followed by one or more `para` elements. In an existing section that has a title and several paragraphs we paste a title in the middle of a paragraph at the location indicated in the example below with a `|` mark.

```
<section>
  <title>My section title</title>
  <para>Some description</para>
  <para>Another | description</para>
</section>
```

A non schema aware editor will just place that element there, resulting

```
<section>
  <title>My section title</title>
  <para>Some description</para>
  <para>Another <title>New title</title> description</para>
</section>
```

A schema aware editor can apply a split strategy and determine that the title will be valid if the split is performed at section level and obtain

```
<section>
  <title>My section title</title>
  <para>Some description</para>
  <para>Another </para>
</section>
<section>
  <title>New title</title>
  <para> description</para>
</section>
```

As you can see from this simple example the schema aware editing opens the door for a lot of opportunities to improve the editing experience.

5. Formatting (pretty print)

The schema information can be used also when formatting a document. Of course the schema information is not the only information that is taken into account during formatting, other sources can be the document content, information from an associated CSS stylesheet and user defined/application options.

One important information that can be found looking into the schema is the presence of default attributes and in particular the presence of an `xml:space` attribute with a default or fixed value of `preserve` or `default`.

Another type of information is related with the content model of an element. Let's assume a sample document like below:

```
<?xml version="1.0" encoding="UTF-8"?>
<p><b>G</b><i>eorge</i> is my first name.</p>
```

If a format and indent action is applied on the document it will determine that the `p` element contains both elements and text and it will never add whitespaces between the `b` and `i` elements.

Now, let's look into a slightly different example:

```
<?xml version="1.0" encoding="UTF-8"?>
<p><b>G</b><i>eorge</i></p>
```

In this case a document analysis will see that the `p` element contains only elements and will format and indent that as

```
<?xml version="1.0" encoding="UTF-8"?>
<p>
  <b>G</b>
  <i>eorge</i>
</p>
```

This will result in adding a space between `G` and `eorge` and a further rendering will show that as "**G** *eorge*" instead of "**G***eorge*"

If the format and indent action will look also in the associated schema it can find that the `entry` element allows also text (has a mixed content model) and that will mean that whitespaces are important so it will not add a space between the two child elements.

6. Validation

Continuous validation.

As you type validation, aka continuous validation is another technique that uses the schema information to provide feedback to the user during editing. This differs from the normal validation action, performed on user request, in the way the validation errors are displayed to the user. Most editors show in document error markers, highlighting the document area that the error relates to.

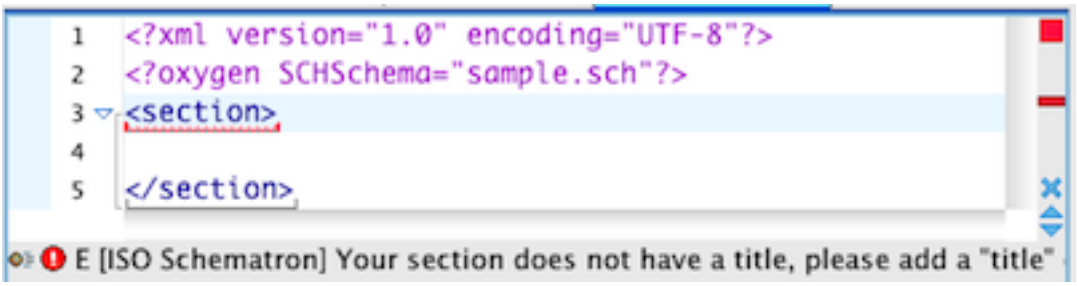


Figure 2. Sample continuous validation markers

Schematron error messages are specified by the schema author.

Although there is no editor to support Schematron for content completion and related functionality the Schematron schemas show their advantages when it comes to validation. Not only that they can check constraints that cannot be checked in other schema languages but the error messages are written in the schema by the schema author. Thus they are not generic errors like *required elements are missing* or similar, but they can be written in a language the user understands and can also point the user to possible solutions to resolve that error.

Let us take a look at an error when we have a missing `title` element in a `section`. The following DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT section (title, para+)>
```

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT para (#PCDATA)>
```

will give an error message (using Xerces) as

```
The content of element type "section" is incomplete,
it must match "(title,para+)".
```

This is a very clear message for a user familiar with XML and DTD content models but not ideal for a beginner. A Schematron schema like below

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern id="checkTitle">
    <rule context="section">
      <assert test="title">Your section does not have a title,
        please add a "title" element at the beginning of the
        section!</assert>
    </rule>
  </pattern>
</schema>
```

will give an error message written by the schema author in a language that the specific audience understands easily

```
Your section does not have a title, please add a "title" element
at the beginning of the section!
```

Quick fixes - automatic correction of reported errors.

A schema aware editor can recognize specific error messages and offer support to automatically fix the reported validation problems. For example it can recognize an error that complains about a required missing attribute and offer as quick fix proposal to add that attribute.

7. Conclusions

The schema information helps dramatically in speeding up the editing process and avoiding getting into invalid states thus creating a better editing experience for both advanced users and beginners.

Tools should try to take advantage of schema information in all operations, not only to offer content completion.

There are cases like the format and indent when the schema information provides critical information that allows the formatter to correctly deal with whitespaces.

How to avoid suffering from markup: A project report about the virtue of hiding XML

Felix Sasaki
University of Applied Sciences Potsdam
<felix.sasaki@fh-potsdam.de>

Abstract

This paper describes the development of a specialized RELAX NG schema for XHTML and a related XSLT processing chain. This development is the XML-related outcome of a markup project. The non-XML-related outcome are two documents about Japanese layout, one in English, one in Japanese. The paper focuses on the interplay between markup constraints and social constraints, and demonstrates the virtue of hiding XML, for fostering its adoption in new communities.

Keywords: Validation, XSLT, Translation, Localization

1. Introduction

This paper describes the development of a specialized RELAX NG schema, using a methodology[1] described in the RELAX NG Tutorial, for XHTML and a related XSLT processing chain. This development is the XML-related outcome of a markup project. The non-XML-related outcome are two documents about Japanese layout, one in English, one in Japanese.

What is so specific about the XML-related outcome of this project? One one hand it demonstrates an application of a validation and transformation processing chain for the publication of multilingual, aligned texts. There is no standardized means to represent and process multilingual data in XML, although there are formats used heavily in the localization industry (see Section 2.4). The design of the processing chain hopefully will provide ideas for readers how to tackle this problem.

On the other hand it shows how these chain supports specific user needs. These needs can be summarized most simply as "Don't show me that I use XML!". Implementing that requirement needs a lot of flexibility in setting up the validation and transformation process. It is not always a simple, and sometimes a painful task.

[1] http://relaxng.org/tutorial-20011203.html#IDAIRZR

Nevertheless, it is worth it: "Hiding XML" is an important means to spread its adoption among communities who are not using it (yet).

We will start with a description of the background of the project, and then give an overview of what has been achieved, that is a summary of the processing chain. When we will describe its development mainly in social terms, that is: What requirements from the users led to the processing setup or changes of the setup? Finally we review these decisions and reflect what could have been made different, in order to avoid the suffering.

2. Background

2.1. A document about requirements for Japanese layout

The project we are describing is a joint work item between four W3C working groups, the so-called *Japanese Layout Taskforce*[2]. The CSS, Internationalization Core, SVG and XSL Working Group have started this effort to gather requirements for realizing Japanese layout on the Web. We will not go into details of the project here, but only summarize some requirements:

- Gain a common understanding of terminology for Japanese layout, like "What is a printing area?" or "What is Ruby?"
- The ability to present Japanese text in vertical layout on the Web.
- Specify terminological information about Japanese characters, e.g. their relation to Unicode characters and their behavior in specific contexts like line breaking.
- Description of specific layout features like "Ruby[3]", "Tatechuyoko[4]" etc.

This project is crossing boundaries of culture and technology. One one hand there is the traditional, Japanese printing industry, on the other hand various instantiations of Web technology. The project is also crossing language boundaries for the participants. Many of the participants from the four Working Groups mentioned above do not speak Japanese. In contrast there are participants from the Japanese printing industry. They are the experts in providing a key input about Japanese layout, and he goal is to bring their knowledge to an international community. However, many of them are not fluent in English and are no experts in Web technology and related terminology. Especially the terminological differences between Japanese layout and Web technology are enormous.

Hence, the social challenge of this taskforce was to set up a working environment that would allow a back and forth between the two groups of participants. To achieve this it was decided soon after the creation of the taskforce that there should be two

[2] http://www.w3.org/2007/02/japanese-layout/
[3] http://www.w3.org/TR/jlreq/#ruby
[4] http://www.w3.org/TR/jlreq/#tatechuyoko

requirements documents, one in English, one in Japanese. XML played a key role in setting up the editing and publication environment and is a technical means for helping the communication between the two groups.

2.2. The result of the project: Japanese-English aligned documents

Before we go into details of the processing chain, we will give a short example from the Japanese and English documents which are being created, see the Figure Figure 1. The figure is a screen shot from an editor's copy of two documents. The public version is available in English[5] and in Japanese[6].

The structure of the two documents looks very similar, but they are not created as a direct translation from one language to the other. Instead the documents are aligned on the level of paragraph units. We say "paragraph unit" and not "paragraph" since there are several other alignment units like headings, figures, tables, captions etc. The methodology of paragraph unit alignment was chosen because a sentence-by-sentence alignment turned out not to be feasible. We will describe the reasons in more detail later.

As mentioned above, one important aspect of this project is terminology. Both documents contain an appendix which provides the same terminological concepts, but separately for readers in each of the two languages, see http://www.w3.org/TR/jlreq/#terminology-en and http://www.w3.org/TR/jlreq/ja/#terminology-ja. A lot care has been taken for alignment of the terminology entries, see as an example the entries for ruby in English[7] and in Japanese[8].

2.3. The processing chain

As in input to the generation of the two monolingual documents described above, a multilingual version is used. It contains all aligned paragraph units, and it is the single source which is being modified by the editors. The processing chain to generate the monolingual documents out of this source visualized in Figure 2.

It might come to a surprise, but the multilingual, aligned document is authored in – slices of XHTML files, see right part of the figure, the files marked as "ja-en". The reason for using XHTML will be explained later in Section 3.4. The reason for slicing the document is the size: including figures it has about 400 pages, 200 for each language. Out of these slices a merged version is being generated (step 2), from which after some post processing (step 3) the two monolingual versions follow (step 4).

[5] http://www.w3.org/TR/jlreq/
[6] http://www.w3.org/TR/jlreq/ja/
[7] http://www.w3.org/TR/jlreq/#ruby
[8] http://www.w3.org/TR/jlreq/ja/#ruby

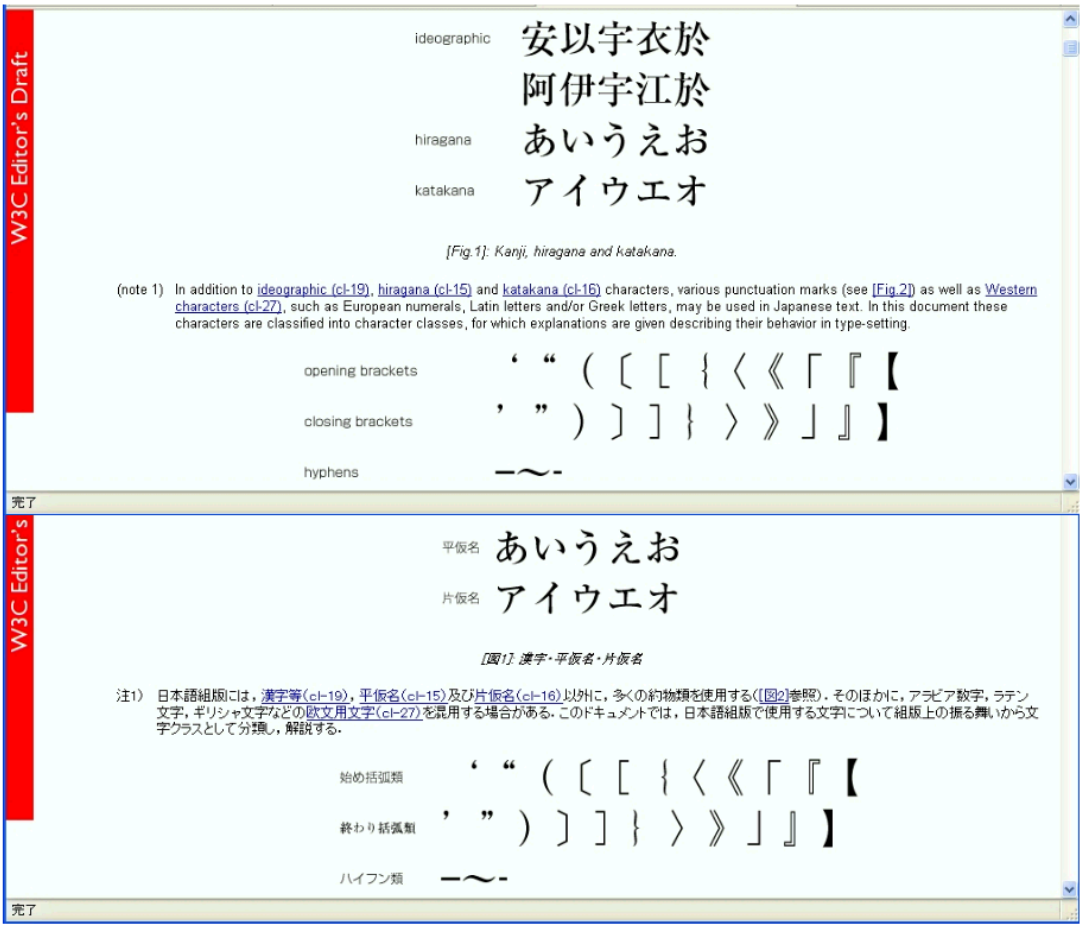


Figure 1. Example of the Japanese-English aligned document

For the editing of the XHTML files before step 1, the editors rely on their favorite HTML editors. They do not see the XML and do not know about the details of the processing chain. However, the DTD-based validation of their editing tools is not sufficient to verify the alignment between the Japanese and English parts. "Verify" here means that the alignment has to follow a defined structure, in order to make sure that the XSLT processing chain does not lead to unforeseeable results. A crucial part of implementing this verification is the validation against a special purpose RELAX NG schema, before the monolingual versions are generated. We will describe this schema in detail later.

The post processing (step 3) encompasses the implementation of numberings for links to figures, sections or the terminology, or the verification of character names used in the text against a naming scheme taken from an XML version of the Unicode data base. Again it was important to hide complexity for the editors, while having a means to verify markup consistency. And again the constrained RELAX NG schema played a crucial role in making sure that this step is successful.

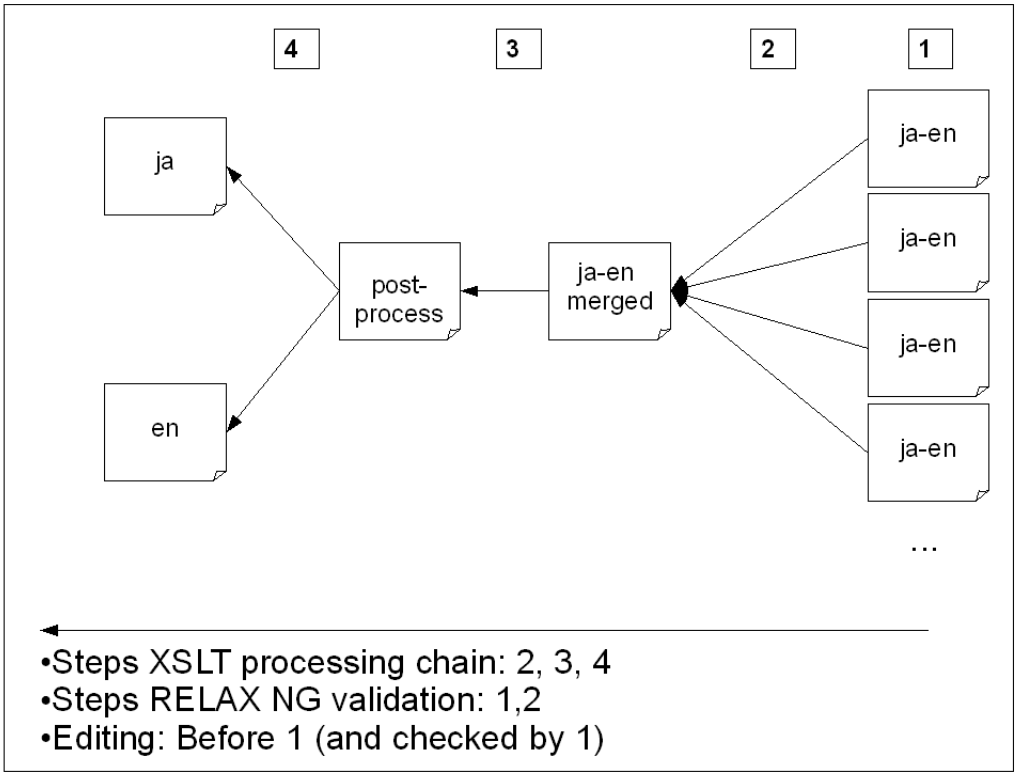


Figure 2. Processing chain for the creation of the Japanese-English aligned document

2.4. Why this approach was chosen

The reader may ask himself "Why XHTML? Why not a specific vocabulary for multilingual, aligned texts?". Actually there is an XML-based format to align translations, which is widely used in the localization industry: XLIFF, the XML Localization Interchange File Format XLIFF 1.2.

The reason for not using XLIFF is the complexity and the high level of specificity in this project. Existing XLIFF processing tools are tailored towards many wide spread XML formats like generic XHTML, or non-XML input formats like Java code. XLIFF allows for taking such formats as an input, easy editing in an aligned WYSIWYG mode, and conversion into the translated output. However, tailoring an XLIFF editing environment for the needs of the English-Japanese document, including the requirement for e.g. terminology, would have been an enormous effort. Another reason is that editors were used to their own editing environments, and there was just no time for them to get used to a different environment, e.g. supporting XLIFF. See Section 3.1 for details. In summary: the decision against XLIFF was not based on particular aspects of XLIFF, but on the (social) circumstances of the project.

The answer to the question "Why not another special purpose XML vocabulary for the project?" is a social one. As mentioned above, the Japanese participants of the taskforce provide the key input for Japanese layout requirements, but they are not experts in Web technology, including XML. Asking them to learn a new XML vocabulary, no matter if XLIFF or a special purpose one, would have meant a severe burden for them. It would have meant a delay for the project or might have become a severe hindrance for its progress.

In the following section we will describe how the rationale for "hiding XML" was developed in social terms or rather social constraints. Then we will describe how these social constraints lead to markup constraints, especially the design of the RELAX NG schema and the XSLT processing chain.

3. The project development

The project can be described in several steps. Each steps demonstrates a working strategy for the taskforce, which was taken up and later abandoned or changed.

- The "Starting in one language" step
- The "Translating sentence by sentence" step
- The "Translating aligned" step
- The "Giving up translation, work in both languages" step
- The "We need more features!" step

These steps were not planned to happen in this order, or to happen at all. They emerged out of the project along the way, and it was necessary to take them for supporting the inter-cultural and inter-technological communication between the various participants.

3.1. Editing tools

Before describing the individual steps, we will explain briefly the various editing environments.

The editors used all kinds of editing software: plain text editors, XML tools like OXygen or XMetal, or XHTML editing tools. Some of the editors had used their software for several years and did not have the time to get used to a new editing environment. Also, esp. for the users of XHTML software, hiding the markup during editing was quite important.

These circumstances resulted into the the requirement not to change individual editing environments. That was one reason why no special purpose XML vocabulary was developed for the "Translating aligned" step, see Section 3.4. With such a vocabulary it would have been possible to hide XML, e.g. with an "XML+CSS" solution. But the editors would have had to change their software to an environment which understands "XML+CSS".

3.2. "Starting in one language" step: A simple XHTML file

At the beginning the project goal was the development of a Japanese document which at some point later should be translated into English. The social circumstance in this step was that the main editors were not experienced in XML or even (X)HTML editing at all. So providing them with a "simple as possible" setup meant to choose a WYSIWYG environment. XHTML was chosen as a format since publication on the web was planned from the beginning, and every editor was familiar with a WYSIWYG environment for this format.

An XHTML template was prepared for the editors. It provides the following document structure: a main part and an appendix with sections, and some common inline elements for references or highlighting of terms. Soon it became obvious that the document would become very large. In order to avoid complexity, the editors were advised to use the template without introducing new kinds of markup.

3.3. "Translating sentence by sentence": Copying the file!

First, a fair amount of text was created in Japanese, including figures and a terminology section being referenced from the main text. After a few months, a decision was made to start with translations to English. The Japanese taskforce participants needed feedback from the non-Japanese participants to understand if they were heading into the right direction.

Analyzing the Japanese XHTML file showed that the editors had been working very carefully. They had preserved the structure of the initial template and had not introduced additional kinds of markup. Hence, the simplest approach for translating into English was taken: the file was copied, and the Japanese content was replaced step by step, that is sentence-by-sentence.

During this step some conventions for identifiers were introduced. For example the editors decided to use the same ID attributes for terminology in the Japanese document and the English version. These and other ID attributes turned out to be a rescue for the next, mostly painful step.

3.4. "Translating aligned": Creating an aligned file and a validation + transformation processing chain

Feedback from the non-Japanese participants of the taskforce revealed that the translation sentence by sentence was not feasible. The writing style in the two languages for the genre of technical documentation turned out to be too different to be easy to understand for all readers. An example taken from http://www.w3.org/TR/jlreq/ja/#ja-subheading0_1 (Japanese version) and http://www.w3.org/TR/jlreq/#en-subheading0_1 (English version) is given below.

Table 1. Example of translation variants

Japanese Version	具体的な解決策を提示することではなく，要望事項の説明をすることにした．それは，実装レベルの問題を考える前提条件をまず明確にすることが重要であると考えたからある
English Version 1 (nearly literal translation)	(We) have decided not to propose actual solutions, but to make an explanation of important issues. This is because (we) think that it is important in the first place to explain precisely precondition constraints for considering implementation level problems.
English Version 2 (translation actually used in the document)	The goal of the task force is not to propose actual solutions but describe important issues as basic information for actual implementations.

Due to such differences, a different strategy was developed: an alignment paragraph by paragraph.

In terms of the editing setup, this step required a complete redesign. Validation against the XHTML DTDs of the Japanese and English files was not sufficient to assure the alignment of paragraph units. An analysis with simple XPath expressions like `count(//p)` showed that the files already were out of sync. It became clear that a more constraint vocabulary than XHTML was necessary to bring them back and to keep them in sync. As described above for this project it was not feasible to create a special purpose XML vocabulary or to use an existing one like XLIFF. The burden for the editors to learn the new element and attribute types would have been too high.

3.4.1. The schema - overview

Finally it was chosen to keep the XHTML elements, as a convenience for the editors, and to develop a different schema for validation "off-line", that is validation not during XHTML DTD-based editing. In addition, the two documents were merged into one, as another means to check the alignment. Physically the merged document was represented in slices of XHTML files, due to the amount of text. The schema is provided as an appendix to this paper.

The schema constitutes a typical document structure via patterns with `<div>` elements: `div1`, `div2` and `div3`. All these patterns consist of a `heading` pattern, e.g. `div1heading`, and some block level elements via the `div-mix` pattern. In the case of `div1`, it is also possible to nest a `<div>` element of level 2, see the `div2` pattern. In the case of `div2`, there is the `div3` pattern accordingly.

This structure is a very restricted version of what you may use in vocabularies like DocBook or TEI. The parts which are highly specific for this project are patterns which make use of class attributes at `<div>` elements with the value “Taiyaku”. The Japanese word Taiyaku “対訳” means “aligned translation”, and that states the purpose of these patterns. Below is a document fragment that may be validated against the pattern called `p-taiyaku`, which is used for paragraph-wise aligned translations.

```
<div class="Taiyaku">
 <p lang="ja" xml:lang="ja">これはインラインのタグも使用できる例文です。</p>
 <p lang="en" xml:lang="en">This is an example sentence which may also contain ▶
inline markup.</p>
</div>
```

The p-taiyaku pattern consists of a `<div>` element with an attribute `class="Taiyaku"`, and it contains two `<p>` elements. The pattern requires the appearance of `lang` and `xml:lang` attributes, which have prescribed values: “ja” and “en”. Transforming this “Taiyaku” pattern into a special purpose XML vocabulary may be represented as follows:

```
<p-taiyaku>
 <p-ja>これはインラインのタグも使用できる例文です。</p-ja>
 <p-en>This is an example sentence which may also contain inline markup.</p-en>
</p-taiyaku>
```

`figure-taiyaku` is an example of another “Taiyaku” pattern:

```
<div class="Taiyaku">
 <p class="figure" xml:lang="ja" lang="ja" ...>
  <img .../>
  <span class="figureCaption">漢字・平仮名・片仮名</span>
 </p>
 <p class="figure" xml:lang="en" lang="en" ...>
  <img .../>
  <span class="figureCaption">Kanji, hiragana and katakana.</span>
 </p>
</div>
```

Describing this example as a special purpose XML vocabulary might result in the following structure.

```
<fig-taiyaku>
 <fig-japanese ...>
  <img .../>
  <caption>漢字・平仮名・片仮名</caption>
 </fig-japanese>
 <fig-english ...>
  <img .../>
  <caption>Kanji, hiragana and katakana.</caption>
```

```
    </fig-english>
  </fig-taiyaku>
```

For the social reasons described above, using such a vocabulary was not feasible in the project. To put it differently, the value of XML making it easier to process markup based on generic identifiers was given up to achieve a social value of XML adoption.

3.4.2. Why RELAX NG?

RELAX NG worked well for modeling "Taiyaku", since it was easy to enumerate the disjunct categories of "Taiyaku" units and to create their RELAX NG patterns. There is a finite number of patterns (only 6), and the structure of these is clearly separated. Using XSD 1.1. conditional type assignment and assertions XSD 1.1 Part 1, or Schematron Schematron assertions, are other means to define the "Taiyaku" patterns, and would have been other possible solutions to the problem we faced. In that sense, the choice of RELAX NG was arbitrary. Nevertheless it was necessary to have an additional validation mechanism, on top of the XHTML DTD, and the expressive power of this mechanism exceeded the power provided by XML DTDs.

3.4.3. From the aligned document to the monolingual versions

In step 4 from fig. 2, the monolingual documents are generated via XSLT. Here the application of constrains imposed by the RELAX NG schema is crucial. As an example, the constraints that a `<div>` element consists of two paragraphs in Japanese and English is used to extract one paragraph for one monolingual version and the other for the other monolingual version. The `<div>` elements used for the alignment are filtered out.

The `xml:lang` attribute is used heavenly during the transformation. The inheritance of `xml:lang` allows using very general rules like the filtering of all elements which are not in the target output language.

3.5. Giving up translation, working in both languages

The paragraph-wise alignment led to a better, that is for both languages easier to understand translation. However, in the meantime the non-Japanese participants had provided major inputs to the document. The result was not a one-way-translation, but parallel work in both languages, with translating back and forth.

Although this was most revolutionary in terms of the work strategy, at least in terms of the social interaction between participants, the effect on the editing setup, including validation and processing chain, was zero.

3.6. "We need more features!"

This step is not finished yet, and it will probably go on for a while. If the project would have been the development of a special-purpose XML vocabulary, the step would have meant "We introduce more markup", e.g. "we add attributes or elements" in terms of the vocabulary, and "We add additional processing" in terms of the XSLT chain. The latter task does not change, however the former task is again different due to the constraint to use only existing elements and attributes from XHTML.

We will explain an example already mentioned above in more detail. A feature requested "on the way" was the ability to assure the representation of Unicode character names. These names are standardized in the Unicode character data base. It would have been possible to use XML entities as a shortcut for the names, however the requirement was too complex: both the names and the code point should be given in the output, in a specific order, and with specific emphasis. For example, the following input text

```
In JIS X 4051, <span class="character">[、]</span> and
<span class="character">[,]</span>...
```

should be given in the output as

```
In JIS X 4051, <span class="character">IDEOGRAPHIC COMMA "、"</span> and
<span class="character">COMMA ","</span>...
```

An XSLT template triggered by `<span class="character">` takes up `[、]` to gather the necessary information from the Unicode character data base. To make sure that such elements are used properly, the following RELAX NG declaration is provided.

```
element span {
 attribute class { "character" },
 xsd:string { pattern = "\[.*\].*" }
}
```

3.7. What could have been made different?

As an input for the review of the technological decisions made, the Table 2 summarizes them and compares them to "ideal" decisions that would have been made without social constraints.

The table makes obvious that an ideal technical solution would have meant less work for the technical setup of the project. Also, the markup-related outcome of the project might have been a starting point for developing a format for multilingual editing, or provide input to further developments of XLIFF. The semantics of such a vocabulary is already described explicitly in the RELAX NG patterns, compare

Table 2. Actual and "ideal" project decisions

Step	"Ideal" technical decision	Actual technical decision
Starting in one language	Special purpose XML vocabulary for technical documentation, e.g. DocBook	XHTML template with usage instructions
Translating sentence by sentence	Specialization of the vocabulary for the alignment purpose, see examples in Section 3.4.1	Copy of the template
Translating aligned	Re-engineering the template, creating validation and transformation chain	
Giving up translation, work in both languages	None	None
We need more features!	Change of the vocabulary	No change of vocabulary, but of "hidden" schema constraints

e.g. the XHTML structure to the XML vocabulary exemplified in Section 3.4.1. Hence, a follow-up on the project might easily lead into that direction.

4. Conclusion: was it worth it?

This paper described a project of imposing additional constraints on top of an existing XML vocabulary (XHTML). The motivation was mainly given by project specific social constraints, mainly the unfamiliarity of participants with XML and their limited time for learning new XML vocabularies. Nevertheless, the exercise was helpful in fostering the adoption of XML.

There are other communities like Microformats which may learn from this approach. They are a similar example of adding application specific constraints to XHTML. However, rarely microformats use a validation means for assuring the adequacy of usage. Social agreement is regarded as being sufficient. Although this paper introduced a special purpose project and a special purpose set of constraints, other approach might benefit from the general approach, that is not replacing the promises of social agreements with validation constraints, but by using validation for supporting them. For a related discussion see also Quin 2006.

To some extend it is foreseeable that experts in designing (markup) languages will criticize this paper. After all the approach of not developing a new vocabulary, but hiding its constraints within XHTML elements and attributes, is laborious, both in terms of schema and XSLT design. The author is hopeful that these experts will see the benefit of bringing communities closer to XML, or to put it differently, of "spreading the word" without speaking it out loudly.

A. Acknowledgements

The author would like to thank the Japanese participants of the Japanese layout task force for letting him participate in an "adventure" of cross-cultural and cross-technological communication, and for taking the effort of making their knowledge available to the international community.

B. The RELAX NG schema

```
# schema for JLTF aligned document. For documentation see comments inline and
# http://www.w3.org/2007/02/japanese-layout/docs/aligned/ (W3C member only).
# For information about the project see ▶
http://www.w3.org/2007/02/japanese-layout/ .
default namespace = "http://www.w3.org/1999/xhtml"
# General structure of the document.
start =
  element html {
    attribute lang { xsd:NCName },
    attribute xml:lang { xsd:NCName },
    element head {
      element meta {
        (attribute content { text },
         attribute http-equiv { xsd:NCName })?
      }*
      & element title { text }
      & element link {
          attribute href { text },
          attribute rel { xsd:NCName },
          attribute type { text }?
        }*
      & element style {
          attribute type { text },
          text
        }?
    },
    (# <body> element  of a slice consists of  one or more div1 element (see ▶
below).
     element body {
       attribute class { "slice" },
       div1+
     }
     | # <body> element of the complete document consists of a <div> element
       # with class "body" and another one with class "back". Each
       # contain a div1 element (see below).
       element body {
```

```
          attribute class { "merged" },
          element div {
            attribute class { "body" },
            div1+
          },
          element div {
            attribute class { "back" },
            div1+
          }
        })
  }
# div1, div 2 and div 3 have the same structure:
# a heading followed by div mix. div1 and div2 optionally
# have nested divs of type div2 and div3 respectively.
div1 =
  element div {
    attribute class { "div1" },
    attribute id { text },
    div1heading,
    div-mix?,
    div2*
  }
# div1heading and div2heading have the same
# structure: <div> element with class "Taiyaku"
# and heading elements in order "Japanese , English"
# inside
div1heading =
  element div {
    attribute class { "Taiyaku" },
    attribute id { xsd:NCName }?,
    element h2 { langAttrJa, a, p-mix },
    element h2 { langAttrEn, a, p-mix }
  }
div2 =
  element div {
    attribute class { "div2" },
    attribute id { xsd:NCName }?,
    div2heading,
    div-mix?,
    div3*
  }
div2heading =
  element div {
    attribute class { "Taiyaku" },
    attribute id { xsd:NCName }?,
    element h3 { langAttrJa, a, p-mix },
```

```
    element h3 { langAttrEn, a, p-mix }
  }
div3 =
  element div {
    attribute class { "div3" },
    attribute id { xsd:NCName }?,
    div3heading,
    div-mix?
  }
div3heading =
  element div {
    attribute class { "Taiyaku" },
    attribute id { xsd:NCName }?,
    element h3 { langAttrJa, a, p-mix },
    element h3 { langAttrEn, a, p-mix }
  }
# Below language attributes.
langAttrJa =
  attribute xml:lang { "ja" },
  attribute lang { "ja" }
langAttrEn =
  attribute xml:lang { "en" },
  attribute lang { "en" }
# Basically we have in divs (in sections) only:
# paragraphs, figures, various lists, tables
div-mix =
  (linkToSlices
   | p-taiyaku
   | figure-taiyaku
   | ol
   | ul
   | dl
   | table
   | terminology-and-charclass-table)*
# Used to link to slices
linkToSlices =
  element a {
    attribute href { text },
    attribute class { "linkToSlice" }
  }
# Ordered list
ol = element ol { li* }
# Definition list
dl =
  element dl {
    element dd { p-taiyaku }+
```

```
  }
# Unordered lists
ul = element ul { li+ }
# <div> class "Taiyaku" element with <p> in
# English and Japanese
p-taiyaku =
  element div {
    attribute class { "Taiyaku" },
    attribute id { xsd:NCName }?,
    p-ja,
    p-en
  }
# Below Japanese and English paragraphs only
# used in p-taiyaku
p-ja = element p { langAttrJa, commonAtts, p-mix }
p-en = element p { langAttrEn, commonAtts, p-mix }
# Inline elements
p-mix = (a | span | img | i | br | ins | del | em | text)*
# em element, used for editors notes. Content will be deleted in the output!
em = element em { p-mix }
# common atts
commonAtts =
  attribute class { "ft" }?,
  attribute id { xsd:NCName }?
# inline elements
br =
  element br {
    empty,
    attribute * { text }*
  }
del = element del { p-mix }
ins = element ins { p-mix }
a =
  element a {
    attribute class { a-classes }?,
    attribute href { xsd:anyURI }?,
    attribute id { xsd:NCName }?,
    attribute name { xsd:NCName }?,
    attribute shape { text }?,
    (text | ins | del)*
  }
a-classes = "figure_ref" | "sec_ref" | "termref" | "characterClass"
i =
  element i {
    attribute class { xsd:NCName }?,
    text
```

```
  }
span =
  # definition for characters
  element span {
    attribute class { "character" },
    xsd:string { pattern = "\[.*\].*" }
  }
  | # other span elements
    element span {
      attribute class { span-classes }?,
      attribute id { xsd:NCName }?,
      p-mix
    }
# <li> element. No mixed content or
# text content allowed here.
li = element li { (div-mix | ol | table)* }
# Tables
table =
  element table {
    attribute class { table-classes },
    element tr {
      element td {
        attribute align { xsd:NCName }?,
        attribute class { td-classes }?,
        attribute colspan { xsd:integer },
        attribute rowspan { xsd:integer },
        div-mix
      }+
    }+
  }
span-classes = "figure_ref"
table-classes = "standardTable" | "t_note"
td-classes = "ft"
# Images
img =
  element img {
    attribute alt { text },
    attribute src { text }
  }
figure-taiyaku =
  element div {
    attribute class { "Taiyaku" },
    attribute id { xsd:NCName }?,
    element p {
      attribute class { "figure" },
      attribute id { xsd:NCName },
```

```
      langAttrJa,
      img+,
      element span {
        attribute class { "figureCaption" },
        attribute id { xsd:NCName }?,
        p-mix
      }
    },
    element p {
      attribute class { "figure" },
      attribute id { xsd:NCName }?,
      langAttrEn,
      img+,
      element span {
        attribute class { "figureCaption" },
        attribute id { xsd:NCName }?,
        p-mix
      }
    }
  }
terminology-and-charclass-table =
  element div {
    attribute class { "Taiyaku" },
    element table {
      attribute class { "termlist" | "charclass" },
      attribute xml:lang { "ja" },
      attribute lang { "ja" },
      (text | anyElement)*
    },
    element table {
      attribute class { "termlist" | "charclass" },
      attribute xml:lang { "en" },
      attribute lang { "en" },
      (text | anyElement)*
    }
  }
anyElement =
  element * {
    (attribute * { text }
     | text
     | anyElement)*
}
```

Bibliography

[1] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. Available at http://docbook.org/.

[2] Liam Quin. *Microformats: Contaminants or Ingredients? Introducing MDL and Asking Questions*. Paper presented at Extreme Markup Languages 2006, Montréal.

[3] James Clark and MURATA Makoto. *RELAX NG Tutorial*[1]. Committee Specification 3 December 2001. Available at http://relaxng.org/tutorial-20011203.html.

[4] *Information technology -- Document Schema Definition Languages (DSDL) -- Part 3: Rule-based validation -- Schematron*. International Organization for Standardization (ISO) ISO/IEC 19757-3:2003.

[5] Lou Burnard and Syd Bauman, editors. *Text Encoding Initiative Guidelines development version (P5)*. TEI Consortium, Charlottesville, Virginia, USA, Text Encoding Initiative. Available at http://www.tei-c.org/P5/.

[6] Yves Savourel et al., editors *XLIFF Version 1.2*[2]. OASIS Committee Specification 24 July 2007. Available at http://docs.oasis-open.org/xliff/v1.2/cs02/xliff-core.html.

[7] Shudi (Sandy) Gao 高殊镝 , C. M. Sperberg-McQueen and Henry S. Thompson. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Working Draft 3 December 2009. Available at http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/.

[1] http://relaxng.org/tutorial-20011203.html
[2] http://docs.oasis-open.org/xliff/v1.2/cs02/xliff-core.html

Authoring XML all the Time, Everywhere and by Everyone

Stéphane Sire
EPFL
<stephane.sire@epfl.ch>

Christine Vanoirbeek
EPFL
<christine.vanoirbeek@epfl.ch>

Vincent Quint
INRIA
<vincent.quint@inria.fr>

Cécile Roisin
INRIA
<cecile.roisin@inria.fr>

Abstract

This article presents a framework for editing, publishing and sharing XML content directly from within the browser. It comes in two parts: XTiger XML and AXEL. XTiger XML is a document template specification language for creating document models. AXEL is a client-side Javascript library that turns the document template into a document editing application running in the browser. This framework is targeted at non XML speaking end users, since it preserves end users from XML syntax during editing. Its current implementation proposes a pseudo-WYSIWYG user interface where the document template provides a document-oriented editing metaphor, or a more form-oriented metaphor, depending on the template.

Keywords: XML, authoring, document template, client-side Javascript library

1. Introduction

Nowadays, most web-based applications take advantage of the XML format to expose information on the web – by use of XSLT transformations and/or CSS style – or, still better, to anchor automatic processes on XML data that follow a conceptual model. Despite the fact that XML originates from research in the structured document do-

main – XML is an application profile of SGML – , the common use of XML on the web is mostly based on data extracted from relational databases. In this sense, web users essentially contribute to populating such databases by introducing data through forms displayed on their browser. On another hand, the semantic web perspective reinforces the trend to produce more document-oriented information as it encourages to provide information that embarks significant meaning by the use of tags and attributes employed in documents. As a consequence XML authoring should address both paradigms.

The use of forms constrains users to provide consistent data-oriented information. It may be performed by using XHTML forms; in this case, a significant effort is required on the server side to check validity of entered data. It may be performed by the use of XForms whose declarative approach clearly facilitates the control of data, but unfortunately XForms is currently not supported by browsers, and thus requires the use of often complex frameworks.

Producing document-oriented information on the web is feasible but is currently addressed in so many different ways: web-based rich text editors, Wikis and blogs, XHTML editors or Content Management Systems. Web-based rich text editors allow web users to provide XHTML content in a way very close to usual desktop word processors. Wikis or blog authoring systems allow users to produce tagged information but, often necessitate the knowledge of a basic syntax which is not appropriate to end-users. Most XHTML editors offer the possibility to introduce micro-formats in the documents, leveraging the level of reusability of common components. Content management systems offer functionality to produce either XHTML or XML content on the web. More recently, web-based WYSIWYG XML editors became available.

As a result, millions of (X)HTML blog entries, wiki pages or database generated content are currently available, but they are difficult to interpret and repurpose. XML content is also available, but XML-based authoring solutions imply using complex client/server architectures. They include the use of XML parsers to validate information against a generic model (DTD, XML Schema, Relax-NG, etc.), usually on the server side.

Imagine what would be possible if non XML-savvy users could use their familiar browser as an XML authoring tool, allowing them to produce mixed data- and document-oriented information, constrained by a specific template. As an extension, with the agreement on common XML vocabularies or domain specific languages, all data entered on the web would be available for automatic processing [12]. Moreover, the use of common XML vocabularies would allow to develop standard editing components. This modularity would allow to quickly create light XML authoring applications adapted for specific content models.

This article presents AXEL (Adaptable XML Editing Library), a client-side Javascript library for authoring template-driven XML content on the Web. It relies on the use of XTiger XML, a template definition language based on XHTML, which is designed to express structural constraints on the edited content. Additionally,

the associated use of CSS provides flexibility in terms of user interface. The template language and the library bring the opportunity to provide users with traditional forms-based interfaces or document-oriented interfaces, allowing them to author XML content with a visual user experience close to WYSIWYG word processors.

The paper is organized as follows: we introduce first the concept of template proposed by XTiger XML and explain its articulation with AXEL, the client-side document template engine that transforms a template into an interactive XHTML page. Then we provide details about the XTiger XML syntax and semantics and we illustrate through concrete examples how templates constrain the document structure, to guide the presentation and to support the mapping to a target XML structure. Then we describe the editing functionality provided by AXEL and justify the customization of user interface in terms of usability. Finally, we explain some server integration aspects and we outline the main features of the library. The article finishes with a comparison of our approach with related work and some perspectives for future work.

2. Templates at a Glance

2.1. Document Templates

In many document production systems such as word processors, templates are typical documents whose organization and style indicate how documents of a certain type should be structured and presented. With this approach, a template is just a sample document that authors use as a starting point and that they develop by providing content while following the structure and style of the initial document. The tool lets authors free to change everything at any time, but the initial sample document is supposed to give them hints about what is expected in the end. In some systems, in XHTML editors for instance, templates contain also some fixed parts that an author can not modify and that will be preserved in the final document, to strongly constrain some parts of the document.

Different types of templates are also used in content management systems for generating HTML pages from a data base. In that case, the template is a HTML page with "holes" that contain queries for extracting content from the data base. Some statements are also interspersed in the HTML code to generate additional parts depending on the data found in the data base.

So, there are different sorts of document templates. The templates we are considering in this paper were initially designed [7] to make it easier for authors to create and edit well structured and semantically rich XHTML documents that do not require complex schemas and transformations, while still allowing documents to provide useful, automatically processable information. The initial language we have developed for this purpose was called XTiger [11].

The idea behind XTiger was to use XHTML as the basic document format and to constrain the way to use it, in order to produce a specific type of document. A number of authors are comfortable with (X)HTML editors, and some of these editors produce valid, well structured XHTML code. Because XHTML is an XML language, the namespace mechanism can be used to allow XHTML documents to include elements and attributes from another XML language, XTiger, that expresses constraints on the XHTML structure. As XHTML can benefit from CSS style sheets, it is easy to specify the visual aspect of the document structure.

With this approach, a template is an XHTML document (with its style sheets) that represents the skeleton of a document, and that contains statements expressed in the XTiger language to indicate how this embryonic document can evolve and grow. We have extended the Amaya web editor [1] to support the original XTiger language. The editor interprets the XTiger statements contained in the template to guide the user in building the intended document structure. The author interacts with a familiar editor on a well formatted document and eventually produces a structured XHTML document, that can then be used directly on the web with any browser. Templates are easy to create: they are basically XHTML documents containing XTiger elements where structural constraints have to be set.

This approach to templates has proven to be very useful for editing rich XHTML documents on the web [7], but it is possible to go a step further and to produce any XML structure, not only XHTML, while preserving the ease of use of XTiger templates. This is done by adding to the original XTiger language a mapping mechanism for specifying what piece of XHTML structure is equivalent to what target XML structure. Another significant step is to let authors free to choose their preferred tool, by implementing the editor as a Javascript library that runs in a browser. When loaded into the browser with the library, the document template is turned into an interactive editor that follows the constraints expressed by the XTiger elements to generate only data following a predefined XML content model.

This extended version of the template language is called XTiger XML [14]. In the remainder of this paper, we often write XTiger when discussing features offered by both versions of the language.

Thus, an XTiger XML template plays three complementary roles:

- It includes constraints that define the data components the user can enter at different places in the document.
- It contains presentation hints (XHTML elements, style sheets) that define how each editing component looks like while editing.
- It provides information for mapping the editing components to a target XML structure.

The figure below shows a typical XML editorial process built with document templates: a template author creates the templates which are turned into interactive editing applications into the browser by the AXEL library. End users can then create

or change documents; when they save them, their XML content is generated and sent to server side applications for further processing or stored into databases or documents.

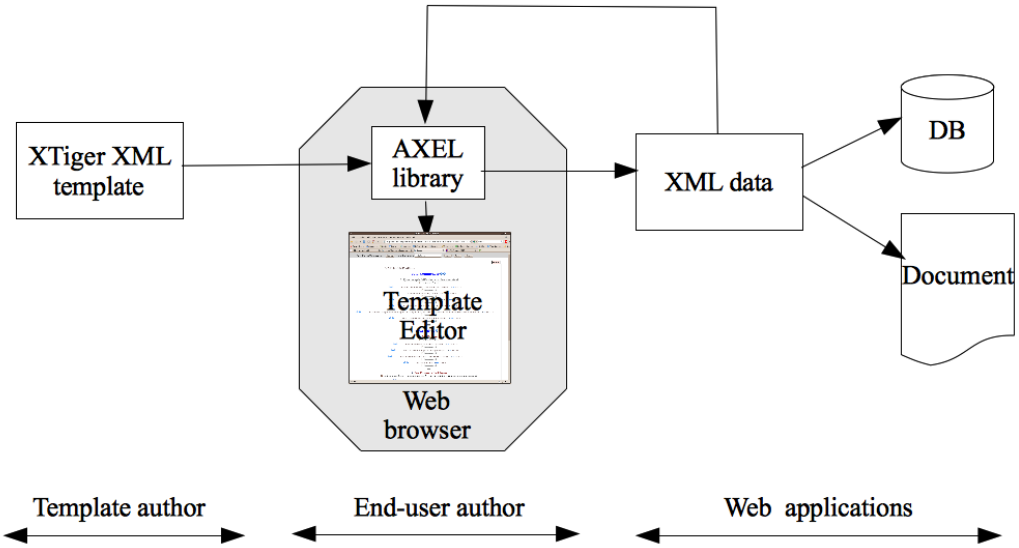


Figure 1. Overall architecture

2.2. Anatomy of an XTiger XML Template

The following code example shows a very simple document template for editing a list of persons to great. As this section explains, the document template defines in the same file: some structural editing constraints, a presentational view for editing data, and an XML content model.

Example 1. The "Greetings" template

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xt="http://ns.inria.org/xtiger">
<head>
 <title>My first template</title>
 <xt:head label="greetings">
   <xt:component name="friend">
     <li><xt:use types="text">name</xt:use><xt:menu-marker/></li>
   </xt:component>
 </xt:head>
</head>
<body>
 <p>List of persons to great:</p>
 <ul>
```

```
    <xt:repeat minOccurs="0" maxOccurs="*" label="persons">
      <xt:use types="friend" label="name"/>
    </xt:repeat>
  </ul>
</body>
</html>
```

A capable document template engine, such as AXEL, transforms the previous template into an interactive XHTML page from which a user can create a list of persons. At any time, the page content can be exported into XML, as in the following example.

Example 2. Example of edited XML content

```
<greetings>
  <persons>
    <name>Charlie</name>
    <name>Oscar<name>
  </persons>
</greetings>
```

The "Greetings" example shows that the document template is an XHTML document. This is because, as it will be run inside a browser, the presentation language is the language for displaying pages in the browser. The document also contains some elements and attributes in the `http://ns.inria.org/xtiger` namespace which is prefixed with `xt:` for XTiger. These elements constrain user's input and define an XML content model.

The document template is divided into two parts. The `xt:head` part, which is declared in the `head` section of the XHTML document, declares some reusable editing components and data types. Each component is declared as a `xt:component` element with a `name` attribute that will be used to refer to it. The content of a component can mix elements from the presentation language, XHTML, and from the XTiger namespace.

The second part of the template is the `body` section of the XHTML document. It contains a mix of XHTML elements and XTiger elements. The XHTML elements are rendered as usual within the browser. They give the document its appearance and can be styled using CSS to generate different looks and feels. The `xt:use` element is a component inclusion element which makes it possible to insert an editing component in place and specify its data type. The document template engine expands each `xt:use` element by replacing it with the content of the component whose name matches the value of its `types` attribute. Ultimately, some editing component names such as the `text` component are reserved types which are associated with *primitive editor components*. The primitive editor components are transformed by the engine into special fields which can be edited by the end user to enter data in the document.

Some other XTiger elements and attributes, such as the `xt:repeat` element and its `minOccurs` and `maxOccurs` attributes are editing constraints that guide the editing process. The `xt:repeat` element indicates that its content can be repeated several times, with a minimum and a maximum number of times.

Finally, the XTiger attribute `label` drives the XML content serialization process. In fact, the edited document is turned into XML content by traversing it from its root. Each time the template engine encounters a `label` attribute, it creates a new XML content fragment with a tag name set to the value of `label`. The XML content of this fragment is the result of serializing the corresponding document subtree.

2.3. Editing With a Document Template

A client-side document template engine, such as AXEL, loads a document template in a browser window, and transforms it into an interactive editing application. As can be seen on Figure 2, some parts of the document template are transformed into user interface controls such as a *minus* and a *plus* button. Some other parts of the document become editing fields, such as the fields to enter a person name. Finally, some parts are just part of the background and are displayed as usual, non-editable XHTML elements, such as the list header in the snapshot below.

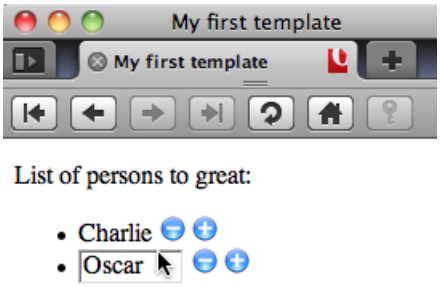


Figure 2. The "Greetings" generated editor

The user interface generated by the document template engine is quite different from a *Rich Text Editor* user interface. The main reason is that there is no need for a command panel to group all the available options in one place (usually at the top of the window), because the template prevents users to insert document formating commands. Instead, an editor inserts directly into the document some user interface controls (such as the *minus* or *plus* buttons) directly within the document flow, at the position where the choice is available to the user.

The areas where users can input text are defined by primitive editor components. Each of these editors is responsible to manage the display and the editing of its content. For instance, the `text` primitive editor component displays its content either as an XHTML `span` element, or as an `input` or `textarea` that dynamically replaces

the content when the user clicks on it for editing. This is illustrated with the *Charlie* and *Oscar* inputs in the figure above.

In order to preserve a document look and feel, the document template editor does not systematically emphasizes the editing component borders, for instance with dashed boxes, as it is the case in other document template editors such as Microsoft PowerPoint. Instead, each document template author can use CSS properties and `class` attributes to create the desired effects.

To some extent, the editing user interface can be seen as something between a full WYSIWYG editing user interface and a more constrained form-based user interface. In any case, it never shows XML tag or attributes to the user, which makes it easily usable by everyone.

The editing user interface is self-contained in the document (no need for extra menus or panels) and the document template engine implementation is client-side. As a consequence, editable document templates can be embedded anywhere in any application (see section on server integration for details). For instance, for the purpose of writing this article, we have deployed the document template engine onto a WebDAV web server. The only addition to the document template of the article (which uses a custom XML content model from which we can easily generate a Docbook or an XHTML document), has been to program a Javascript menu bar to load and save XML data from and to the document template. The resulting editing application with its menu bar is displayed below:

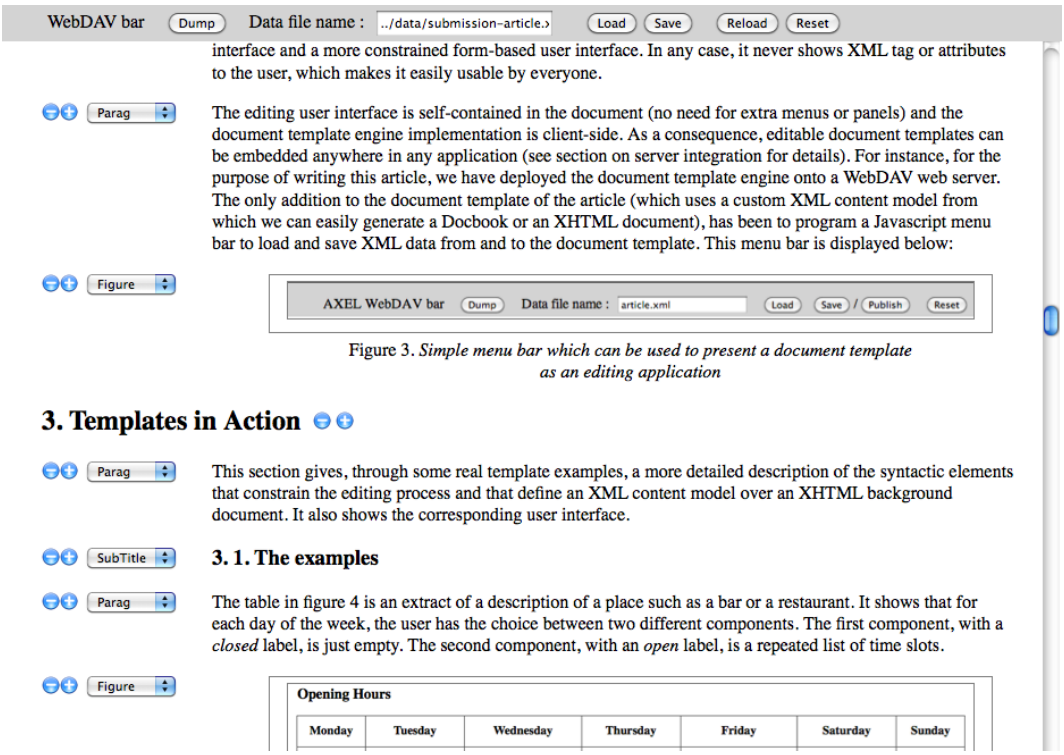


Figure 3. A simple editing application with it's menu bar for authoring this article

3. Templates in Action

This section gives, through some real template examples, a more detailed description of the syntactic elements that constrain the editing process and that define an XML content model over an XHTML background document. It also shows the corresponding user interface.

3.1. The examples

The table in Figure 4 is an extract of a description of a place such as a bar or a restaurant. It shows that for each day of the week, the user has the choice between two different components. The first component, with a *closed* label, is just empty. The second component, with an *open* label, is a repeated list of time slots.

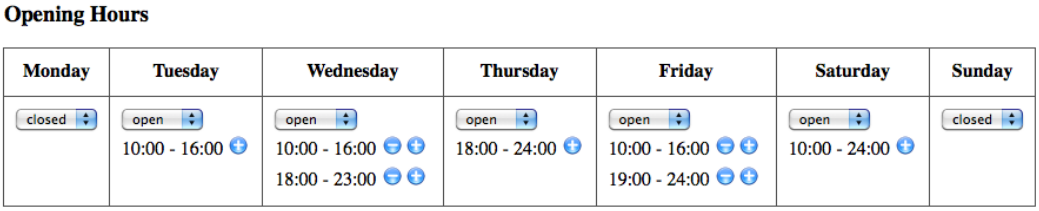


Figure 4. Editing a timetable

The Figure 5 shows a restaurant menu. As it is displayed, the user has already entered two courses (*Entrées* and *Plats*). In the first course, she has entered three dishes (*Salade de crudités*, *Salade de cabécou* and *Salade de coeurs de canard*), and she has entered various information about the price of dishes. This example shows that repeated components can be nested to create hierarchical structures (e.g. dishes within courses). As it will be explained later, the template author can control the insertion point of the *plus* and *minus* buttons inside the document, and their size. Also, some information, such as the specialty, some comments, or the prices are optional.

The bibliographic entry in Figure 6 is an extract of a bibliographic reference list. It shows that a bibliographic entry is made of two nested components. The first component sets the general category of the bibliographic entry, such as a *Paper*, while the second component sets a subcategory. In the figure, the choices for the second component are between *Article*, *ArticleInJournal*, *ArticleInProceedings* or *InBook*. These are all different types of bibliographic references that can describe a *Paper* publication. The presentation and the nature of the information displayed in the user interface may be quite different for each type.

Figure 5. Editing a menu

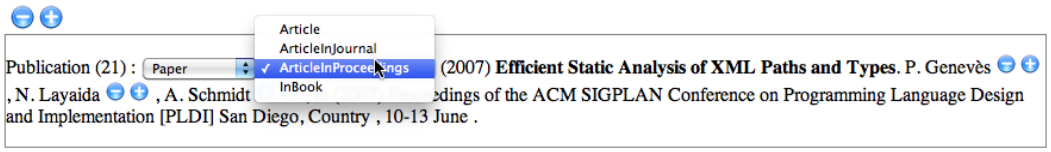


Figure 6. Editing a bibliography

The XML content below is generated from the document shown on Figure 6. This illustrates the kind of XML content that can be edited with AXEL and XTiger XML.

Example 3. XML content of a bibliography

```
<Publication>
  <PublicationId>21</PublicationId>
  <Paper>
    <ArticleInProceedings>
      <PublishingYear>2007</PublishingYear>
      <Title>Efficient Static Analysis of XML Paths and Types</Title>
      <Authors>
        <Author>
          <FirstName>P.</FirstName>
          <LastName>Genev&#xE8;s</LastName>
        </Author>
```

```
        ...
      </Authors>
      <Proceedings ShortName="PLDI">
        <PublishingYear>2007</PublishingYear>
        <ProceedingsTitle>Proceedings of the ACM SIGPLAN...
        </ProceedingsTitle>
        <Location>
          <City>San Diego</City>
          <State>California</State>
        </Location>
        <Dates>10-13 June</Dates>
      </Proceedings>
    </ArticleInProceedings>
  </Paper>
</Publication>
```

3.2. Constraining document structure

The examples above show the three types of structural constraints that define the directions into which the user can develop a document. These are: repetition, optionality, and choice.

Repetition is expressed as a `xt:repeat` element. It allows the user to insert its children elements between `minOccurs` and `maxOccurs` times. The example below shows how the *open* component of the opening hours table shown in Figure 4 defines a repetition of time slots which are themselves declared in a *slot* component. The generated XML content is visible on Example 4.

Example 4. Extract from the timetable template with repetition

```
<xt:component name="slot">
  <p>                    <!-- 7:00 opening hour hint -->
    <xt:use types="text" label="begin">7:00</xt:use>
    <span> - </span>     <!-- 24:00 closing hour hint -->
    <xt:use types="text" label="end">24:00</xt:use>
    <xt:menu-marker size="16"/>
  </p>
</xt:component>

<xt:component name="open">
 <br/>
 <xt:repeat minOccurs="1" maxOccurs="*" label="slots" >
   <xt:use types="slot" label="slot"/>
 </xt:repeat>
</xt:component>
```

Example 4 also shows the `xt:menu-marker` element that indicates where to insert the repetition buttons. It can have a `size` attribute to set the size of these buttons. In absence of the `xt:menu-marker` element, the buttons will be inserted after the last children of the `xt:repeat` fragment. You can notice that it can be declared inside the definition of a component type which is repeated, and not only directly inside the `xt:repeat` element.

Optionality is expressed as an `option` attribute which can be declared on the `xt:use` element inserting a component type in the document as shown on Example 5. When the value of `option` is `set`, the component target XML content is by default generated in the document but this can be changed by the user, by unchecking the checkbox which is displayed in the editor. When the value of `option` is `unset`, the component target XML content is not generated automatically, but this can be changed by the user, by checking the checkbox. When the attribute `option` is not defined, the XML content is always generated in the document. It is also possible to make parts of a document template optional using an `xt:repeat` element with `minOccurs` set to 0 and `maxOccurs` set to 1.

Example 5. Extract from the menu template with optionality

```
<p class="specialtyComment">
  <xt:use types="text" label="comment" option="unset">commentaire</xt:use>
</p>
```

Finally choice is expressed as an `xt:use` element where the `types` attributes declares several component types that can be inserted. The extract below shows that for a day of the week in the timetable template, in that case *Wednesday*, the user can select between two components, `open` or `closed`.

Example 6. Extract from the timetable template with choice (i.e. open vs. closed)

```
<xt:component name="open">
  ...see example above...
</xt:component>

<xt:component name="closed">
  <span/> <!-- this generates no XML content -->
</xt:component>

...
<td><xt:use label="wednesday" types="open closed"/></td>
...
```

3.3. Defining the XML content model

The template language drives the generation of a target XML content model with only two instructions: the `label` attribute and the `xt:attribute` primitive component type inclusion element.

The `label` attribute can be set on any `xt:use` component inclusion element, on any `xt:repeat` component repetition element, or on the `xt:head` element. Each `label` attribute generates a new XML element in the target content model during the serialization process. Similarly, when an `xt:use` element includes a choice of several component types, the selected component type generates a new XML element name after the component type name. The `label` attribute set on the `xt:head` element defines the target content model root node.

For example, the `<xt:use label="wednesday" types="open closed"/>` instruction in the opening hours table example generates a `<wednesday>` element that may contain either an `<open>` or `<closed>` child. If the user selects the *open* component type, this component generates a `<slots>` element (see the label of the `xt:repeat` in Example 4) that contains a repetition of `<slot>` elements with their own content model. This is illustrated below:

Example 7. XML content extract generated from the document on Figure 4

```
<wednesday>
  <open>
    <slots>
      <slot>
        <begin>10:00</begin>
        <end>16:00</end>
      </slot>
      <slot>
        <begin>18:00</begin>
        <end>23:00</end>
      </slot>
    </slots>
  </open>
</wednesday>
```

The `xt:attribute` element is a special component type inclusion element which can be used as an alternative to an `xt:use` element. The difference is that the target XML content model will be treated as an XML attribute instead of an element. This attribute, named after the *name* attribute of `xt:attribute`, will be attached to the current target XML element. As a consequence, the `xt:attribute` element can only include primitive component types that generate text data in the content model.

As an example, the `xt:attribute` element in the *price* component of the menu template visible on Figure 4 and listed below generates a `currency` attribute with

the possible values *EUR*, *CHF* or *USD*. In that particular case the choices in the popup menu are labelled from the content of the `i18n` attribute, here some currency symbols. It uses the primitive editor `select` which is described in Section 4.2.

Example 8. Price component type definition in the menu example

```
<xt:component name="price">
  <xt:menu-marker size="12"/>
  <xt:use types="text">prix</xt:use>
  <xt:attribute types="select" name="currency" values="EUR CHF USD"
                i18n="&#8364; CHF $" default="EUR"/>
</xt:component>
```

The *price* component of Example 8 can be inserted with `<xt:use types="price" label="price"/>`, which generates XML elements such as `<price currency="EUR">10</price>`.

4. Editing User Experience

AXEL employs two types of user interface controls: structure editing controls and primitive editor input fields that you can see in action in the examples of Section 3.1. The next sections explain these controls. The global editing user interface also supports tab navigation to jump from one primitive editor to the next (resp. previous) one. It is also possible to shift-click on a *minus* button to cut an item instead of deleting it, and to shift-click on a *plus* button to paste it at another position in the same repeat group.

4.1. Structure Editing Controls

Unselected document parts, either because they are repeated and their current count is 0, or because they are optional, are grayed out. We have implemented this behavior directly within the library with some CSS rules and some CSS generated `class` attributes.

The `xt:repeat` element generates a *minus* button and a *plus* button to respectively insert or delete a component. It can also replace the *minus* button with a checkbox if its `minOccurs` attribute value is 0: the checkbox is unchecked and the *plus* button is not visible if the user has not yet inserted a single item. The user can then insert a first item by checking the box. Then, if `maxOccurs` is greater than one, a *plus* button also appears to insert more items. The checkbox is replaced by a `minus` button as soon as the user has inserted more than one item, if this is allowed by `maxOccurs`.

The checkbox is also displayed when an `option` attribute is present on an `xt:use` element. In that case it is displayed as an isolated checkbox which is selected or not, depending on the existence of the component.

After some trials we finally decided to not give more feedback to help the user identify the boundaries of the components which are repeated or optional. Some subjective testing with automatically added borders were not satisfactory, mainly because they broke the document appearance. Our feeling is that in most cases the template content gives enough cues about the document internal structure. However, template authors can still create additional feedback, such as borders, using CSS and Javascript to achieve special effects on a per-template basis.

The `xt:use` element with multiple component types inclusion generates a pop-up menu that presents the different type names. These names can be internationalized by the individual component type definitions with an `i18n` attribute. Each time the user selects a different component type for inclusion, the currently visible component is replaced by the selected one. Actually the pop-up menu, an HTML `<select>` element, is inserted in place of the `xt:use` element, just before the included content.

4.2. Primitive Editor Input Fields

The library needs also to handle text user input so that users can create XML content, and not just an XML structure. This is done through primitive component types which are implemented by primitive editors. AXEL has a plug-in mechanism for creating new primitive editors by adding Javascript classes. Currently we use two of them: a `text` primitive editor, and a `select` primitive editor.

The `text` primitive editor manages a block of text which is displayed within a `<span>` XHTML element when not editing, and within an `<input>` or a `<textarea>` element when editing. It is possible to configure several parameters with the `params` attribute of the `xt:use` element that inserts a primitive editor. This attribute contains key-value definitions that influence the behavior of the editor. For instance the `layout` parameter can be set to *placed* so that the input field (i.e. `<input>` or `<textarea>`) dynamically replaces the `<span>`, or to *float* so that it is displayed over, with an initial equivalent shape. In all cases, these input fields grow as the user is typing, which gives a user experience similar to editing a WYSIWIG document such as in Word or Google Docs, and very different from editing a form, such as with XForms. Moreover, it is possible to configure them so that the input field inherits the typography from the document template, or to use `class` attributes to apply CSS effects to it.

We have extended the `text` primitive editor with the notion of *filters* that can be set on the editor. These filters, as the primitive editor, are implemented with Javascript classes following a specific API. Basically they can filter the target XML content to support complex XML structures which cannot be defined just with XTiger canonical constraints. They can also dynamically change the `<span>` static text display to any other XHTML presentation for the data. For instance, this article has been edited mainly with a `text` primitive editor that uses a `wiki` filter for creating

verbatim or emphasized text within paragraphs. This filter is shown in action on Figure 7. We have also managed for some configurations to position the cursor just after the character that was clicked to bring up the editing view. This creates a nice *click through* illusion that contributes to give a WYSIWIG feeling.

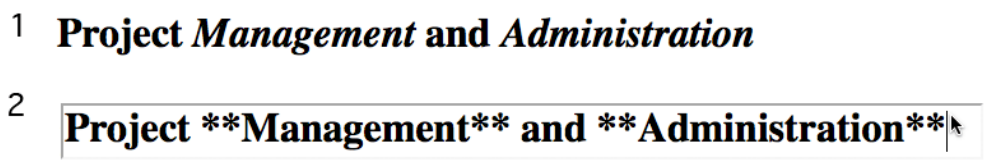


Figure 7. Text primitive editor with wiki filter while viewing (1) and editing (2)

The `select` primitive editor is rendered with a pop-up menu for directly selecting a value for an `xt:attribute` element as presented previously. It appears when the user clicks the current value of the field, which is displayed within a `<span>`, as with the `text` primitive editor. Figure 8 shows the popup menu after the user has clicked on the top *CHF* currency to change it.

Figure 8. Example of popup menu of the primitive editor "select"

Additional primitive editors will be created in the future to enhance the user experience, ranging from calendar selection controls – to pick up dates –, to file uploader editors – to insert images. This is fully supported by the XTiger XML concept of primitive component type.

4.3. Customization of Editing Behavior

The authors of template have the freedom to customize the editing user experience through CSS and Javascript programming. For this purpose, AXEL offers predefined CSS classes which are added to the generated user interface controls. For instance, we have experimented a very useful feature that consists in hiding all the structure editing controls by adding a *preview* class attribute to the `<body>` element of a template and some CSS rules. This gives an overview of the document, as in Figure 9.

The document of Figure 9 is still editable as all the primitive editors are functional, however the user cannot change the structure (adding or removing components in repetitions and changing the selected component of a `xt:use` choice). This can be useful to make light corrections (spell-checking, typos) to a document or to

translate it. This opens up future possibilities to generate different user interfaces for different tasks.

Figure 9. Menu document with all structure editing controls hidden but still editable

5. Server Integration

AXEL is designed to serve as the XML authoring layer for REST applications. It is fully implemented in Javascript; hence it can be executed in any browser providing reasonable support for the DOM model. This section outlines the main features of the library for that purpose.

5.1. Client-side Library

AXEL is a client-side Javascript library that takes as input an XHTML DOM tree containing the template. It transforms it into an XHTML tree containing the editing user interface and some Javascript objects for controlling editing. A programmer creates an editor by first instantiating an `xtiger.util.Form` object. The constructor parameter is a location path to a folder that contains the four images used by the editing user interface. Then s/he can set the template to transform with a call to `setTemplateSource` before calling the `transform` method to actually generated the editing user interface.

Example 9. Javascript code to transform a template loaded in the document object

```
var form = new xtiger.util.Form("{path-to-images}");
form.setTemplateSource(document);
form.transform();
```

By default the template is transformed in place. It is also possible to indicate a target XHTML document and a node inside which the result should be placed. The document that hosts the result of the transformation must include some specific AXEL CSS rules.

This very thin API is designed to embed XML authoring in any web application. For instance a programmer can build a generic XTiger XML editor that loads an XTiger XML template into an `iframe` element and transforms it to generate the editing user interface. It is even possible to directly put the code to generate the editor directly within a template file. This enables funny applications such as sending a template document by email to somebody, who can open it in his or her browser to make some editing, and then save the results to a server (this requires a little Javascript programming as explained in Section 5.2).

5.2. XML Serialization

The editor object has a function that serializes the current content of the document into the target XML content model. That function takes as input a logger object for accumulating the result while iterating on the document. A specific `xtiger.util.DOMLogger` is available for that purpose. It provides a method to dump its content into a string which can then be sent to the server as an HTTP PUT or POST request using the Ajax XHR object, now implemented in all major browsers.

Example 10. Javascript code to serialize the XML content of a document

```
var logger = new xtiger.util.DOMLogger ();
data = form.serializeData (logger); // form from above
var xmlString = logger.dump();
```

Reciprocally, the editor object provides a function that loads some XML data into a document. That function takes as input a DOM data source for iterating on the XML data while iterating on the template. A specific `xtiger.util.DOMDataSource` is available for that purpose. This object accepts an XML document object as input, such as the *responseXML* object constructed by an Ajax XHR object, or it can directly parse a string into an XML document object using a *DOMParser* object (or a simulated version if it is not available). This makes it quite easy to retrieve the XML data from a server side application and to inject it into the document. For experimental purposes we have also built a DOM data source based on the E4X Javascript API to natively parse XML data.

Example 11. Javascript code to load XML content from an Ajax XHR object into a document

```
var xhr = new XMLHttpRequest();
...
var source = new xtiger.util.DOMDataSource();
source.initFromDocument(xhr.responseXML);
form.loadData(source, result); // form from above
```

We currently have successfully integrated the library with WebDAV servers, with Orbeon Forms applications, and even with a Ruby on Rails application.

5.3. Experiments and Performances

The Javascript library has been developed and tested with Firefox (version 3 and above). Thanks to this client-side approach combined with the use of standard web technologies, we provide a platform-independent solution. It runs on recent versions of all major browsers although we did not do extensive testing on all platforms yet. The Javascript library compressed with the Yahoo UI compressor is less than 96 kB, including the basic editor plug-ins that were required to write this article.

We have experienced this Javascript library with several document templates in order to evaluate this solution in terms of performance and usability. The tested templates range from simple form-based structures, such as the menu example given previously, to more complex templates, like the template used for writing this paper. The table below lists some properties and some results obtained with different use cases of templates. Time results of these tests have been obtained on a MacBook Pro with 2.16 Ghz Intel Core Duo with 2GB memory running Firefox 3.5.3 (a 3 years+ computer).

Template Name	# nodes	Tree depth	Typical document size	Download time	Click response time ("-" means not noticeable)
Menu	89	5	2kB	-	-
Curriculum	173	4	2kB	-	-
Article	317	7	10-100kB	0.2-2s	-
Specification	243	6	50-500kB	1-36s	-

Figure 10. Loading XML documents into a template and editing with AXEL

We can notice that the download time is not noticeable for normal size XML documents. Download time depends on the complexity of the template characterized by its total number of node (*#nodes*) and the number of nested components (*tree depth*), but we haven't precisely characterized the relation yet. This is left as a future work in order to give guidelines to template authors. XML download time is linear

with the XML document size and, for instance, it took up to 36 seconds for a 500 kB XML document on this 3 years+ hardware configuration, which can be considered very long. However, the document was 190 printed pages in the browser and it was by itself an excellent surprise to be able to edit so long documents within the browser. This template also uses a specific plugin editor to edit logical expressions that take time to convert when loading and saving. The times given in the table do not take into account the preliminary download time of the template and the transformation time to turn the template into an editor, which are not noticeable in all the cases. The files were directly loaded from the local drive. We could also have presented the results with Safari which are even better.

We do not show a similar table for the time taken to generate XML content from the edited document because it is usually not noticeable, about 5 seconds in the worst case from the examples in the table above.

In all cases we noticed pretty good response time in the browser, between a click and the display of the input field. Usually not noticeable, it stays below half a second even with 500kB+ XML documents. These results obviously come from our client-side editing code, where only load and save operations pay for communication time. Thanks to recent web technologies, modern web browsers can become valuable XML document editors with the addition of a very tiny amount of code.

The various templates we have tested aim at evaluating both the expressive power of the XTiger XML template language and the quality of the authoring interface. The original XTiger language implemented in the Amaya stand-alone application has demonstrated its ability to define templates for editing a wide range of web documents [13], and due to its filiation with that language, XTiger XML inherits this feature. But with XTiger XML we gain in usability for simple applications, while keeping the expressive power for complex XML structures. Indeed, menu and curriculum vitae documents, for instance, can be straightforwardly edited by XML unaware people (for instance, a restaurant chef). For such "minimal" templates (few element types and little structural depth), the interface can be considered as minimal and allows users to focus on what they have to do (Hick's law [3]). Moreover, editing controls (menus for choices, add/suppression buttons for repetitions, text editing areas) are always contextually located and therefore mouse movements are kept minimal (Fitts's law [2]). We can notice that this simplicity at the authoring step does not prevent rich server-side services to be provided, such as databases feeding or high quality publication.

6. Related Works

By design, XTiger XML is an extension of the original XTiger template language. We tried to keep both languages as close as possible to each other, and indeed XTiger XML adds only a few elements and attributes. The main difference lies in the editors that are based on the languages. The original XTiger template language has been

implemented in the Amaya web editor, and as far as we know, it is available only in this editor. For XTiger XML, a very different approach was taken. It is implemented as a Javascript library, and then, it can run in any modern web browser.

As stated above, XTiger XML can be used as a form editing framework. It could even be possible to create plug-in modules to handle all the usual form interactors (drop lists, radio button, etc.), but the current syntax is not powerful enough to express non-structural constraints on the editable data model. However, this is not our main objective. Our priority is to develop further the library for editing document-oriented data and for managing collaborative editing constraints in the future.

The approach presented in this paper is different from wiki-based authoring environments, which are common on the web, such as MediaWiki in use by Wikipedia. First, it does not require the user to learn a specific syntax. Second, it is not limited to creating HTML content models. To some extent, semantic extensions of wikis such as Semantic MediaWiki make it possible to create annotated documents that contain structured information, but the requirement to learn a domain specific syntax still holds true. To cope with this issue, several template-based wikis have been proposed with more or less freedom for the author: the wiki template can be used simply as a seed document, without any control on the further authoring process, or it can on the contrary act as a very strict editing framework, preventing the author from deriving from the expected structure unless he modifies the template [10].

An in-between approach is proposed in [4] where a post-hoc validation mechanism can be applied on instances. While this can be satisfying for simple web pages, it becomes boring for the author when structures are more complex. Again, users have to choose between expressiveness and usability [5]. We claim that both can be provided thanks to the use of a rich template language and new editing paradigms such as those described in this paper. It can be noticed that a similar approach was recently proposed in [6] for the authoring of semantic annotations in MediaWiki, where editing controls are generated from ontologies.

XTiger XML with AXEL is very close in its objectives to generic XML authoring applications, especially those that use style sheets or XSLT transformations to associate a visual presentation with an XML content model defined by a schema. Such applications generate a WYSIWYG editor based on this visual presentation and driven by the schema. For a long time these tools were available only as desktop applications, such as XMLSpy or Oxygen. But with the success of web applications, browser-based WYSIWYG XML editors are appearing now, such as XOpus.

However, there are still two important differences:

- WYSIWYG XML editors, even when they run in the browser, require at least an XML schema and some XSLT transformations. This implies that they are used only for documents and data for which such resources exist, or that new schemas and transformations must be created for new types of documents and

data. We believe that the use of document templates instead of schemas and transformations languages makes XTiger XML easier to learn and to manipulate.

- XTiger XML and AXEL work on the web. They use the browser not only as the local platform for running the editor, but also as a web client that can download and upload documents on remote web servers. This allows everyone on the web to edit XML data and documents and to feed servers with well structured data. This also provides a global infrastructure for sharing and collaborating on XML contents.

7. Future Works

XTiger XML and AXEL are still under development, firstly to extend authoring services with new primitive editors. We have also identified some issues that will be addressed shortly.

7.1. Validation

In its current state, AXEL can be compared to XML tools based on XSLT, regarding validation: there is no formal way to make sure that the XML data generated by the tool are valid against a given schema, except by validating every instance. But there are solutions to this issue. XTiger can be seen as a regular tree grammar, and then an XTiger XML template can be compiled into a schema language (DTD, XML Schema, Relax-NG). But instead of producing schema language syntax equivalent to a template, our plan is to compile XTiger into the logical representation introduced in [8]. We can then use the validation services offered by the XML Resolving Solver [9], which is based on this logical representation. The Solver can statically check properties of XTiger XML templates, in the same way it checks properties of schemas.

A property that the Solver could statically check is whether a template always generates XML instances that are valid against a given schema. This may ensure for instance that the data entered with a template can be safely stored in an XML data base defined by its XML Schema.

7.2. Data Delivery

A second issue concerns data delivery on the web. Producing pure static XHTML read-only format remains of importance. This is required to enhance the probability to be indexed by major search engines. Explicitly including micro-formatted elements is another concern; for example, applications aiming at aggregating information (such as the Kritx review aggregator) rely on the availability of such information in XHTML documents. Those requirements are not met if documents are published only through a client-side Javascript transformation process.

We have manually transformed several document templates into an XSLT transformation that takes an XML content conform to the implicit document model as input, and generates an XHTML document that looks like the document generated from the document template for editing. We are confident that this process can itself be programmed as an XSLT transformation taking as input a XTiger XML template and generating the target XSLT transformation. Doing this will allow us to provide a full XML editorial chain within the browser.

7.3. Template Editing

The templates we have used up to now were created by various means. The simplest templates were written totally "by hand". In some cases we have used an XHTML editor (especially Amaya), for creating a skeleton document, that was then edited as a text file for adding the XTiger XML elements. Some other templates were almost entirely created with Amaya, which includes a feature for editing the original XTiger language. But, as stated earlier, there are differences between the XTiger language implemented in Amaya and the XTiger XML language. This implies that templates produced by Amaya have to be tweaked "by hand" when used by AXEL for editing XML documents. An obvious work item for the future is to adapt Amaya and implement all the extra XTiger features included in XTiger XML.

In addition to these means, we think that it would be interesting to also have a template authoring application running in the browser, to make that feature more widely available, without the need for installing a dedicated application. Such a tool could start from an initial XHTML document and allow the user to insert XTiger XML elements, in the same way as in Amaya. It would also be interesting to go a step further, when an XML Schema (or Relax-NG) is available for the target XML language. This schema could then be used to drive the interaction with the user in order to create a template consistent with this schema.

8. Conclusion

The XTiger XML language and its implementation in the AXEL library make it easy to create and update XML data and documents on the web. The language is very simple: it contains only a handful of elements and a few attributes. This allows anyone familiar with HTML to create templates. The editing engine provides a simple and intuitive user interface, which allows any web user to create and edit XML data.

Being implemented as a light Javascript library that runs in a web browser, the editor can be deployed very efficiently and it lets users free to work with their preferred tool. Because templates are based on XHTML, they allow XML data to be involved in any web application and to be collected by average web users. Building

on the ubiquitous web infrastructure, this approach helps to put XML within reach of everyone.

9. Acknowledgements

Early work on this project has been initiated in the framework of the PALETTE Integrated Project supported by the IST programme of the European Commission (DG Information Society and Media, no. 028038). We especially thank Thibaud Latour from CRP Henri Tudor, our partner in Luxembourg, for his active support. Further development of the XTiger XML language is currently supported by the Innovation Promotion Agency of Switzerland under the grant No 10813.1 PFES-ES, a project in collaboration with the MadeinLocal company (www.madeinlocal.com[1]).

Bibliography

[1] Amaya: Home page. http://www.w3.org/Amaya/

[2] P. M. Fitts: The information capacity of the human motor system in controlling the amplitude of movement. Journal of Experimental Psychology, Vol 47 pp. 381-391, 1954

[3] W. E. Hick: On the rate of gain of information. Quarterly Journal of Experimental Psychology, Vol 4 pp. 11-26, 1952

[4] A. Di Iorio – F. Vitali: Wiki content templating. WWW 2008, Beijing, China, pp. 615-624, ACM Press, 2008

[5] A. Di Iorio – S. Zacchiroli: Constrained wiki: an oxymoron?. WikiSym '06: Proceedings of the 2006 international symposium on Wikis, pp. 89–98, ACM Press, 2006

[6] A. Di Iorio – S. Duca – S. Righini – D. Rossi – F. Vitali: Customized Edit Interfaces for Wikis via Semantic Annotations. Workshop on Adaptation and Personalization for Web 2.0, UMAP'09, June 22-26, 2009

[7] F. Flores – V. Quint – I. Vatton: Templates, Microformats and Structured Editing. Proceedings of the 2006 ACM Symposium on Document Engineering, DocEng 2006, pp. 188-197, ACM Press, October 2006

[8] P. Genevès – N. Layaïda – A. Schmitt: Efficient static analysis of XML paths and types. PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 342–351, ACM Press, 2007

[1] http://www.madeinlocal.com

[9] P. Genevès – N. Layaïda: XML Reasoning Made Practical. Proceedings of the 26th IEEE International Conference on Data Engineering, ICDE 2010, IEEE, March 2010. http://wam.inrialpes.fr/publications/2010/ICDE10demo.pdf

[10] A. Haake – S. Lukosch – T. Schummer: Wiki-templates, adding structure support to wikis on demand. WikiSym' 05, San Diego, USA, pp. 41 – 51, ACM Press, 2005

[11] É. Kia – V. Quint – I. Vatton: XTiger Language Specification. http://www.w3.org/Amaya/Templates/XTiger-spec.html

[12] P. Nálevka: Advanced Automated Authoring with XML. XML Prague 2009

[13] V. Quint – I. Vatton: Structured Templates for Authoring Semantically Rich Documents. Proceedings of the 2007 international workshop on Semantically aware document processing and indexing, ACM International Conference Proceeding Series; Vol. 259, pp. 41-48, ACM, 2007

[14] S. Sire: XTiger XML Language Specification. http://media.epfl.ch/Templates/XTiger-XML-spec.html

Multimedia XML

Robin Berjon
Robineko
<robin@berjon.com>

Abstract

The past couple of years have seen a drastic improvement of the web platform in its multimedia capabilities. From styled text with a few pictures, embedded objects, and limited interactivity we are now moving towards far greater integration of multiple media technologies resulting in a much richer platform being available. As multimedia XML reaches a level of maturity that makes it usable in the large, it is time to take a global look at what is available or becoming available today.

Keywords: XML, multimedia, SVG, SMIL, Web, XHTML, Javascript, APIs, W3C

1. Introduction

When the Web went mainstream, everyone was abuzz about how it finally brought multimedia and interactivity to the masses. Looking back to those heady days and opening up a website from the late nineties, it can be quite amazing to think that anyone did indeed become excited: multimedia meant Comic Sans and animated GIFs, and interactivity was largely about slow round-trips to the server. We have come a long way over the past decade, perhaps longer than many yet realise.

At about the same time, XML was unleashed upon a largely unsuspecting world, brimming with hopes that it would be the Web's SGML. While XML has without a doubt succeeded, but its usage inside of browsers is nowhere near as common as it is almost everywhere else in the ecosystem. Some are quick to mark it as having failed there, but is that really the case? We will look at how multimedia XML technologies are indeed available inside browsers, and also at whether greater integration is truly needed.

2. SVG

SVG has long been the darling poster child of multimedia XML. It certainly has been through a rather rocky first decade, or as Brad Neuberg put it talking about Web standards in general and SVG in particular:

First they're ignored, then they're hyped, then they're written off for dead, then they start getting real work done.

—Brad Neuberg

A quick survey of today's situation shows that SVG has finally reached that final stage. Every browser except Internet Explorer supports it natively, and Microsoft has recently joined the SVG Working Group and is actively participating, which has spurned speculation that the next release of their browser will also support it.

But a hypothetical upcoming release is insufficient to make SVG usable. The major step forward that was announced during the Google-hosted SVG Open last year was the release of svgweb[1]. It is a drop-in Javascript library that can will make SVG fully functional on any browser with the same penetration level as Flash: if the browser supports SVG natively it will not interfere, if it doesn't it will bring in a Flash emulation of SVG support. This provides precisely the backwards-facing transition tooling that one needs in order to start deploying SVG today, including embedding it inside HTML.

As for any other technology, support level from browsers is not at 100%, but it is improving fast. Perhaps more importantly, while the SVG support table[2] may give the impression that browsers are lagging behind, one should keep in mind that the SVG specification is extremely large, and contains a lot of features that many people rarely need. Because of that, even 60% coverage is enough to make it very useful.

As an amusing aside, not only do we now have a Flash shim to support SVG on browsers that don't yet run it natively, but we also have Gordon[3], a pure-Javascript implementation of SWF Flash that runs on browsers that don't have Flash support (e.g. the iPhone and friends) and emulates it using SVG. If anything, this should give an idea of the implementations' maturity.

Looking forward, what is needed to improve SVG support in browsers is a clearer definition of the core in order to improve interoperability. This will probably involve removing some of the less-used features from the core so that implementers can focus on the important parts more easily.

3. X3D & WebGL

3D on the Web has also had its share of ups and downs. In 1997, everyone wanted to do VRML but it vanished from view almost as fast as it had arrived. From the ashes of the VRML project X3D was born, but it too spent many years being largely ignored, supported only in plugins that few people knew of or applets that took

[1] http://code.google.com/p/svgweb/
[2] http://www.codedread.com/svg-support.php
[3] http://github.com/tobeytailor/gordon

too long to load. It was used extensively in some specific vertical industries, but most people wrote it off as dead.

The situation changed when a group of browser vendors joined forces inside the Khronos Group[4] in order to develop the WebGL[5] specification. WebGL is a low-level 3D API similar in concept to OpenGL but designed to work with Javascript that is intended for use inside browsers, specifically with the `canvas` element (current implementations of `canvas` only expose a 2D context, but the element's interface was designed so that it could expose many different types of graphical APIs).

This technology is still young and in the process of being developed, but it has solid traction and an alpha implementation of it has become available in Firefox, Safari, and Chrome builds. More importantly for XML aficionados, an open source Javascript implementation of X3D called X3DOM[6] (pronounced X-Freedom) has been created that can render X3D inlined inside HTML using WebGL. In the same way that svgweb makes SVG work in browsers that do not support SVG, X3DOM makes X3D work in browses that don't support X3D. Given that it relies on WebGL being available it is still a rather far away from being universal, but it nevertheless needs to stay on one's radar.

4. XHTML 5

This year's hype is all about HTML5, to the point that anything new is now branded "HTML5" in the same way that they used to be branded "Web 2.0" or "AJAX". There is a level of resentment in parts of the XML community against HTML5, sometimes because XHTML 2.0 was abandoned, sometimes because it places greater emphasis on the HTML serialisation. This resentment needn't exist: all the improvements that HTML5 brings (except those concerning the parsing model, naturally) are available equally in the XML serialisation. What's more, it is easy to author polyglot documents that can be processed identically as XML or HTML, giving the joint benefits of using the XML toolchain and the `text/html` media type that all browsers understand.

A complete overview of the improvements that the newer version brings would be the subject of a different article, but some items are worthy of being noted.

Of particular interest within this topic is the availability of native audio and video, alongside APIs to support them. This means that no plugin will be needed to include media object, but more importantly that they will be controllable in a way that is better integrated with the Web stack. Videos can be completely composited with the rest of the display, and the user interface to interact with them can be built entirely with web technologies. There are shims available that will swap in a

[4] http://www.khronos.org/
[5] http://www.khronos.org/webgl/
[6] http://www.x3dom.org/

plugin for browsers that don't yet support these new elements, so that for the most part they can be deployed today. The one big issue is that of codecs: some browsers only support MPEG H.264, while others only support the open Ogg Theora. The reason for this divide is that H.264, as all that MPEG produces, is crippled with strict licenses, patents, and royalties which naturally breeds strong opposition to it, while some vendors doubt that Ogg Theora can be sufficiently reliable and can be implemented in hardware (an aspect that is important for mobile browsers). Users can work around this problem by making both formats available (the `audio` and `video` elements can have multiple `src` child elements and the browser will use the first that points to a document that it can understand) but that requires one to encode the content multiple times. Hopefully this situation will be clarified over time; meanwhile several video websites such as DailyMotion and YouTube have released experimental versions that make use of this new functionality.

HTML5 also adds a number of new elements with improved semantics: `section`, `article`, `header/footer`, `nav`... While these may not be rocket science, they do provide commonly understood containers that allow for the content to be more easily repurposed (e.g. by moving the navigation bar to a different location on small screens).

Improvements to forms handling, while nowhere near the level of functionality offered by XForms are also convenient and alleviate the need for dumb validation.

Many more improvements, if only in clarity, have been made such as the improved integration of namespaces into the DOM, SVG inlining, or a `DOCTYPE` that you can actually type. Many are still in flux and it is common to hear concerns that the specification will not be released within one's lifetime. In some ways, that doesn't matter. The HTML5 specification is written in a manner similar to that in which Web software is now written: there is no version, just continuous improvement. And indeed, browsers do take the improvements into account as they go, without waiting for an official release.

5. SMIL

SMIL is such a tentacular piece of technology that it is hard to come to grips with. Version 3 was release a little over a year ago, and its table of contents alone prints out to twenty pages. Browser support is generally rather spotty, except for SMIL Animation which is supported by several — though certainly not all — and for which a shim (SmilScript[7]) was developed. The bits and pieces that were reused here and there in SVG or in MMS seem to either make people unhappy, or to go unused. It is therefore tempting to write it off as overengineered and dead.

But there is more to the SMIL story than that. One aspect that is hard to get right in any interactive scenario that involves timed media is the timing model: synchronising things that may not run at the speed at which they are supposed to (e.g. because

[7] http://en.wikipedia.org/wiki/SmilScript

of bad network or slow hardware) and keeping multiple timed elements harmoniously functioning together is a tough problem. SMIL does provide a concrete and solid solution here. As a result, even if it means that SMIL as a complete technology may not be used all that much, bits and pieces can be cannibalised and wired together in order to produce useful functionality.

One such example are SMIL Timesheets[8]. A commonly cited issue with SMIL Animation's integration into SVG is that animation does not change the actual DOM (`getAttribute` will return the base, non-animated value and will therefore be wrong) so that one has to use the clumsy and globally hated `baseVal/animVal` API to access values from script in an animated document. Furthermore, SMIL does not cleanly define its relationship to the DOM so that inserting a SMIL element at runtime may produce random results. Timesheets help with the latter problem by making the SMIL information external to the DOM that it applies to in the same way that CSS externalises styling information. This solves the problem with manipulating SMIL content at runtime, and opens the door to a new API that would expose animation information in the same way that computed style is.

So while SMIL is not quite ready for prime time, parts of it are becoming available in browsers and there is hope that its more useful segments will be salvaged in future.

6. CSS

CSS is another huge family of several dozen specifications[9], but unlike SMIL it is seeing a renaissance in browser implementations after the long lull of the Browsers' Peace (or Pax Explorana).

Oftentimes, CSS isn't seen as an XML technology because not only is it not expressed in XML syntax, but further it uses its own selectors instead of XPath. I contend that that view is incorrect. While the situation could be improved, CSS can on its own bring life to XML documents, without requiring any intervention from HTML. There will be limits to the interactivity that it can produce without script (for instance, it won't be able to generate links) but it can be used effectively to display XML inside of a browser.

The two first somewhat newer tools that it brings to the table are its support for namespaces[10] (which is now mature) and its recently upgraded Selectors[11] that provide new abilities such as matching on attribute values, on position and position of the element type, negations, and better sibling matching. There is no doubt that compared to XPath this set is small, but it is designed to be fast when used within

[8] http://www.w3.org/TR/timesheets/
[9] http://www.w3.org/Style/CSS/current-work
[10] http://www.w3.org/TR/2008/CR-css3-namespace-20080523/
[11] http://www.w3.org/TR/2009/PR-css3-selectors-20091215/

a document that is updated continuously, and is sufficient much of the time for styling purposes.

Browsers are now finally bringing together much improved support for CSS Fonts[12], which mean that we are being progressively liberated from a choice between Comic Sans, Verdana, and using images. Combined with improved support for paged media[13] and printing[14] this produces the ability to provide Web documents that can be elegant in print as well without having to resort to PDF. Again, some may complain that it is not as powerful as what XSL:FO provides, but it is an undeniable step forward.

Additional improvements also mean that much better-looking documents are progressively becoming possible within the browser, in XML, and without having to resort to intensive image editing and ugly tricks. Examples of that include improved backgrounds and borders[15] with its multiple background effects, native round corners, or slicing, increasing support for multi-column rendering[16], or Media Queries[17] that enable authors to produce content that automatically adapts itself to multiple target devices. A way of specifying gradients, compatible with SVG, is also on its way.

More visually impressive, the recent addition of transitions, animations, and 2D/3D transformations to the CSS toolset also help bring multimedia richness to potentially any XML document.

7. Getting It All Mixed Up

Not quite multimedia, MathML is nevertheless both XML and beautiful when properly rendered. It currently enjoys a relative amount of support in browsers (natively in Firefox and Opera, using a plugin elsewhere). Likewise distantly related, W3C Widgets are becoming available. They are essentially Web documents wrapped in a Zip archive with a simple XML configuration file and ran as applications.

What's interesting with the above two is that while not exactly in the multimedia XML category they can nevertheless be mixed up with all the aforementioned technologies, and are reaching outside the browser. Indeed, for all the limitations in XML support that the browser platform currently has, it does carry the promise of truly compound documents and demonstrations have been made that integrate all of the above together in a single document.

Discussion is ongoing as to whether namespaces are indeed the best way of achieving this mix, and it is not impossible that a redefinition of vocabulary com-

[12] http://www.w3.org/Style/CSS/current-work#fonts
[13] http://www.w3.org/Style/CSS/current-work#paged-media
[14] http://www.w3.org/Style/CSS/current-work#print-profile
[15] http://www.w3.org/TR/css3-background/
[16] http://www.w3.org/TR/2009/CR-css3-multicol-20091217/
[17] http://www.w3.org/TR/2009/CR-css3-mediaqueries-20090915/

posability will emerge in the coming year that will produce even better integration than what is currently available.

The one most sorely missing component in this family sadly remains XBL2[18]. It is a powerful technology, derived and improved from the original Mozilla XBL, and several browser implementers have had it as the second or third thing on their TODO lists for several years. Unfortunately, it seems to stay stuck there.

That is a shame because XBL2 could in fact be used to implement SVG, MathML, or any number of other languages including FO assuming you had the technology to produce the graphics rendering (e.g. Canvas2D and CSS). It might not be trivial for some languages and it might not be fast in all cases but it would work. It has the power not only to render elements but also to make them behave in specific manners and expose the regular APIs to scripting.

XBL2 solves a lot of the HTML extensibility debate that is currently ongoing by putting the choice in the users' hands as to whether they want to use namespaces or other mechanisms, leaving only the question of how to extend core, low-level browser functionality (such as access to device capabilities).

Say that in order to better capture email discussions that are archived as HTML documents I wished to create an element representing philippics. It would naturally be stylable, and would also expose its own specific API as part of the DOM (e.g. philippic.onhalfwaythrough = doze;). Using XBL2, I can build a binding that does just that. Then when it comes to syntax, the author using my binding has a choice of:

- if using XHTML (or any other XML language), have a `<robin:philippic>` element:

```
@namespace robin url("http://berjon.com/ns/diatribes#");
robin|philippic { binding: url("diatribes.xml#philippic") }
```

- just adding a `<philippic>` element to HTML:

```
philippic { binding: url("diatribes.xml#philippic") }
```

- binding based on a class (`<div class='philippic'>`):

```
.philippic { binding: url("diatribes.xml#philippic") }
```

- or on an attribute (`<aside type='philippic'>`):

```
aside[type=philippic] { binding: url("diatribes.xml#philippic") }
```

Or in fact any CSS that draws my fancy (though the above would be the expected big ones).

XBL2 is, effectively, an extensibility mechanism for any format that has a DOM and runs in UAs that support CSS. It is limited in the higher-level semantics that it

[18] http://www.w3.org/TR/xbl/

can provide but then so are namespaces; and it would be possible to use RDDL in conjunction with XBL2. It also isn't necessarily tied to JS and it could be used as a GRRDL-like mechanism.

Some may balk at the notion of author-chosen syntax but that's not very different from the fact that authors can today turn pretty much any element into any other with a good dose of CSS. It also means that user style sheets can override disliked UA behaviour.

This does make repurposing content harder since one has to process the CSS as well as the content but this problem has two sides: if you're doing fully open ended indexing of the web you probably need to process CSS anyway as otherwise content can game you by setting "display: none" on things a crawler should see but not users; conversely if you're processing content within a reasonably controlled environment you can impose a choice of syntax.

Extensibility-wise this only leaves the case of core functionality that cannot be easily added without extending browsers (I wonder how much of this there is that isn't about accessing a device's capabilities). Say you wanted to make audio processing available in the browser using for instance a syntax similar to SVG's filters. You would probably need some form of audio functionality added to the browser. I'd make the case that much if not all of this functionality ought to be added as an API and have its declarative side handled by XBL2. It may be time to explore whether the Web's architecture should generalise such an approach or not, and if so to what degree.

Within the extensibility debate, this leaves open the question of how to handle the case of an XBL binding becoming popular enough that it ought to migrate into, say, HTML 8. Could maybe XBL2 have a form of use-native-if-available attribute? It will be interesting to see going forward if we may simply dub XBL2 as HTML5's extensibility mechanism and declare victory,

8. Does XML in the Browser Really Matter?

Having looked at what's available and upcoming, and taking into account the fact that it's unlikely that in the close future browsers will be adding much in the way of core XML support, is it important that we have more of it?

I contend that it isn't. If we think of the browser as the platform, as the operating system that it has become, and if we take into account the fact that Javascript is now massively faster and more reliable than it used to be, I believe that — perhaps with the exception of XBL2 — we do not need more than what is already becoming available, and the rest (e.g. FO or XQuery) can be built atop the existing using Javascript libraries. After all, if people have implemented Flash and Zip in Javascript, there is no good reason why XML technologies cannot be supported just as easily, in fact probably more easily.

As such, my recommendation to XML enthusiasts the world over is that we stop whining and whinging about the level of XML support in browsers and build the tools we need atop what we have.

XQuery in the Browser

The same as JavaScript, just with less code

Ghislain Fourny
ETH Zurich
<gfourny@inf.ethz.ch>

Markus Pilman
ETH Zurich
<mpilman@student.ethz.ch>

Daniela Florescu
Oracle
<dana.florescu@oracle.com>

Donald Kossmann
ETH Zurich
<donaldk@inf.ethz.ch>

Tim Kraska
ETH Zurich
<tim.kraska@inf.ethz.ch>

Darin McBeath
Elsevier
<D.McBeath@elsevier.com>

Abstract

Since the invention of the Web, the browser has become more and more powerful. By now, it is a programming and execution environment in itself. The predominant language to program applications in the browser today is JavaScript. With browsers becoming more powerful, JavaScript has been extended and new layers have been added (e.g., DOM-Support and XPath). Today, JavaScript is very successful and applications and GUI features implemented in the browser have become increasingly complex. We aim at improving the programmability of Web browsers by enabling the execution of XQuery programs in the browser. Although it has the potential to ideally replace JavaScript, it is possible to run it in addition to JavaScript for more flexibility. Furthermore, it allows instant code migration from the server to the client and vice-versa. This enables a significant simplification of the technology stack.

The intuition is that programming the browser involves mostly XML (i.e., DOM) navigation and manipulation, and the XQuery family of W3C standards were designed exactly for that purpose. We developed an extension to XQuery for Web browsers and implemented it as a plugin for Internet Explorer and Firefox. Plugins for Chrome, Safari and Opera are on their way. The purpose of our contribution is to give a demo showing the usefulness and simplicity of XQuery for the development of AJAX-style applications.

Keywords: XML, XQuery, browser, script, scripting, JavaScript, DOM, HTML, XHTML, events, stylesheets, CSS, Client-side programming, mash-up

A full paper was published in the proceedings of the WWW 2009 conference [2]. A demo paper was published in the proceedings of the SIGMOD 2008 conference [1]. The plugin itself is available for download [3].

Bibliography

[1] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska: XQuery in the Browser. Demo paper, SIGMOD 2008, June 2008, Vancouver, Canada. http://portal.acm.org/citation.cfm?doid=1376616.1376769

[2] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, D. McBeath: XQuery in the Browser. 18th International World Wide Web Conference (WWW2009), April 2009, Madrid, Spain. http://www2009.eprints.org/102/1/p1011.pdf

[3] XQIB Web site and download http://www.xqib.org

Topic Maps run from XML and is coming back with Flowers

FLWOR-Style in the Topic Maps Query Language using a TMAPI-based implementation

Benjamin Bock
Topic Maps Lab, University of Leipzig
<bock@informatik.uni-leipzig.de>

Sven Krosse
Topic Maps Lab, University of Leipzig
<krosse@informatik.uni-leipzig.de>

Lutz Maicher
Topic Maps Lab, University of Leipzig
<maicher@informatik.uni-leipzig.de>

Abstract

In its history, Topic Maps developed from a syntax-based standard to a pure data model without any syntax defined within its core data model. The syntaxes defined by the ISO for the exchange of Topic Maps are conforming to the generic data model, one of them, XTM, being based on XML. The usage of XTM without a Topic Maps engine is cumbersome because of the generalized schema and the merging rules. For example, extracting useful information from XTM using XSLT requires to query for the typing topics, which is a new subquery just for selecting the right subject whereas it was the entity name in a domain specific XML format. Querying the properties, called Names and Occurrences in Topic Maps, requires additional subqueries because their types and scopes are again Topics and not simple XML entity- and attribute names. The Topic Maps Query Language which is the latest draft in the ISO standardization presented here allows formulating queries against a Topic Maps store in a concise way and outputting the result in various representations. Our implementation TMQL4J uses any TMAPI-compatible store to operate on and allows optimized queries and outputting domain-specific XML. This is demonstrated by generating an ATOM feed for the subject identity record service subj3ct.com.

Keywords: Topic Maps, XML, Topic Maps Query Language, TMQL, TMAPI, XTM, XSLT, FLWOR

1. Introduction

Topic Maps was developed as an SGML and HyTime-based standard [1] a decade ago. With the advancements and popularity of XML, the 2001 version was completely transitioned to XML [2]. The XML version of Topic Maps disclosed the paradigm to a wider audience and enabled several implementations. In the continuing progress of both, development and standardization, the stakeholders and editors came to the conclusion that the data model should be independent of a notation or syntactic representation [3]. While XML may be a good way to *exchange* topic maps, it is not always ideal to *handle* them as XML internally. Especially the merging of constructs - one of the core functionalities of the integration model ([4], Section 6) is described significantly easier using the specialized data model instead of a notation-based model. Consequently, the Topic Maps community focussed on the development of a data model and using XML Topic Maps just as one of several exchange formats.

Due to the way XML is intended to be used from a Topic Maps standards perspective, the XML schemata defined in [2] and [5] are not designed to be extended or modified ([2], Introduction). This major drawback eliminates the possibility to exploit one of the core features of the *Extensible* Markup Language: Extensibility. Another way is to be found.

First we are going to describe the Topic Maps Data Model and how it is currently accessed using APIs. Then we will illuminate different ways of representing and querying a topic map in different containers, document-based and in relational databases. After that we will compare several possibilities to query a topic map. Finally we will show an application of the here-proposed and recommended way to query a Topic Map for XML data, creating a feed for the Web 3.0 identity service subj3ct.com. We conclude with a self-assessment and outlook.

2. The Topic Maps Data Model

The Topic Maps Data Model [4] is defined in terms of XML Infoset [7]. It is a generic model to encode knowledge and connect encoded knowledge to relevant information resources. This means it is not dedicated to a specific usage or model but generally usable for any particular or general application. A *topic map* consists solely of *topics* and role-based *associations* between topics. Topics represent subjects of discourse which may be anything whatsoever, regardless of whether it exists or has any other specific characteristics, about which anything whatsoever may be asserted by any means whatsoever ([4], section 3.14). Topics consist of their characteristics, i.e. *names* and *occurrences*. Names in turn may consist of several *variants* thereof. Associations consist of *roles*, each role referring to a player which is one topic. Figure 1 shows the hierarchical aspect of this meta-model, the class names being representative for a set of instances thereof.

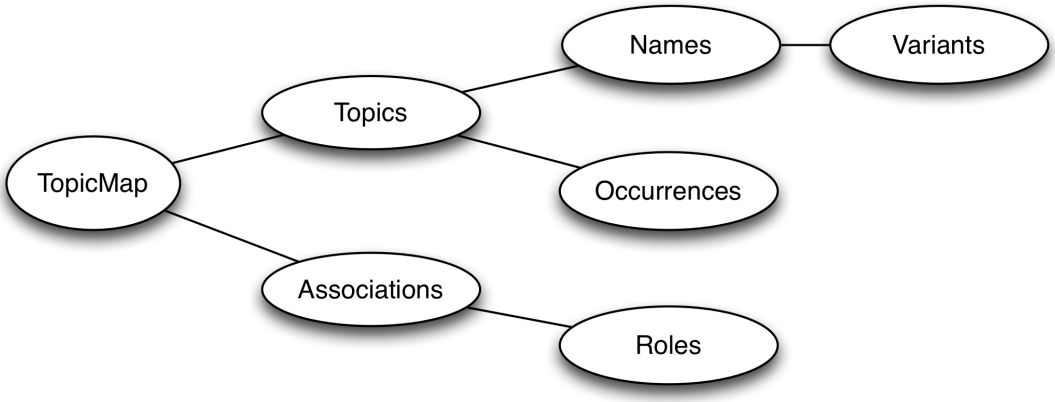


Figure 1. Hierarchical aspect of the TMDM

Names, occurrences, associations, and roles are typed. Types are again topics. Names, variants, occurrences, and associations are referred to as *statements*, while roles are not statements on their own but only within their specific association. The validity of a statement may be constrained by a *scope*. A scope is a set of topics and may be empty. The empty scope is called the *unconstrained scope*. Figure 2 shows the seven entities of the meta-model, i.e. topic map, topic, name, variant, occurrence, association, and role, in their inheritance hierarchy. The inheritance hierarchy shown also introduces another abstract construct: *reifiable*. In the class model of TMDM, all constructs except topics have a restricted set of properties. The concept of reification allows to make additional assertions about a construct by creating a topic representing this construct. All assertions made about this topic are in fact assertions about the reified construct. Obviously a topic is not reifiable because all assertions can be made directly about it.

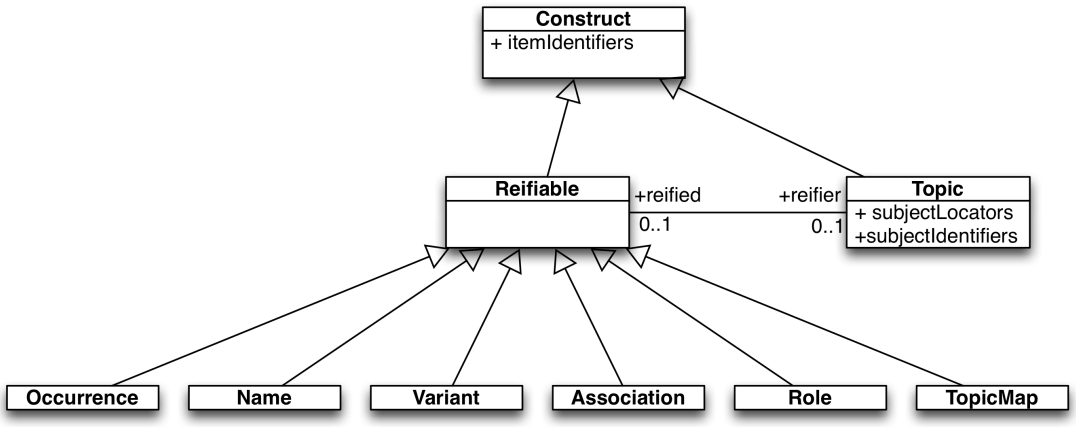


Figure 2. Class diagramm showing inheritance in the TMDM

A topic may have any number of types, instances, supertypes, or subtypes; each of these is again a topic. Additionally, some of the topics are used as type or in the scope of a construct. These topics constitute the *ontology*. A topic map usually contains a model of the real world. For practical reasons, the ontology for a specific application should be further restricted. This task is done using a topic map schema, defined using the Topic Maps Constraint Language [8]. A topic map schema defined in TMCL is again a topic map.

All constructs (including topics) can uniquely be adressed using item identifiers. An item identifier is a *locator*, i.e. a string conforming to the IRI notation defined in RFC 3987. Additionally, topics are addressable using subject identifiers and subject locators. Subject identifiers point to subject indicators in an attempt to unambiguously identify the subject represented by a topic to a human being ([4], section 3.16). Thus subject identifiers are indirect, symbolic representations of a topic. A subject locator refers to the information resources that *is* the subject of a topic. The subject *identifier* `http://www.topicmapslab.de/` represents a working group at the University of Leipzig while the subject *locator* `http://www.topicmapslab.de/` represents the website of this working group. These two different locators refer to two different subjects and therefor point to two different topics. However, these topics would most likely be associated somehow to each other in a topic map.

A topic can be refered to by any number of identifiers (greater zero). Any topics sharing a common item identifier or subject identifier must be merged by the Topic Maps engine. Item identifiers and subject identifiers share the same notion of *indirectly* adressing a subject, consequently they're treated equally and combined (i.e. given two topics, one with an item identifier `x` and one with a subject identifier `x`, these two are merged). Any two topics sharing a common subject locator must also be merged because they describe the same resources. When merging two topics, these two are replaced by a new one having the union of all their properties (with duplicates removed). The concept of merging facilitates having one single construct representing a real world concept, thus eliminating the need for joins etc.

2.1. TMAPI

The Topic Maps Application Programming interface [9] is a community-driven effort to create a minimal but universal API for programmatically accessing and manipulating data conforming to the TMDM. TMAPI consists of a number of interfaces defined in Java. Each Topic Maps construct is represented by an interface. Additionally there are interfaces for several indexes, exceptions, and locators. A Topic Maps engine is a library implementing these interfaces. The defined goal of TMAPI is to do for Topic Maps what SAX and DOM did for XML: providing a single common API which all developers can code to and which means that their applications can be moved from one underlying platform to another with minimum fuss ([9], section "Why Bother?").

The original TMAPI 1.0 was designed to represent an in-memory representation of (an earlier version of) XTM. During its development, TMDM was in an early stage. This led to the decision to fix TMAPI to an intermediate state between XTM 1.0 and XTM 2.0, the latter being the biunique counterpart to TMDM. This discrepancy is being fixed with TMAPI 2.0 which is aligned to TMDM. However, it has insufficient support for the supertype-subtype relationship and allows the implementation some freedom of interpretation regarding the TMDM's requirement to represent the type-instance relationship between topics as queryable associations. This may require additional handling for some queries and hinders exchange of implementations. On the other hand, this relief simplifies the implementation of engines and is therefor an understandable approach.

TMAPI being defined using the Java-language inspired interfaces for other programming languages like PHPTMAPI for PHP. Similar APIs are being used in Ruby, Python, C#, and C++. Some of the implementing engines offer enhancements, the one for Ruby is described below. When refering to TMAPI in the following, any of the before-mentioned interfaces on the abstraction level of Java TMAPI 2.0 and their implementations are included.

2.2. Ontopia Interfaces and Tolog

For a long time Ontopia was the strongest commercial Topic Maps engine vendor and implementation. It is implemented in Java and comes with a huge set of tools and extending packages. The recent open sourcing of Ontopia brought a popularity boost to the engine and therefor its interfaces and query language *tolog*. Ontopia's interfaces are similar to TMAPI's but they additionally feature transactions, concurrency management etc. There is no need to look at Ontopia's interfaces in detail as they are easily mapped to TMAPI, even though they were not inspired by TMAPI in the first place.

Tolog (written in all small letters originally) [10] is a logic-based query language inspired by Prolog's Datalog and SQL. It was Ontopia's (the company's) approach to the a Topic Maps Query Language and since serves as a temporary solution as long as the official TMQL standard is not completed [TOLOG, chapter 6]. Ibidem has been shown that tolog queries can be implemented efficiently on top of Topic Maps engines using a relational database as backend.

While the tolog language's implementations can execute quite sophisticated queries, it lacks on flexibility regarding the query results. It can only return topics and the other contructs but not any other representation like custom XML. This drawback excludes the otherwise useful language for our purpose: returning custom XML from a Topic Maps query.

2.3. Ruby Topic Maps' Approach to TMAPI

Ruby Topic Maps [11] is a Topic Maps engine written in the Ruby programming language. From a language style perspective, Ruby has several advantages compared to Java and therefor enabled several tweaks in the API. Besides language-specific enhancements, other features where implemented to minimize the effort needed to implement an application on top of RTM's API. Finally, the functional features of Ruby offer powerful functions on sets, simplifying the further processing of the query results.

As mentioned before, type and scope of a construct is always a topic resp. a set of topics. Hence the typing or scoping topic(s) must be queried before they can be used in another query. Here, query refers to a method call, not to a complex query with multiple processing steps. In TMAPI this requires to

1. create a locator representing the string of the IRI: `topicMap.createLocator("some:IRI");`
2. query the topic using the locator either as subject identifier, subject locator, or item identifier: `topicMap.getTopicBySubjectIdentifier(theLocator);`
3. use the topic in the query.

Wherever the TMAPI interfaces expect a topic, it is possible to use a string, a locator, or a topic in RTM. This eliminates the first two of the three steps above but creates the necessity to specify how the IRI string is to be interpreted. RTM allows two syntaxes for that, one inspired from the Compact Topic Maps notation [12]: Prefixing item identifiers with "`^`" and subject locators with "`=`", while subject identifiers are used without prefix. The other is inspired from the JSON Topic Maps notation [13], using the prefixes "`ii:`", "`sl:`", and "`si:`" with the obvious meanings. Another feature to point out here is the (switchable) inference of supertypes resp. subtypes when querying.

While RTM's API provides several benefits over TMAPI, its disadvantages are the restriction to use Ruby, only indirectly supported output formats using programming, and the lack of integrated query optimization facilities.

3. Storing a Topic Map and Accessing the Store

3.1. XTM and Other Document-based Formats

The XML Topic Maps Serialization Format, Version 2.0 [5] is defined using RELAX NG [6] and an exact counterpart of the TMDM. It features some shortcuts to allow smaller files like an `instanceOf`-Element instead of a type-instance association. Furthermore it uses different XML elements for the value of an occurrence resp. variant to allow a simpler schema while being precise about the content. Names, occurrences, and variants share the value property in TMDM. The datatype of a

name is imlicitly `xsd:string`. The datatype of occurrences and variants can be anything, including `xsd:anyURI` where the value is persisted in a `resourceRef` element instead of a `resourceData` element. A XTM file does not need to be completely merged. It is allowed that there exist two topic definitions in an XTM file which are semantically identical but not yet brought together. Under certain conditions this complicates queries against an XTM file. This restriction is ignored in the example of querying an XTM2 file using XQuery presented later.

For the same reasons XQuery is not the right tool for this job, neither XSLT solves this problem. The disadvantage lies in the non-matching layers of modeling: XQuery and XSLT are optimized to operate on domain-specific XML documents where the schema of a XTM document is generic from a users perspective. Of course the XTM schema is domain-specific from an XML-perspective, but it just serves as a container format in the case of XTM.

The above-mentioned Compact Topic Maps notation is, besides XTM, the only other ISO-standardized format for Topic Maps. It is optimized for users to write down a topic map directly in a file. This file format is not supported by any query languages directly and, compared to XML, hard to parse. Another common format is the also above-mentioned JTM. Neiter CTM nor JTM is free of the drawbacks explained for XTM.

3.2. Relational Databases and NOSQL Approaches

There is no standarized schema in which a topic map should be represented with a relational database. However, several imlementations, including Ontopia and RTM, are able to use relational databases to persist their data. A topic map represented in one of the schemas of the two mentioned implementations is persisted fully merged. It is theoretically possible to query a topic map using SQL directly from the database. This assumes detailed knowledge of the specific database schema, of additional index tables etc. and is not recommended by the authors.

A trend in these days is called NOSQL. NOSQL is loosely summing up all non-relational data stores. There are no citable academic publications for NOSQL yet, so we have to refer to the homonymous entry in the English Wikipedia. NOSQL focusses on horizontal scalability while not necessarily requiring the ACID guarantees provided by relational databases. To short cut this divagation: All query languages created for other data models suffer the mismatch between their particular way of modeling data and the Topic Maps way. The consequent and obvious solution is to create a query language which fits the TMDM and allows advanced processing to fulfill the user's expectations.

4. Querying a Topic Map

4.1. Querying an XTM File Using XQuery

It was already argued not to use XQuery [14] to query an XTM file. Nevertheless we will show an XQuery example to directly compare it with the TMQL approach. The Italian Opera topic map is a commonly used example in the Topic Map community. We're assuming a completely merged topic map so we don't have to care about duplicates. Take the following example for a query of all cities mentioned in this topic map:

Example 1. XQuery cities from the Italian Opera topic map by id

```
declare default element namespace "http://www.topicmaps.org/xtm/";
.//topic[./instanceOf/topicRef/@href="#city"]
```

In this query, a city is refered to by its fragment identifier `#city` which points to a topic element within the same file. This topic element contains additional information like a subject identifier (`http://psi.ontopedia.net/City`), names in different languages (each language designated by a scoping topic in the name elements) and so on. To combine and integrate information from several sources - one of the core competences of Topic Maps - a published subject identifier (PSI) should be used.

Example 2. XQuery cities from the Italian Opera topic map by PSI

```
declare default element namespace "http://www.topicmaps.org/xtm/";
.//topic[./instanceOf/topicRef/@href=concat(
    "#",
    //topic[./subjectIdentifier/@href=
        "http://psi.ontopedia.net/City"]/@id
)]
```

If we now (simplifying but wrongly) assume that for each result `/count(./name)` equals 1, we can append `/name/value/string(.)` to get all city names. Still, we did not query for city names constrained using a given scope. We did not take into account that `instanceOf` can be equally modeled as association and so on. We stop here and move on to TMQL which fits the given purpose much more natural.

4.2. TMQL Path and SELECT Queries

It was shown that within an XML environment, using Topic Maps in form of generic XTM documents may be cumbersome. It is highly desirable to create documents which adhere to a given schema or may be easily transformed to one using XSLT. One approach to this was the creation of TM/XML which provides a domain-specific

representation of a topic map. The schema of TM/XML is dependent of the topic map's schema, not the desired output schema. We present a draft of the Topic Maps Query Language [15] standard which features a query style which is heavily inspired by XQuery FLWOR and allows outputting domain-specific XML documents queried from a Topic Map. This draft uses a path style as its fundamental processing model. The path style is also exposed to the user and can be used, especially for queries for a result with a uniform structure. A third query style is the SELECT style which was inspired by SQL. Within a TMQL processor, both FLWOR style and SELECT style can easily be converted to Path style. The expressiveness of these three styles is identical. The only differences are that path expressions cannot have explicit variables and that FLWOR is the only style able to output complex content [BA06].

The expressions in a path style navigate along predefined axes. The separator used is >>. The resulting values can be filtered using boolean conditions or used as new starting points for further navigation ([15], section 6.6.1). The following TMQL path query extracts all cities from the Italian Opera topic map.

Example 3. TMQL path querying cities from the Italian Opera topic map

```
http://psi.ontopedia.net/City >> instances
```

The alignment of the query languate to the data model obviously simplifies the structure of the query significantly. Every navigation item is a set of Topic Maps constructs, locators or simple values (e.g. string). All merging is provided by the engine. All inferencing of subtypes etc is done automatically. If there was a subtype of City, e.g. Capital, the list of capitals would be included in the list of cities. There is a shortcut for the instances axis shown in the following snippet:

Example 4. TMQL path querying cities from the Italian Opera topic map using shortcut

```
// http://psi.ontopedia.net/City
```

Except for another single character to signify the traversal of the instances axis this cannot be any shorter. There are shortcuts for many axes but their terminal symbols are still disputed. We look forward to a normative standardization of the query language as an ISO standard.

4.3. The TMQL FLWOR Query Style

The FLWOR style is a reason to look at a slightly more sophisticated query and using XML output. The following code snipped shows a TMQL FLWOR query to get all operas and their composers. As we use the namespace `http://psi.ontopedia.net/` multiple times it is useful to define a prefix `o` for this. The query result is to be wrapped within a root element, named `root` in the example. In the following a naïve

approach to the processing model is explained using the example: The set of all instances of opera is iterated and assigned to the variable `$OPERA` variable. The set in the variable is then iterated over to find all instances matching the pattern in the WHERE-clause. In each iteration, the variable `$COMPOSER` is assigned to the matching counterpart in the association. The boolean expression in the WHERE-clause matches an association typed by `o:composed_by` which contains two roles, one of type `o:Composer` and one of type `o:Work`. For each matching association an XML template is filled with the elements given in the templates and the subqueries in curly braces evaluated.

Example 5. TMQL FLWOR querying operas and their composers

```
%prefix o http://psi.ontopedia.net/
RETURN
<root>
{
  FOR $OPERA IN // o:Opera
  WHERE o:composed_by( o:Composer : $COMPOSER , o:Work : $OPERA )
  RETURN
  <composed_by>
    <opera>
      { $OPERA / tm:name }
    </opera>
    <composer>
      { $COMPOSER / tm:name }
    </composer>
  </composed_by>
}
</root>
```

5. Application: A subj3ct.com feed

5.1. About subj3ct.com

Subj3ct describes itself as infrastructure technology for Web 3.0 applications. The criterion for being "Web 3.0" is the focus on subjects and semantics rather than documents and links. The goal of Subj3ct is to provide technology and services to enable applications to define and exchange subject definitions [17]. Its goal is not to collect any data but only the information relevant to uniquely identify subjects. Subj3ct provides a web based search interface which provides links to resources about the subjects as well as a REST API for identifier and full text search.

The Subj3ct data is updated using ATOM and SKOS feeds. ATOM is an XML-based format and SKOS is transferred in RDF/XML format which is, as the name suggests, also XML-based. To update subj3ct.com from a Topic Maps-based applic-

ation, the most obvious way is to use a TMQL FLWOR query which is suggested in this paper. The ATOM variant will be discussed here.

Because everybody can publish everything about any subject, some way of weighting relevant sources is useful. Subj3ct assigns trust scores to identifiers. "The Trust Score provides a rough measure of how likely it is that the creator of an identifier agrees with any statement being made about the identifier" ([17], chapter 4.1). This does not mean a trust in the creator of an identifier or the individual who creates statements about that identifier. Neither does it qualify the content which may be returned when requesting any URLs contained in those statments.

5.2. Updating subj3ct.com via ATOM

The following figure shows an excerpt of the ATOM feed provided by the topicmapslab.de community portal. Besides the standard ATOM elements used in the header part, the entries are the interesting part. Additional to the classic atom entries, a subj3ct.com ATOM feed includes links with special `rel` attributes. The attribute values `SubjectIdentifier` and `SubjectEquivalence` are used to designate the subject identifiers. We assume that the different names were chosen to distinguish between those primarily used in the source providing the feed and external ones.

In the ideal case, the subject identifier points to a subject indicator (which is a resource describing the real world concept, as said before). Practically often the representation is at another address which is thought to be provided using `link` elements with a `rel="SubjectRepresentation"`. As the topicmapslab.de portal is a subject-centric portal, all these links match in the example.

Example 6. Excerpt of the ATOM-based topicmapslab.de people update feed.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Subject Identifier for People at Topic Maps Lab</title>
  <id>http://www.topicmapslab.de/people?locale=en</id>
  <updated>2010-01-22T11:45:14Z</updated>
  <author>
    <name>Topic Maps Lab</name>
    <uri>http://www.topicmapslab.de/</uri>
  </author>
  <entry>
    <title>Benjamin Bock</title>
    <id>http://www.topicmapslab.de/people/Benjamin_Bock</id>
    <updated>2010-01-18T18:12:23Z</updated>
    <summary>Is involved in &quot;Ruby Topic Maps&quot;
and &quot;Topic Maps Lab Community Portal&quot;</summary>
    <link href="http://www.topicmapslab.de/people/Benjamin_Bock"
      rel="SubjectIdentifier"/>
```

```
    <link href="http://www.topicmapslab.de/people/Benjamin_Bock"
      rel="SubjectRepresentation"/>
    <!-- snip -->
    <link href="http://bock.be/njamin" rel="SubjectEquivalence"/>
    <link href="http://psi.ontopedia.net/Benjamin_Bock"
      rel="SubjectEquivalence"/>
  </entry>
  <entry>
    <title>Sven Krosse</title>
    <id>http://www.topicmapslab.de/people/Sven_Krosse</id>
    <!-- snip -->
  </entry>
  <entry>
    <title>Lutz Maicher</title>
    <id>http://www.topicmapslab.de/people/Lutz_Maicher</id>
    <!-- snip -->
  </entry>
</feed>
```

5.3. Using TMQL FLWOR to Create a Feed for subj3ct.com

The following code snipped shows a single TMQL FLWOR query to create the complete ATOM feed for subj3ct.com in one step. In this example the header is returned in one step. For simplicity the current time is chosen for the `updated` element. Of course, this could also be determined by the modification time of the topic which was modified last. In the first `FOR` part of the query all people are queried. The `WHERE` clause constraints the list to only those person-topics which have been marked as published using the corresponding occurrence. For each person an entry is returned which is then filled using subqueries for all properties requested. Please notice the use of built-in functions for matching regular expressions.

Example 7. A TMQL FLWOR query to create a subj3ct.com ATOM feed

```
%prefix tml http://www.topicmapslab.de/
%prefix t http://www.topicmapslab.de/types/
RETURN
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Subject Identifier for People at Topic Maps Lab</title>
  <id>http://www.topicmapslab.de/people?locale=en</id>
  <updated>{ fn:current-dateTime }</updated>
  <author>
    <name>Topic Maps Lab</name>
    <uri>http://www.topicmapslab.de/</uri>
  </author>
{
```

```
  FOR $person IN // tml:people
  WHERE $person / t:published
  RETURN
  <entry>
    <title>{ $person / t:firstname } { $ person / t:lastname}</title>
    <id>{ $person >> indicators [0] }</id>
    <updated>{ $person / t:updated_at }</updated>
    <summary>{ $person / t:summary }</summary>
    {
      FOR $si in $person >> indicators
      WHERE fn:regexp( $si >> atomify,
          "http://www.topicmapslab.de/people/.*")
      RETURN
      <link href="{$si}" rel="SubjectIdentifier"/>
      <link href="{$si}" rel="SubjectRepresentation"/>
    }
    {
      FOR $si in $person >> indicators
      WHERE not fn:regexp( $si >> atomify,
          "http://www.topicmapslab.de/people/.*")
      RETURN
      <link href="{$si}" rel="SubjectEquivalence"/>
    }
  </entry>
}
</feed>
```

Compared to a previous approach (which collected the information needed using RTMAPI method calls and then outputting a string), the time needed for implementation was reduced; the amount of code is smaller while maintaining a good readability.

6. Conclusion and Outlook

The creation of the ATOM feed query is mostly a work of taking an existing feed and replacing all repeating parts with `FOR-WHERE` expressions and selecting the properties also easy to create. The syntax of TMQL FLWOR has many parallels to XQuery and thus can easily be learned. A big advantage of TMQL is the alignment to the data model and the therefor concise and fitting syntax. The ability to output XML eases the integration with other applications significantly.

We implemented TMQL in Java using the before mentioned TMAPI interfaces. The implementation is called TMQL4J and we use it for several academic projects. Our implementation is published as part of the Ontopia open source project [18].

Additionally, Ruby Topic Maps will be enhanced by an additional package rtm-tmql allowing ease of use within the Ruby library.

The query language is still a work in progress and may change before its final standardization in ISO, therefor the implementation will also change. The naïve implementation of the processing model showed significant worse performance compared to the highly optimized tolog engine used in Ontopia. Some obvious optimizations already show massive performance increases. An reasonable next step would be not to implement TMQL on top of TMAPI but to directly access a backend optimized for the engine.

The current draft of the upcoming TMQL standard describes only read access. Several parties are working on drafts supporting modifing access. With a stable query language for reading and modifying Topic Maps, a solid foundation for wide adoption will be laid.

Bibliography

[1] ISO/IEC IS 13250:3004: Information Technology - SGML applications - Topic Maps (Second Edition). International Organisation for Standardization, Geneva, Switzerland, 2003

[2] Members of the TopicMaps.Org Authoring Group: XML Topic Maps (XTM) 1.0. http://topicmaps.org/

[3] ISO/IEC JTC 1/SC34 N0266: Topic map foundational model requirements. International Organisation for Standardization, Geneva, Switzerland, 30 October 2001. http://www1.y12.doe.gov/capabilities/sgml/sc34/document/0266.htm

[4] ISO/IEC IS 13250-2:2006: Information Technology — Document Description and Processing Languages: Topic Maps — Data Model. International Organisation for Standardization, Geneva, Switzerland, 18 June 2006. http://www.isotopicmaps.org/sam/sam-model/2006-06-18/

[5] ISO/IEC IS 13250-2:2006: Information Technology — Document Description and Processing Languages: Topic Maps — XML Syntax. International Organisation for Standardization, Geneva, Switzerland, 2006. http://www.isotopicmaps.org/sam/sam-xtm/2006-06-19/

[6] Clark, James – Cowan, John – MURATA, Makoto: RELAX NG Compact Syntax Tutorial. Working Draft, 26 March 2003. OASIS. http://relaxng.org/compact-tutorial-20030326.html

[7] W3C: XML-Infoset, XML Information Set (Second Edition), W3C Recommendation, 4 February 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204

[8] ISO/IEC FCD 19756: Information Technology — Document Description and Processing Languages: Topic Maps Constraint Language. International Organisation for Standardization, Geneva, Switzerland, 15 July 2009. http://www.isotopicmaps.org/tmcl/2009-07-15/

[9] Ahmed, Kal; Garshol, Lars M.; Grønmo, Geir O.; Heuer, Lars; Lischke, Stefan; Moore, Graham: TMAPI 1.0. http://tmapi.org/

[10] Garshol, Lars M.: tolog A topic map query language. Ontopia, 24 August 2007. http://www.ontopia.net/topicmaps/materials/tolog.html

[11] Bleier, Arnim; Bock, Benjamin; Schulze, Uta; Maicher, Lutz: JRuby Topic Maps. in: Linked Topic Maps, LIV Series volume XIX, 12 November 2009.

[12] ISO/IEC WD 13250-6: Information Technology — Document Description and Processing Languages: Topic Maps — Compact Syntax. International Organisation for Standardization, Geneva, Switzerland, 15 May 2008, http://isotopicmaps.org/ctm/ctm.html

[13] Cerny, Robert: JSON Topic Maps 1.0. 23 June 2009. http://www.cerny-online.com/jtm/1.0/

[14] W3C: XQuery 1.0: An XML Query Language 23 January 2007. http://www.w3.org/TR/xquery/

[15] ISO/IEC WD 18048: Information Technology – Document Description and Processing Languages – Topic Maps – Query Language. International Organization for Standardization, Geneva, Switzerland, 2007-07-13 http://www.isotopicmaps.org/tmql/

[16] Barta, Robert: Towards a Formal TMQL Semantics. in: Leveraging the Semantics of Topic Maps, Springer 4 September 2007.

[17] Moore, Graham; Ahmed, Khalil: Subj3ct - A Subject Identity Resolution Service. in: Linked Topic Maps, LIV Series volume XIX, 12 November 2009.

[18] Garshol, Lars M.; Grønmo, Geir O.: Ontopia: Tools for building, maintaining, and deploying Topic Maps-based applications. Ontopia 5.0.0, 8 July 2009, http://code.google.com/p/ontopia/

Extending XQuery with Collections, Indexes, and Integrity Constraints

Cezar Andrei
Oracle Corporation
<cezar.andrei@oracle.com>

Matthias Brantner
28msec Inc.
<matthias.brantner@28msec.com>

Daniela Florescu
Oracle Corporation
<dana.florescu@oracle.com>

David Graf
28msec Inc.
<david.graf@28msec.com>

Donald Kossmann
28msec Inc.
<donald.kossmann@28msec.com>

Markos Zaharioudakis
Oracle Corporation
<markos.zaharioudakis@oracle.com>

Abstract

The standard XQuery language lacks the ability to define and manipulate persistent artifacts like collections, indexes, and integrity constraints. This paper introduces a first attempt to standardize the syntax and semantics of such extensions, and it studies the implications on the static context, dynamic context and processing model of XQuery while dealing with persistent data. The paper presents example modules that show how collections, indexes, and integrity constraints are declared, created, maintained, or accessed.

Keywords: XML, XQuery, DDL, indexes, collections, integrity constraints

1. Introduction

XQuery has been designed by the World Wide Web Consortium as a general purpose XML information processing language,useful in a variety of architectures and environments. For example, XQuery can be used to process *XML data on the edge* of existing software architectures, where the information is temporary, and is being searched, transformed ,or modified, just before being passed along for further processing to other programming languages (e.g. SQL, JAVA, Phyton, Ruby, JavaScript). Another increasingly popular usage of XQuery is in *XML databases* or *XML end-to-end architectures*. In such architectures, XML *is* the primary form in which the information is stored and being processed, the information is *persistent* across successive invocations of programs, and XQuery is the primary language for accessing this information for search, filter, transform, update, and for writing more complex application workflows.

Unfortunately, XQuery, as it is currently standardized by the W3C, is incomplete and cannot be used as such (without proprietary language extensions, or rich APIs from other programming languages) in the second type of architectures: persistent databases or XML end-to-end architectures. Unlike its cousin query language, SQL, XQuery lacks the capability to model, describe, and reason about the persistent state of the "database". XQuery 1.1 does indeed have the capability to access collections of nodes at runtime, which could be envisioned as modeling the persistent state of the XML database, yet the language is underspecified in this area. Such collections have no detailed semantics (about copy, order, or multiplicity for example), the language lacks the ability to declare statically such collections, it lacks the static and/or dynamic information that is required for proper compilation and/or execution (e.g. type, update patterns), and it lacks operations to create and modify such collections. Moreover, the language lacks the ability to declare and manage access structures (e.g indexes) and integrity constraints.

All such concepts are required for a complete XML/XQuery database story. Unless such concepts are included in the standard language itself, each XQuery implementation will have proprietary extensions to overcome such limitations, or such functionalities will be supplied through non XQuery rich APIs. In both cases, the portability of XQuery applications will be limited or the simplicity and elegance of XML end-to- end architectures will be hurt.

In this document, we describe an extension of XQuery 1.1 [2] called XQuery Data Definition Facility (or XQDDF) to deal with such persistent artifacts: collection, indexes, and integrity constraints. The document describes the lifetime and evolution of such artifacts: how are they declared, how do they come into existence, how are they used in the compilation and execution of XQuery programs, and how are they shared by multiple XQuery programs.

Specifically, XQDDF extends

1. the static context with the definitions of collections, indexes,, and integrity constraints
2. the dynamic context with the runtime aspects of collections, indexes, and integrity constraints
3. the syntax (and semantics) of the prolog of library modules with the declaration of collections, indexes, and integrity constraints
4. the semantics of the import module statement
5. the XQuery Update Facility [3]
   - with new update primitives for creating, deleting, and modifying collections and indexes
   - by adding new expressions for modifying collections

   the Function and Operators [4] by adding functions for creating, deleting, and modifying collections and indexes, and activating integrity constraints. All such functions are in a new namespace whose prefix in this document is *XQDDF*.

All such extensions are described along the lines of tiny XQuery snippets which are iteratively build up on each other. They illustrate how to declare, manage, and use each of the components of XQDDF. Please note that the purpose of this document is not to be a complete specification. Instead, such a specification is provided online at [6].

The remainder of this paper is describes as follows: First, we give an overview over the different components of XQDDF and their impact on the XQuery processing model, i.e. their impact on the static- and dynamic context. After that, we illustrate collections, indexes, and integrity constraints, each in a separate section. At the end, we conclude and give an outlook in Sec.

2. Overview

According to the XQuery 1.1 specification [2], collections are sequences of nodes that are potentially available using the fn:collection function. Unfortunately, as we mentioned before, XQuery 1.1 collections have no static information, and there underspecified in many aspects. This specification introduces a new kind of collection, together with a complete semantics for declaring, creating, modifying and accessing them. In the remaining of the document by collections we will refer to this new kind of collections introduced in this specification, and not the XQuery 1.1 notion.

XQDDF collections are disjoint sequences of parent-less nodes identified by QNames (not URIs). The sequence of nodes can be retrieved using the xqddf:collection(QName) function. They are created by invoking specific functions. Their content can be modified either by specific expressions (e.g. insert, delete) or by invoking the equivalent side-effecting functions.

Indexes are access structure whose contents are defined by a "domain" expression defining the set of nodes to be indexed and a number of "key" expressions on which the index is being built. Indexes can be either used implicitly by the query processor (if such an evaluation is equivalent to the original program) or used explicitly in expressions using xqddf:probe functions. While defining an index there are a lot of questions to be considered and answered: is it a multi-key index or a single-key index? is the index required to have homogeneous set of keys? which kind of equality or comparisons is the index able to solve (value comparisons, general comparisons) ? is the index able to solve only equality point search or it is able to solve range queries? how is the index maintained (automatically or explicitly)? is the index maintained up-to-date with respect to the original data in an atomic fashion or can be updated asynchronously? how is the indexed used in programs (automatically used by the compiler or explicitly used by the user)? is the index stable, i.e. does it return the nodes in the (original) order in the collection of not?

This document attempts to give users control over the answer to such questions while they define and manipulate indexes. If the XML data is stored (e.g. relational stores, LDAP stores), the propagation of updates on collections or indexes to such an underlying persistent store is beyond the scope of this specification.

Integrity constraints (ICs) specify rules that ensure the accuracy and consistency of data that is available in collections. One can imagine lots of different kinds of integrity constraints (e.g. uniqueness of keys, foreign keys, or global collection integrity constraints). This document attempts to gives syntax and semantics for a comprehensive and useful set of such constraints, and a way to activate and deactivate them at runtime.

The collections, indexes, and integrity constraints have a dual representation: (a) the static information about such persistent artifacts – that is populated though compilation of their declarations (i.e. during the static analysis phase) – and (b) the dynamic context (runtime) aspect – that is independently populated via the execution of specific creation functions (i.e. during the dynamic analysis phase). The set of statically known collections, indexes, and integrity constraints do not have to be perfectly consistent with the set of dynamically available such artifacts: each dynamic artifacts has to be described in the static context, but the reverse is not required. A runtime access to an artifact that is not found in the dynamic context will result in an execution error.

In general, collections, indices, and ICs are expected to be persistent artifacts, that is, they may survive across and be shared by multiple XQuery programs. This is accomplished by sharing the same XQuery static and dynamic context across programs.

Figure 1 shows an example. In this example, a first module (Module A) declares the collections and indexes and, hence, populates a static context with the information. A second module (Module B) executes in this static context and creates the dynamic structures for collections and indexes, hence modifying/populating the

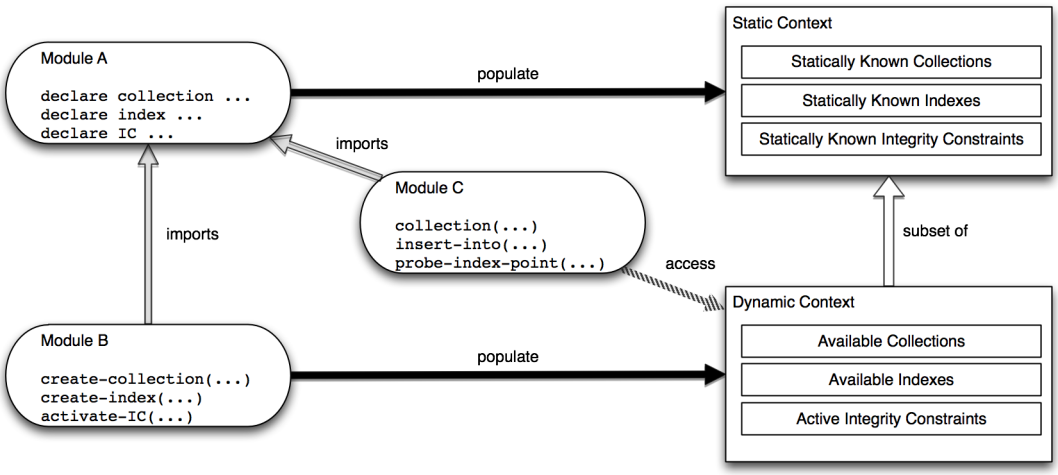


Figure 1. XQDDF Overview

dynamic context. A third module (Module C) executes in the static and dynamic context populated by the first two, and hence can use the persistent artifacts (created by the previous two: collections and indexes).

3. Collections

Let us assume an application that models a news organization. The application models its data as XML documents grouped into collections of logically related entities. In the example introduced in this section, we show how three such collections may be created and used; the first collection contains employee data, the second contains news articles, and the third contains information about the months of the year (e.g., the name, number of days, and fixed holidays for each month).

3.1. Declaration

Before a collection can be created, it must be declared. A collection declaration is an extension to the prolog of XQuery's library modules and describes the collection by providing (1) a unique name (as QName), (2) specifying certain properties for the collection itself (e.g. modifier properties), and specifying properties for the documents contained in the collection (e.g. types).

In the example query presented below, the declaration of the collections are placed inside the “news-data” library module. These declarations assign the names news-data:employees, news-data:articles, and news-data:months to the three collections, respectively. Documents in both, the employees and the months collections are assumed to have a well-known structure. This structure is reflected in an XML schema (“news-schema”) which declares two global elements for employees and months, respectively. Accordingly, the collection declarations for employees and

months specify that their (root) nodes are elements whose name and type matches the name and type of the corresponding global element declarations in the "news-schema". In contrast, articles may come from various sources and, as a result, article documents do not have any particular schema. Therefore, the declaration for the articles collection specifies node() as its type. Both employee and article documents may be updated during their lifetime. In contrast, the months-related information is fixed (i.e. it can not change), so the nodes of the months collection are declared as "read-only". Furthermore, the collection itself is declared "const", meaning that no months may be added to or deleted from this collection after it is created and initialized. Finally, we want the order of the month documents within their containing collection to be the same as the actual order of the months within the year. To achieve this, we have to declare the collection as "ordered", so that when we later insert the month documents in the collection, the system will store and return them in the same order as their insertion order. In contrast, the position of employees or articles inside their respective collections does not have any special meaning for the application. Hence, the corresponding declarations do not specify any ordering property. This allows the system to store and access the contents of these collections in what it considers as the optimal order.

```
module namespace news-data = "http://www.news.org/data";

import schema namespace news = "http://www.news.org/schemas";

declare collection news-data:employees
  as schema-element(news:employee)*;

declare collection news-data:articles as node()*;

declare const ordered collection news-data:months
  as schema-element(news:month)*
  with read-only nodes;

declare variable $news-data:employees := xs:QName("news-data:employees");
declare variable $news-data:articles := xs:QName("news-data:articles");
declare variable $news-data:months := xs:QName("news-data:months");
```

Besides declaring collections, the XQuery snippet also declares three variables whose values are the QNames of the collections. Those variables might be used by functions in (other) modules in order to refer to those collections.

3.2. Life Cycle Management

Having been declared, the collections can now be created, i.e. added to the dynamic context. Collection creation is illustrated by the XQuery script shown below. First, the collection descriptions must be made visible to the script. This is done by im-

porting the “news-data” module that contains the declarations of the collections. Then, the collections are created by calling the xqddf:create-collection function. There exist two variants of this function: the first takes a QName as input (i.e. the $news-data:employees variable declared in the "news-data" module) and the second takes both a QName and a node-producing expression. In the first variant, a map is created and registered in the dynamic context that maps collection names to empty sequences. In the second variant, the given expression is evaluated first, and (deep) copies are made of the nodes in the resulting sequence. This way, a sequence of distinct documents is produced which is called the “insertion sequence”. Then, as in the first version of the function, this sequence is added to the dynamic context. In the script below, the latter variant is used to create and initialize the months collection. In fact, months must be initialized during creation because it is a constant collection, so no documents can be added to it later. The months are inserted in the collection in the order from January to December, and since the collection was declared as ordered, this order is preserved in the dynamic context. Collection can be destroyed (i.e. removed from the dynamic context) using the xqddf:delete-collection function.

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import schema namespace news = "http://www.news.org/schemas";
import module namespace news-data = "http://www.news.org/data";

xqddf:create-collection($news-data:employees);

xqddf:create-collection($news-data:articles);

xqddf:create-collection($news-data:months, (
    validate { <month name="Jan">...</month> },
    ...,
    validate { <month name="Dec">...</month> })
);
```

Note that in the "xqddf" module, most functions are declared as updating, i.e. they return a pending update list (PUL; see *XQUERYUPDATE10*). For example, the first variant of the create-collection functions returns one update primitive to create the collection in the dynamic context. Such changes are only made visible if the corresponding PUL is applied. This can be done (1) either at the end of the query or (2) using an apply expression as provided by the *XQUERYSCRIPTING10*. In the example above, the latter approach is used (see the semi-colon that is used after the function call expression).

3.3. Accessing and Deleting Collection Data

In this section, we provide an example that shows how data can be added to collections, deleted from collections, or retrieved from collections. In order to get access to the collections, the corresponding modules and schemas are imported.

Next, the employees collection is populated using the xqddf:insert-nodes function. The first argument to this function is the QName of a collection, and the second is a node-producing expression (called the source expression). The QName is used to lookup the collection declaration and the collection itself. Then, the nodes produced by the source expression (source nodes) are copied and the copies are added to the document container, making sure that the actual type of each node matches the static type given in the collection declaration. Copying the source nodes (and their sub-trees) guarantees that the nodes in the insertion sequence (1) are indeed parent-less nodes that do not belong to any other collection already and (2) are distinct from each other. Notice that the need to validate the root nodes against the type specified in the collection declaration is the reason why the "news-schema" must be imported, even though no type defined by the schema is referenced explicitly in the query. In this example, the employees collection is populated by a single call to the xqddf:insert-nodes function, whose source expression is a concatenation of explicitly constructed documents.

After populating the collection, the script runs a query expression that uses the xqddf:collection function to access the employee and article collections' root nodes. The expression returns, for each journalist, the articles authored by that journalist ordered by their date.

Finally, the xqddf:remove-nodes function is used to remove the articles that were published before 2000 from the article collection. Like xqddf:insert-nodes, xqddf:remove-nodes takes as input the QName of a collection and a node-producing source expression. The source nodes must be parent-less nodes that belong to the collection. The function looks up the collection declaration and the collection entry from the dynamic context, and removes the source nodes from the collection. Notice that the whole script is organised as a concatenation of three block expressions (see *XQUERYSCRIPTING10*). Only the second block produces an actual result. The other two are purely updating blocks (their result is the empty sequence and a pending update list). Writing the query as a concatenation of blocks (instead of a single sequential expression), allows the result of the script to be the concatenation of the results of each block (instead of the result of just the last expression in a sequential expression).

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import schema namespace news = "http://www.news.org/schemas";
import module namespace news-data = "http://www.news.org/data";

block {
```

```
  xqddf:insert-nodes($news-data:employees, (
    validate { <employee id="100">....</employee> },
    ...,
    validate { <employee id="500">...</employee>) }
  );
},
block {
  for $emp in xqddf:collection($news-data:employees)[./position/@kind eq ▶
"journalist"]
  let $articles := for $art in ▶
xqddf:collection($news-data:articles)[.//author//name eq $emp/news:name]
                  order by $art//date
                  return $art
  return <result>{$emp}<articles>{$articles//title}</articles></result>
},
block {
  xqddf:delete-nodes($news-data:articles,
    xqddf:collection($news-data:articles)[.//date lt xs:date("01/01/2000")]
  );
}
```

4. Indexes

Let us assume the same news application that we used in the previous section. In this section, we will illustrate how to create and use indexes on the collections of the news organization. Specifically, we extend this application with two indexes: First, let us assume that each employee has a city where she is currently stationed at. We want to create an index that maps city names to the employees that are stationed in those cities. The first index will contain one entry for each city where at least one employee is stationed in. Moreover, let us assume that we want to search for journalists based on the number of articles they have written. For this, we will create a second index that maps article counts to the employees who are journalists and have produced that number of articles.

4.1. Declaration

Similar to collections, indexes must be declared before they can be created. An index declaration describes the index by providing its domain expression, its key expressions, and certain index properties; it also specifies a name for referencing the index in subsequent operations. Like collections, indexes are declared inside the prolog of library modules.

The example below, contains the declaration of two indexes. The first index declaration assigns the name news-data:CityEmp to the index. It uses the "on nodes" and "by" keywords to specify the domain and key expressions, respectively. The

"as" keyword specifies a target atomic data type which the results of the key expression must match with (after atomization). The index is declared as a "value equality" index. This means that it can be used to find the employees in a particular city, but not in a "range" of cities. In other words, the index is not aware of any ordering among city names. Finally, the maintenance property of the index is set to "automatically maintained". Briefly, an automatically maintained index is one whose maintenance is the responsibility of the XQuery processor rather than the XQuery programmers.

The second index declaration assigns the name news-data:ArtCountEmp to the index. Its domain expression selects all employees who are journalists. Its key expression computes the number of articles written by the "current" journalist. This index is declared as a "value range" index, which means that it can be used to find journalists whose article count is within a given range. Finally, the index is declared as "manually maintained", which means that programmers must explicitly request the index to be synchronized with the underlying data.

```
module namespace news-data = "http://www.news.org/data";

import schema namespace news = "http://www.news.org/schemas";
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

(: collection declarations go here :)

declare automatically maintained value equality index news-data:CityEmp
  on nodes xqddf:collection(xs:QName("news-data:employees"))
  by .//news:station/news:city as xs:string;

declare manually maintained value range index news-data:ArtCountEmp
  on nodes ▶
xqddf:collection(xs:QName("news-data:employees"))[./news:position/@kind eq ▶
"journalist"]
  by count(for $art in xqddf:collection(xs:QName("news-data:articles"))
           where $art/empid = ./id
           return $art) as xs:integer;

(: variables for collection QNames go here :)
declare variable $news-data:CityEmp := xs:QName("news-data:CityEmp");
declare variable $news-data:ArtCountEmp := xs:QName("news-data:ArtCountEmp");
```

As for collections, the module declares two variables that may be used by subsequent expressions to access the indexes, e.g. as first parameter to the xqddf:create-index function described below.

4.2. Life Cycle Management

Having declared the indexes in the "news-data" library module, they can now be created. This is done using the admin-script shown below. This script must first import the “news-data” module. As far as indexes are concerned, the effect of this import is to create two entries in the static context of the main module, mapping the index names to the index definitions (domain expression, key specification, and properties). Then, the query creates the indexes by invoking the ddf:create-index function, passing the name of the index as input.

Let us consider the creation of the CityEmp index (the process is the same for the ArtCountEmp index). Index creation starts with retrieving the index definition from the static context, using the index name. Then, an index container is created, whose entries will be pairs associating a city name with a set of employees. Next, this container is populated using the following process: The domain expression is evaluated, and for each employee node E in the domain sequence, the name of the city C where the employee is currently stationed in is retrieved by evaluating the key expression, atomizing its result, and checking that the atomic value matches the specified target type. Finally, the pair [E, C] is inserted in the index: if an entry for C exists already, E is inserted in the set associated with C; otherwise, an new entry is created mapping C to the set { E }. The last step in index creation involves registering the index inside an indexes table in the dynamic context that maps index names to index containers. The index container will remain registered until it is destroyed by a call to the ddf:delete-index function.

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import module namespace news-data = "http://www.news.org/data";

xqddf:create-index($news-data:CityEmp);

xqddf:create-index($news-data:ArtCountEmp);
```

4.3. Probing and Maintaining Indexes

The next step in this example is to show how the index can be used to optimize query performance, which of course, is the primary motivation for supporting indexes in any data-processing system. XQDDF provides two functions for index probing: probe-index-point and probe-index-range. The latter function is only available for indexes declared as “value range”.

4.3.1. Probing Indexes

The first query below illustrates the use of the xqddf:probe-index-point function. This query returns the names of all employees stationed in Paris. As shown, the xqddf:probe-index-point function takes the index name and the keyword “Paris”

as inputs. It uses the index name to find the index container via the indexes tables, looks-up the entry for "Paris" inside this container, and returns all the associated employee nodes.

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import module namespace news-data = "http://www.news.org/data";

xqddf:probe-index-point($news-data:CityEmp, "Paris")
```

The second query illustrates index probing via the xqddf:probe-index-range function. It returns all journalists who have written more than 100 articles. As shown, the first parameter of the xqddf:probe-index-range function is the index name, followed by six parameters per key expression. The six parameters specify a range of values for the key values: the first two are the lower and upper values of the range, the next two are booleans that specify whether the range does indeed have a lower and/or upper bound, and the last two are also booleans that specify whether the range is open or closed from below or above (i.e., whether the lower/upper bound are included in the range or not).

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import module namespace news-data = "http://www.news.org/data";

xqddf:probe-index-range($news-data:ArtCountEmp, 100, (), true(), false(), ▶
true(), false())
```

4.3.2. Maintaining Index Data

Now, let us consider what happens when the data on which an index is built gets updated. In general, index maintenance is the operation where the index contents are updated so that they reflect the index definition with respect to the current snapshot of the data. There exist two maintenance modes: manual and automatic. If an index is declared as "manually maintained", index maintenance is done only when the updating function ddf:refresh-index is invoked inside a query. Essentially, in manual mode maintenance invoking this function is the responsibility of the programmer, and the index may become stale between two consecutive calls to the ddf:refresh-index function. In contrast, if an index is declared as "automatically maintained", the query processor guarantees that the index stays up-to-date at any given time.

Consider the following updating query as an example:

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";
import module namespace news-data = "http://www.news.org/data";

replace node value
   xqddf:collection($news-data:employees)/employee[@id eq "007"]//station/city
    with "Beijing"
```

In this example, the CityEmp index was declared as automatic. The index-maintenance query transfers the employee with id "007" from his current city, say Paris, to Beijing. Since index CityEmp is automatic, after the update is applied, the query processor will initiate a maintenance operation on the index, whereby the employee node will be removed from the node set associated with Paris and inserted into the node set associated with Beijing (if there is no other employee stationed in Beijing already, an entry for it will be created first). Notice that, although the index is not explicitly referenced anywhere in this query, its definition must still be available to the query because it is needed to perform the index maintenance. In this example, the query imports the "news-data" module because it contains the declaration of the employees collection, which is referenced by the query.

The ArtCountEmp index is more complex than the CityEmp index, so the system may not be able to maintain it in an efficient way. Furthermore, the index contains "statistical" information, so it may be acceptable if its contents are not always in sync with the underlying data. Therefore, the ArtCountEmp index was declared as "manually maintained".

5. Integrity Constraints

Analogously to collections and indexes, XQDDF defines an additional extension to XQuery library modules which allows the declaration of (static) integrity constraints (ICs). Static ICs[1] can be used to ensure that, in every moment in time, all data which is stored in collections is accurate and consistent according to the semantics of an application. As in the relational world, XQDDF defines several types of ICs: Entity, Domain, and Referential ICs. Entity ICs check for the accuracy and consistency of all nodes in a collection. For instance, a special case of the Entity IC is the IC that checks for unique keys among all nodes in a collection. The Domain IC validates that each node in a collection satisfies a given expression. The Referential IC is used to ensure a foreign key relationship between the nodes in two collections.

In this section, we describe how such ICs are declared in a library module and how a particular IC can be (de-)activated. All ICs are described using examples for the news application. Specifically, we declare ICs for the data stored in the news-data:employees and the news-data:articles collections.

5.1. Declaration

As for collections and indexes, ICs must be declared before the user can activate them. An IC declaration specifies (1) the name of the IC for being used by function call to (de-)activate it (see next section), (2) the name of the collection(s) whose data

[1] Note that XQDDF doesn't define any dynamic integrity constraints which check the validity of a particular *update*.

should be validated, and (3) the expression(s) that guarantee the accuracy and consistency of the data. Analogously to indexes, ICs are declared inside the prolog of the library module that declares the collection(s) which is/are referenced by the IC.

5.1.1. Entity Integrity

An Entity IC is used to state the uniqueness of a key among all nodes of a collection. For example, the IC (named news-data:UniqueId) in the example below states that the value of the id attribute of each employee is unique among all other nodes in the news-data:employees collection.

```
declare integrity constraint news-data:UniqueId
  on collection news-data:employees
  node $id check unique key $id/@id;
```

The name of the collection is specified after the "on collection" keyword. The path expression following the "check unique key" keyword returns the value to be checked for uniqueness. The result of this path expression must not be empty and is wrapped to return an atomic value. The variable $id is successively bound to each node of the news-data:employees collection and available in the check expression..

5.1.2. Domain Integrity

The Domain IC allows the user to specify constraints that a particular node in a collection must satisfy. Domain ICs can be use in addition to XML Schema types or if no XML schema is available.

With the following example, we want to make sure that the name of each author of an article is not the zero length string. This can be particularly useful since there is no XML schema for articles.

```
declare integrity constraint news-data:AuthorNames
  on collection news-data:articles
  foreach node $article check fn:string-length($article/author/name) != 0;
```

The name of the IC is news-data:AuthorNames and it is defined on nodes belonging to the news-data:articles collection. The "foreach node" expression specifies a variable (using a QName) which is bound to each node in the collection. For each such node, the check expression is executed. For each node, the boolean effective value of the result of this expression must be equal to true.

5.1.3. Referential Integrity

The Referential IC requires every value of a node in a collection to exist as a value of another node in another collection. For example, in the database of the news or-

ganization, we want to make sure that each article is maintained by an (existing) employee. This can be done by declaring a so called foreign key IC. In the following example, this IC is given the name news-data:ArticleEmployees.

```
declare integrity constraint news-data:ArticleEmployees
  foreign key
    from collection news-data:articles node $x key $x/empid
    to   collection news-data:employees node $y key fn:data($y/@id);
```

The QName following the "from collection" and "to collection" keywords specify the source and destination collections, respectively. Each result of the key expressions are wrapped to return an atomic value. For each atomic value in the source collection, an atomic value in the sequence returned by the key expression on the destination collection must exist. The IC is violated if this is not the case for any node in the source collection. This semantics is equivalent to the following XQuery expression.

```
every $x in xqddf:collection(xs:QName(”news-data:articles”))
satisfies
  some $y in xqddf:collection(xs:QName(”news-data:employees”))
  satisfies $y/id eq $x//sale/empid
```

5.2. Life Cycle Management

ICs can be checked manually (if requested by the user) or automatically on updates apply time, after validation and indexes are computed. In order to be checked automatically, an IC needs to be active. ICs can be (de-)activated using the two updating functions xqddf:activate-integrity-constraint and xqddf:deactivate-integrity-constraint, respectively. Each function takes the name of the IC to (de-)activate as parameter. The flag indicating whether an IC is active or not is stored in the dynamic context.

Deactivating an IC might be useful if the corresponding check is expensive and, hence, inconsistency of the data might be acceptable and only checked (and fixed manually) from time to time. To check an IC manually, the XQDDF defines an updating function called check-integrity-constraint which triggers the IC, identified by a QName passed as parameter, to be checked.

Similar to collections and indexes, the module declaring the integrity constraints (i.e. with namespace http://www.news.org/data) can also declare variables whoes values are the QNames of the ICs. This allows their names to be easily referenced by subsequent expressions. For example, such a variable can be passed as a parameter to the activate-integrity-constraint in the importing admin-script module (see above). For the ICs from the section above, those variables are declared as follows:

```
declare variable $news-data:UniqueId := xs:QName("news-data:UniqueId");
declare variable $news-data:AuthorName := xs:QName("news-data:AuthorNames");
```

```
declare variable $news-data:ArticleEmployees := ▶
xs:QName("news-data:ArticleEmployees");
```

6. Conclusion

In this document, we have presented an extension to XQuery called the XQuery Data Definition Facility (XQDDF). It extends XQuery with three artifacts: collections, indexes, and integrity constraints. The document describes - using examples - the lifetime and evolution of such artifacts: how they are declared, how they come into existence, how they are used in XQuery programs, and how they are shared among multiple XQuery programs.

In its current state, XQDDF is implemented in version 1.0 of the Zorba XQuery processor ([7]) and version 1.0 of Sausalito ([8]). Sausalito implements an XML end-to-end architecture that allows to create RESTful web applications - entirely in XQuery and deploy them in the cloud. In its heart, Sausalito implements an XML database whose collections, indexes, and integrity constraints are declared and managed using components of XQDDF.

Bibliography

[1] Scott Boag et al. XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/

[2] Jonathan Robie et al. XQuery 1.1: An XML Query Language. W3C Working Draft, 15 December 2009. http://www.w3.org/TR/2009/WD-xquery-11-20091215/

[3] Don Chamberlin et al. XQuery Update Facility 1.0. W3C Candidate Recommendation, 09 June 2009. http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/

[4] Ashok Malhotra et al. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, 23 January 2007. http://www.w3.org/TR/2007/REC-xpath-functions-20070123/

[5] Don Chamberlin et al. XQuery Scripting Extension 1.0. W3C Working Draft, 3 December 2008. http://www.w3.org/TR/2008/WD-xquery-sx-10-20081203

[6] Cezar Andrei et al. XQuery Data Definition Facility 1.0. http://www.zorba-xquery.com/xqddf10.pdf

[7] Zorba XQuery Processor 1.0 http://www.zorba-xquery.com/

[8] Sausalito - An XQuery Application Server for the Cloud http://www.28msec.com/

The Future of XML at W3C

The World Wide Web Consortium and the Extensible Markup Language

Liam Quin
W3C
<liam@w3.org>

Abstract

The World Wide Web Consortium (W3C) published the specification for the Extensible Markup Language (XML, not an acronym) in 1998; since then, W3C has produced a wide range of specifications central to XML, including namespaces, xml:id, XLink, XPath, XPointer, Schema, XSLT, XSL-FO, XQuery, and, more recently, SML, XProc, EXI and more. Several pieces of work are coming to a close in 2010. In the future, should the XML Activity at W3C grow again, or shrink? What work should we be doing that we are not? (and what should we not be doing that we are?)

The session will explore work that W3C could consider in the future, and will ask the audience to participate with comments and suggestions. It should be understood that W3C is funded by Membership fees, so that the work we can do is limited to what Members (both existing and future) might support, but, within that framework, there is a lot of flexibility.

Several other standards organizations and industry consortia have produced specifications relating to XML; some (such as ISO) may also have endorsed one or more W3C specifications. Should W3C endorse the use of some other specifications alongside our own? What relationship would the conference attendees like to see between W3C and ISO or OASIS, for example, or the XML Consortium in Japan?

This is an interactive session to engage with the community, to ask questions and to hear people's views.

Keywords: XML, W3C, standards

1. Introduction

The Extensible Markup Language (XML) was developed by a small group of people, together with approximately two hundred industry experts, at the World Wide Web Consortium (W3C); the original purpose was to devise a subset, or profile, of ISO 8879 (SGML) that could be used interoperably in Web-based viewers. Today,

more than ten years later, XML is one of the world's most widely-used specifications for sharing text-based information with internal structure.

XML has passed from being new and unheard of, through being the golden boy, the subject of industry hype and fairy-dist, into a more mature phase in which it is largely taken for granted as a basic and ubiquitous part of computing infrastructure. Work at W3C continues in several areas, detailed below.

Today, what people ask for most in XML is stability, the property that any well-formed XML document will continue to be well-formed, and that any valid XML document will continue to be valid, and furthermore that the interpretations of such documents by XML processors is not changed. But there is still considerable room for further technical work, and perhaps also for liaison with more orgnizations.

This document does not answer questions about the future of XML and XML work at W3C: it exists instead as a framework for exploring what such work might be, based on the needs of the wider XML community.

2. Types of Work

Today, XML-related work at W3C can be broadly characterized into the following groups:

- Processing errata and fixing increasingly minor bugs;
- Revising XML as necessary as other standards change: for example, it was recently revised to keep it aligned with Unicode, clarifications to IRI syntax in system identifiers and namespaces have been made, and xml:lang values updated (in separate documents) as needed, both within W3C and elsewhere.
- Continuing to develop existing XML-related specifications: new versions of XPath, XSD (Schema), XSLT, XSL-FO and XQuery are in development in response to users' needs.
- Completing some specifications: the Efficient XML Interchange format is requesting implementation experience for its low-memory streamable XML-specific compression; The XML Processing Model Working Group has produced the XProc XML Pipeline language; the Service Modeling Language (SML) was recently published as a W3C Recommendation and that work has ended.
- Liaising with other activities within the W3C, such as the HTML 5 work, SVG, CSS and others, as well as (to a much lesser extent) other organizations such as OMG, WTF, ISO and IETF.
- New work: currently (Spring 2010) there are no new Working Groups within the W3C XML Activity, but this could change at any time.

3. The Future

The question for the audience, and for the wider community, is what work should W3C be doing in the area of XML: what would be most useful? What new work should we begin? What should we develop further? Or should we just shut up shop and go home?

Should we be doing more outreach, education, perhaps giving training, white papers, tutorials?

What about relationships with ISO, OASIS, The XML Consortium, and other organizations?

Real time, all the time, ragtime XML

Mark Howe
Jalfrezi Software Limited
<mark@cyberporte.com>

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

Alan Hazelden
Jalfrezi Software Limited
<alan@draknek.org>

1. Beyond document serving

All-XML environments such as XRX [10] can potentially simplify the deployment of client-server document delivery systems to browser-type technology. Multiple mature implementations of the component technologies of XRX (XQuery, REST, XForms) reduce resistance to the adoption of an approach that, at a systems level, is quite different from the traditional development model. However, the use of these technologies has a price:

- "Because XRX uses a single model for data (XML) it avoids the translation complexity of other architectures."[10] The same, however, cannot be said of XQuery itself. Standard XML tools are of limited help when constructing or inspecting XQuery, and machine generation and parsing of XQuery is a challenge no different in kind to machine generation and parsing of C++ or Word macros.

  This matters because in a strictly RESTful application "The current 'state' of the HTTP 'session' is not stored on the server [...] but tracked by the client as an application state, and created by the path the client takes through the web" ([5] p95), and the algorithm defining that path is represented in XQuery. This may not be a major concern for clients that assemble RESTful components in a simple way. But as soon as the client behaves intelligently (for example by comparing the results of requests to different RESTful services before deciding which combination of resources to use to tackle the task), key parts of the system algorithm are embedded in XQuery.
- Deducing the semantics of a RESTful URI using XML technology is not trivial, eg `http://localhost:8080/orbeon/exist/rest/db/orbeon/bookmarks/queries/bookmark.xq?hash=d3346bda2344033cf77460a4ef7363d8&user=drench` (from [3]). Even representing that URL in DocBook requires a CDATA section because of the clash between XML and CGI usage of the ampersand. It would be more ac-

curate to say that REST provides all-XRX server responses - there is nothing XML-like about the requests.

Furthermore, Richardson argues that the non-XML HTTP request types should be used to represent the semantic difference between, say, reading and modifying a resource (eg p97), and that information cannot be inspected by looking at the URL. HTTP provides a uniform interface, but only for the application programmer. The semantics of those HTTP headers are opaque to XML-technology application code unless the HTTP headers themselves are converted into an XML-friendly format (at which point REST starts to look a lot like XML-RPC).

Because a URL is not XML, standard XML tools cannot be used to manipulate them. For example, how should one validate the above URL?

- REST assumes a blocking request-response typified by HTTP over which it is often deployed. This is an extremely simple model to work with, and is supported by some extremely robust client and server technology, but it isn't a good fit for all types of problem. Also, non-blocking concurrency is an increasingly important paradigm for both hardware and software design, because 'blocking' and 'concurrent' are axiomatically hard to combine.
- "REST is more about a mindset than code, more about design than implementation" ([4] p246). "The traditional definition of REST leaves a lot of open space, which practitioners have seeded with folklore" ([5] p80). Appropriate mindsets and design matter, but folklore, for all its charm, is an unreliable guide for implementation. While the first X in XRX at least ensures that a Java developer will not find himself patching code in Lisp or Modula 2 on the other side of a RESTful URI, REST remains underspecified as an API.

  Attempts to define REST more rigorously do so at the expense of making REST less universal. Thus Richardson argues that his Resource-Oriented Architecture embodies the true spirit of REST (as 'specified' in a dissertation), while dismissing most well-known REST services as a hybrid of REST and RPC ([5] p79).
- How does a sys admin bring up, monitor or tear down an XRX system? How gracefully will the system degrade if one of the components fails? Can the system be modified dynamically? There are answers to such questions for any instantiation of an XRX system, but there is no generic way to get that answer, and no portable metadata at this level. Thus, while the components of XRX may lend themselves to combination in flexible ways, the system as a whole tends to be tied to a particular mix of implementation-specific detail.

Xcruciate [9] was originally developed as an all-XML solution for realtime systems such as chatrooms and virtual worlds, but the first practical application for Xcruciate was a web-based content management system called Xcathedra. There were commercial and technical reasons for proceeding this way, but there are a lot of other

ways to implement a CMS, eg [1], and this is the sort of problem domain for which XRX seems to be a good fit.

We therefore decided to rework the Xcruciate architecture with the following goals:

- Small, simple daemons as the unit of code reuse.
- Services that are not restricted to a blocking request-response model, and which publish an XML API in the form of a schema.
- Routing of data over a wide range of transports in a way that is transparent to the application code.
- Support for dynamic, redundant service provision.
- A generic, scalable mechanism for starting, stopping, monitoring and debugging clusters of daemons.

2. Xcruciate in theory

Figure 1 shows the building blocks of a distributed Xcruciate system. Processing is handled by daemons, which use and provide services that are defined in terms of input and output events. Gateways translate between the language-specific I/O mechanisms (eg a Java queue or a Unix pipe) and a transport mechanism such as TCP..

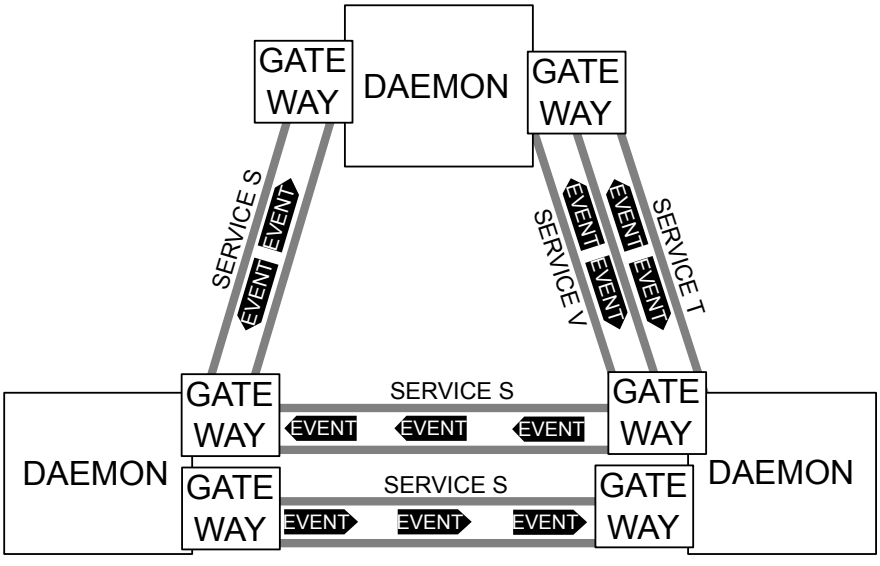


Figure 1. Daemons, gateways, services and events

Services generally make no assumptions about the gateway through which they are provided. A daemon may serve the same service via different gateways, and

may serve different events from the same service to the same daemon via different gateways. In this respect the Xcruciate model is more like XML-RPC than REST, because 'wrapper' information is in XML rather than in gateway-specific headers ([5] p14).

While services can run over any gateway, some gateways are more equal than others for particular tasks. The application code does not directly control the gateway method, because this would tend to break portability, but it can express a preference for either rapid (UDP-type) or safe (TCP-type) transport as shown in table 1. Essentially, internal gateways such as Java queues or Scala actor mailboxes are always preferred, and after that it's a choice between the reliability and predictability of TCP and the speed and scalability of UDP.

Table 1. Preferred gateways (higher number = higher priority)

Gateway	Priority for "fast"	Priority for "safe"
Scala actor message	100	100
Java queue	80	80
UDP	60	20
xcrdlib TCP	40	60
HTTP	20	40

All daemons use the xcrdnoc service, which links daemons within a system into a hierarchy for management purposes. (There is no requirement or indeed expectation that application data will be routed according to this hierarchy.) When the first daemon is started, it is passed an XML document containing configuration information for its own use, but also information about the services provided by other daemons. On the basis of this information it can start other daemons, passing them a subset of the information provided in its configuration file, and so on.

The xcrdnoc service allows the providing daemon to control startup, shutdown and other daemon-like behaviour, and provides logging and reporting services to the daemons it starts. It also provides a mechanism to enable each daemon to locate and connect to the services it requires. xcrdnoc uses a fixed priority system to determine how each daemon satisfies its need for services, but allows for redundant provision of services. Thus, for example, an Xcruciate system can use a cheap mailer when available, but switch to a more expensive mailer if the cheap one goes down, and revert to the cheaper one when it comes back up.

Figure 2 shows the logical structure of a generic Xcruciate daemon, the implementation of which is language-dependent (actors, threads, pipes between processes...) The wrappers of incoming events are checked, after which they are passed via an optional validation stage to the application code. The application code typically runs in a loop, which enables it to generate output events without the trigger

of a specific input event. Application code output passes through an optional validation stage before being routed to the appropriate daemon via the appropriate gateway. xcrdnoc events are handled by the daemon without intervention from the application code.

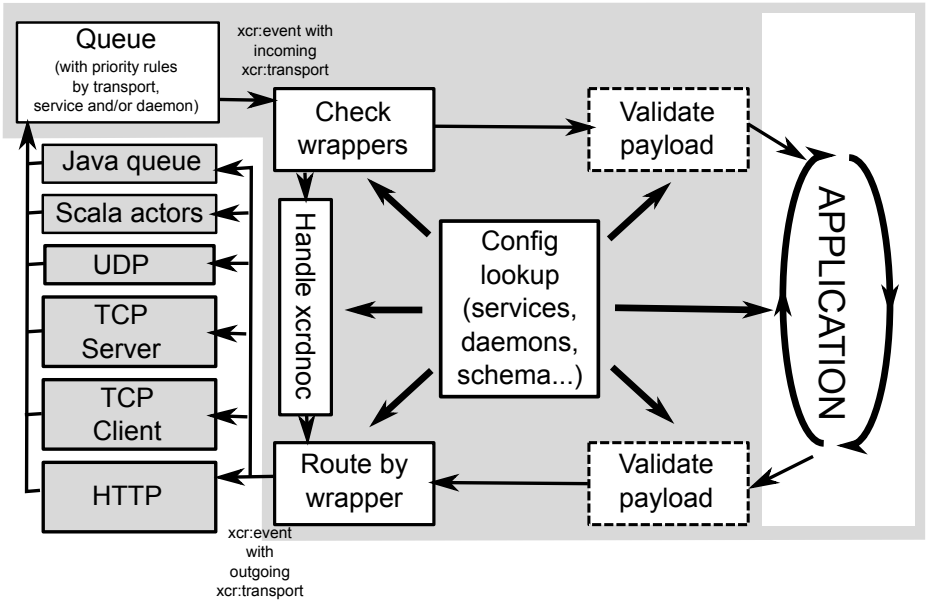


Figure 2. Generic Xcruciate daemon

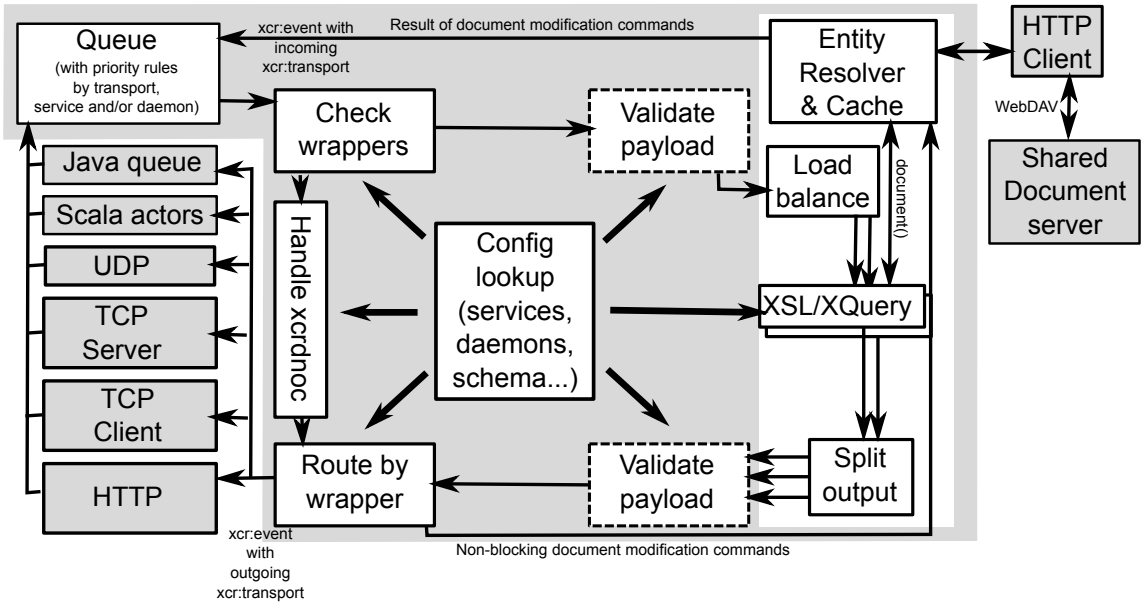


Figure 3. Xacerbate daemon

Figure 3 shows how the generic Xcruciate daemon structure applies to XSLT processing. The XSL processor only runs when there is event input - Xacerbates daemons that do housekeeping or other time-dependent tasks need an external source of

timer events. Depending on the semantics of the events, it may be possible to run several transformations in parallel. The output document is split into zero or more events which are processed as for the generic daemon.

Local resources are accessed via blocking reads from an entity resolver and cache using the XPath document() function. The entity resolver may store stateful data locally as files, or it may use a web service via HTTP, WebDAV or some other protocol. The shared document server, if present, could be implemented using XRX-type technology.

3. Xcruciate in practice: a distributed live music system

In order to test our model we designed a somewhat quirky music production framework, summarised in Figure 4. Multiple keyboards can be used to play music that is distributed on a note by note basis between connected players. A daemon provides sound sample information and the samples themselves, while a recorder can record key presses as they are played for future replay via a dukebox (digital jukebox).

Cataphonix is about the performance of music rather than the representation of musical scores at which markup schemes such as SMDL excel ([6][4]). The Cataphonix representation of note information is therefore based on a subset of the widely-used MIDI 1.0 standard [2]. There is an event to start a note (p22), an event to stop a note (p23) and an option to stop all notes (p23), which is apparently required in MIDI to handle the analogue equivalent of UDP packet loss leading to a never-ending note. Cataphonix keyboards and the dukebox in playback mode are broadly equivalent to MIDI controllers (p88) while the recorder is broadly equivalent to a MIDI sequencer (p102).

We are not suggesting that this is an entirely sensible way to design a music system. The aim was rather to provide a simple but non-trivial test case involving multiple types of daemon that are connected and configured dynamically, and which process XML information in real time. (One benefit of choosing musical output is that latency in music is very easy to hear).

Figure 5 shows the basis of decisions about the implementation language for the various Cataphonix daemons. XSLT has been used wherever possible, but exceptions have been made where the daemon needs to perform tasks that are not easy to achieve directly through XML manipulation. It is relatively easy to revisit these implementation decisions at some later point because all Xcruciate daemons use the same bootstrap and communication protocols.

[4] Links to the original specification seem to be broken.

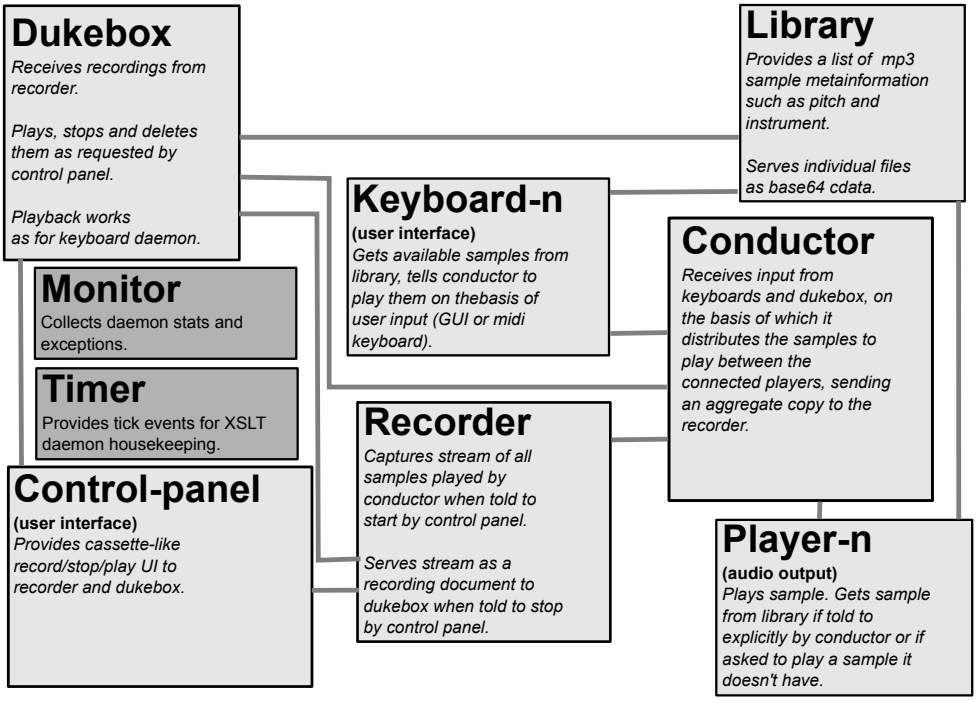


Figure 4. Overview of Cataphonix

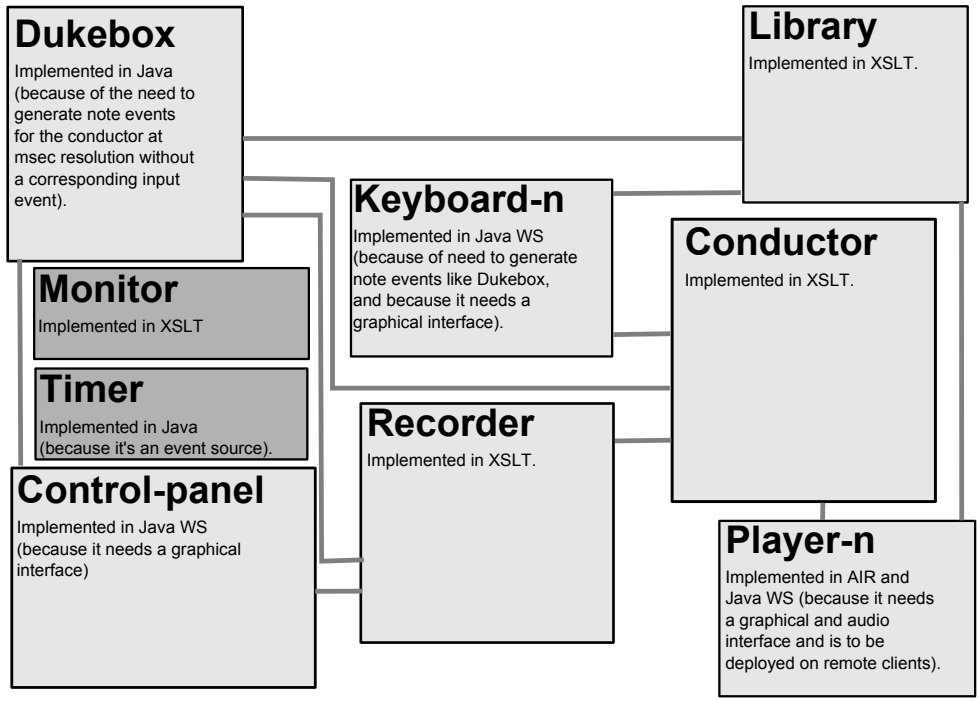


Figure 5. Choice of language for Cataphonix daemons

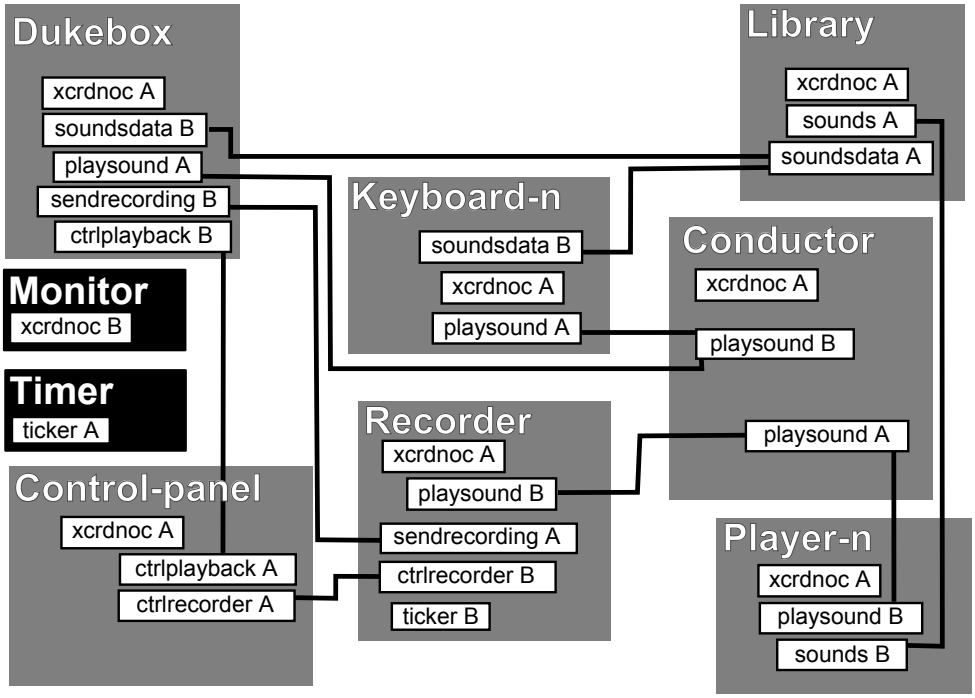


Figure 6. Cataphonix services

Figure 6 shows the services provided and needed by the Cataphonix daemons. It should be noted that the "ends" of the services are named A and B, to avoid the connotations associated with "client" and "server". In general, the net flow of data is from A to B but, for example, either end may involve a TCP server. It is useful to label the ends ot the service in order to know which Schema to use to validate input and which to use for output. This point is made explicit in Figure 7, which shows one possible set of gateways over which the services could be routed. HTTP has been used for the daemon which is closest to a classic document server, while UDP has been used for rapid transfer of small documents such as notes and daemon statistics.

Figure 8 shows an alternative set of gateways, assuming that the core Cataphonix daemons are written in Scala and running within a single JVM. (This makes sense because of Scala's excellent support for non-blocking concurrency via actors, but also because the memory footprint of the JVM prohibits large numbers of single-daemon JVMs per machine). All intra-JVM communication is via actor messages.

Figure 9 shows the same setup as figure 8, from outside the JVM. Effectively, the multiple daemons inside the JVM appear to the keyboard and player daemons like one daemon that implements five services over HTTP and UDP. Of course the same effect could be achieved without the JVM by restricting access to "internal" services to particular daemons. It is thus possible to implement encapsulation, where the actor-based traffic between JVM daemons in this example are analagous to private methods in an object-oriented paradigm.

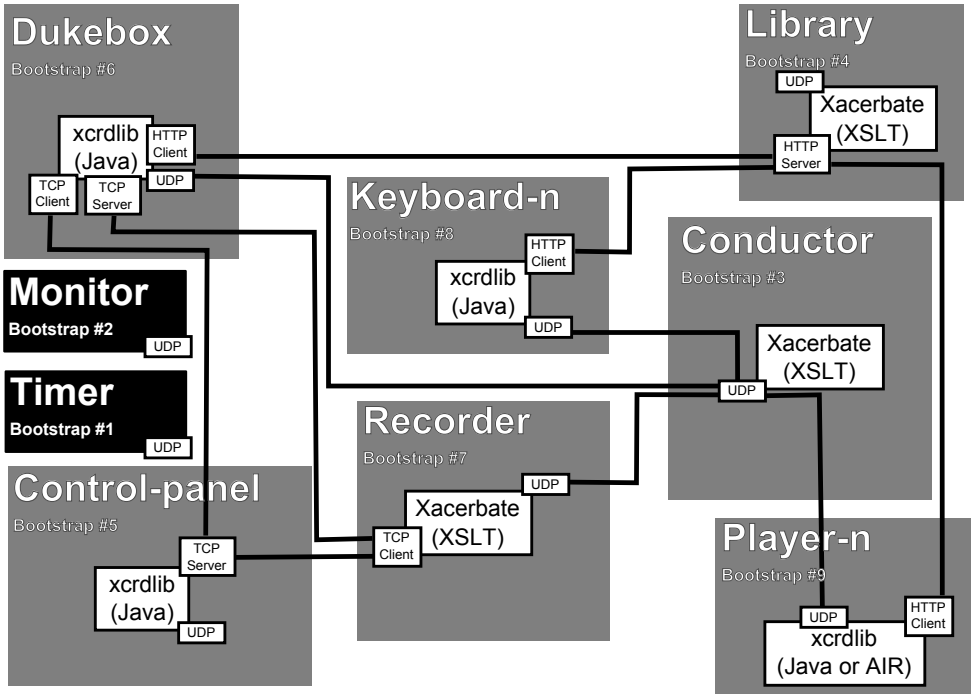


Figure 7. Cataphonix daemons and gateways (non-JVM)

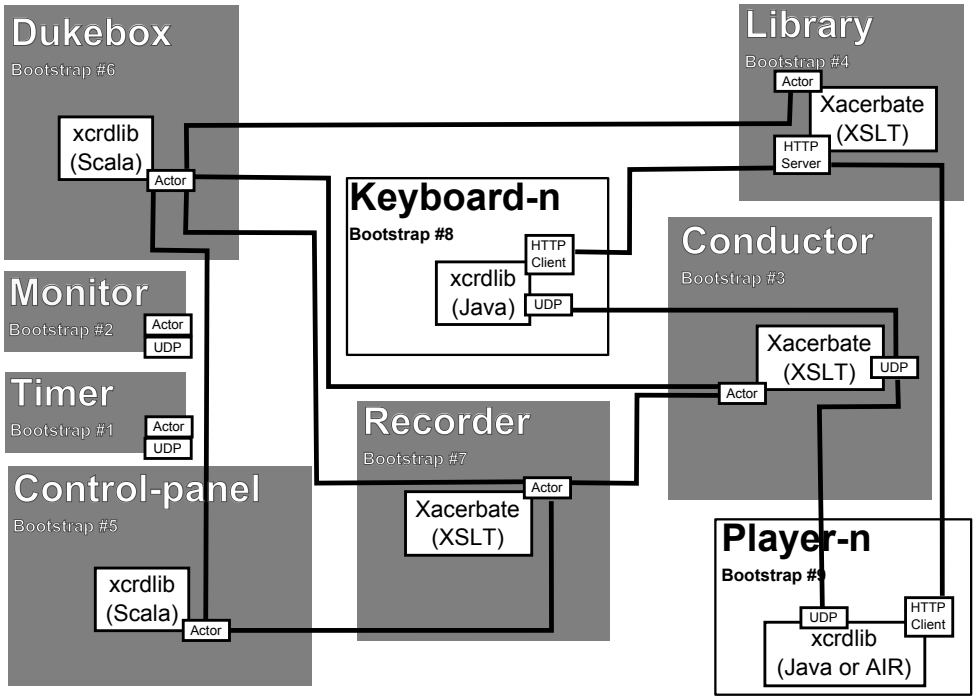


Figure 8. Cataphonix daemons and gateways (Scala)

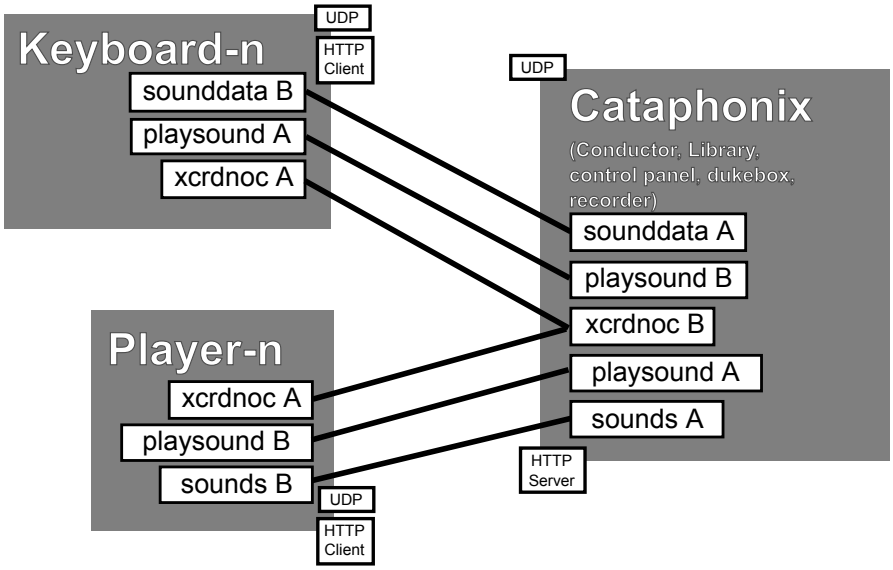


Figure 9. Encapsulation of daemon functionality

4. Issues to explore

Cataphonix has helped us to clarify many issues concerning realtime inter-daemon communication, not all of which are addressed in the current design.

- The service-based implementation of Xcruciate scales much better than the previous version because it can make full use of concurrency features in both hardware (multiple cores) and software (Scala actors). However, we have yet to extend our scheme to work across multiple machines, where additional issues such as remote control of daemons must be considered. Linked to this is the need for more sophisticated authentication mechanisms between daemons - Cataphonix currently uses shared pass strings which, while workable within one secure machine, are not ideal for sending across the Internet.
- Unit testing of an Xcruciate system is more complicated than testing a RESTful service, because there is not necessarily a single response per request. Our current thinking is that a test suite would need to be able to control the whole system, which is possible via xcrdnoc, but we have yet to see how this works out in practice.
- In this phase of development we have concentrated on linking together server daemons. Some of our potential applications involve rich clients, and we are attracted by the idea of handling client-server communication in a way analagous to intra-server communication. The challenge, of course, is identifing the points at which the two scenarios are significantly different. For example, client connections need to be more dynamic, and the servers they connect to need to be less trusting of clients than they would be of other daemons on the same machine.

5. The RESTlessness of Xcruciate

By way of conclusion, it may be useful to compare the behaviour of Xcruciate with the four properties of Richardson's Resource-Oriented Architecture:

- Xcruciate does not provide URI-style *addressibility* (p84), where each resource has at least one unique label, because it is not fundamentally a resource-based model. It is hard to imagine what persistent resource could correspond to a request to play a note, for example - an XML representation of the loudspeaker's contribution to global warming?

  However, there is no reason why particular services cannot provide addressibility where that makes sense. Thus the sounds and sounddata services provided by the Cataphonix library provide addressible resources.
- Similarly, Xcruciate services are not guaranteed to be *stateless* (p86). For many realtime applications, server state is rather important. For example, in a chatroom, the ability to send typed speech to the people in the room at the time, rather than the people in the chatroom specification, is generally considered to be a feature. Statelessness assumes a clear "before" and "after", which maps better onto request-response interaction than to data sources and sinks.

  However, as with addressibility, particular Xcruciate services may be stateless, such as those provided by the Cataphonix library, keyboard and player.
- A similar argument applies to *connectedness* (p94). It is possible to refer to Xcruciate resources that are addressible. For example, the sounds event of the sounddate service returns a list of sample metadata, including a unique identifier that can be used to request the sound sample itself.
- Finally, Xcruciate services do provide a *uniform interface* (p97). Indeed, it could be argued that the interface is more uniform than the REST interface because it is independent of implementational details. Richardson points out that REST over WebDAV "makes your service incompatible with other RESTful services" (p104). Xcruciate could potentially offer the same service over HTTP, WebDAV and UDP, because its semantics do not depend on the specifics of HTTP or WebDAV.

Bibliography

[1] Cocoon http://cocoon.apache.org

[2] The MIDI Manual, 3e. David Miles Huber. Focal Press 2007

[3] XRX: XQueries in eXist, by Jeni Tennison, July 3, 2008 http://news.oreilly.com/2008/07/xrx-xqueries-in-exist.html

[4] Programming web services with Perl. Randy J. Ray, Pavel Kulchenko. O'Reilly 2002

[5] RESTful Web Services. Leonard Richardson, Sam Ruby. O'Reilly 2007

[6] SGML: SMDL overview http://www.coverpages.org/smdlover.html

[7] The Extensible Stylesheet Language Family (XSL) http://www.w3.org/Style/XSL/

[8] Xcruciate: real-time XSLT for cross-media social networking http://www.xcruciate.co.uk/

[9] Imagining, building and using an XSLT virtual machine. Mark Howe, Tony Graham. XML Prague 2009, pp. 189-206

[10] XRX: Simple, Elegant, Disruptive, by Dan McCreary, Friday May 23, 2008 http://www.oreillynet.com/xml/blog/2008/05/xrx_a_simple_elegant_disruptiv_1.html

Film Markup Language

Automating Cinemas Using XML

Ari Nordström

Abstract

Film Markup Language (FML) is a DTD and a processing model for automating commercial cinemas. This whitepaper outlines the principles behind cinema automation, breaking down the process into semantic components, and applies XML to put the principles into practice.

1. Introduction

FML (Film Markup Language) is a DTD and a processing model for automating cinemas. This paper outlines both. First, however (assuming you're not familiar with the practical details of running a cinema or the technical history of showing films), I will start by explaining some of that since I believe it's valuable to know some of the gritty details before proclaiming how to revolutionise it. Then I'll break down a typical screening into its semantic components, and in light of the breakdown, walk through the FML DTD and its process model.

1.1. A Hundred Years of Cinema Technology

The cinema as an institution is more than a hundred years old. The basic idea, to show a series of still images in rapid succession so that the eye is cheated and that series of images is magically transformed into a moving image, has not changed since. Apart from improved mechanical and electronic solutions in projectors, the basic mechanism is still the same. Projectors still stop each and every individual frame of the film print at the film gate to project them on the silver screen.

Of course, technology's more advanced now. There are better optics, used to produce lenses that projecting sharper images, and there are better light sources to illuminate the film frames with. Some thirty years after the film projector was first introduced, sound was added to the experience. Stereo sound took another twenty years, and in the mid-nineties, the sound was made digital, with acronyms such as *SRD* and *DTS* helping to sell tickets.

The cinema industry is a very conservative business and every innovation, every new solution, is regarded with scepticism. If a box-office hit happens to use a new technology, the new technology might succeed, but then again, it might not. The history of the cinema is full of clever innovations, from Smell-O-Vision to 3D, that never really took off or died in spite of millions of cinemagoers embracing them.

The history of the cinema is also about the war against the television, cable-TV, DVDs, and, lately, Blu-Ray discs. When television made its entrance, cinemas all over the world died in the thousands, and the film industry was forced to try one new technology after another. The audience was treated with first CinemaScope and stereo sound and then CINERAMA, 70mm presentations, Ultra Panavision, and so on, but still too many people stayed home and watched Archie Bunker, [The Lucy Show], and the like instead.

The real reason was costs. A cinema then consisted of a single screen: only a single feature could be run at a time. If that feature wasn't a success, well, too bad. The four or five ushers still had to be paid, just as the four candy girls, five box-office ladies, and the projectionist.

As time went by and fifties turned into sixties and then seventies, the cinema staff shrank from fifteen people to four, with extras coming to work during premieres and long runners. As the staff became smaller, everyone got more to do, everyone except the projectionist.

Upstairs, in the projection booth, the projectionist still did what he had been doing for the last seventy years. He ran the show. At his disposal were two projectors, a sound rack, and dimmers to handle the auditorium lighting. He would thread the first reel in one projector, dim the lights, and start the show. He would then thread the second reel in the other projector and wait until the first was about to end, at which time he would perform a *change-over*, and without any interruption apparent to the audience switch over picture and sound to reel two. He would then thread the third reel in the first projector, rewind the first reel for the late show, and wait until the time came to perform another change-over. This would go on from anywhere between five and twenty reels, depending on the film's length and the film lab's ambition level, until the show was finished. The projectionist would then turn up the auditorium lights and pull the curtain, and the process would start anew.

Inevitably, the idea was brought forward to automate the cinema so that the projectionist could become a part-time usher, admitting patrons into the auditorium while the trailer reel was running. Of course, a number of tasks could be remotely handled, without having to *automate* anything, but in the end, new technology was required.

The first problem to solve was how to avoid change-overs. The obvious answer is, longer reels, but surprisingly, a number of systems with automatic changer-overs were tried. Part of the reason was that the projection light source then was [carbon arc]-based lamphouses, where the light was generated by driving a strong current through carbon rods. The carbon burned out in about twenty-five minutes, setting a practical limit to the reel length.

When Xenon bulbs were introduced, reels became longer, and eventually non-rewind systems were introduced. On these, reels are spliced together on a horizontal platter (see Figure 2), with the start of the film nearest the uptake core. Once the print is assembled on the platter, the core is removed, and the film can be threaded

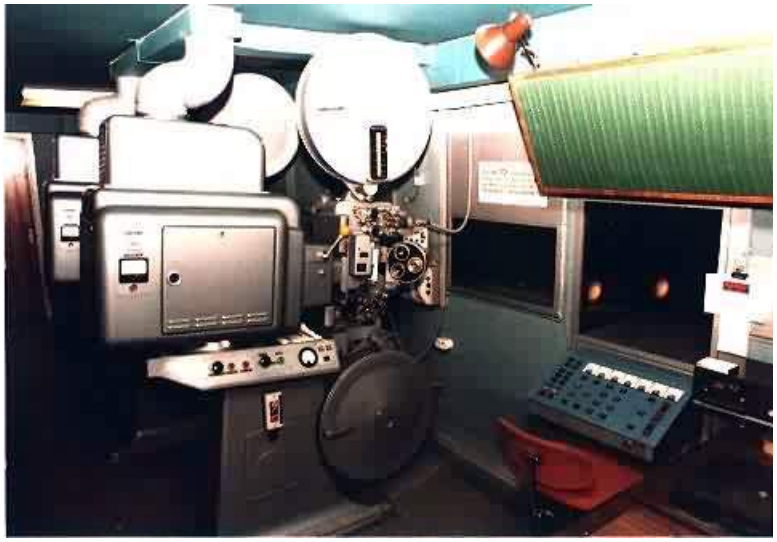


Figure 1. The Draken Cinema's Projection Booth

through a platter speed regulator mechanism and a set of rollers to lead the film to the projector and back to the uptake platter. In this way, the whole film can be run on a single projector, and doesn't have to be rewound after the show.

With the introduction of the platter came automation, and with automation, the idea of multiplex cinemas where a number of auditoriums are placed under one roof. The multiplexes are obviously more economical to run since a smaller staff can handle more features. Often, a single projectionist can handle up to a dozen auditoriums, provided that the booth(s) are well planned. Keeping this is mind, it is easy to see why the introduction of multiplex cinemas came to mean the death of most single-screen cinemas.

1.2. Cinema Automation

Most types of cinema automation systems work like mechanical pianos. They are event-based, and the placement of the [instruction] decides what will happen. Most Swedish cinemas use a type of automation where events are initiated from the print itself—the film is [programmed] by placing pieces of aluminum tape on the print, either on the edge of the film strip or on the area between frames. A sensor placed on the projector reacts when a piece of tape passes and sends this information to the automation device, basically a step-by-step matrix mapping events to the passing aluminum tapes on the film, causing the device to bump up to the next event. Examples of such events may be to dim auditorium lights, change a sound format, pull the curtain after a show, or ignite the Xenon bulb. Anything that can be *event-driven* can be automated.

Other types of automation also deal with events and actions to be taken for each event, but instead of the print and pieces of aluminum tape, events are generated

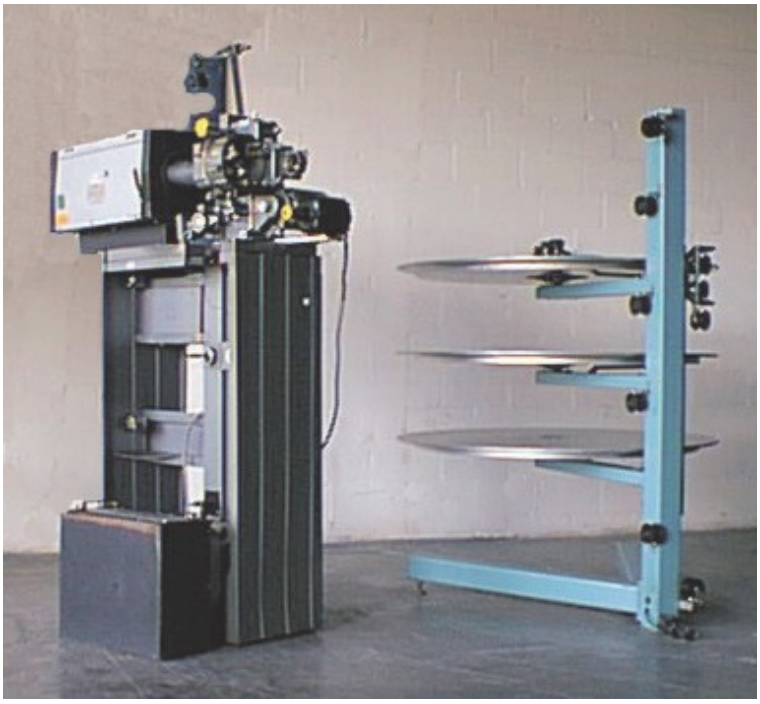


Figure 2. A Platter System

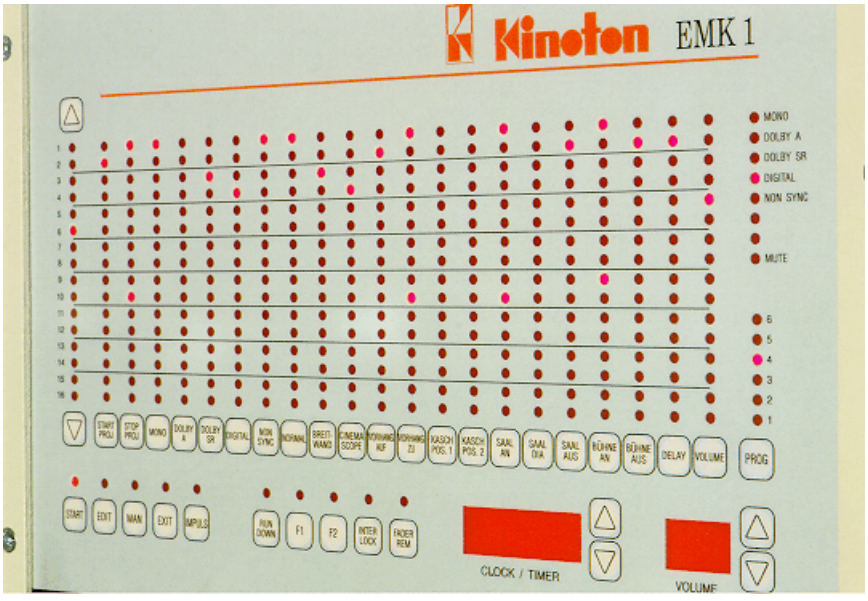


Figure 3. Kinoton Automation Matrix

by *timing* them: for example, [switch the sound format from mono sound to four-channel stereo and Dolby SR type noise reduction twelve minutes and fifteen seconds into the show] or [pull the curtain one hour, fifty-four minutes, and thirty-eight seconds after start]. These values are cinema-dependent, unfortunately, since few

projectors will run at exactly the same speed. Yes, the standard is 24 frames per second, but deviations from this are common.

Also a complete screening will have a fairly large number of events since the show consists of not only the film, but commercials, trailers, logos, and, perhaps, light shows or other events specific to an auditorium. But before we take a look at the components of a cinema from an object-oriented view, allow me to trace the origins of FML (Film Markup Language).

1.3. How Film Markup Language Was Born

Long before I became an SGML and XML consultant, I worked as a projectionist at old single-screen cinemas to have at least some money to spend while studying. I loved the job so I've stayed on, showing old classics once a week at one of last few large single-screen auditoriums in Göteborg, the Draken cinema. As I became involved with SGML and later, XML, I got the idea to write screening instructions for other projectionists working at Draken—basically schemata with instructions on the picture and sound formats of the show, change-over information (how to locate the cue marks on the image), and curtain calls to create an aesthetically pleasing show—in XML. Then, it was a question of creating a simple fact sheet. It seemed like a fun thing to do so I jumped into it. But while gathering data and thinking about a suitable DTD, I realised that XML would be an ideal way to describe both a screening and the cinema itself in an object-oriented way, and that the XML instance or instances could be used to automate the whole thing. As I said, cinema automation is a lot like a mechanical piano and XML is ideal for describing that kind of event-driven action.

> **Note**
> The fact that I love old cinemas has not prevented me from working at the local multiplex at times. I have more experience running automated cinemas than I care to admit.

The DTD grew and I had long and fruitful discussions on suitable processing models with my good friend, XML consultant and programmer Henrik Mårtensson. What we came up with is what is presented here. Henrik still disagrees with me what exactly the objects of the model are, and what their relations are, but the model works.

2. A Cinema's Components

Before digging into the FML DTD, it is probably wise to discuss the things it describes. So here's [Cinema 101], the basics of the technical side of a cinema.

2.1. The Auditorium

From a patron's point-of-view, the most important part of any cinema is the auditorium, preferably with a large screen. The auditorium is lit with a groups of lights of which some are usually spots aimed at the screen. All of these are controlled from the booth; it wouldn't make sense otherwise. Sometimes there's also on or more curtains but they are mostly a thing of the past, when attending a show was a happening.

Sometimes the auditorium also has a light show of some kind, or some other gimmick (one of the more unusual ones I've seen was a statue of a lady raised from the proscenium floor between shows) to enhance that feeling of a *happening*. In these times of automation, they're rare since having one means that the often limited space on the automation matrix must be devoted to the gimmick.

The screen is usually framed using black velvet-clad masks. The masks are movable so that the screen's aspect ration may be changed to correspond to the film's intended *projection format*. [Widescreen] is such a format, and uses an aspect ration of 1:1.85 or 1:1.66. [CinemaScope] is another and uses an aspect ratio of 1:2.35. A third format is the still common television aspect ratio, 1:1.37. This format is called [Academy] (and don't even ask why; we won't go there).

Really large cinemas (like the Draken cinema) sometimes handle other formats used for the more esoteric and rare screenings, for example, 70mm film presentations.

Behind the screen are (usually) three speakers, [left], [centre], and [right]. Often, there's also at least one subwoofer. On rarer occasions you'll find cinemas with five screen speakers with the extras placed between the centre and side channels (either for 70mm presentations or for a recent, but rare, digital sound system from Sony). On the auditorium side and back walls are another dozen or so speakers for the [surround] channel; sometimes, they're split into two or more channels.

2.2. The Projection Booth

The projector booth contains, in essence, one or more projectors and a sound rack. The newer multiplex cinemas have just one projector per screen while the older single-screen booths often have two or three, and sometimes even four. Consequently, the multiplexes always have some kind of platter or other non-rewind system. Some of the older ones have them, too, so that the poor projectionist can double as an usher.

There's also a sound rack. Normally, it will feature at least sound processor of some kind, often manufactured by Dolby Laboratories Inc., and a number of power amplifiers to feed the speakers in the auditorium.

Somewhere near the projector is probably also an automation box. It could be of the matrixed type discussed, a PC handling [timed events], or some other construction, most likely an old one.

2.2.1. Projecting Images

The projector's job is to get an image to the screen, be the film analogue or digital. Every single frame to be projected must stop at a film gate where it must be perfectly still while the projector's shutter is open or the illusion of movement will be lost in the blur that ensues.

Remember the screen aspect ratios I talked about in Section 2.1? They're required because the film is shot in that way; the actual frames on the film strip have that aspect ratio. So to keep out unwanted parts of the frame on the screen, the gate must be masked using that same aspect ratio. This is done using an *aperture plate*, and since every aspect ratio that the cinema can handle needs its own, we must be able to change aperture plates during a screening (for example, if a trailer uses another aspect ratio than the main feature).

There's the matter of the change-over mechanism; to block the picture when the other projector is running (or when the projector stands still, or when changing lenses, etc.) there needs to be some kind of change-over shutter.

Different lenses with different focal lengths are required to enlarge the frame to its proper size on the screen, often one lens per aspect ratio (sometimes one lens can handle several aspect ratios, but that depends on how the velvet masking of the screen works). So in order to be able to switch between lenses during a show, automatically, the lenses must be placed on a turret that can be rotated using an electrical motor.

Most projectors today use lamphouses with Xenon bulbs to produce the light that is required. The Xenon bulb is fed anything between 50 and 150 Amperes DC or more, depending on the size of the screen and the projection distance. The DC current is created in a rectifier that may or may not be integrated with the projector. Digital projectors frequently use more than one light bulb but the basic principle is the same.

2.2.2. Cinema Sound

Cinema sound is traditionally optical. The print has two optical tracks, one left and one right, that are fed into the processor, and is electronically remade into four channels. This is what in layman's terms is called [Dolby Surround] or [Dolby Prologic]. Two different noise reduction systems are commonly used, [Dolby SR] and [Dolby A]. The optical soundhead on the projector consists of a lamp, optics to focus a thin *slit* on the film, and two cells to read the variations in the optical tracks.

Newer and vastly better is digital sound, delivered either directly on the print (either [Dolby Digital] or [Sony Dynamic Digital Sound]), or on CDs ([Digital Theatre Sound]). These sound formats require an additional sound processor to decode the track. Soundheads installed on the projector are also required, obviously.

Note
If the cinema cannot handle digital sound formats, the print's optical sound tracks are used instead.

Older sound systems include six- or four-channel magnetic strips on 70mm or 35mm film, used to produce either [Academy sound] (in a few variants) or Dolby's magnetic sound formats (of which there are a couple). Magnetic sound, again, requires an additional sound processor. These sound systems are now extremely rare.

Finally, most cinemas include some sort of system for tape or CDs, to be used between shows, and a PA system.

2.3. The Components of a Screening

Any show in a cinema can be divided into easily recognizable components:

- *The show starts:* The curtain, if one exists, is pulled, and possibly, light shows or other special features of the auditorium are performed. The correct picture and sound formats are chosen, both in the booth and on the screen The projector is started, and a first changeover is done to switch in the picture and sound. Auditorium lights are dimmed to about 50%.
- *Commercials and trailers:* If the cinema shows commercials or trailers, these are shown first. Possibly, they will require different sound or picture formats. Quite possibly, the commercials presentation will end with the cinema owner's logo.
- *The main feature starts:* After finishing trailers, sound and picture are switched off, and the curtain is closed (if one exists). Spotlights light it while the correct picture and sound formats are chosen. Then, the curtain is again pulled, lights are completely faded out, and the projector picture and sound are switched on. First in line are logos such as a Dolby Digital or THX logo. Then, for about two hours, nothing more will happen unless the cinema has two projectors and change-overs are required, or if something goes wrong.
- *End credits roll:* During end credits, lights are usually faded up to about 50%. When they're over, the curtain is closed, spots light up the curtain, and auditorium lights come up to full strength. Automated cinemas often now switch on intermission music.

3. The FML DTD and Process Model

The FML DTD consists of three parts:

- A `show` structure that describes the screening itself, as a series of events.
- A `theatre` structure that describes the properties of the cinema.
- A `film` structure that describes the properties of the film.

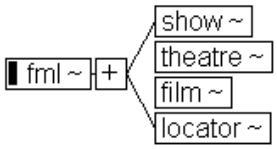


Figure 4. The FML Root Model

In addition to the three main structures, there's also a `locator` element, used to link to other FML documents (see Section 4.2 for more information about the philosophy of creating FML documents). The `locator` element uses the Simple XLink mechanism for pointing out the documents.

3.1. The Film Structure

The `film` structure, shown in Figure 5, describes the properties of a film. Any film, I might add, from the Paramount logo to [Ben-Hur].

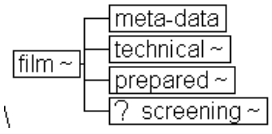


Figure 5. The `film` Structure

3.1.1. Film Meta-data

The `film` structure starts with meta-data about the film, for example, its original and translated title, the print number, as well as information about any subtitling or dubbing the print may have. The `meta-data` element also contains production information about the film, for example, cast and crew, taglines, a description of the story, and other PR-related data. The production notes can easily be used when booking ads in newspapers, or when presenting a new feature in the cinema's lobby monitors.

An FML document package could also contain images and trailers to be shown on the lobby monitor screens.

3.1.2. Technical Data about the Film

The `film` structure includes a technical data structure, shown in Figure 6.

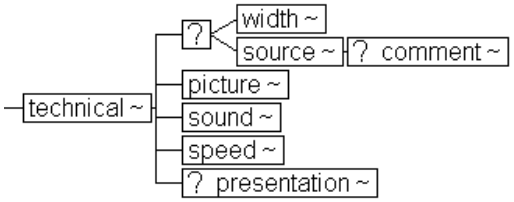


Figure 6. The `technical` Structure

The `technical` element describes the technical properties of the film:

Width ([Gauge]) — The vast majority of films are distributed in 35mm (implying the width of the film strip), but prints are still sometimes produced in, for example, 16mm or, in very rare cases, 70mm. This structure cannot be used with a digital print (`source` must be used instead).

Source — A number of feature films now come in a digital format and therefore the `source` structure can be used instead of `width` to denote the source format of the feature film.

Aspect ratio — Aspect ratios are fairly standardised, as I've indicated in Section 2.1, but variations do exist. These films are often either old, or produced using some proprietary system.

Sound format — Sound formats, again, are reasonably standardised. Most films have at least optical Dolby sound, or sound derived from Dolby formats (the patent for Dolby's original [Dolby A] noise reduction expired so a number of Dolby clones exist; later this also happened with their improved [SR] noise reduction). There are also a number of digital sound formats, as well as older magnetic-strip formats. And then, of course, there is mono sound, often called [Academy].

The `sound` structure also includes an optional volume setting for the print, replacing the letter from the production company often accompanying a print that states that the film should be played at [7] at the Dolby logarithmic volume scale. If the receiving cinema has a sound processor calibrated according to the Dolby standard curve, any volume setting can be given.

There's also a structure for equalizer settings; if the film deviates from the Dolby standard curve, this structure may be used to describe how.

Speed — Normally, 35mm film is projected at 24 frames per second, but variations do exist. New productions, however, especially those intended for television, may use 25 fps. Silent films

often use other, slower, speeds, for example, 18 or 16 fps, while 70mm productions, as well as *IMAX* and *OMNIMAX* features, sometimes use 30 fps or higher.

> **Note**
> While newer projectors will actually hold exactly 24 fps (or 25, or any other desired speed), most projectors out there do not; variations exist and are significant enough to affect any time-based automation.

Presentation — The `presentation` element identifies the film's visual properties, for example, if the film is in black and white or if it uses a 3D system.

Note that the `film` substructure describes the technical properties of the *film* itself. The `film` structure, therefore, should be regarded as a [wish list]. When referred to from the `show` structure (see Section 3.3), the values here will act as commands that may or may not be executable, depending on the cinema's technical capabilities (see Section 3.2.2).

3.1.3. Prepared

The `prepared` structure, shown in Figure 7, is practical information entered and used at the cinema. The names of the projectionists that inspected and assembled the print are given here, as well as the date of the assembly, the cinema where the assembly was made, and additional notes, if needed.

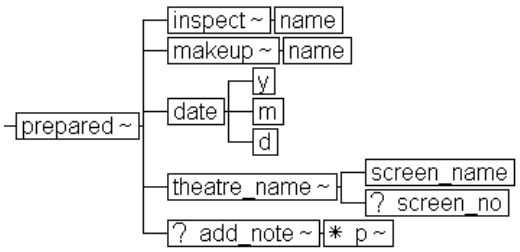


Figure 7. The `prepared` Structure

Unless additional notes need to be made, this information can be entered by the FML system, when the show is prepared.

3.1.4. Screening

FML began as a simple DTD for writing change-over instructions for cinemas without automation. This structure, the `screening` element, shown in Figure 8, can still be

used for that purpose but in an FML implementation, it is used to list *timestamps* and *offsets* in the film print.

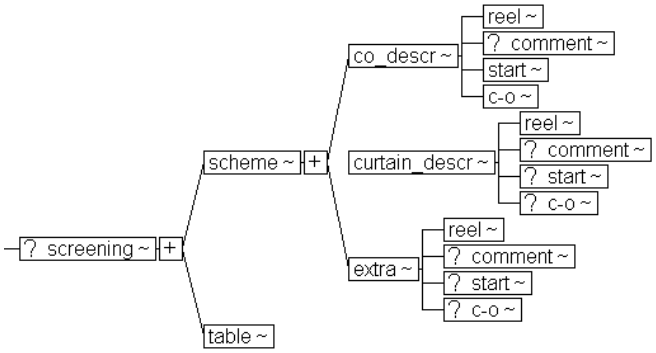


Figure 8. The `screening` Structure

The change-over instructions consist of a number of change-over descriptions. There's a reel number, a description of when to look for the cue marks (normally placed in the upper right corner of the image, usually 7.3 seconds before the last frame), and a description of the appearance of the cue mark itself. A curtain close instruction can also be described, to perform an aesthetically pleasing show.

In an FML implementation, the `screening` structure basically uses a series of three types of events in the `scheme` structure, of which two directly correspond to the number of reels. Every such event includes timestamps and offsets for reel length, change-over marks (from the start of the film, that is, from timestamp 0, to the change-over mark), end credits start, curtain calls, and so on. In an FML implementation, these timestamps and offsets can completely describe the timeline for a complete show.

The third type of event, `extra`, is used by the film's producers to include timestamps and offsets for additional events if such are required. This structure can also be used by the cinema's staff to add cinema-specific events for a show.

A CALS table structure is also included, because a table is sometimes preferred by projectionists for creating change-over instructions. Also, for paper descriptions, it may sometimes be useful to have both.

3.2. The Theatre Structure

The `theatre` structure, shown in Figure 9, describes the properties of the *cinema* rather than the film.

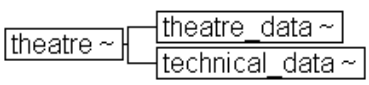


Figure 9. The `theatre` Structure

The `theatre` structure is divided into two parts: non-technical data, and technical data.

3.2.1. Theatre Data

Theatre data, shown in Figure 10, is about the theatre; its name, owner information, location, and number of seats are given.

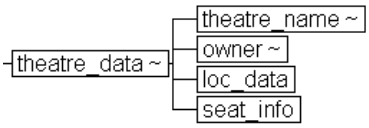


Figure 10. The `theatre_data` Structure

The theatre data is mainly used when publishing information about the cinema as such, for example, in newspaper ads, or on the Web.

3.2.2. Technical Data about the Cinema

If the `film` structure's `technical` structure is a wish list, then the `technical_data` structure in the `theatre` element is a response, stating [this is what we can do]. If a `show` structure's *action* refers to the `film` structure describing a film's technical properties, but the cinema cannot handle the request, an override is done (see Section 3.3), either with a reference to the technical data described here, or by creating a manual override.

> **Note**
> The `technical_data` structure is also available in the `show` structure (see Section 3.3) that is used to describe the actual show.

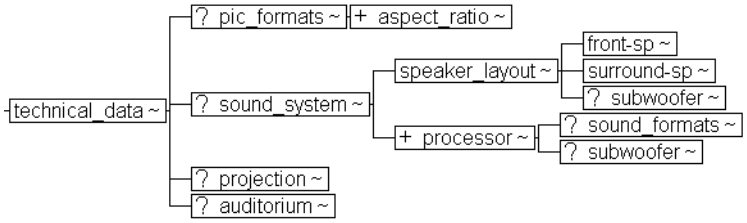


Figure 11. The `technical_data` Structure

The available aspect ratios (that is, the physical screen formats) and the sound system(s) are listed here. Cinema sound systems are specialised sound processors that can handle a variety of sound formats. Most common are the Dolby Laboratories sound processors, such as the CP650.

Note

The `pic_formats` structure lists the picture formats (aspect ratios) that apply to the screen's masking. The cinema's projectors also use this structure, but in that context, they list the *projector's* available picture formats. Ideally these two contexts should match each other, but it is entirely possible for a cinema to be able to project a film in a certain picture format and aspect ratio without being able to mask the screen accordingly.

The sound system description also contains information about the speaker layout, from the number of screen speakers to how many surround speakers exist and in what configurations, as well as whether or not the cinema is equipped with a subwoofer to handle low-frequency sound. The subwoofer information includes information about the cut frequency, sound level, and use; these settings can vary, depending on the sound curve (their use is mainly in the `show` structure; see Section 3.3).

The `technical_data` structure also contains information about the projection equipment and the auditorium's technical setup (lighting, doors, etc). These are described below.

The technical descriptions are referenced from the `show` structure; the references are used as instructions to the cinema.

3.2.2.1. Projection

The `projection` structure, shown in Figure 12, describes the projection equipment present in a projection booth. If a multiplex is described in an FML document instance, then it will contain several `theatre` structures, since auditoriums are likely to have different properties.

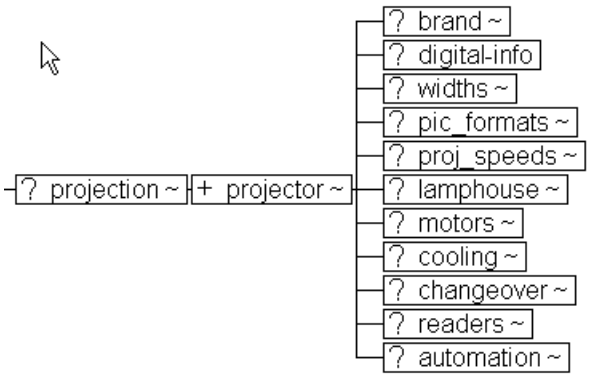


Figure 12. The `projection` Structure

The projection equipment is described per projector:

Brand — The projector brand is simply information about the model and make of the projector. Ideally the manufac-

turer should supply this information, even though the setup might be customised once installed.

Digital Information — *Digital information*, when used, identifies the type of digital projector used. Most digital cinema projectors are of type DLP while cheaper systems may use LCD or even CRT technology. Custom types also exist. The resolution(s) that the projector supports are also given.

Widths — *Widths* is information about the film gauges the projector can handle. Machines capable of projecting both 35 and 70mm film are fairly common. The *Zeiss Favorit 70* pair of projectors at the Draken cinema, for example, handles both.

Picture formats — *Picture formats* control the *projector's* available picture formats and aspect ratios. This includes controlling the different lenses used for different aspect ratios on the film strip, and the physical masking of the frames in the film gate.

> **Note**
> The `pic_formats` structure is also used in the cinema's technical data section and controls the *screen's physical aspect ratios*. If the image is to be projected correctly, these two contexts usually have to match.

Projection speeds — Many projectors can run at several speeds, or can be customised to do so; thus the available speeds should be given. These settings may be used to change the projection speed during the show.

Lamphouse — The lamphouse information includes information about the light bulb's (often Xenon) maximum and recommended current, as well as the maximum and set current of the rectifier(s) feeding the bulb. Note that some digital projectors include more than one light bulb. There's also information about the cooling of the lamphouse.

Motors — The `motors` structure contains information about the projector's motor(s).

Cooling — The `cooling` structure contains information about how the film gate is cooled.

Changeover — The `changeover` structure indicates the design of the change-over shutter and its state (during show).

Automation — The `automation` structure contains information about the type of automation used. If FML is used, the type of event notification must be noted. FML will handle both pulse-driven events (by placing aluminum tape on the print), and timestamps. If timestamps are used, the show may have to be [recorded manually] the first time, unless the film distribution company has supplied the relevant time intervals in the `screening` structure. In that case, the FML implementation will insert the relevant values into the `show` instance.

Programming shows is discussed in Section 3.3.

All of the above elements have attributes that control on/off-settings for the components they describe. They are used when referencing them from the `show` structure.

3.2.2.2. Auditorium

The auditorium's technical layout is described using the `auditorium` structure, shown in Figure 13.

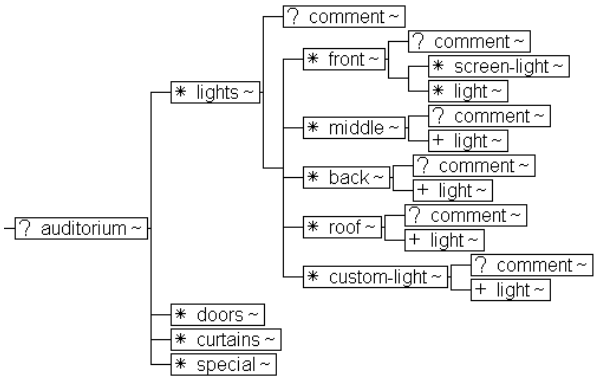


Figure 13. The `auditorium` Structure

The `auditorium` structure is divided into four basic parts:

Lights — The `lights` structure describes the auditorium lighting, divided into five areas (front, middle, back, roof, and custom). Each area contains a number of individually identifiable `light` elements, that thus may have separate properties. The most important of these properties are the time intervals it will take to dim them, and the values that will be used when controlling them during the show. When processed in the

`show` structure (see Section 3.3), these values will decide just how dark the auditorium will be during commercials, the main feature, and the end credits.

Doors — Doors leading to and from the auditoriums are often controlled by cinema automation. The doors should, for example, automatically close when the main feature begins.

Curtains — If the cinema uses curtains, the `curtains` structure is used to describe them (and, in the `show`, structure, control them).

Special — Any special features of the auditorium can be described using the `special` structure. Of course, any such feature will require customizing the FML implementation at the cinema.

The `auditorium` structure's contents are referenced from the `show` structure. The references function as instructions to the cinema, and therefore, every element described above includes attributes controlling on/off and, where applicable, intensity levels.

Note
Each main structure, as well as their contents, is allowed multiple times so that the auditorium's different lighting states can be adequately described. Any number of states is possible and each state, whether a single light or a group of lights, can be referenced from the `show` structure.

3.3. The Show Structure

The show is described by a series of *actions*, as shown in Figure 14.

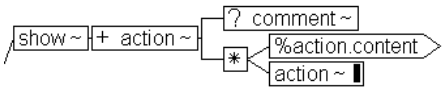


Figure 14. The `show` Structure

Actions are described in `action` elements. The `action` element's main use is as a link element to relevant instructions but it can also include instructions directly and group other `action` elements when a number of actions are logically part of the same event in the show.

Note
The FML processor might also use the recursive `action` structure to reorganise actions so that processing can be minimised.

The link from the `action` element is made to the relevant structure in either the `film` (for example, to fetch the correct picture format) or the `theatre` (for example, to dim auditorium lights) structures. Normally, no further information is required. The FML processor will, if an `action` element has been linked to an instruction in the `film` or `theatre` elements, read the target instruction and act accordingly. However, if an override is required, most of the instructions in `technical_data` and `film` can be used directly in an `action` to describe what should happen instead.

Every action is timed for the show, done either by supplying the events from the film using aluminum tape, setting timestamps or offset values (from start, or any other convenient [base time definition]), or both. A convenient unit for offsets is *frames* rather than seconds, minutes, or hours, since using frames as units will make the information independent from variations in projector speeds. The `action` element and the `film` structure's relevant elements contain a number of attributes to this end.

The `action` element uses Simple XLinks (see http://www.w3.org/TR/xlink for more information) as the link mechanism. Simple XLinks are very easy to implement, and provide a standardised way of linking that is already supported by a number of tools useful when developing FML.

Note
The `show` structure is normalised during processing so no links have to be traversed when the show is running.

3.4. The Mechanical Piano Analogy

The FML process model is fairly straight-forward but allow me to use the mechanical piano analogy again: the `show` structure is the roll and the `action` elements are the holes that tell which notes to play. Unless overrides are made, the actions are linked to corresponding structures in `film` and `theatre` structures, forming instructions on the exact [note] to be played. That [note] can be changed by creating an override within the `action` element (by using the appropriate instruction directly).

4. FML Implementation

4.1. FML Development

FML was originally to use the XFO (eXstensible Filter Objects) development framework for the creation of FML software. XFO describes how to modularise software development by splitting complicated tasks (called *filters*) into chains of simpler ones (called *operators*). Any software can be used as XFO operators by surrounding them in thin wrappers.

Note
These days, it is likely that we'll use an XProc-based approach instead. XProc does everything that XFO does, and has the added advantage of being a recognised standard.

The FML user interface will use graphical objects representing the FML DOM objects since that approach. will make it easier to manipulate `show` structures when linking to other structures or when creating overrides. A drag & drop interface, combined with forms for entering offsets, timestamps and a film's or cinema's technical information, is the preferred approach as most projectionists will probably prefer to spend as little time as possible programming shows; a traditional XML editor is therefore a bad idea.

Note
Much of the preprocessing to create a show can be automated if the cinema already has defined FML templates for its capabilities and basic shows, and if the film distributor delivers relevant film data in FML format. Programming a show in that case is a simple matter of a few XSLT transformations, perhaps with a few manual overrides.

4.2. Creating FML Document Instances

Every auditorium should start by creating an FML document instance with auditorium-specific information in the `theatre` structure, from picture and sound format information to the description of lights in the auditorium. A standard template for a generic show should also be made so that the process of having to create a complete show every time a new feature film arrives can be avoided. The template should preferably avoid using timestamps or offsets because they can be supplied by the film distributor. These both documents will then serve as the [this is what we can do] statement.

Film distributors can considerably ease the process by creating FML document instances using the `film` structure. The distributor can also create an FML instance for the show itself as they should have access to all the information about film lengths, reel lengths, offsets, and so on. If an FML document instance is supplied for trailers and commercials as well as the main feature, the programming still left to the projectionist would be to link to the appropriate actions in these FML documents.

Note
FML also allows timestamps and offsets to be given in the `screening` structure, in which case these values can be included in the `show` using simple transformations.

The FML document instances from the film distributor serve as the [this is what should be done] statement. Obviously, if film distributors cannot be made to create these documents, they will have to be created in the projection booth.

4.3. Moving Films to Other Auditoriums

When films are moved from larger auditoriums to smaller ones in a multiplex—something that happens after a few weeks, when audiences start to fail and new films arrive—FML will minimise or eliminate the required reprogramming of the show in a new auditorium. Today, most automation systems require reprogramming; aluminum tapes have to be removed, moved, or new ones have to be inserted, simply because every auditorium is unique. With FML, because every auditorium already has its own FML document instance describing the auditorium's properties, and because a generic `show` template will have been created for that auditorium, a simple, easy-to-automate, redirection of the links in the `show` structure from the previous auditorium will be sufficient.

5. FML Hardware

If FML's to be implemented at a real cinema, the FML processor will require some hardware, and I'd better at least mention some of the basic ideas and principles.

5.1. Multiplexes Already Using Automation

Today's multiplexes (that in all likelihood already use some kind of automation) use electronically controllable hardware, from the projector(s) to dimmers. Most of these will require a short pulse to perform a function, for example, starting a projector, switching a sound format, or dimming the lights. The exact nature of this pulse, from its length and voltage required to exactly what it controls, varies from device to device. Therefore, it is not possible to create a complete [FML control unit] and implement it everywhere.

What is possible, however, is to construct the electronics that interpret signals from a serial (for example, USB) port of the computer, to which the FML software will direct the actions from the `show` structure. The signals from the USB port will at each auditorium be directed to different hardware, depending on the action; thus, an FML interpreter, basically a box that features a number of pins to be connected to the cinema's various hardware, can be built. These outputs may then have to be amplified or transformed in some way, depending on the hardware.

Some multiplexes use serial channels to direct automation signals to the various devices (basically a token ring-like network). Therefore, it may be easier to change the nature of the serial output from the computer instead.

Digital cinemas (with hard disks containing the feature film) are far easier to automate because in most cases they already include standardised USB (or Firewire) ports and extensive remote control capabilities. Standard interfaces for handling every aspect of the digital projector already exist.

5.2. Single-screen and Manual Cinemas

At single-screen cinemas without automation, as well as other [manual] cinemas, the hardware must be built from scratch. Older projection equipment may well be unsuitable for electronic control as such, and may have to be modified. Otherwise, the principle is the same as with multiplexes—an FML control unit is still required.

6. In Closing

The FML DTD is in version 1.5 at the time of this writing and a first implementation of the software is being written.The conservativeness of the cinema industry is in the way (apart from the fact that none of the people I speak to knows, or even wants to know, what XML is), however, so the work is progressing very slowly.

6.1. FML Advantages

Why should the cinema industry pay for FML automation?

- Because the film distributors can make sure that their instructions on how a film should be screened are followed, or at least put into the programmed show. It's very difficult to stop projectionists from doing overrides but since they usually don't have to, why should they?
- Because the film distributors can distribute trailers, images, and, generally speaking, any kind of PR material along with the FML package.
- Because it will *standardise* the programming of the show, and make it easier, and make it more fun. Even the most modern multiplexes out there still handle the programming of a show more or less manually, with a lot of tweaking when a print is moved from one auditorium to another.
- Because it will greatly simplify and automate the boring task of having to reprogram an old feature just because it's moved to a smaller auditorium.

What more reasons do you need?

6.2. The Future

FML can easily be extended to include the box-office or other parts of the cinema. I can see a number of benefits with doing so. One of the more bizarre is probably that if I know exactly how many people that are attending a show and which seats

are taken, I can use that information to calibrate the volume levels and equalizer settings (more people means changed acoustics and changed levels) accordingly.

A *diagnostics* structure should be developed to assist the projectionist in fault-tracing. Most projectors in use are rather old, however, so the interactivity (for example, when comparing to, say, the automotive industry) would be very limited. Modern digital projectors and sound processors are a different matter, however; basically they are computers and as such, possible to customise.

Some cinema processors include subtitling functionality to handle foreign-language subtitles electronically, using a digital projection in addition to the analog 35mm projectors. Such equipment is not handled explicitly by FML although provisions exist to customise an FML implementation accordingly. Subtitling functionality should probably be included in FML 2.0.

Digital cinemas, where [films] are electronic images fetched from hard disk drives instead of from a 35mm print and then fed to digital projectors, are part of FML 1.5 but their many variants, 3D cinema in particular, are not explicitly handled. There's nothing to say that these cannot be implemented using version 1.5—the show will still have to be programmed and coordinated with events in the auditorium and FML is just as convenient for handling electronic images—but currently most digital formats would require custom definitions.

I'd like to extend my warmest thanks to my friend and colleague Henrik Mårtensson, expert programmer and information analyst as well as the creator of XFO, for great ideas and infinite patience in listening to my rants. I'm forever in his dept for his merciless pursue of any errors in my ways concerning the FML process model and the idea of the projection booth as an object-oriented model. Also, thanks to my dear wife and my both children. Without them, I would probably still work in a projection booth. Without XML.

A Time Machine for XML: PUL Composition

Ghislain Fourny
ETH Zurich
<gfourny@inf.ethz.ch>

Daniela Florescu
Oracle
<dana.florescu@oracle.com>

Donald Kossmann
ETH Zurich
<donaldk@inf.ethz.ch>

Markos Zacharioudakis
Oracle
<markos_za@yahoo.com>

Abstract

As storage - main memory as well as disk - becomes cheaper, the amount of available information is increasing and it is a challenge to organize it. Our broader aim is to provide a unified framework for efficiently versioning and querying data, documents, as well as any kind of semi-structured information between data and documents, which can be stored as XML. In order to query this information, we started with the XQuery programming language, and extended its data model, its syntax and its processing model to make it seamlessly time-aware. More specifically, to do this, we made the assumption that the changes made by an XQuery program can always be expressed as a PUL. This is not obvious with an XQuery Scripting Extension program, as it can apply several, possibly interdependent, PULs. The contribution of this paper is to introduce a new operation on pending update lists, called PUL composition, which allows to summarize the changes made by an entire XQuery Scripting program with a single PUL instead of a sequence of PULs.

Keywords: XML, Versioning, XQuery, CVS, SVN

1. Background

A full paper describing the versioning system is available as a technical report [1]. In this section, we summarize the most important concepts.

XML nodes are organized in trees. During the execution of an XQuery program, or several XQuery programs, trees evolve.

In our versioning system, the evolution of a tree is modeled as a tree timeline. A tree timeline is a sequence of trees, which are actually "the same tree" at successive moments in time, as represented on Figure 1 (a).

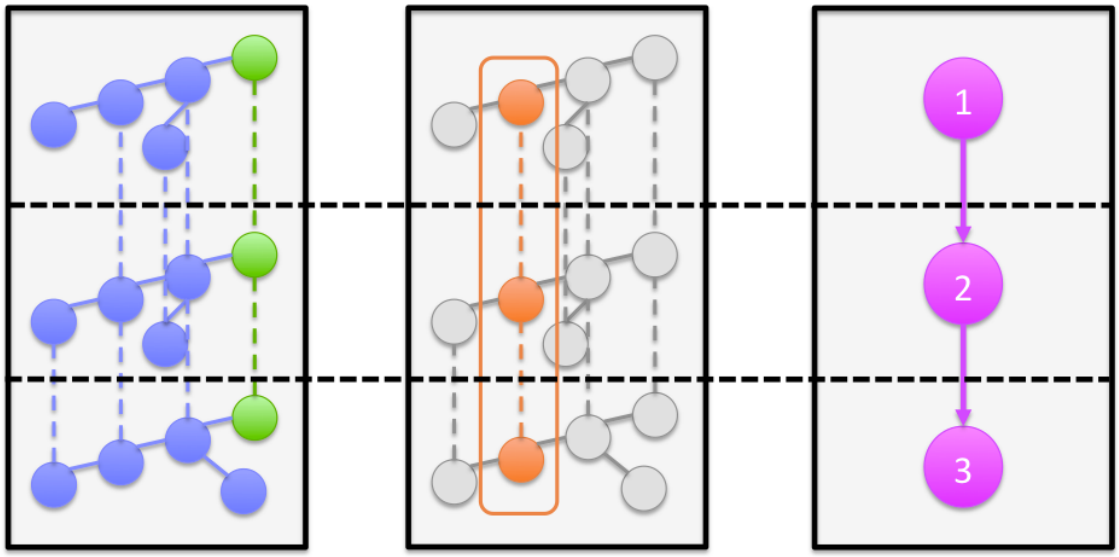


Figure 1. Tree timelines, node timelines and versions

Exactly like trees, nodes are organized in node timelines (b). A node timeline is a sequence of nodes, which are actually "the same node" at successive moments in time (same reference URI, distinct identites). Within a node (tree) timeline, a node (tree) is uniquely identified by a version (c).

Deltas between several trees in a timeline are expressed as single PULs, as can be seen on Figure 2. It is this very assumption that we aim to justify.

2. PUL Composition

While in the Update Facility, it is obvious that changes made by the program can be expressed as a PUL, this is not straightforward for the Scripting Extension. In the Scripting extension, several PULs might be applied locally, and these PULs might be interdependent (e.g., the target of a PUL might be in the contents previously inserted by another PUL), so that they cannot just be merged (with *upd:mergeUpdates*, described in the XQUF specification). The Scripting Extension introduces a time

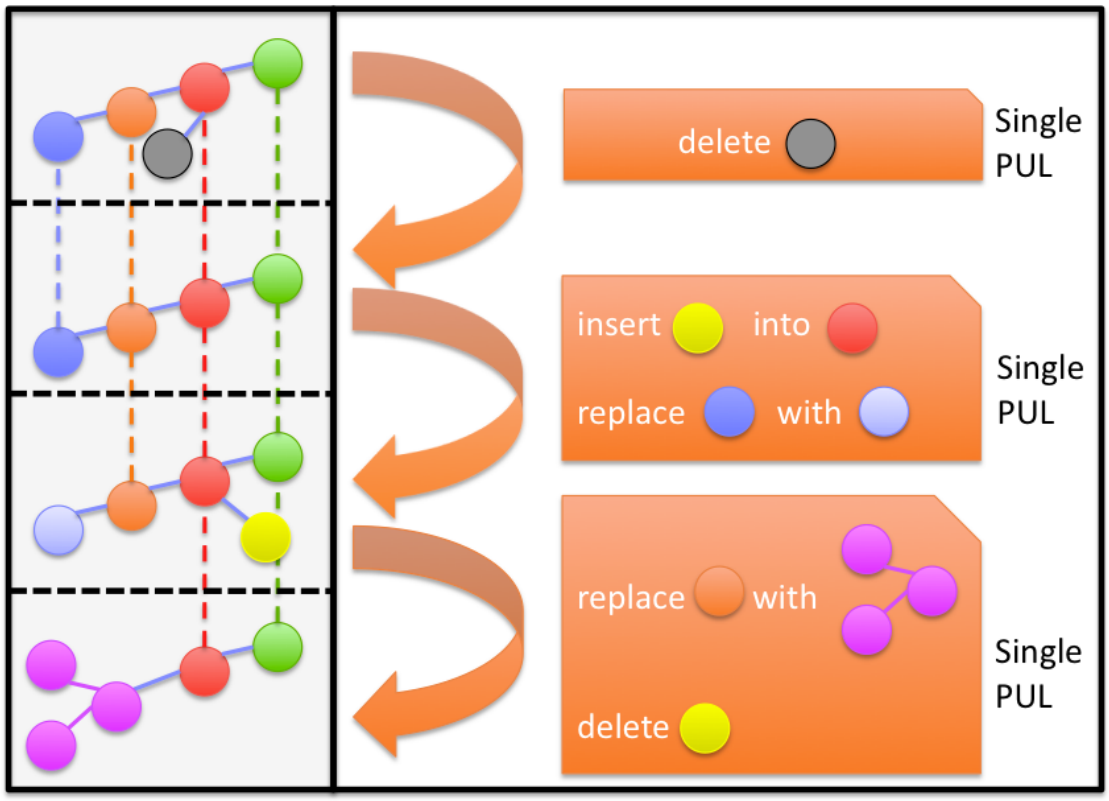


Figure 2. Deltas as single PULs

component which has to be taken into account. At first, it seems that deltas must be expressed by sequences of PULs (e.g., a and b on Figure 3).

In this section, we show how two, possibly interdependent, PULs can be composed to one single PUL (e.g., c on Figure 3). This is what allows us to express our deltas as single PULs. Note that PUL composition is different from PUL merging (*upd:mergeUpdates*), since the PULs are possibly not independent, i.e., they are not applied to the same snapshot.

We give an operational definition of PUL composition. A local PUL is maintained locally, which summarizes all local changes made since the beginning of the execution of the program, or since the last checkout. It is normalized (see Section 3), and each time a new PUL is applied by the (XQuery Scripting Extension) program, a copy of this new PUL is also composed with the local PUL (see Section 6). Note that the local PUL is not meant to be applied - rather, it is meant to be committed (checked in) to the repository in our versioning system.

The idea behind PUL composition is very simple. For each update primitive being composed with the local PUL:

- Either the target of this update primitive was already here at the beginning of the program (or at the last checkout), in which case the update primitive is accu-

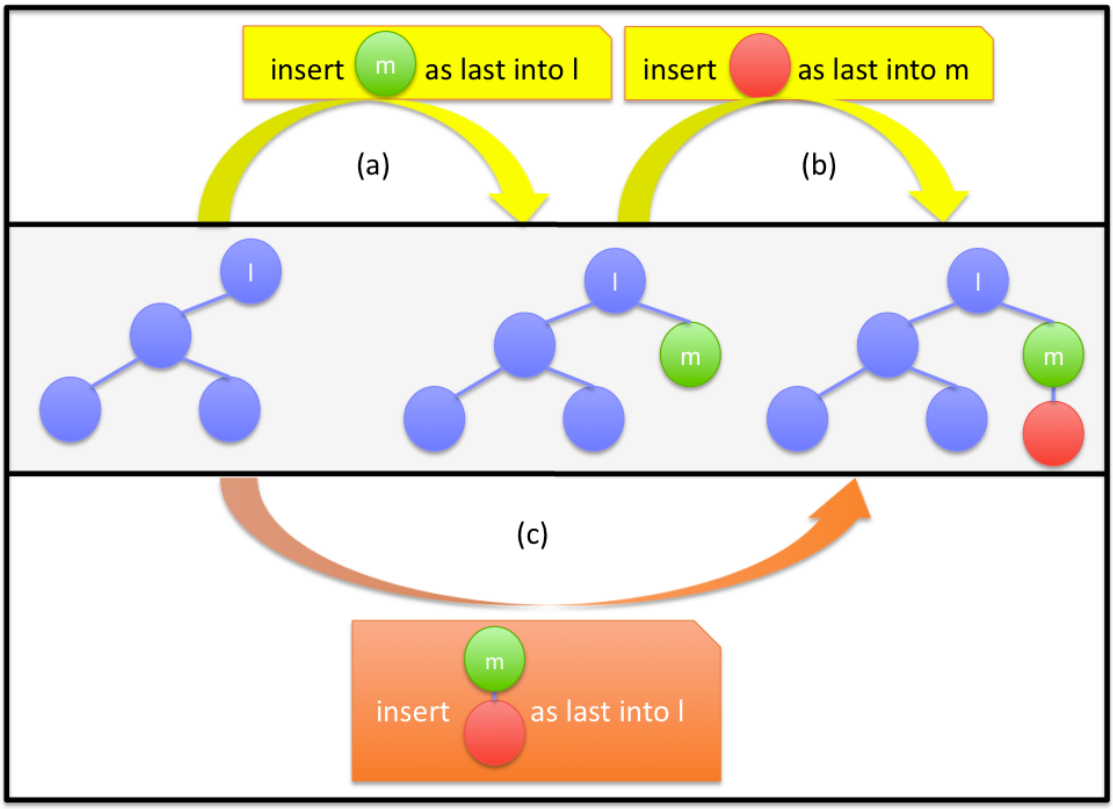


Figure 3. Two PULs (a, b) can be summarized in a single PUL (c) (Aggregation Case)

mulated against the local PUL (accumulation, shown on Figure 4 and detailed in Section 4).

- Or its target is in the contents of the local PUL, in which case the extended semantics of the update primitive is made effective (aggregation, shown on Figure 3 and detailed in Section 5)

3. The normalized local PUL

In XQuery, there is a dynamic context which contains dynamic information about the execution of a program. We extend the dynamic context with a local PUL, which contains information about all changes which have been made since a certain point in time (e.g., the beginning of the program, or say the last checkout). The local PUL can be seen as the delta between this point in time and now.

The local PUL is normalized, which means that, without loss of generality, for each target, there is at most one update primitive of each kind (*upd:insertBefore, upd:insertAfter, upd:insertIntoAsFirst, upd:insertIntoAsLast, upd:insertAttributes, upd:replaceNode, upd:replaceValue, upd:replaceElementContent, upd:delete and upd:rename* - *upd:put* is out of the scope of this paper, *upd:insertInto* deserves a special treatment

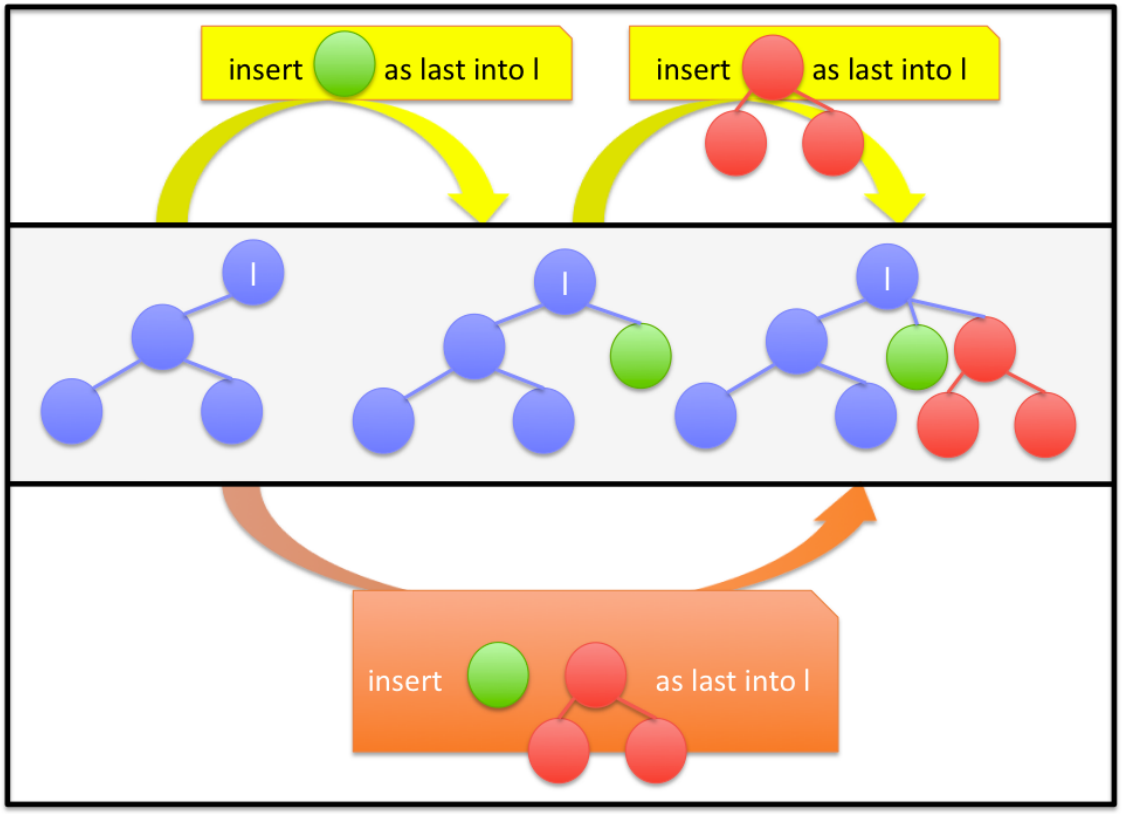


Figure 4. Two PULs (a, b) can be summarized in a single PUL (c) (Accumulation Case)

and can always be replaced with another inserting update primitive) with this target. This was already the case for replacing and renaming update primitives in the specification, and in case there are several inserting update primitives of the same kind with the same target, the XQuery Update Facility specification says that "ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent." Whenever a PUL contains several inserting update primitives of the same kind sharing the same target, this allows us to group the contents of these update primitives inside a single update primitive of that kind, in an implementation-dependent way, without altering the semantics of the PUL. The local PUL is kept normalized when aggregating or accumulating update primitives, as defined in the next sections.

Also, there are backpointers from the nodes of the local trees in the store (read by the XQuery program) to their copy in the local PUL whenever there is one (i.e., whenever they were added by the program), in order to know where they came from (see Figure 5). These backpointers are followed when the PUL to be applied is copied to a new PUL which is to be composed with the local PUL (see Section 6). From now on, as on Figure 5, we take the convention that the nodes encircled in black are copies which belong to contents of the local PUL or to the contents of the

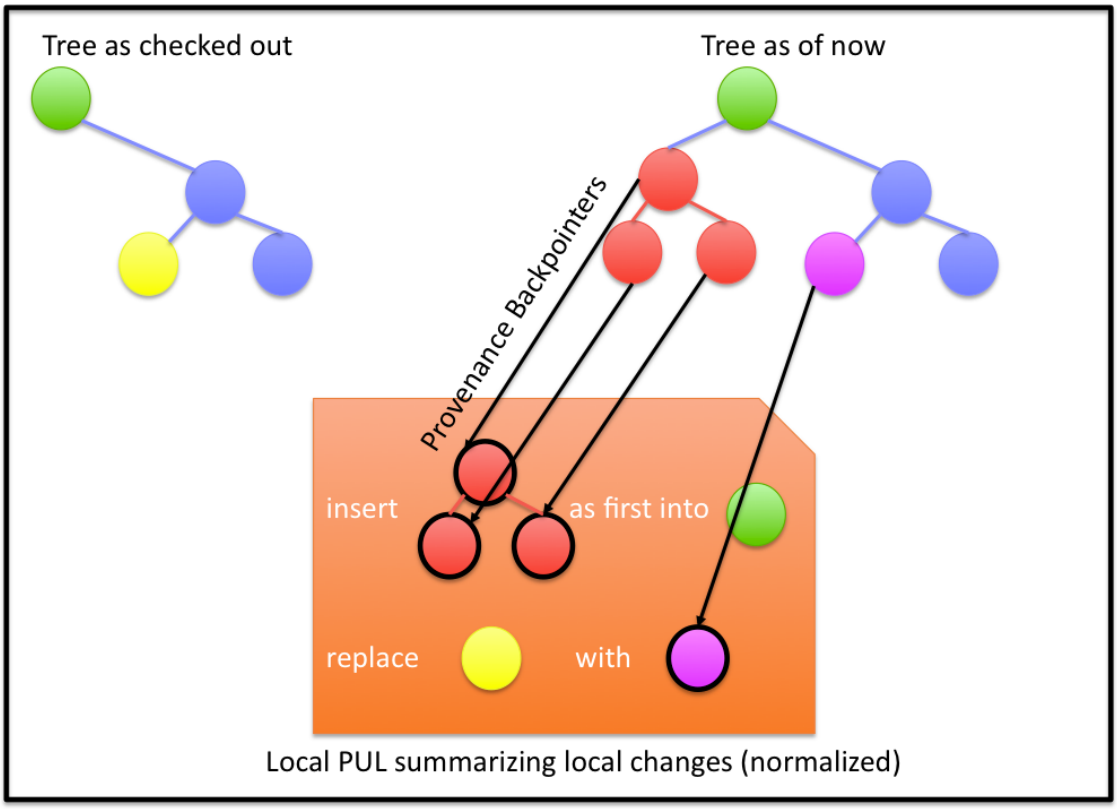


Figure 5. The local PUL: overview

update primitive being composed. The identities of these copies are distinct from those of the original nodes.

4. Accumulation

When the target of an update primitive was already here at the last checkout (i.e., it is not to be found anywhere in the contents of the local PUL - to put it simply, it is not encircled in black), then this update primitive is accumulated against the local PUL. It is mostly like merging them (upd:mergeUpdates), with the additional constraint that the local PUL remains normalized (Figure 6).

For *upd:insertBefore*, *upd:insertIntoAsLast* and *upd:insertAttributes*, the contents of the update primitive are inserted at the end of the contents of the update primitive of the same kind in the local PUL (or the update primitive is inserted in the local PUL if not available).

For *upd:insertAfter*, *upd:insertIntoAsFirst*, the contents of the update primitive are inserted at the beginning of the contents of the update primitive of the same kind in the local PUL (or the update primitive is inserted into the local PUL if not available).

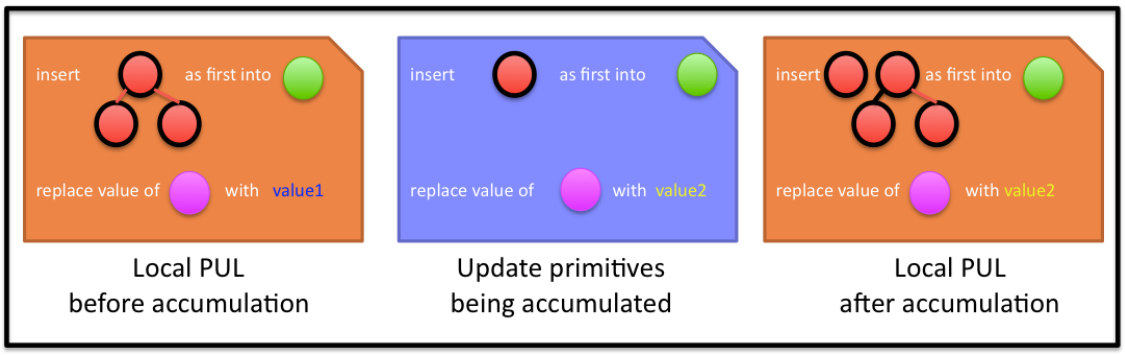


Figure 6. Accumulating against the local PUL

For *upd:delete*, *upd:replaceNode*, the update primitive is inserted into the local PUL if there is not already an update primitive with the same target and the same kind in the local PUL. Otherwise, an error is raised (this should never happen during the normal execution of an XQuery program).

For *upd:replaceValue*, *upd:replaceElementContent*, the update primitive is inserted into the local PUL if there is not already an update primitive with the same target and the same kind in the local PUL. Otherwise, the contents (or string) of the update primitive in the local PUL are replaced with the contents of the update primitive being accumulated.

For *upd:rename*, the update primitive is inserted into the local PUL if there is not already an *upd:rename* update primitive with the same target in the local PUL. Otherwise, the name mentioned in the update primitive of the local PUL is replaced with the name of the update primitive being accumulated.

5. Aggregation

When the target of an update primitive was not here at the last checkout, but was put into place by a former PUL, it means that it is to be found somewhere in the contents of the local PUL (to put it simply, it is encircled in black). In this case, the update primitive is aggregated against the local PUL (Figure 7).

The semantics of aggregation is the same as the semantics of applying a PUL, with just the following modifications for *upd:insertBefore*, *upd:insertAfter* and *upd:replaceNode*.

For these three update primitives, the specification says that the parent property of the target must be non-empty. We relax this constraint. However, if the parent property is empty, the target must be in the contents of an update primitive in the local PUL (encircled in black). This corresponds to (a) on Figure 7.

In case the parent property of the target is empty, the semantics of applying these three udpate primitives is extended as follows: for *upd:insertBefore* (*upd:insertAfter*), the content of the update primitive is inserted right before (after) the target

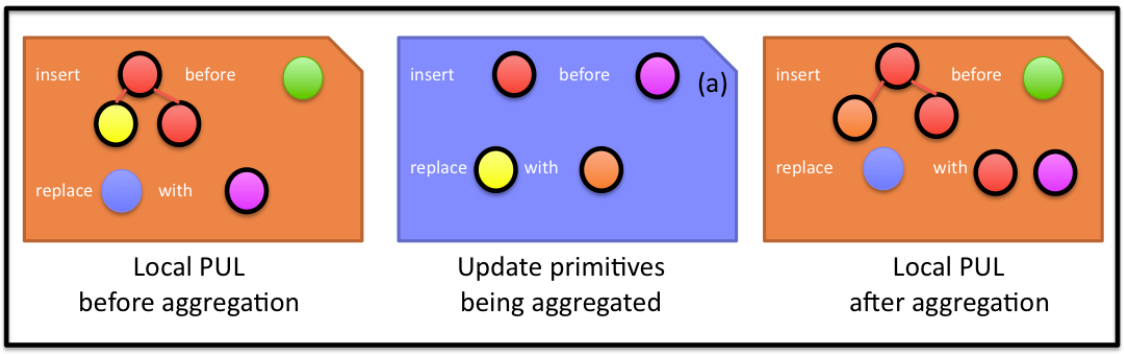


Figure 7. Aggregating against the local PUL

in the contents of the update primitive of the local PUL to which this target belongs (like on Figure 7); for *upd:replaceNode*, the content of the update primitive replaces the target of the update primitive ot the local PUL to which this target belongs.

6. Extension to Apply Expressions

Apply Expressions are introduced in the XQuery Scripting Extension specification. In an apply expression, after each operand is evaluated, the PUL it returns is applied with upd:applyUpdates.

We perform PUL composition right here. Whenever a PUL is applied, a copy of it is also composed with the local PUL. This is described on Figure 8.

For nodes which were not here at the last checkout, but were subsequently added, we introduce provenance information in order to know from which update primitive in the local PUL they come from. During the copy of the PUL, these backpointers are followed for each target, whenever possible, and new backpointers are created when copying the contents. On Figure 5, the nodes encircled in black are in the contents of the local PUL.

More precisely, we extend the semantics of Apply Expressions as follows. For each operand, before the PUL is applied:

- The PUL is copied to a new PUL (the contents are copied too), where:
  - Backpointers are put from each node in the contents of the update primitives in the original PUL to the corresponding node in the copied PUL. (a)
  - In this copied PUL, the original targets (which are never in a PUL) are replaced, whenever there is a backpointer to the local PUL, by the corresponding target in the local PUL. (b)
- The copied PUL is then composed with the local PUL

Then the original PUL is applied (with *upd:applyUpdates*, as defined in the XQUF specification).

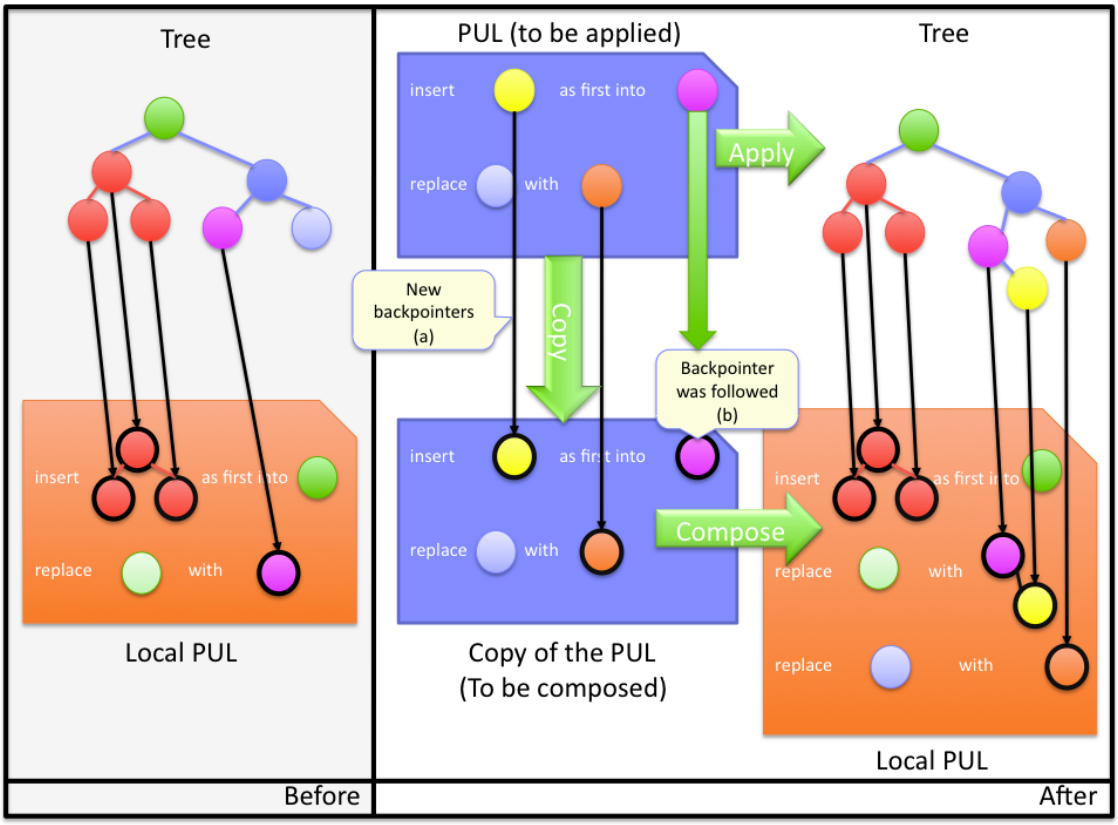


Figure 8. PUL: applying, copying and composing

Note that after this entire process, the original nodes, applied, will be in the store (not in a PUL), whereas the copied nodes, composed, will be in the content of the local PUL. This means that the backpointer always points from a node in the store to the corresponding node in the local PUL (see Figure 8).

7. Conclusion

With this new composition operator on PULs, it is possible to summarize the semantics of several successive PULs in a single one, without introducing any order between the update primitives. For our versioning framework, it means that it is legitimate to assume that the delta between two trees in a timeline can be expressed by a single PUL.

In a separate document, we came up with a formal proof that PUL composition is correct, i.e., the composed PUL has the same semantics on the XML tree than the original sequence of interdependent PULs. This document is available on request.

Bibliography

[1] G. Fourny, D. Florescu, D. Kossmann, M. Zacharioudakis: A Time Machine for XML. Technical report. https://www.dbis.ethz.ch/research/publications/timemachinexml.pdf

[2] XQuery Update Facility 1.0, W3C Candidate Recommendation, June 9th, 2009. http://www.w3.org/TR/xquery-update-10/

[3] XQuery Scripting Extension 1.0, W3C Working Draft, December 3rd, 2008. http://www.w3.org/TR/xquery-sx-10/

Posters

XQuery Update Facility in Enterprise Database Systems and in SQL/XML

Dusan Petkovic
University of Applied Sciences, Rosenheim, Germany
<petkovic@fh-rosenheim.de>

Abstract

XQuery is the standard language for querying sources with XML content. The specification of XQuery 1.0, which is released in January 2007 does not include features for updating XML documents. In absence of such a recommendation and due to user's requirements for the ability to update XML data, enterprise database systems have already implemented their proprietary solutions.

In this paper, we first compare the proposed specification for XQuery Update Facility in the standard document with already existing implementations of analogous functions in three enterprise database systems. Second, we discuss several problems concerning SQL/XML and XQuery update operations. In relation to existing implementations the paper shows that all implementations of XQuery updates in enterprise database systems are significantly different, although their semantics are similar. Also, we strongly advocate for the use of update in-place, when adding new capabilities to SQL/XML.

Keywords: XML, XQuery Update, SQL/XML

1. Introduction

XQuery Update Facility is an extension of XQuery 1.0, which specifies how existing XQuery Data Model (XDM) instances can be modified. The proposal of XQuery Update Facility is currently in the "Candidate Recommendation" status. This proposal introduces, as its main part, several expressions that can be used to insert new nodes, delete or replace the existing ones etc. According to the proposal, there are altogether four updating expressions: insert, delete, replace, rename, which correspond to a set of standard update primitives (insertion, deletion, replacement and renaming of XML nodes), and one non-updating expression: transform. (The goal of the transform expression is to apply one of the update primitives and to present the results in the same query.)

In the meantime, enterprise database vendors implemented proprietary extensions for updating XDM instances. The characteristic of all of them is that they significantly differ in their approach.

The main contribution of this paper is twofold. First, we compare the updating expressions specified in XQuery Update Facility with the implementations of analogous features in enterprise database systems (IBM DB2, Oracle and SQL Server) and evaluate the existing implementations. Second, we discuss problems in extending SQL/XML by XQuery update functions and give some ideas, how they can be resolved.

1.1. Related Work

One of the first research papers concerning XQuery Update introduced a set of updating expressions [1]. The similar ideas have been presented in papers [2] and [3]. The former addresses the question of updating relational databases through XML views using query trees, while the latter shows how updated views are propagated to the underlying relational database.

Until now, the XML XQuery working group has finished XQuery 1.0 as the official specification for the XQuery language [4]. The same group has published the XQuery Update Facility 1.0 proposal [5]. This document introduces several new expressions, which allow the modification of XML data. The perspectives on the concepts behind the XQuery language are discussed in detail in [6]. The requirements as well as different scenarios, how the syntax and semantics of XQuery Update can be defined, are given in [7]. The SQL standard does not specify SQL functions in relation to updates. It supports XQuery Update Facility by simply including a Normative Reference to the specification [8].

Oracle capabilities for updating XDM instances using XQuery are described in [9]. The comprehensive discussion, how the XQuery language can be used with Oracle is given in [10]. The implementation and the functionality of XQuery updating expressions in IBM DB2 are presented in [11]. The general way, how the XQUery language can be used with IBM DB2 is given in [12].

Corresponding SQL Server capabilities are described in [13], while XQuery retrieval and update methods for SQL Server are discussed in [14].

1.2. Sample Data

To demonstrate examples given in this paper, we use the simple table called cust_xml, with two columns of the types INT and XML, respectively. The rows of this table describe departments with employees. Table 1 shows the creation of the cust_xml table and the content of the doc column of that table after insertion of two rows.

Table 1. Sample table and data

```
create table cust_xml (id INT, doc XML);
                <customer>
                    <name>
                        <first>Sybille</first>
                        <last>Moser</last>
                    </name>
                    <address country="USA">
                      <street>First</street>
                      <city>Baltimore</city>
                      <state>Maryland</state>
                      <zipcode>21201</zipcode>
                   </address>
                   <phone type="home">410- 499-0499</phone>
                </customer>
                <customer>
                    <name>
                        <first>Matthew</first>
                        <last>Smith</last>
                    </name>
                    <address country= "Canada">
                       <street>Marshal</street>
                       <city>Vancouver</city>
                       <state>British Columbia</state>
                       <zipcode>V6E 1C1</zipcode>
                    </address>
                    <phone type="home">905-403-2222</phone>
                    <phone type="work">647-521-3333</phone>
                </customer>
```

1.3. Roadmap

The rest of the paper is organized as follows. Section 2 gives an overview of W3C Candidate Recommendation of XQuery Update Facility 1.0. The description of each expression is also given in this section. Section 3 discusses the implementation of the proposed XQuery Update Facility in IBM DB2. The Oracle's proprietary functions, which are syntactically similar to the standardized SQL/XML XQuery functions (XMLTable, XMLQuery and XMLExists) are the topic of Section 4. MS SQL Server supports a single method in relation to XQuery updates, and this method, with its clauses is described in Section 5. Section 6 summarizes the results of the previous sections. Section 7 discusses several problems concerning direct use of update expressions for SQL/XML and recommends the ways, how these problems can be solved.

2. XQuery Update Specified by XQuery Working Group

According to [4], XQuery is a read-only language, meaning that it can read an XDM instance, but can't update an existing instance. On the other hand, applications using a query language need to read and to modify data stored in XML documents.

In 2005, W3C published a first document: "Working Draft of the Requirements" for XQuery Update Facility. This document specifies, among other things, the requirements for update functions for XQuery.

Now, XQuery Update Facility exists as W3C "Candidate Recommendation" [5]. Because of the new (update) paradigm, XQuery Update Facility contains extensions to the following topics:

- processing model
- prolog and static context
- new expressions and extensions

The XQuery Update processing model supports expressions that return a value ("simple expressions") as well as expressions that return a pending update list. A pending update list is a list of update primitives, which are applied after the query is processed. Update primitives represent node state changes that are checked for conflicts and applied after that, if no conflicts appear. The scope, within which update primitives are evaluated with respect to a fixed XDM instance is called snapshot. (XQuery Update Facility defines an entire query as one snapshot.)

The prolog of XDM instance is extended with so called revalidation declaration. The revalidation declaration sets the revalidation mode ("strict", "lax" or "skip"), and it controls the behavior of the revalidate operation. (Revalidate operation specifies the "recovery" type for an updated document.)

In relation to new expressions, XQuery Update Facility classifies all of them in updating and non-updating expressions. There are four updating and one non-updating expression:

- insert
- delete
- replace
- rename
- transform (this one is non-updating)

The transform expression is a non-updating expression, because its goal is to apply one of the updating expressions listed above and to present the results in the same query.

The following subsections describe briefly these expressions. The detailed description of their syntax and semantics can be found in [5]. (The examples for all four updating expressions are given in Section 3.)

2.1. Insert

An insert expression inserts copies of zero or more nodes into a specified position of a target node. There are two forms of this expression: The first one (using the into clause) inserts children nodes at the specified position (first, i.e. last). The second one uses the after (before) clause to insert sibling nodes after (before) the specified source node.

2.2. Delete

Delete expression deletes zero or more nodes from an XDM instance. The effect of the delete expression is that the deleted node(s) are first appended to the pending update list and, at the end of the query execution, removed. (This is called a "snapshot semantics".)

2.3. Replace

Replace expression is an expression with two different forms. If the node clause is specified, the corresponding expression replaces a single node with a new sequence of zero or more nodes. The replacement nodes occupy the position that was occupied by the node that was replaced. The new nodes have the different node identity than the replaced node.

If the value of node clause is specified, the value of the target node is replaced by the text node, preserving node identity. The target node can be of any node kind, and is converted into a string by atomizing the sequence and concatenating the resulting items.

2.4. Rename

One of the specific requirements in the "Working Draft of the Requirements" document is the capability for modifying properties of a node. Generally, it should be possible to change several properties (the name of the node, the type of the node etc.). Xquery Update Facility specifies the rename expression, which replaces the name of a node with a new QName. The effects of the rename expression are applied only to the target node. In other words, descendants and attributes of the target node are not affected.

2.5. Transform

The transform expression is used to create updated copies of existing nodes of an XDM instance. As we already stated, this expression is n't an updating expression because it copies nodes, and than applies an updating expression to these copies. The copy clause assigns the input document to a variable, which holds the copy of

it. The modify clause applies one or more updating operations to that variable, and finally, the return clause produces the result of the entire expression. Example 1, which deletes the zipcode node, if the customer has a Canadian address, clarifies the use of these three clauses.

Example 1.

```
for $c in //address[@country=="Canada"]
                return
                copy $cc := $c
                modify delete node $cc/zipcode
                return $cc
```

3. XQuery Update Functions Supported by IBM DB2

Of all enterprise database systems discussed in this paper, IBM DB2 is the only system, which supports updating expressions according to W3C XQuery Update Facility. IBM DB2 uses the convenient syntax of the SQL UPDATE statement and its SET clause to assign a new value to the corresponding XML column using the standardized XMLQUERY function. (XMLQUERY is an SQL/XML function, which takes as parameter an XQuery expression, as well as arguments for XQuery variables, and returns an instance of XML data type.) The XQuery expression is a transform expression that starts with the optional transform keyword, followed by copy, modify and return clauses (see section 2.5).

3.1. Insert

All forms of the insert expression discussed in Section 2 are supported by IBM DB2. The following example uses the insert expression with the last into option to insert the new phone element node as the last entry of all phones in the XML document with id=2.

Example 2.

```
UPDATE cust_xml
        SET doc=XMLQUERY('transform copy $new :=
        $DOC
        modify do insert
           <phone type="cell">674-521-4444</phone>
        as last into $new/customer
        return $new')
        WHERE id= 2;
```

The following example shows the use of the before clause to insert the new phone element as the first entry of all the phones for the XML document with id=2.

Example 3.

```
UPDATE cust_xml
        SET doc=XMLQUERY('copy $new := $DOC
        modify do insert
         <phone type="cell">674-521-4444</phone>
        before $new/customer/phone[1]
        return $new')
        WHERE id = 2;
```

3.2. Delete

If the modify clause of the transform expression contains the delete option, zero or more nodes are deleted. (Note that the delete operation is the only operation, which can act on a sequence of nodes.)

The following example deletes all customer phones in the XML document with id= 1.

Example 4.

```
UPDATE cust_xml
        SET doc = XMLQUERY('copy $new := $DOC
        modify do delete $new/customer/phone
        return $new')
        WHERE id = 1;
```

3.3. Replace

According to the specification in XQuery Update Facility, IBM DB2 supports two forms of the replace expression: replace and replace value of. The former expression replaces the whole node, while the latter expression replaces the content of a node. Example 5 shows the use of the replace expression to replace a value of a node.

Example 5.

```
UPDATE cust_xml
        SET doc = XMLQUERY('copy $new := $DOC
        modify do replace value of $new/customer/phone/@type
        with "cell"
        return $new')
        WHERE id = 1;
```

3.4. Rename

The rename expression changes the name of an element, attribute or PI node. (Text nodes and comments cannot be renamed, because they do not have a name.) The

rename clause in the following example changes the name of the zipcode element to zip.

Example 6.

```
UPDATE cust_xml
            SET doc = XMLQUERY('copy $new := $DOC
            modify do rename
            $new/customer/address/zipcode as "zip"
            return $new');
```

4. XQuery Update Functions Supported By Oracle

In contrast to IBM DB2, which supports the standardized transform expression, Oracle implemented a set of XML update functions, which look like the existing XQuery functions (XMLQuery, XMLTable and XMLExists), specified in the SQL/XML standard. Oracle's XQuery Update functions have two constraints: They can use only XPath expressions as parameters and the capability of the renaming a node isn't supported.

4.1. Insert

Oracle supports several forms of the insert function, which allow the insertion of copies of zero or more nodes into a specified position of a target node. These functions are analogous to the different forms of the insert expression of XQuery Update Facility. The following functions exist:

- insertChildXML
- insertChildXMLbefore
- insertChildXMLafter
- appendChildXML
- insertXMLbefore
- insertXMLafter

The insertChildXML function inserts new children under parent XML elements. The insertChildXMLbefore and insertChildXMLafter functions insert one or more elements as children of the specified target parent elements. The insertion occurs respectively immediately before (after) the child element, which is specified as a parameter of the function. The appendChildXML function inserts one or more nodes of any kind as the last children of the specified element node.

The insertXMLbefore and insertXMLafter functions insert one or more nodes of any kind immediately before (after) a target node. (The target node can be any node, except an attribute node.)

All insert functions discussed above return a copy of the input XDM instance. The copy can be subsequently used with the SQL UPDATE statement to modify data in the database, as the following two examples show.

Example 7.

```
UPDATE cust_xml
            SET doc = insertChildXML(doc,
            '/customer', 'phone',
              XMLType('<phone type="cell">674-521-4444</phone>'))
            WHERE id = 2;
```

The above example inserts the new phone element in the XML document with id=2. (If the parent node already has children, as in this case, the new element becomes the last child of that parent node.)

Example 8.

```
UPDATE cust_xml
            SET doc=insertXMLbefore(doc,
            '/customer/phone', 'phone',
              XMLType('<phone type="work">674-521-9999</phone>'))
                WHERE id = 1;
```

The UPDATE statement in the example above inserts the new phone element immediately before the element, which is specified as the parameter of the XMLType function. (The XMLType function is Oracle's proprietary function that stores the XML document, specified as its parameter as the CLOB value, and makes it available to a SQL application.)

4.2. Delete

The deleteXML function deletes XML nodes of any kind. The result of the function is a copy of the input XDM instance. After that the UPDATE statement can be used to store that copy in the database.

Example 9.

```
UPDATE cust_xml
             SET doc = deleteXML(doc,'/customer/phone')
             WHERE id = 2;
```

The UPDATE statement in the example above uses the deleteXML function to delete all existing phone nodes in the XML document with id=2.

Note that all Oracle's XQuery Update functions can be "embedded" in the SQL SELECT statement, too. In this case the corresponding update function creates the

transformed copy of the XDM instance, leaving the persisted XML document unchanged, as the following example shows.

Example 10.

```
SELECT deleteXML(doc, '/customer/phone')
            FROM cust_xml
            WHERE id = 2;
```

4.3. Replace

The updateXML function replaces XML nodes of any kind. This function takes as arguments an input XDM instance and an XPath value pair, and returns a new XDM instance with either nodes or values of nodes changed.

Example 11.

```
UPDATE cust_xml
            SET doc = updateXML(doc,
            '/customer/phone[1] ',
             XMLType('<phone type="work">674-521-9999</phone>'))
                WHERE id = 2;
```

The UPDATE statement in Example 11 uses the updateXML function to make the persistent replacement of the first phone element in the XML document with id=2. This form of the updateXML function corresponds to the W3C's replace expression with the node clause.

Example 12.

```
UPDATE cust_xml
            SET doc=updateXML(doc, '/customer/phone[1]/@type','work')
            WHERE id = 1;
```

The UPDATE statement in Example 12 uses the updateXML function to replace the existing value of the type attribute of the first phone element with the new value. This form of the updateXML function corresponds to the replace expression with the value of node clause.

5. XQuery Functions Supported by SQL Server

MS SQL Server provides proprietary extensions for XQuery updates. Unlike IBM and Oracle, Microsoft's implementation of this feature depend neither on XQuery Update Facility 1.0 nor on SQL/XML capabilities. They are provided through the method called modify(), which operates on instances of the XML data type.

The modify() method is invoked through the SET clause of the UPDATE statement. The parameter of the modify() method is a character string, which specifies the form of the update operation. The first part of the string defines the operation (insert, delete or replace), while the second part specifies the XQuery expression that identifies the nodes. SQL Server doesn't support the rename operation.

5.1. Insert

The insert clause of the modify() method is used to insert one or more nodes in the XML document, given as the method's parameter. This clause accepts an XQuery expression that identifies the nodes to be inserted, and another XQuery expression that specifies the reference node.

Example 13.

```
UPDATE cust_xml
            SET  doc.modify ('insert
             <phone type="cell">111-2223333</phone>
            after (/customer/phone)[1]')
            WHERE id = 1
```

The UPDATE statement in the example above inserts a new phone element in the XML document with id=1.

5.2. Delete

The modify() method with the delete clause can be used to delete one or more nodes from an XML document. (The semantics of the delete clause is analogous to the semantics of the delete expression specified in XQuery Update Facility).

Example 14.

```
UPDATE cust_xml
            SET  doc.modify
            ('delete  /customer/phone')
            WHERE id = 1
```

The UPDATE statement in Example 14 deletes all existing phone elements in the XML document with id=1.

5.3. Replace

SQL Server supports only the replacement of a value of the specified node. The value of option with the replace clause is used to replace the value of an existing node that is specified in the XQuery expression.

Example 15.

```
UPDATE cust_xml
            SET  doc.modify
            ('replace value of
            (/customer/phone/@type)[1] with
            "cell"')
            WHERE id = 2;
```

The UPDATE statement in the example above modifies the value of the first phone element.

6. Comparison of Implementations

In this section we compare existing implementations of XQuery Update Facility in enterprise database systems.

IBM DB2 supports the updates on XDM instances according to according to W3C XQuery Update Facility. The Oracle's approach is different: the company supports a set of XML update functions, similar to the existing SQL/XML functions and constructs, such as XMLQuery() and XMLTable. That way, the update of XML documents at the node level can be done declaratively.

Microsoft SQL Server has provided the capability of XML node level update with the single modify() method, in which the updating expressions of XQuery Update Facility are embedded. The expression (insert, delete or replace) is specified as the first parameter of the modify() method.

7. Conclusions about Adding New Capabilities to SQL/XML

At this moment, SQL/XML supports XQuery Update facility only by including a Normative Reference to the existing document. To add capabilities for SQL/XML to make direct use of the InsertExpr, DeleteExpr, ReplaceExpr and RenameExpr requires a solution of at least the following problems:

- How existing XML values can be renamed?
- How existing XML values can be modified?
- What the concept of node identity means?

The first problem that arises is how an existing XML value can be renamed. (We don't discuss the question about creating new XML values or deleting existing ones with respect to SQL/XML's use of XQuery Update Facility, because it can be solved easily.)

From the SQL's viewpoint, there are two possible solutions to solve this problem: either to change the name of the node in the XML document or to replace the entire document with a new one, which has a different name. The replacing of the entire

document with a new one isn't efficient, because database systems may use physical structures, such as indexes that have to be maintained each time, when such a replacement of the entire document happens. (Another reason for the inapplicability of the latter solution is that the "old" XML value has been validated and the validation and the accompanying annotations will be lost in that case.)

The same discussion, which is applied to the renaming operation, is valid for the replace operation, too. Updates usually leave the most part of an XML document unchanged. To simulate this, one has to specify explicitly how the transformation preserves the unchanged parts of the input document. This is inefficient for the same reasons discussed in the last paragraph. Also, the more unchanged parts exist, the more transformation specifications are necessary (see [15-17]).

For the reasons mentioned above, we strongly recommend that the direct use of ReplaceExpr in SQL/MLX must be in place". (The same is true for RenameExpr.)

The question of the object identity for SQL applications is already solved [18]. On the other side, the direct use of the updating expressions (that is, not in the context of the transform expression) requires understanding what the concept of node identity means in the context of an SQL environment. For this question, there is no agreement today. (As can be seen from [19], the related discussion concerning the question of whether or not XQuery Update Facility guarantees to preserve node identity has not been completed yet, and there are a lot of controversial opinions about the topic.)

Bibliography

[1] Tatarinov, I.; Halevi, A. Y.; Ives, Z.G.; Weld, D.S. – Updating XML, SIGMOD 2001, pp.413-24.

[2] Braganholo, V; Davidson, S.; Heuser, C. – From XML View Updates to Relational View Updates, VLDB 2004, pp. 276-87

[3] Braganholo, V; Davidson, S.; Heuser, C. – UXQuery: Building Updatable XML Views over Relational Databases, SBBD 2003, pp. 26-40

[4] XQuery 1.0: An XML Query Language, www.w3.org/TR/xquery

[5] XQuery Update Facility 1.0, W3C Candidate Recommendation, August 2008, www.w3.org/TR/xquery-update-10/

[6] Katz, H.; Chamberlain, D.; Draper, D.; Fernandez, M. - XQuery from the Experts, Addison Wesley, 2004

[7] Melton, J.; Buxton, S. - Querying XML, Morgan Kaufmann, 2006

[8] ISO/IEC 9075-14:2006 Information technology – Database languages SQLXML – Part 14: Related Specifications (SQL/XML:2006)

[9] Liu, Z.H.; Krishnaprasad, M.; Warner, J.W. - Effective and Efficient Update of XML in RDBMS, SIGMOD, June 2007

[10] Using XQuery with Oracle XML DB V11g, http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb_xquery.htm

[11] Nicola, M.; Jain, U. - Update XML in DB2 9.5, www.ibm.com/developerworks/db2/library/techarticle/dm-0710nicola

[12] XQuery Reference for IBM DB2 V9.5, ftp://ftp.software.ibm.com/ps/products/db2/info/vr95/pdf/en_US/db2xqre951.pdf

[13] Pal, S.; Dobrowolsky, I. – XML Support in MS SQL Server 2005, http://msdn.microsoft.com/en-us/library/ms345117.aspx

[14] Vithanala, P. – Introduction to XQuery in SQL Server 2005, http://msdn.microsoft.com

[15] Cheney, J. - Flux: FunctionaL Updates for XML, ACM Int. Conf. on Functional Programming, 2008, Victoria.

[16] Ghelli, G.; Re, C.; Simon, J. - XQuery!: An XML Query Language with Side Effects, EDBT Workshops 2006, pp. 178-191.

[17] Raghavachari, M.; Shmueli, O. - Conflicting XML Updates, EDBT 2006, pp. 552-569.

[18] ISO/IEC 9075-2: 1999 Information technology – Database languages SQL – Part 2: Foundation (SQL/Foundation)

[19] Key, M. - Xquery Update and Node Identity, http://saxonica.blogharbor.com/blog/_archives/2008/2/14/3523047.html

Yet Another XML Binding

Wolfgang Laun
Thales Rail Signalling Solutions GesmbH
<Wolfgang.Laun@thalesgroup.com>

Abstract

This paper is a technical report about the experience of implementing an XML binding to a programming language, i.e., the technique for converting between data in variables and its XML representation. Based on reflection techniques for obtaining information about a program's datatypes, a set of library modules was developed for marshalling, unmarshalling and XML schema generation. The techniques for mapping various data structuring concepts, such as arrays (with arbitrary index types), structures (with variants) and linked data are discussed in detail, as well as the way, the XML schema is constructed. Finally, an outline of applications and lessons learned is given. Although the concrete programming language of this project is obscure and almost extinct, these experiences might be valuable for a range of classic languages.

Keywords: XML, XML languag binding, XML Schema generation, XML marshalling, XML unmarshalling, CHILL

1. Introduction

1.1. Motivation

The motivation for starting the project "CHILL XML Binding" (CXB) is a result of the continuing use of the programming language CHILL [8] in applications for Thales IXL (interlocking system) LockTrac 6131 [1]. The language, first published in 1980, is a collection of features that were state-of-the-art at that time, and considered well-suited for implementing telecom applications; its module and threading features, a strong typing concept and extensive run-time checks are very valuable for safety-relevant applications. The development of the Thales IXL is based on a CHILL compiler, which was, up to version 2.95.3, available as part of the GNU Compiler Collection [2], and a tool chain for data preparation. Recent additions to the IXL system have introduced components written in Java, and a new generation of data preparation tools is being developed, both of which has suggested that XML might be used, as a format for communication between heterogeneous subsystems and for configuration data. Additionally, logging is considered as another application area for XML.

It should be obvious that the task of XML marshalling and unmarshalling would not be agreeable with programmers unless they were given an API with which data can be transformed with a single call. Moreover, an easy way of creating the corresponding binding on the Java side is highly desirable. Therefore, the project CXB was launched with the goal of creating a library for marshalling, unmarshalling and XML schema [6], [7] generation.

1.2. The Solution's Outline

CXB is implemented as a set of CHILL modules providing an API for marshalling, unmarshalling and schema generation. It uses the XML C Parser and Toolkit developed for the Gnome project [5], commonly referred to as libxml2, for document tree manipulations. Only a very slim interface layer is required for calling C functions from CHILL.

For mapping CHILL types to XML, full information about datatypes is essential. This is available through a language extension.

For requesting a CXB function, client programs have to set up a context, add a data item, and call the appropriate function, as shown below.

Example 1. Requesting CXB Functions.

```
-- declare and initialize a variable
DCL someVar SomeType := createSomeType(...);
-- a default context, no options
DCL ctxt XmlContext :=  makeXmlContex([]);
-- add someVar as a context item
addItem( ctxt, modeinfo(SomeType), ->someVar );

-- marshal to standard output
marshalFile( ctxt, "-" );

-- unmarshal from standard input
unmarshalFile( ctxt, "-" );

-- write an XML schema to a file
genXmlSchema( ctxt, "some.xsd" );
```

The implementation of a schema compiler for the generation of CHILL type definitions from an XML Schema was not considered, simply because there was no apparent application.

Below, Section 2 describes CHILL types and discusses various options for binding them to XML Schema types; then Section 3 provides a collection of technical issues arising from the implementation, and Section 4 demonstrates a simple use case. Finally, Section 5 outlines applications and summarizes lessons learnt.

2. An XML Binding for CHILL

2.1. A Brief Survey of Datatypes in CHILL

All CHILL datatypes can be found in other programming languages (in the Pascal tradition), although occasionally in a more or less restricted form. A general property common to all predefined and declared CHILL types is that they have a fixed size (in bytes). The essential properties of individual CHILL types are:

- Integer types are defined to have a size of 1, 2 and 4 bytes. Restricted types can be defined by specifying range bounds.
- CHAR and WCHAR are character types, with 7 bit and 16 bit representations of Unicode code points, respectively.
- Character strings may be composed from both character types, either with a fixed length or with a limited varying length.
- SET types are enumerations, whereas POWERSET types represent sets of values from any discrete member type.
- BOOLS types are bit strings of fixed length.
- REF types are reference values bound to a specific type.
- ARRAY types have static index types, given by an arbitrary discrete value range, and arbitrary element types.
- STRUCT types may contain tagged and tag-less variant fields.

Notice that this enumeration omits CHILL types which have no representation as values in memory, e.g., event and buffer types.

2.2. The Binding between CHILL and XML Schema Types

Table 1 shows the mapping of CHILL data types to XML Schema types, and the facets that may be used for limiting the base type's value range.

Table 1. Binding of CHILL to XML Schema Types

CHILL type	Schema type	Facet
BYTE	`xs:int`	minInclusive, maxInclusive
UBYTE	`xs:int`	minInclusive, maxInclusive
INT	`xs:int`	minInclusive, maxInclusive
UINT	`xs:int`	minInclusive, maxInclusive
LONG	`xs:int`	minInclusive, maxInclusive
ULONG	`xs:unsignedInt`	minInclusive, maxInclusive

CHILL type	Schema type	Facet
BOOL	`xs:boolean`	
CHAR	`xs:string`	length value="1"
CHARS(*)	`xs:string`	length value="*"
CHARS(*) VARYING		maxLength value="*"
WCHAR	`xs:string`	length value="1"
WCHARS(*)	`xs:string`	length value="*"
WCHARS(*) VARYING	`xs:string`	maxLength value="*"
SET	`xs:string`	enumeration
POWERSET	`xs:list`	Note 1
POWERSET CHAR	`xs:complexType`	Note 2
POWERSET WCHAR	`xs:complexType`	Note 2
BOOLS(*)	`xs:string`	maxLength value="*", note 3
REF	`xs:IDREF`	Note 4
ARRAY element	`xs:complexType`	Note 5
STRUCT	`xs:complexType`	Note 6

1. Powersets are represented as elements or attributes with Schema type `xs:list` except when the member type is CHAR or WCHAR.
2. Powerset elements are represented according to the member type. For member types CHAR and WCHAR, a powerset is represented as a complex type, with the set elements being represented as individual XML elements.
3. Bit string values are represented as a string consisting of '0' and '1'. The string may be shorter than the length of the bit string, with omitted bits being assumed to be unset.
4. Referenced types are currently restricted to STRUCT types. The attribute `_id` is added to the type in order to carry the `xs:ID` value.
5. In an XML schema, arrays have to be expressed as repeatable elements, by adding the attributes `minOccurs` and `maxOccurs` to an `xs:element` declaration. An anonymous complex type is generated for such an element's type, extending the schema type representing the array element's CHILL type. The extension adds two optional attributes. Attribute `_ix` represents index values according to the index type, and the boolean attribute `_def` indicates that the element represents the array element default value, to be applied to all absent elements.
6. Structure fields with a simple type may be represented as elements or attributes. Elements are inside a <sequence>, requiring a specific order. The unmarshaller,

however, will tolerate aberrations. Tag-less variant fields are marshalled as hexadecimal data.

2.2.1. Why are all integer types bound to `xs:int`?

XML schema types such as `xs:short` or `xs:unsignedByte` would be the exact counterparts for CHILL types INT or UBYTE, whereas `xs:int` requires additional facets for restricting the range of permissible values. The choice of `xs:int` was made because xjc, Sun's XML Schema compiler for JAXB [3], generates the Java types byte and short where possible, which is more of an inconvenience than a safeguarding.

2.2.2. Using `xs:list` for POWERSET types

Powersets are terse representations of boolean property sets, and they are frequently used in our application domain. It seemed desirable to use the most compact representation. For POWERSETs of character types, `xs:list` cannot be used, since Unicode 0x0020 cannot be represented in such lists.

2.2.3. Representations of Array Values

Whereas languages such as C and Java represent arrays with a uniform index range from zero upwards, CHILL (and other languages following Pascal's [4] style of array declaration permitting arbitrary ranges of discrete values as index type. This means that an array element is actually a pair: index value and element value. To represent this in XML, which merely provides for lists of elements, one has to define a wrapper type, extending the element type with an attribute for the index value.

An additional complication arises from the fact that index types are fixed ranges. This could be expressed in an XML schema by setting the facets `xs:minOccurs` and `xs:maxOccurs` to the same value. But there is a snag with arrays where not all elements are used, i.e., a part of the elements is set to some dummy value. To reduce the volume of XML text, the wrapper type is defined with an additional boolean attribute, to indicate that the contained element value is the default. (The two attributes are mutually exclusive.)

2.2.4. Reference Types and Linked Data Structures

Lists, trees and other linked data structures are constructed in the usual way, using reference types (“pointers”). The obvious XML Schema counterpart is `xs:IDREF`, but then the referenced type has to have a matching `xs:ID` value, as an attribute or element. Since this isn't necessarily available as a STRUCT field, and, if it were, could not be identified by looking at the type declaration, an additional attribute (`_id`) is added to types used as referenced types.

The current implementation does not handle pointers to arbitrary locations, e.g., to a field within a STRUCT value, even when that field is a structure. Such usage of pointers is rare, it is difficult to detect, and it cannot be mapped easily to XML.

2.2.5. Structures with Variant Fields

Structures with variant fields cannot be represented as multiple extensions of a type containing the fixed header, because a STRUCT type may contain more than one variant field. Therefore, the XML Schema particle `xs:choice` is used. For a generic way of selecting the appropriate choice variant from a structure variable's data in memory, the structure must be associated with a tag field, i.e., a fixed field preceding the variant field. Thus, both the marshalling and the unmarshalling algorithm can easily distinguish the applicable variant.

2.2.6. Representing Fields as Elements or Attributes

Using `xs:element` for all fields of a STRUCT type is simple to implement, whereas using attributes (where possible) reduces the volume of the created XML text and may, occasionally, be preferred for better readability. This has led to the addition of an option which lets user decide which strategy to use.

3. Design and Implementation Issues

3.1. Essential Compiler Support: Reflection

Reflection (as a program's capability of obtaining information about its own structure) is a relatively new concept that was not deemed to be essential at the time CHILL was designed. But the requirement of being able to write generic algorithms for processing variable contents has motivated the addition of a function `modeinfo(`*type*`)` which returns a recursive data structure containing metadata describing the type and all types used in its definition.

Each type is described by a structured node indicating

- the category of the type;
- for discrete types: lower and upper bounds;
- for enumeration types: the list of element names and values;
- the byte size;
- for powerset types: a pointer to the member type node;
- for reference types: a pointer to the referenced type node;
- for array types: pointers to the index and element type node;

- for structure types: the list of fields, as pairs of field names and type node pointers;
- the type name (unless anonymous).

3.2. Names for Tags, Attributes and Types

Tag names for XML elements and attributes are readily available from structure field names. Tag names for the root element and top level elements may be set using procedures from the context API.

Names for schema types are derived from the CHILL type name. If no name is associated with the type, a synthetic name is generated from the type's attribute. (Due to the coding rules in effect, anonymous types are rarely used.)

3.3. The Context

The API type `XmlContext` maintains options and parameters for the operative functions, which avoids cluttering their signatures with complex parameter lists and permits multiple use within the same program. Therefore, the context maintains

- a set of properties: formatted writing, omitting the XML header, tolerance for missing fields and elements, or preference for using attributes;
- a set of one or more variables constituting the XML document;
- the root element's tag;
- the document encoding;
- the namespace prefix and URI;
- the value of the document's `version` attribute;
- adapters for converting between binary data and non-standard XML text representations (see Section 3.6);
- handlers for unmarshalling values of a certain STRUCT type (see Section 3.7).

Setting a variable is sufficient for simple marshalling and unmarshalling operations.

3.4. Marshalling

Given a variable's address and the address of the node describing its type, the basic algorithm for marshalling is straightforward. It is driven by a pass through the node tree, processing the next chunk of memory according to the type described by the current node. In parallel, the document tree is set up, with the current parent node passed down and on to procedures handling the next type node:

- Type nodes for structure types proceed through the list of fields, while nodes for arrays iterate over the elements, using the type nodes for the index and element types.

- Procedures for single valued types simply process memory contents, creating the string representation for the XML document.
- Upon encountering a reference value, the `xs:IDREF` value is constructed from the address taken as an unsigned integer. If the referenced variable has not been emitted as an XML element yet, it is marshalled and added a child of the document root, including the address as an `xs:ID` value for attribute `_id`.

For advancing the pointer advancing through memory, not only the byte size of the current item must be considered but also whether packing is in effect or not.

3.5. Unmarshalling

The backbone of this algorithm is similar to the one for marshalling. One major difference results from the necessity of checking the string value taken from the document tree for being a correct lexical representation of the expected value. Additional checks are:

- For discrete (integer, character, boolean and enumeration) types, values must be within the implied or declared range.
- Fixed length string values must have correct length, and variable length string values must not exceed the maximum length.
- For a set value, the name string must be present in the list of set elements linked to the type node.
- For legacy (7-bit) strings, the codepoint must be in range.
- For structure types, if strict checking is in effect, all fields of a structure type must be present, either as an attribute or as an element.
- For array types, if strict checking is in effect, all elements must be present unless a default element is provided.
- If a reference type variable is to be unmarshalled where the `xs:IDREF` value is not null, then there must be a child of the document root containing this value as the value of its attribute `_id`. This element is unmarshalled into an allocated memory buffer, and its address is used as the binary value for the reference variable. It is also filed away, together with the id value, so that upon encountering the same id value once more, the corresponding address can be used again.

3.6. Adapters

An adapter is a pluggable mechanism for converting between a string and the binary representation of some datatype. It could be used if

- a scalar datatype is to be represented in a non-standard way, e.g., an integer value as a hexadecimal literal;
- a structured datatype should be represented as a string.

The adapter's "print" procedure is called during marshalling, and the "parse" procedure during unmarshalling.

3.7. Emulating Lists

In programming languages where lists are not available as a data structuring mechanism, they have to be implemented using techniques such as linked lists. For XML, however, the natural representation of a list is by repeating an element. For CXB, this can only be bound to an array, which, in CHILL, has to have a fixed (or at least: limited) number of elements.

To permit unmarshalling XML element sequences of arbitrary length where the corresponding array has a fixed, short length, CXB provides a "callback" feature. This associates a structure datatype with a handler procedure, to be called whenever an element corresponding to an item of the type is to be unmarshalled.

3.8. Schema Generation

Generating an XML Schema for a context is done by walking the node trees of the items defined for the context, dealing with each type as it is encountered. For an XML Schema's suitability for generating, say, Java code, repeated generation of anonymous simple or complex types from the same type node should be avoided. This is achieved by keeping track of the generated types.

The limitations imposed by CHILL type definitions containing lower and upper bounds for scalar ranges, or fixed or maximum string lengths are added as facets to the corresponding type definitions.

All referenced types must be defined with an additional attribute `_id` of type `xs:ID`. These types are detected by an initial pass through the tree of modeinfo nodes.

Arrays are declared by adding facets `xs:minOccurs=0` and `xs:maxOccurs` to an element. For the array element, two schema types are created: one for the element proper, and another one, as an extension, where attributes `_ix` and `_def` are added.

Schema generation can be augmented with the insertion of `xs:annotation` elements, either for comments, or with `xs:appinfo` elements containing processing instructions for XML Schema compilers such as xjc.

4. Example

This example demonstrates how a message interface between a CHILL client and a Java server for computing simple arithemtic operations can be set up. The first listing shows a CHILL program which, according to its command line parameter, either marshals, unmarshals or creates a schema. (For simplicity's sake, socket input and output has been omitted.)

Example 2. A CHILL Client.

```
client: MODULE

-- imports
<> USE_SEIZE_FILE "XmlContext.grt" <>
SEIZE XmlContext;
SEIZE makeXmlContext;
SEIZE addItem;
<> USE_SEIZE_FILE "XmlMarshal.grt" <>
SEIZE marshalFile;
<> USE_SEIZE_FILE "XmlUnmarshal.grt" <>
SEIZE unmarshalFile;
<> USE_SEIZE_FILE "XmlSchemagen.grt" <>
SEIZE genXmlSchema;

SYNMODE m_opcode = SET (
  plus, minus, times, divide_by );
SYN opsym ARRAY(m_opcode) CHAR = ['+', '-', '*', '/' ];

SYNMODE m_operation = STRUCT (
  opcode   m_opcode,
  operand1 LONG,
  operand2 LONG,
  opresult LONG
);

DCL operation m_operation;
DCL option CHARS(20) VARYING;
DCL ctxt XmlContext := makeXmlContext([]);
addItem( ctxt, modeinfo( m_operation ), ->operation );

CHILL_GETARGV( 1, option );
IF option = "-m" THEN
  operation :=  m_operation[ times, 42, 100, 0 ];
  marshalFile( ctxt, "-" );

ELSIF option = "-u" THEN
  unmarshalFile( ctxt, "-" );
  WRITETEXT( stdout, "%C %C %C = %C%/",
             operation.operand1,
             opsym(operation.opcode),
             operation.operand2,
             operation.opresult );

ELSIF option = "-g" THEN
  genXmlSchema( ctxt, "chill.xsd" );
```

```
    FI;

    END client;
```

After this program is run to generate the XML Schema, JAXB's Schema compiler is called to generate the corresponding Java classes. They are used in the following Java main program.

Example 3. A Java Server.

```
import javax.xml.bind.*;
import generated.*;

public class Server {
  public static void main( String[] args ) throws Exception {
    JAXBContext ctxt = JAXBContext.newInstance( "generated" );
    Unmarshaller u = ctxt.createUnmarshaller();
    Chill doc = (Chill)u.unmarshal( System.in );

    MOperation op = doc.getMOperation();
    switch( op.getOpcode() ){
    case PLUS:
      op.setOpresult( op.getOperand1() + op.getOperand2() );
      break;
    case MINUS:
      op.setOpresult( op.getOperand1() - op.getOperand2() );
      break;
    case TIMES:
      op.setOpresult( op.getOperand1() * op.getOperand2() );
      break;
    case DIVIDE_BY:
      op.setOpresult( op.getOperand1() / op.getOperand2() );
      break;
    }
    Marshaller m = ctxt.createMarshaller();
    m.marshal( doc, System.out );
  }
}
```

After compiling, a demonstration can now be run as a simple pipeline:

Example 4. Execution.

```
$ ./client -m | java Server | ./client -u
unmarshal XML
marshal XML
42 * 100 = 4200
```

5. Conclusions

5.1. Applications

CXB was used for adding an additional input format to a configuration data compiler, which generates a set of tables (several large arrays of structures) in binary form. Whereas the old format required the data to be in a format prescribed by CHILL's syntax rules, the XML format is free from any such dependencies. The XML Schema generated from the CHILL types describing the set of tables is useful in two ways:

- After generating Java code from the schema, the data can be processed easily by tools written in Java, e.g., ones that derive test cases from the assembled data.
- The schema itself is valuable as a documentation asset.

For an IXL subsystem implemented in CHILL, an additional set of configuration data was defined by CHILL type definitions. From the generated XML schema, Java code was compiled and used in the data preparation tool for marshalling, whereas the CHILL application uses CXB's unmarshalling API for loading a configuration file.

5.2. Lessons Learnt

Although the programming language CHILL is now rarely used, its features are typical for procedural languages in the Pascal tradition; therefore the experiences from the CXB project should be applicable for a range of similar programming languages.

The CXB project has demonstrated the feasibility of implementing an XML language binding for a traditional programming language. Even though structuring concepts such as arrays and linked structures require a circumspect approach, the overall effort for implementing the CXB library was surprisingly small, which, of course, is also due to the excellent support provided by the XML C Parser and Toolkit. The project has also shown that being able to use reflection on a program's datatypes is a valuable asset.

Adding XML as an (additional) data format removes dependencies from programs providing that data, and making data available in XML form enables the application of a wide range of standard XML tools.

Generating an XML Schema from which code for other languages, such as Java, can be produced provides a convenient way for establishing smooth data transfer between programs written in the traditional language and, for instance, Java programs.

Bibliography

[1] Fuß, W.: Tailored Solutions for Safety-Installations in the Loetschberg Tunnel -- A Project with Importance for the Trans-European Rail Traffic. DATE'08 Proceedings, pp. 21--25 (2008)

[2] GCC, the GNU Compiler Collection. http://gcc.gnu.org/

[3] Sun Microsystems, IncThe JavaTM Architecture for XML Binding (JAXB) 2.0. http://www.jcp.org/en/jsr/detail?id=222

[4] Jensen K., Wirth N.: PASCAL User Manual and Report. Springer Verlag New York Heidelberg Berlin, 1978

[5] Veillard D.:The XML C Parser and Toolkit. http://xmlsoft.org/

[6] W3C: XML Schema Part 1: Structures, Second Edition. http://www.w3.org/TR/xmlschema-1/

[7] W3C: XML Schema Part 2: Datatypes, Second Edition. http://www.w3.org/TR/xmlschema-2/

[8] ITU: CHILL -- The ITU-T Programming Language. CCITT/ISO/IEC International Standard ISO/IEC 9496, ITU Recommendation Z.200, Geneva (2003)

Introduction to XQC – the C Language Binding for XQuery

Matthias Brantner
28msec, Inc.
<matthias.brantner@28msec.com>

John Snelson
Oracle Corporation
<john.snelson@oracle.com>

Vinayak Borkar
University of California, Irvine
<vborkar@ics.uci.edu>

Christopher Hillery
Oracle Corporation
<chillery@lambda.nu>

Abstract

XQC is the C language binding for XQuery. It is intended to provide a standard API allowing applications to make use of any one of several available XQuery implementations. It was developed as a cooperative effort between developers of the XQilla XQuery engine of Oracle Berkeley DB XML and the Zorba XQuery engine from the FLWOR Foundation.

XQC offers a variety of features for programmatic control over all aspects of XQuery processing, including: compiling queries into re-usable expressions; executing compiled queries and navigating through the results; binding external variables and the context item for use during query execution; modifying many components of the static and dynamic query contexts; and receiving and handling errors which may occur at any point during the process. XQC also offers several convenience features and the option for extensions to the basic functionality.

XQC is a young specification. The authors would like to encourage adoption of the API among a larger number of XQuery engines. Further, we would very much like to have additional contributors to the XQC specification itself and to any reference implementations of XQC.

1. Overview

XQC is the C language binding for XQuery. It is intended to provide a standard API allowing applications to make use of any one of several available XQuery implementations. It was developed as a cooperative effort between developers of the XQilla XQuery engine of Oracle Berkeley DB XML [1] and the Zorba XQuery engine from the FLWOR Foundation [2].

XQC offers a variety of features for programmatic control over all aspects of XQuery processing, including: compiling queries into re-usable expressions; executing compiled queries and navigating through the results; binding external variables and the context item for use during query execution; modifying many components of the static and dynamic query contexts; and receiving and handling errors which may occur at any point during the process. XQC also offers several convenience features and the option for extensions to the basic functionality.

The benefits to having a standard, cross-engine API are numerous. Several specific ones include:

- The ability for developers to migrate between different XQuery engines without needing major code rewrites. This allows developers to try out several engines to see which is best for them, and to change engines later as new features or licensing options become available.
- The ability for tools to be written without depending on a particular XQuery implementation. Tools developers can deliver their products to a wider range of potential customers if they can develop to a common API.
- Eventually, the possibility of standardized training for XQuery/C developers.

The Java Database Connectivity (JDBC) standard [3] is a good example of how much benefit can be derived from a robust cross-application API standard. The Java world also has a standard API for XQuery, known as XQJ [4]. XQJ is an inspiration for XQC, although XQC does differ from XQJ in a number of respects.

XQC is an object-oriented API, even though it is for the C language. The current release (version 1) of the API comprises seven C structs, each of which contain only function pointers. Instances of these structs can be treated as objects; internally they contain implementation-specific data, and each function pointer acts as a method call operating on that data. As such, this document will refer to the structs as "classes", and to the functions contained in these structs as "methods". It is intended that later versions of XQC will also include a C++ language binding which closely mirrors the C API.

[1] http://oss.oracle.com/xqilla-xquery.html
[2] http://www.zorba-xquery.com/
[3] http://java.sun.com/products/jdbc/overview.html
[4] http://jcp.org/en/jsr/detail?id=225

The current version of the XQC specification and mailing list archives are available via Sourceforge [5].

2. Basics of Query Compilation and Execution

The entry point for users of XQC is the `XQC_Implementation` class. Client code obtains a pointer to an `XQC_Implementation` in an implementation-specific manner. For example, when using the Zorba XQuery processor, the following code will initialize the backing data store (using Zorba's basic "simple store") and return an `XQC_Implementation*`:

```
#include <xqc.h>
#include <zorba/simplestorec.h>

XQC_Implementation* impl;
void* store = create_simple_store();
zorba_implementation(&impl, store);
```

Note

`zorba_implementation()` returns `XQC_Error`, an enum defined by XQC for specifying the success or type of failure for all XQC methods. This is why the `XQC_Implementation*` is returned via an output argument. Error checking and handing will be discussed in more detail in a later section.

To compile a query, the method `prepare()` is called on the `XQC_Implementation`. The result of preparing a query is an `XQC_Expression`. This class represents a compiled query, and allows the query to be executed repeatedly.

```
XQC_Expression* expr;
impl->prepare(impl, "(1+2, 3, 4)", NULL, &expr);
```

Note

Because C is not object-oriented, it is necessary to pass the instance itself as the first argument to any method. This idiom is used for all methods in XQC.

To execute a prepared query, the method `execute()` is called on the `XQC_Expression`. The result of executing a prepared query is an `XQC_Sequence`. This object represents an instance of the XQuery Data Model (XDM) [6].

```
XQC_Sequence* sequence;
expr->execute(expr, NULL, &sequence);
```

[5] https://sourceforge.net/projects/xqc/
[6] http://www.w3.org/TR/xpath-datamodel/

3. Results of Query Execution

`XQC_Sequence` provides a cursor-like interface for exploring the query results. Recall that an instance of the XDM is defined to be a sequence of Items, where each *Item* is either a *Node* or an *Atomic Value*. The `next()` method of `XQC_Sequence` allows you to advance through the sequence one item at a time. Details about the current item of a sequence can then be read via accessor methods. For example, to retrieve the value of each item as a C string, the `string_value()` accessor method can be used as follows:

```
const char* string;
while (sequence->next(sequence) == XQC_NO_ERROR) {
  sequence->string_value(sequence, &string);
  printf("%s", string);
}
```

Other accessors on `XQC_Sequence` include:

- `integer_value()` and `double_value()` - return the current item as a C `int` or a C `double`, respectively.
- `item_type()` - return the basic type of the current item as an `XQC_ItemType`, which is an enum comprising all seven XDM node types and the 23 non-derived XML schema simple types.
- `node_name()` - return the name of the current node (an element or attribute) as a QName (a namespace URI / localname pair).
- `type_name()` - return the type of the current item as a QName.

4. Static and Dynamic Contexts

You may have noticed the `NULL` arguments passed to the `XQC_Implementation::prepare()` and `XQC_Expression::execute()` methods. Together, these allow the user to provide the *Expression Context* [7] for evaluating the query.

`prepare()` can be passed a *Static Context* in the form of an instance of `XQC_StaticContext`. Client code obtains an `XQC_StaticContext` via the `XQC_Implementation::create_context()` method. Methods on `XQC_StaticContext` allow the client to look up and configure a large number of components of the static context, such as:

- the *statically-known namespace definitions*
- the *default element/type namespace* and the *default function namespace*
- the *base URI*
- the *construction mode* ("preserve" or "strip")

[7] http://www.w3.org/TR/xquery/#context

- the *ordering mode* ("ordered" or "unordered")
- the *default order for empty sequences* ("least" or "greatest")
- the *boundary-space policy* ("preserve" or "strip")
- the *copy namespace mode* ("preserve" or "no-preserve", and "inherit" or "no-inherit")
- the *XPath 1.0 compatibility mode* setting

For all components which have enumerated possible values, XQC provides a C enum matching those values which the corresponding methods on `XQC_StaticContext` accepts. For instance, `XQC_BoundarySpaceMode` has two values, `XQC_PRESERVE_SPACE` and `XQC_STRIP_SPACE`, which can be passed to `XQC_StaticContext::set_boundary_space_policy()`.

Analogously, `execute()` can be passed a *Dynamic Context* in the form of an instance of `XQC_DynamicContext`. Client code obtains an `XQC_DynamicContext` via the `XQC_Expression::create_context()` method. Methods on `XQC_DynamicContext` are primarily for binding values to external variables in the prepared query via the `set_variable()` method, and for binding a value to the context item for the expression via the `set_context_item()` method. Both of these methods take an `XQC_Sequence` for the value, which allows chaining queries. The client can also specify the implicit timezone with `set_implicit_timezone()`.

Here is a complete example, which demonstrates: setting a static context component (base URI); executing a query with that static context; and chaining queries by binding the result of the first query to an external variable in a second query.

```
XQC_StaticContext* stat = NULL;
XQC_DynamicContext* dyn = NULL;
XQC_Expression* query1 = NULL, query2 = NULL;
XQC_Sequence* result1 = NULL, result2 = NULL;

// prepare and execute query1
impl->create_context(impl, &stat);
stat->set_base_uri(stat, "http://www.example.com/");
impl->prepare(impl, "fn:resolve-uri(\"index.html\")", stat, &query1);
query1->execute(query1, NULL, &result1);

// prepare, bind variable, and execute query2
impl->prepare(impl,
  "declare variable $uri external; fn:concat($uri, \"#anchor\")",
  NULL, &query2);
query2->create_context(query2, &dyn);
dyn->set_variable(dyn, "", "uri", result1);
query2->execute(query2, dyn, &result2);

// retrieve the string-value of the first item in result2
const char* str;
```

```
result2->next(result2);
result2->string_value(result2, &str);
/* prints "http://www.example.com/index.html#anchor" */
printf("%s", str);
```

5. Error Handling

As mentioned earlier, virtually all methods in XQC return `XQC_Error`. This enum has values describing various error conditions that can occur when calling XQC methods, such as "invalid argument", "not a node" (returned by `XQC_Sequence::node_name()`), and "no current node" (returned by accessors on `XQC_Sequence` if the sequence has been exhausted or not yet begun). It further has values for some non-error conditions, such as "end of sequence" (returned by `XQC_Sequence::next()` when the sequence is complete) and the successful "no error" condition. It also has values for implementation failures such as "internal error" and "not implemented".

Finally, it has values for describing various classes of errors that can occur during query compilation and processing, such as *static errors*, *dynamic errors*, and *type errors*. However, frequently these will be insufficiently specific. The XQuery specification details a large number of explicit error conditions, each identified by a QName, that an implementation can raise to describe a wide range of processing errors. XQC client code can receive these values via an `XQC_ErrorHandler`. This class is slightly different than the five XQC classes we have seen so far in that the client code itself is responsible for allocating the object and implementing its single method. For client code to inform the XQC implementation about an `XQC_ErrorHandler`, it may be passed to the `set_error_handler()` method on either `XQC_StaticContext` or `XQC_DynamicContext`.

The single method on `XQC_ErrorHandler` is named `error()`. It will be called by the XQC implementation when an error is about to be raised. It will be passed the QName of the error, as well as a string description. If the error occurred because the XQuery called the `fn:error()` function, the argument to `fn:error()` will also be passed to the `XQC_ErrorHandler` as an `XQC_Sequence`.

6. Memory Management and Thread Safety

All of the XQC classes, except `XQC_ErrorHandler`, have a method `free()`. Client code is required to call this method when it is done with a given XQC object. There are documented correct orders for freeing objects; generally, any object which was initially created by a method on another XQC object must be freed before the object which created it. For example, users must free any `XQC_Expression` objects created by calls to `XQC_Implementation::prepare()` prior to freeing the corresponding `XQC_Implementation`.

There are two exceptions to this general rule:

1. If an `XQC_Sequence` is bound to an external variable via `XQC_DynamicContext::set_variable()`, the implementation takes ownership of that `XQC_Sequence` and now has the responsibility for freeing it (presumably when the `XQC_DynamicContext` is freed). Somewhat oddly, the same is not true for an `XQC_Sequence` passed to `XQC_DynamicContext::set_context_item()`; the client code remains responsible for freeing that sequence.
2. The `free()` method of `XQC_InputStream` (discussed in the next section) is actually called by the XQC implementation, not client code. This will be explained below.

Three XQC classes have documented rules regarding thread safety. `XQC_Implementation` and `XQC_Expression` are explicitly thread-safe, such that instances may be shared between multiple threads. `XQC_StaticContext` is explicitly not thread-safe; each thread should create its own when required.

7. Other Features

7.1. Additional `XQC_Implementation` factories

In addition to the `prepare()` and `create_context()` methods discussed earlier, `XQC_Implementation` has several other methods which create instances of XQC classes.

1. `prepare_file()` and `prepare_stream()` allow compilation of queries from a C `FILE*` and from an `XQC_InputStream` (discussed below), respectively.
2. `create_empty_sequence()`, `create_singleton_sequence()`, `create_integer_sequence()`, and `create_double_sequence()` create instances of `XQC_Sequence` with the corresponding contents.
3. `parse_document()`, `parse_document_file()`, and `parse_document_stream()` parse XML from a C `char*`, a C `FILE*`, or an `XQC_InputStream`, respectively, returning the contents as an `XQC_Sequence`.

7.2. `XQC_InputStream`

The seventh and final XQC class is `XQC_InputStream`. Like `XQC_ErrorHandler`, this is intended to be allocated and implemented by client code. It is used as a way for client code to pass data into the XQC implementation, specifically for compiling a query or parsing an XML document via the `prepare_stream()` and `parse_document_stream()` methods discussed in the last section. The main method the user must implement is `read()`, which the implementation will call to read a number of bytes from the client code. Also, as mentioned in the section on memory management, there is a `free()` method which the client code must also implement. This is

called by the XQC implementation when it has completed reading the data from the stream.

7.3. Extension interfaces

Each implementation of XQC will likely have some custom functionality it would like to provide. XQC allows for implementation-specific extension functionality via the method `get_interface()`, which is defined on the five XQC classes that the implementation creates (that is, all except `XQC_ErrorHandler` and `XQC_InputStream`). These methods allow client code to request implementation-specific interfaces by name.

8. Conclusions and Future Outlook

XQC is a young specification. The authors would like to encourage adoption of the API among a larger number of XQuery engines. Further, we would very much like to have additional contributors to the XQC specification itself and to any reference implementations of XQC. There are several outstanding projects which need additional insight and implementors. For example:

1. As mentioned in the overview, it is intended that XQC will comprise both a standard C API and a parallel standard C++ API. Version 1 defines only the C API.
2. To allow truly cross-engine compatibility, especially for tools developers, it would be ideal if the current requirement for an implementation-dependent entry point to `XQC_Implementation` could be dropped.
3. `XQC_Sequence` has no methods for most of the `node accessor` methods defined by the XQuery Data Model specification. Critically, there are no `children()` and `attributes()` accessors for navigating into the node hierarchy.
4. XQC does not provide a way to specify collations or the default collation. (Zorba, for example, provides this functionality via an extension interface.)
5. XQC version 1 provides no mechanism for implementing external XQuery functions. (Again, Zorba also provides this functionality via an extension interface.)

We believe that XQC in its current form represents a useful and usable standard, and look forward to significant expansion and refinement in the future.

Approaches to change tracking in XML

Robin La Fontaine
DeltaXML
<robin.lafontaine@deltaxml.com>

Abstract

There are many different approaches to tracking document changes in XML. This paper looks at different use cases for situations where change to XML is important. We then review the different approaches used by some of the more popular formats, including OpenDocument, Open XML, DocBook, DITA and editors including XMetaL, oXygen and Xopus. These different approaches are discussed in order to illustrate the advantages and disadvantages of each approach in the context of the different use cases. As a prerequisite to the need to represent change, we first consider what constitutes a change in an XML document, as opposed to a change in an unstructured text document. This sets the scene for our more detailed discussion on the representation(s) of change.

Keywords: XML, delta, change tracking

1. Introduction

XML is widely used to represent documents and data, and is acknowledged to be the de-facto standard in this area. However, almost all documents and data are subject to change, but XML does not in itself provide a way of representing change. This has resulted in a number of different approaches to the representation of change in XML, in order to meet a wide variety of use cases.

In reviewing the way that change is represented in XML, it is first necessary to look in some detail at what can be considered to be a change in an XML document or data file. In order to do this, it is necessary to ask the more fundamental question which is, "when are two documents or XML data files the same?". This may seem a rather trivial question, but there are many factors that mean that it does need to be considered carefully.

Having established what is a change, it is then necessary to look at the reasons why changes are important, and why they need to be recorded. There are different use cases for this, including needs of editors, reviewers, and software testers.

This paper reviews a number of different approaches to representing change in XML. If there are different requirements, and different solutions, it is interesting to

look at the advantages and disadvantages of each solution against each of the different use cases.

NOTE: When we refer to 'XML documents' this includes the use of XML for data, as the discussion applies to the use of XML for both data and documents.

2. When are two XML documents the same?

Although it is obvious that if two XML documents have exactly the same sequence of characters, then they must be equal, there are many differences that are legitimately allowed between two XML documents without changing their meaning. For example, the XML specification states that the order of attributes has no semantic meaning, and therefore if two documents differ only in the order of the attributes they can be considered to be the same. Similarly, a user would also consider that a pretty-printed version of a document is the same as the original document. Therefore the handling of whitespace and layout is important when considering equality. There are different ways of representing namespaces, and namespace prefixes may be different, without any semantic difference between the documents.

Canonical XML [1] seeks to address some of these issues by specifying a defined way to write an XML file such that files that contain exactly the same information (or as the specification says, "that are logically equivalent within an application context") will be represented by the same sequence of characters. Canonical XML recognises "that two documents may have differing canonical forms yet still be equivalent in a given context based on application-specific equivalence rules for which no generalized XML specification could account.".

These issues are important in actual use cases and might include one or more of the following:

1. ID attributes: The use of ID attribute may be important, because these are defined as unique within a document, and are often used as internal pointers. Therefore if two documents are the same in terms of their structure, attributes and elements, except that the ID values are different (but consistent as internal pointers) then it may be reasonable to consider that the two documents are still the same.
2. Ordering: It is often the case that in a document some information, for example meta-data, may be presented in any order, again with no semantic difference.
3. Processing Instructions (PIs) and Comments: Differences in comments and processing instructions again do not constitute changes to the actual XML documents, and may or may not be important as a difference to a user.
4. Equivalent data: A specific XML format may allow different representations for the same information.

Therefore we can see that it is necessary to be very clear about how to determine whether or not two documents are the same, before we can make a reasonable assessment of how best to represent change.

2.1. ID attributes

An example of the use of an ID attribute within a document is shown in the listing below. This is an extract from an OpenDocument [2] XML file, and the ID is used to relate the start of the index marker and the end of the index marker. In this example, the actual values of the ID attributes `text:id="IMark665666424"`do not matter, but it is important that they are the same. If ID attributes like this are generated each time a document is saved, typically they will be different even when there is no change to the underlying document.

```
<text:p text:style-name="Standard">The quick brown fox
                <text:alphabetical-index-mark-start
                text:id="IMark665666424"
                /> jumped<text:alphabetical-index-mark-end
                text:id="IMark665666424"/> over the
                lazy dog.</text:p>
```

It is only possible to ignore changes like this using knowledge of the semantics of the document and the use of particular ID attributes.

2.2. Ordering

In the OpenDocument example shown below, a number of meta-data items are shown. As each individual item of data is enclosed in an element, the order of these items could be changed without changing the semantics of the document. We would like therefore to be able to say that some or all of the elements with in the `office:meta` element can appear in any order.

```
 <office:meta>
    <meta:document-statistic meta:table-count="0"
 meta:image-count="0" meta:object-count="0"
 meta:page-count="1" meta:paragraph-count="1"
 meta:word-count="9" meta:character-count="45"/>
    <dc:date>2010-01-12T16:31:22</dc:date>
    <dc:creator>Robin </dc:creator>
    <meta:editing-duration>PT00H01M40S</meta:editing-duration>
    <meta:editing-cycles>1</meta:editing-cycles>
    <meta:generator>OpenOffice.org/3.1$Unix
 OpenOffice.org_project/310m19$Build-9420</meta:generator>
</office:meta>
```

According to the definition of XML, the order of elements is always significant, and therefore it is not possible to cater for the situation described above in a standard way. However, it is simple to add an attribute to the parent element to indicate this, for example `dx:ordered="false"`.

2.3. Processing Instructions and Comments

Processing Instructions (PIs) allow documents to contain instructions for applications. PIs and comments do not form part of the document character data and therefore they may not be relevant when deciding if two documents are equal or not. Canonical XML may be generated either with or without comments, and these nodes may or may not be relevant to determining whether or not two documents are equal.

2.4. Equivalent data

A simple example of equivalent data is in XHTML where multiple whitespace characters are considered to be equivalent to a single whitespace character. Other examples might be different representations of the same numeric value, or different representations of a date. Two documents may be generated using different units of length, and then a length of "1000mm" would be the same as a length of "1m".

Similarly, it may be true that a `strong` (or `bold`) element which spans three words means exactly the same as if there were three `strong` elements each spanning one of the words. This of course is a function of the semantics of the particular XML schema in use.

The equivalence of two representations of the same data cannot be covered within a standard XML definition, but nevertheless it is very important when comparing documents to determine what has changed. If it is only the representation that has changed, then a user will generally prefer that this change is ignored.

3. Use cases for recording change in XML documents

There are many different use cases for the requirement to determine when documents have changed, or to represent the change in some way.

3.1. Documents

It is often important to see the changes between two or more versions of a document, either simply to review these or to accept or reject the changes. The reviewer may be interested in changes to the content and/or changes to the formatting. A reviewer probably does not expect to be able to see changes to some aspects of the document, for example meta-data.

One of the arguments for adopting XML is in order to separate content from layout, to allow repurposing of content into different types of media for publication. But XML is also extensively used to represent graphical items and layout. Although it is quite easy both to record and to show changes to the textual content of a document, it is much more difficult to record and show changes to formatting and layout.

Some document formats have built-in ways to represent changes. For example changes that are tracked or recorded will be recorded in the XML representation in OpenDocument or Open XML [OpenXML]. Other mechanisms include revision or status flags (DocBook [4] and DITA [5]) to show where content has changed, even if the actual changes themselves are not represented. These built-in mechanisms generally do not record every sort of change, but only the changes that are deemed important to a user.

3.2. Editing

When an editor is working on a document, he or she may wish to track the changes that he is making. This is useful either as a record of the changes that have been made, or in introducing the ability to reverse one or more of those changes.

One of the issues with tracking changes is that the document may at certain times during the editing process be invalid XML or even not well-formed XML. As only well-formed XML can be saved in a file, and it is preferable not to save invalid XML, it only makes sense to track changes that are complete 'transactions' from one valid state of the document to another. Where the XML format is not used as the internal data representation for the document, the internal data structure of the editing system may be able to handle these intermediate states.

One other use case for editing needs to be mentioned: many editing systems allow two documents to be compared (typically in a side-by-side view) and each change accepted or rejected. However, as this only requires a representation of the changes within the editing system, it is not relevant to our discussions about representations of change in XML. XML does present challenges to editing systems here because it is all too easy for the acceptance/rejection of changes to destroy the tree-structure of the XML file.

3.3. Processing

It is often necessary to produce reports about how data has changed. If this data is represented as XML, then it becomes important to have the ability to determine changes and then process these changes in order to produce a report.

As a variation on this, it may be useful or necessary to generate actions as a result of change, for example to ensure that the changes in some XML data are replicated in another representation of the same data in a database. It may also be necessary to merge two XML documents, and in order to merge a comparison must be made in order to do determine which parts of the documents are the same.

In this use case, any change to the XML content or attributes may be important and therefore potentially all changes to be represented.

3.4. Testing

Any software which is generating XML as an output will need to be tested. The need therefore arises to be able to compare the output from different versions of the software in order to determine whether or not anything has changed. Although much regression testing is satisfied simply by determining that two documents are the same, when they are different there is a need for some representation of the changes to be reviewed. The changes could be identified by line number, for example, or by XPath.

3.5. A Note on Comparison and Change Tracking

Editors often assume that they will get the same results from comparing two documents as they would have obtained by having change tracking 'on'. A comparison of two documents will show what has been changed, change tracking shows how the changes have been made. Therefore these are not the same, and the same result should not be expected. However, it often makes sense for the result of a comparison to be rendered into a change tracking format so that a user is then able to accept or reject the changes.

4. Representation of change in XML

There are several different ways of representing changes in XML. Although in general these are applied to the changes between two documents, some of them can be extended to show or represent changes between multiple documents, or multiple versions of the same document.

The example used here is to change "The **very** quick brown fox jumped over lazy dog." to "The quick brown fox jumped over **the** lazy dog.", where **bold** text shows changes. The examples have been shortened by removing some information that is not relevant to the discussion, and have been pretty-printed for clarity.

4.1. Line based diff

The traditional output of the UNIX `diff` utility shows changes between two text documents on a line by line basis. It is obviously possible to show differences between two XML documents in a similar way.

```
<         <p>The very quick brown fox jumped over lazy dog.</p>
---
>         <p>The quick brown fox jumped over the lazy dog.</p>
```

This representation has a number of limitations for XML because of its sytnax and tree structure neither of which is reflected in the line-based structure. It may be useful to accept or reject changes based on lines for a regular text document, but

this is unlikely to work for an XML document where the structure is often easily destroyed by moving lines from one document to another.

4.2. Processing instructions

Because processing instructions are in effect external to the main structure of an XML document, they are commonly used to mark additions and deletions in XML editors. Examples include XMetaL [6], Xopus [7], and oXygen [8].

One of the great advantages of using processing instructions to represent changes is that the underlying XML file can still be validated by ignoring the processing instructions. This implies that any deleted content will be within a processing instruction, and any added content will be marked by a start and end marker, each of which is a processing instruction.

```
<topic id="topic-1">
    <title>Topic title</title>
    <body>
        <p>The <?oxy_delete author="robin"
                timestamp="20100113T140621+0000"
                content="very"?> quick
            brown fox jumped over
            <?oxy_insert_start author="robin"
            timestamp="20100113T140625+0000"?>the
            <?oxy_insert_end?>lazy dog.</p>
    </body>
    </topic>
```

In this example, you can see that if all of the processing instructions are removed, the result is a valid file which represents all the changes being accepted.

4.3. Revision flags

Attributes are often used to show revisions to parts of an XML document in order to generate output showing where a document has been revised. This mechanism is built into the XML format itself, and any processor would need to know about this in order to reflect the changes. Examples include DocBook and DITA.

```
<topic xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">
    <title>Topic title</title>
    <body>
        <p>The <ph rev="deltaxml-delete">very </ph>
                quick brown fox jumped over
                <ph rev="deltaxml-add">the </ph>lazy dog.</p>
```

```
    </body>
</topic>
```

In this example, the revision flags have been inserted by comparing two versions of a document, and putting revision flags around text that has been either added or deleted. The added or deleted text can then be decorated in the publishing pipeline, either to PDF or HTML.

4.4. Tracked changes

Some document formats use a more sophisticated version of revision flags to show where text has been added or deleted. These tracked changes are represented in the XML structure, and the editing system may enable the editor to accept or reject them. Tracked changes are typically not able to represent all the possible changes to a document, but will satisfy the needs of a typical editor. Examples include OpenDocument Format (ODF) and Open XML. The example below is OpenDocument text format, ODT.

```
<office:body>
 <office:text>
  <text:tracked-changes>
   <text:changed-region text:id="ct528047904">
    <text:deletion>
     <text:p text:style-name="Standard">
      very
     </text:p>
    </text:deletion>
   </text:changed-region>
   <text:changed-region text:id="ct645104016">
    <text:insertion />
   </text:changed-region>
  </text:tracked-changes>
  <text:p text:style-name="Standard">
   The
   <text:change text:change-id="ct528047904" />
   <text:s />
   quick brown fox jumped over
   <text:change-start text:change-id="ct645104016" />
   the
   <text:change-end text:change-id="ct645104016" />
   lazy dog.
  </text:p>
 </office:text>
</office:body>
```

In this example of tracked changes, notice that the deleted text is held in a separate place from the main body text. This means that the main body of the document is very close to the new version of the document, i.e. with all the changes accepted. However, it is not trivial to reinsert the deleted text in its correct position, with all of the text decoration intact.

4.5. Generic XML deltas

A delta file can be defined such that it represents the differences between two arbitrary XML documents, in XML. Any XML format that is capable of updating one XML document into another could be described as a generic delta, for example XQuery Update Facility [9], XSLT [10] or DeltaXML [11]. Typically this will operate in one direction only (XQuery Update Facility, XSLT), though a symmetrical representation is also possible (DeltaXML). The delta may be a transformation, defining how to get from one document to another, or a data representation, defining what is different between the documents.

XQuery Update Facility and XSLT are both declarative transformations and can represent complex changes. They are intended to be executable transformations between two XML documents and will, in conjunction with the execution engine, convert one document into another. They are not intended to be used as a change-tracking mechanism. They do not meet any of the needs of the use case scenarios described here.

A delta file that is a data representation describes in some way the differences between two documents. A useful derivation of such a generic XML delta file would be one that contained not only the changes but also the original data, all in XML. Both of the original documents could be generated from such a delta representation. Ideally such a delta representation would not duplicate content that is common to the two documents. It is possible to transform this type of data representation into any of the other types of representation listed above, although the reverse is in general not possible. Therefore this generic delta representation is very versatile. An example of this is the full-context delta used by DeltaXML, shown in the example below.

```
<para deltaxml:deltaV2="A!=B"
        deltaxml:version="2.0"
        deltaxml:content-type="full-context">
 The
 <deltaxml:textGroup deltaxml:deltaV2="A">
  <deltaxml:text deltaxml:deltaV2="A">
   very
  </deltaxml:text>
 </deltaxml:textGroup>
 quick brown fox jumped over
```

```
 <deltaxml:textGroup deltaxml:deltaV2="B">
  <deltaxml:text deltaxml:deltaV2="B">
   the
  </deltaxml:text>
 </deltaxml:textGroup>
 lazy dog.
</para>
```

In this example, all of the data from both documents, A and B, is present and has the same look and feel as the original documents. The delta element wrappers and attributes indicate where the documents differ. Either version of the document can quite easily be extracted from this representation, for example using XSLT or XQuery. Note that the actual delta file will not comply with the original DTD/schema because of the additional delta wrapper elements and attributes, but each version that is extracted will be valid against the DTD/schema. Although not shown in this example, the format is capable of representing changes to attributes and elements as well as text. The format also extends to represent changes between more than two documents, for example the changes between two concurrent edits and the document from which the edits are derived.

5. Discussion

Having established that there are different use cases for needing to have a representation of change in XML, we can look at the advantages and disadvantages of the different representations for each of these use cases.

The issues here in include:

1. Does it allow all changes to be represented, e.g. changes to attributes?
2. Is it generic to all XML or specific to a particular format?
3. Does it allow processing using standard XML processors, e.g. XSLT?
4. Does it support accept/reject changes?
5. Is the result, after accepting or rejecting one or more changes, still valid and/or well-formed?

For a particular use case, not all of these features will be relevant. The table below provides a summary of which features are needed within each particular use case.

Table 1. Features Required for each Use case

Feature	Required for Documents?	Required for Editing?	Required for Processing?	Required for Testing?
All changes need to be represented	Not needed because it is mainly the content changes that need to be shown	Not needed when editing documents, though could be useful for a generic XML editor	Yes	Useful though not necessary
Needs to be generic to all XML not specific to a particular format	No, a document can have its own 'tracked change' format	Useful	Yes	Yes
Processable using standard XML processors	Useful	Not needed	Yes	Not needed
Supports accept/reject changes	Yes	Yes	Yes	Not needed
Result, after accepting or rejecting one or more changes, must be valid and/or well-formed	Valid	Valid	Well-formed only needed because output may be different XML	Not needed

We can also look at how each change represention performs against these features.

Table 2. Features of each Change Tracking format

Feature	Tracked changes or Revision flags	Processing instructions	Generic XML deltas	Line based diff
All changes need to be represented	No	No, cannot handle attributes	Yes	Yes but not XML-aware
Needs to be generic to all XML not specific to a particular format	No	Yes	Yes	Yes but not XML-aware
Processable using standard XML processors	Yes	Possible but not easy as deleted items need to be re-parsed	Yes	No
Supports accept/reject changes	Yes	Yes	Yes	No

Feature	Tracked changes or Revision flags	Processing instructions	Generic XML deltas	Line based diff
Result, after accepting or rejecting one or more changes, must be valid and/or well-formed	Valid	Valid	Well-formed, may be valid	No

The two tables have been deliberately lined up so that the middle three columns show a good match between the requirements for particular use cases, and specific change tracking representations:

The feature requirements for documents are met by tracked changes and revision flags.

The requirements for editing are a good match with processing instructions.

The requirements for general processing are met by generic XML deltas.

The line based diff does not really meet any requirements for XML documents. It is also possible to see from the second table that it would be possible to convert the processing instructions into tracked changes or revision flags, although this may not be easy. However the generic XML delta could be converted into any of the other formats because it is a more complete and generic representation of changes, as well as being processable.

6. Design of XML for change

Given the wide range of use cases, someone who is designing an XML format may want to consider some of these at the design stage. For example, is it always clear when two documents are the same? Is there a clear distinction between the textual content and the formatting? Is it easy to represent changes to either or both of these?

It is possible to draw some conclusions and design guidelines for a DTD/schema designer such that the resultant DTD/schema specifies documents that are easy to compare, or where change can be represented well.

Do not use attibutes for data that may change. If some information is liable to change and the change needs to be represented, then it is better to include the information as content rather than as an attribute. This is because it is easier to show changes to content than to show changes to attributes. SVG [12] is an example of an XML format which uses attributes inappropriately for information content, e.g. a list of geometric points. When comparing SVG, it is difficult to show which point in a list has changed, because XML structure cannot be introduced into an attribute. For documents, traditionally the text content is held as PCDATA and all formating and other information tends to be held as attributes. This works as a convention for documents, but for data representation consider having all content in PCDATA and use attributes sparingly. Remember you can never develop an attribute later to give it more XML structure, whereas you can always add substructure to elements.

ID attributes should be persistent. This cannot always be achieved, but it does make comparison - and therefore testing - much easier if applications that read and write the XML are required to maintain the value of ID attributes.

Unordered content should be enclosed in a single element. We have discussed the fact that some content can appear in any order, and it is best if such content is not mixed with ordered content. Therefore it is good practice to enclose such content in a container element. In particular, do not put data in attributes just because data is unordered, because that is a very poor reason for using attributes.

Consider which attributes or content constitute keys, and make these persistent. Keys are needed in many situations and help cross-references within and from outside a document. If you keep them persistent, then you can use them to reference from outside a document and comparison and testing is easier. Keys can be used to control alignment in comparison and therefore more accurate and predictable results can be obtained.

Avoid mixed content. For documents in XML this is almost impossible, but for data it is almost always possible to avoid mixed content, i.e. having both PCDATA and elements as siblings. Representing change to mixed content is difficult because the PCDATA needs to be parsed and divided, for example into words, in order to get sensible results.

7. Conclusions

There are several different ways of tracking change to XML documents. Each representation has its advantages in a particular use case scenario, except line-based diff which is inappropriate for XML. Many document formats have change tracking built into the format, and these range from basic revision flags on added or deleted text to a richer recording of changes to text and formatting. For editing documents, processing instructions enable a degree of 'undo' support for changes made to the document. Generic delta representations enable sophisticated processing of all changes to any XML document, generating reports or update scripts in almost any format. XML therefore has many advantages over line-based formats for generic change processing.

It is possible to convert from one change representation into another, in particular the generic delta can be converted into the other representations.

It is useful for anyone involved in designing or processing XML to have a basic understanding of how change can be detected and represented in order to choose appropriate solutions either at the DTD/schema design stage or when processing XML.

Bibliography

[1] Canonical XML Version 1.0: W3C Recommendation 15 March 2001. http://www.w3.org/TR/xml-c14n

[2] ISO/IEC 26300:2006 Open Document Format for Office Applications (OpenDocument). http://www.iso.org/iso/catalogue_detail.htm?csnumber=43485

[3] Standard ECMA-376 Office Open XML File Formats. http://www.ecma-international.org/publications/standards/Ecma-376.htm

[4] Norman, Walsh: The DocBook Schema. Working Draft 5.0a1, OASIS, 29 June 2005. http://www.docbook.org/specs/wd-docbook-docbook-5.0a1.html

[5] Darwin Information Typing Architecture (DITA) http://dita.xml.org/

[6] XMetaL authoring systemhttp://na.justsystems.com/content-xmetal[1]

[7] Xopus online editinghttp://xopus.com/xopus-web-based-wysiwyg-xml-editor.html

[8] oXygen XML editor http://www.oxygenxml.com/

[9] XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/

[10] XSL Transformations (XSLT). http://www.w3.org/TR/xslt

[11] DeltaXML: Managing change in an XML environment http://www.deltaxml.com/

[12] Scalable Vector Graphics (SVG). http://www.w3.org/Graphics/SVG/

[1] http://na.justsystems.com/content-xmetall

XML Pipeline Performance

Nigel Whitaker
DeltaXML Ltd
<nigel.whitaker@deltaxml.com>

Tristan Mitchell
DeltaXML Ltd
<tristan.mitchell@deltaxml.com>

Abstract

This paper considers XML pipeline performance of a case study constructed from real code and data. It extends our original small study into 'Filter Pipeline Performance' on the saxon-help email list. The system used in this case study is a comparison tool for OASIS OpenDocument Text or '.odt' files.

We will not describe the code in detail, but rather concentrate on the mechanisms used to interlink the various pipeline components used. These components include XML parsers, XSLT filter stages, XML comparison, Java filters based on the SAX XMLFilter interface and serialization. In the previous study we compared two pipelining techniques; this paper will extend this to consider three mechanisms which can be used to construct pipelines; these are:

- *The Java API for XML Processing (JAXP) included in Java Standard Edition 1.4 and subsequent releases*
- *The s9api package provided by the Saxon XSLT processor 9.0 and subsequent releases*
- *The Calabash implementation of the XProc XML pipeline language*

Our primary focus is performance and we will look at run times and memory sizes using these technologies. The overall runtime of the complete system will be described, but we will then concentrate on the performance of running a chain of filters (XSLT and Java based) as this is likely to be of more general interest. We will also discuss some optimization techniques we have implemented during the development process and further ideas for future performance improvement work.

1. Introduction

In this section we will introduce the system being studied, briefly describe the data and also some aspects of the experimental approach.

1.1. The case-study system

The system used for this case study is an ODT comparator. ODT or 'Open Document Text' is a document format supported by a number of word processors and office systems including OpenOffice.org and derivatives such as IBM Lotus Symphony, KWord (from KDE/KOffice) and Google Docs. The format is a standard developed by OASIS [4]. The ODT comparator is one of a number of ODT products from DeltaXML and this one is available for free-of-charge use, either online or through an OpenOffice.org plugin.

Our implementation of an ODT comparator consists of 5 processing pipelines roughly associated with the XML components of an ODT file (content.xml, styles.xml, meta.xml). The pipelines use XSLT 2.0 [9] and Java filters to perform various data manipulations (for example list and table rearrangements, style rationalization and reconstruction), comparison and post-processing and serialization back into the components of the ODT 'zip' file. Some filters are relatively simple (33 lines of code), the longest is 585 lines of XSLT 2.0. Java extension functions are also called, for example to compare binaries such as images and perform measurement unit conversions. The longest pipeline, for content.xml, consists of: input filter chains of 7 filters, a comparator, 22 output filter stages together with parsing and serialization.

1.2. Performance Objectives

We are proponents of using pipelined architectures for processing XML [5]. We prefer the divide and conquer approach and reusability benefits of simple filters composed into larger systems. Some of the filters we used are general purpose (work on any well-formed XML) and others are ODF specific. These filters are implemented in either XSLT 2.0 or Java code. We often test individual templates/functions in the filters. We also test individial filters and sets of filters as well as whole pipelines as black boxes.

However, we also recognize the need to minimise the overheads of multiple XSLT transformations or Java filtering process. Past experience has made us avoid using temporary intermediate files to link pipeline stages together (disk IO is too slow). Similarly using in-memory buffers (Strings or ByteArrays in Java) to hold intermediate results is avoided as there is also an overhead to reparsing lexical XML. Our inter-stage communication should be efficient ideally using parsed, pre-processed or event-based XML. But the primary reason for starting the performance work described in this case study was to minimize the memory footprint.

1.3. Experimental method

In order to appreciate the numbers presented later in this paper it may help to understand some details of the systems used and experimental techniques.

- The system used to perform the measurements was an Apple MacBook Pro with an Intel Core 2 Duo processor running at 2.53Ghz. The system has 4 GBytes of RAM and was running MacOS 10.6.2
- More recent JVMs offer better performance and fewer bugs. We used the most recent JVM/JDK combination available on the above system. This was Java version 1.6.0_17, and more specifically the: `Java HotSpot(TM) 64-Bit Server VM (build 14.3-b01-101, mixed mode)`
- When measuring runtimes we used a large heap size, typically a **-Xmx2g** heap setting, to avoid excessive garbage collection which often occurs when using close to minimal heap sizes.

When measuring memory sizes we used -Xmx virtual machine arguments to determine to nearest 1MB the smallest heap size which does not cause an OutOfMemoryError.

Where possible timings were done post XSLT compilation and also avoided measuring the JVM start-up times. Memory measurements were made using the facilities provided by `java.lang.manangement` APIs included in Java 1.5 and subsequent releases.

The data files used when reporting results were large (over 700 pages) annual financial reports. Two revisions with several hundred differences were compared.

2. The JAXP Pipeline

The Java API for XML Processing (JAXP) [3] has been available for a number of years and allows Java programmers to construct XML processing pipelines. Its provides classes such as the `SAXTransformerFactory` to create `TransformerHandler` and `XMLFilter` instances. A JAXP pipeline is constructed and then invoked through a single trigger point (a single `transform()`, `parse()` or when used in conjunction with our components a `compare()` method).

The pipeline construction involves linking the stages togther using the `setParent()` method for an `XMLFilter` or `setResult()` for a `TransformerHandler`, often via an intermediary `SAXResult` object. All of the stages communicate via the SAX [6] `ContentHandler` interface. Logically it can be considered that events flow between the pipeline stages. In an ideal world all of the filters in a pipeline would stream and there would be very few large in memory data structures. However, in reality this is often the exception and many stages will use in-memory data structures as part of their processing. For example, in order to support navigation using all of the XPath axes in an XSLT processor, an in-memory tree, or array-based-tree, data structure is often convenient. Similarly, the Longest Common Subsequence (LCS) algorithms used in comparison work well with in-memory data strucutures. Only when chaining `XMLFilter` or other callback or 'event' based code together does information 'stream' down a pipeline.

Using JAXP pipelines the following overall result was obtained:

Minimum heap space	933 MB
Runtime (2GB heap)	5 min 14 sec

3. The s9api Pipeline

Introduced in Saxon [7] version 9.0, this API allows `XdmNode` trees to be used as the inputs and outputs of a transformation. This allows chaining via the `setDestination` method and a single trigger method as shown in Example 1.

Example 1. Chained s9api pipeline example

```
XsltTransformer stage1=
      comp.compile(new StreamSource(new File("stage1.xsl"))).load();
XsltTransformer stage2=
      comp.compile(new StreamSource(new File("stage2.xsl"))).load();
XsltTransformer stage3=
      comp.compile(new StreamSource(new File("stage3.xsl"))).load();
...
stage1.setDestination(stage2);
stage2.setDestination(stage3);
stage3.setDestination(...);
stage1.setInitialContextNode(...);
stage1.transform();
```

However as our earlier email discussion [2] indicated this approach consumed significant amounts of memory for long pipelines and appeared to have linear memory consumption with pipeline length. A 'multitriggered' model allows explicit control over intermediate trees, including explicit nullification of references to help the garbage collector. The code for a simplified three stage multi-triggered pipeline is shown in Example 2.

Example 2. Three stage s9api pipeline code example

```
XdmDestination stage1result= new XdmDestination();
stage1.setDestination(stage1result);
stage1.setInitialContextNode(in);
stage1.transform();

in= null;
XdmDestination stage2result= new XdmDestination();
stage2.setDestination(stage2result);
stage2.setInitialContextNode(stage1result.getXdmNode());
stage2.transform();
```

```
stage1result= null; stage1= null;
XdmDestination stage3result= new XdmDestination();
stage3.setDestination(stage3result);
stage3.setInitialContextNode(stage2result.getXdmNode());
stage3.transform();
```

The actual code used is more complex and uses a `java.util.List` to store the filters and then iterates over the list. It also handles Java filters (depicted in Figure 1) and holds the previous filter's `XdmNode` tree which is used as input while the result of the current filter is being generated as another `XdmNode` tree. Care was needed when writing this code to ensure objects could be garbage collected. For example, `XsltTransformer` objects may retain references to their input, or initial context trees after they are executed.

Using multi-triggered s9api pipelines the following initial comparison results were obtained:

Minimum heap space	457MB
Runtime (2GB heap)	4min 5sec

These results demonstrated a significant memory improvement and reasonable speed increase. However these initial results were known to be non-optimal and subsequent optimizations are discussed in Section 5.

The multi-triggered s9api implementation achieves the objective of reducing the memory requirements of the overall pipeline. We have observed, using both code profilers and the monitoring facilities provided by the `java.lang.management` package that for the execution of a pipeline step the required memory approximately corresponds to:

- The size of the input and output `XdmNode` trees of the step
- The size of the internal data structures used by the step
- The size of pipeline wide data strucutures such as the Saxon Processor and its associated `NamePool`.

As the pipeline steps are executed, the memory requirement fluctuates according to the current step. The overall requirement for the pipeline corresponds to the step with the largest input and output trees and internal data structures. In the case of the ODT Comparator used in this case study, this step is the XML comparator used in the centre of the main content.xml pipeline, which unlike many of the other pipeline steps has two large input trees and the largest output tree.

4. The Calabash Pipeline

Calabash [1] is an XProc [8] implementation being developed by Norman Walsh. Its implementation internally uses Saxon `XdmNode` trees. XProc allows us to implement pipelines without the custom Java code needed for the previous Pipelines. We chose to use Calabash as it supported our preferences for Java and XSLT 2.0. Other XProc implementations are available and could also have been used. XProc does not allow us to implement our pipelines directly as it does not support our use of Java filter components and Calabash did not provide an extension step to run these filters. However, using the s9api pipeline code for running a Java filter as a basis it was relatively easy to implement an extension step in Calabash for these filters. The step declaration is provided in Example 3.

Example 3. Calabash extension step for Java filters

```
<p:declare-step type="dx:java-filter"
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:dx="http://www.deltaxml.com/ns/extensions/xproc"
  xmlns:cx="http://xmlcalabash.com/ns/extensions"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <p:input port="source" primary="true" kind="document"
        sequence="false"/>
    <p:input port="parameters" kind="parameter"/>
    <p:output port="result" primary="true" sequence="false"/>
    <p:option name="classname" required="true" cx:type="xs:string"/>
</p:declare-step>
```

This step requires the full classname, which must extend a SAX `XMLFilterImpl`, to be specified as a string and also allows parameters to be specified. Java reflection is used to locate a method using the parameter name prefixed with 'set', as the method name, and which takes a single Java `String` argument. A fragment of the ODT pipeline illustrating such a step is provided in Example 4.

Example 4. Using the Java filter extension step

```
<dx:java-filter>
  <p:with-option name="classname"
    select="'com.deltaxml.pipe.filters.dx2.wbw.OrphanedWordOutfilter'"/>
  <p:with-param name="orphanedThresholdPercentage"
          select="$orphanPercentage"/>
  <p:with-param name="orphanedLengthLimit" select="$orphanLength"/>
</dx:java-filter>
```

This extension step certainly proved useful when making our transition from JAXP. We hope to make the extension step available for use in Calabash and possibly other XProc implementations.

The XProc implementation of the ODT comparison pipeline made use of the standard `p:xslt` step, a modified version of `cx:delta-xml` and the `dx:java-filter` step described above. Six files comprising around 1,000 lines of XProc statements were needed. The performance of this implementation is as follows:

Minimum heap space	1631 MB
Runtime (2GB heap)	4 min 42 sec

The runtime is impressive and an improvement over the JAXP implementation that was our starting point. The memory requirements are higher than we hoped, but it must be remembered that we are using a 0.x implementation and may improve in future releases.

While this paper concentrates on performance issues it is worth pointing out that the conversion of Java pipeline code into XProc was a fairly easy process, once we appreciated the intricacies of XProc pipeline construction. The five pipelines or `p:step-declaration`s used to implement the ODT comparator comprised around 1,000 lines of XProc declarations.

5. Optimizations

In this section we will discuss some of the optimizations used to improve performance (both runtime and memory) of our s9api implementation. These optimizations could also be applied to Calabash.

5.1. Java filter combining

With our initial, simple s9api implementation we chose to make every stage communicate via XdmNode trees. In some cases we knew this was less efficient than JAXP where adjacent Java based XMLFilters do 'stream'. Our initial implementation is depicted in Figure 1.

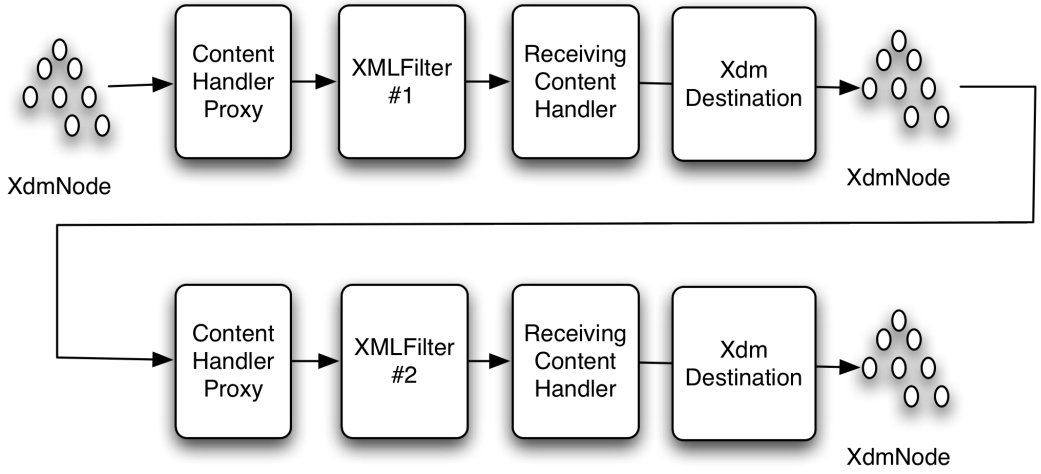


Figure 1. Adjacent java filters linked via Xdm trees

It is possible to detect adjacent Java or more generally streamable filters and connect them together more efficiently. This is depicted in Figure 2

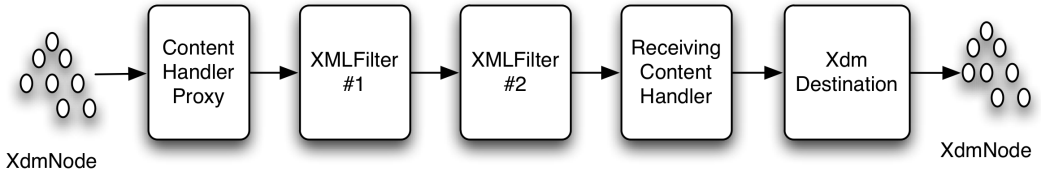


Figure 2. Adjacent java filters combined to stream

The screen output shown in Example 5 captures the debugging/progress information from a sequence of five Java filters used in the main content.xml comparison result pipeline. At this stage of the pipeline these filters are processing fairly large amounts of data, over 100MBytes of none-whitespaced XML when serialized, which accounts for the relatively long runtimes.

Example 5. Progress report for 5 adjacent Java filters

```
finished r[0] (Id: com...filters.dx2.wbw.OrphanedWordOutfilter)
              (elapsed: 2897 ms, cpu: 2507 ms)
finished r[1] (Id: com...filters.dx2.wbw.OrphanedWordOutfilter)
              (elapsed: 2825 ms, cpu: 2468 ms)
finished r[2] (Id: com...filters.dx2.wbw.WordSpaceFixup)
              (elapsed: 3063 ms, cpu: 2634 ms)
finished r[3] (Id: com...pdf.filters.ChangeMetricsFilter)
              (elapsed: 2724 ms, cpu: 2400 ms)
finished r[4] (Id: com...filters.dx2.wbw.WordOutfilter)
              (elapsed: 2585 ms, cpu: 2257 ms)
```

When these are combined and streamed together the result becomes:

```
finished r[0]-r[4] (elapsed: 5347 ms, cpu: 4724 ms)
```

The output indicates that by combining and streaming these filters together we can halve or better their runtime. However the output also demonstrates a disadvantage of combining - it is harder to observe progress and the relative performance of individual filters. Furthermore, another disadvantage not demonstrated here is related to exception and error handling. When we process the filters individually we can report precisely which one throws for example a SAXException. When combined together it becomes much harder to indentify the source of an exception from a Java stack trace.

5.2. Java filter conversion

Our initial Java filters were based on the `org.xml.sax.XMLFilter` interface, which is reasonably well supported in JAXP. However, s9api and Saxon more generally, uses a different type of interface/event internally, defined by the `net.sf.saxon.event.Receiver` interface. This interface describes elements for example using integers (to locate data in the Saxon NamePool) and has seperate events/callbacks for attributes, instead of them being bundled with `startElement`. We initially used some conversion classes (as shown in Figure 1) to be able to run an XMLFilter using XdmNode trees as input/output. However there are some inefficiences, related to converting String data to/from integers referencing the name pool, in this process.

At the time of writing we have converted one of our Java filters so that instead of extending the `XMLFilterImpl` class it extends the Saxon `ProxyReceiver` class. The performance is greatly improved as shown in Example 6.

Example 6. Progress report for ProxyReceiver based Java filter

```
finished r[2] (Id: com...wbw.WordSpaceFixup)
              (elapsed: 992 ms, cpu: 809 ms)
```

The performance gained from removing the 'impedence mismatch' of event conversion looks promising and we are now working towards a goal of replacing filters and/or providing additional `Receiver` support where we use `ContentHandler` implementations in our code.

Finally, we must emphesise a note of caution included in the Saxon documentation. The `Receiver` interface is not a public interface of Saxon and could change in future releases. When using these classes, and our code more generally, we will be careful to ensure that it is used in conjunction with a known version of Saxon.

5.3. Additional optimizations

The two techniques discussed above could be used together to support the running of chains of `ProxyReceiver` based filters. Running chains of filters could also be further optimized so that rather than starting from their own XdmNode input tree they are combined with a previous step. So for example an `XsltTransformer` could use `setDestination()` to a `ProxyReceiver` also implementing `Destination`, to subsequent filters via `setUnderlyingReceiver()` and finally to an `XdmDestination`.

We have implemented the optimizations described above using Java code to improve the performance of our s9api based pipelines. However the code structures could also equally be used by Calabash and possibly other XProc processors. Our experience of writing extension steps suggests that a step supporting a list of Java filters (either XMLFilter or Receiver based) as an option or parameter could be easily implemented.

6. Conclusions

We believe this case study confirms that Saxon's s9api package and the Calabash XProc implementation are both viable alternatives to using the JAXP package for building XML processing pipelines. We found the best performance using custom Java code and s9api. As well as the performance benefits the s9api interface is easier to understand and generally more flexible than JAXP. While we are relatively new XProc and Calabash users we also found it provided good results and was reasonably easy to use given its younger heritage. XProc could be a more readily accessible pipelining technology for non-expert users, particiularly those who are not Java programmers.

We have looked at optimization techniques, particularly those related to using event-based and streaming filters coded in Java. These techniques are perhaps useful to the smaller subset of users who are willing to invest development time to gain application performance. We hope that the performance results we have presented are useful to other users considering optimization techniques. This is work in progress (we need to complete more conversions to `ProxyReceiver`), however, the initial optimizations look promising. Here is a summary of the results for the overall ODT Comparator performance:

Pipeline	**runtime (min:sec)**	**minimum memory (MB)**
JAXP	5:14	933
s9api	4:05	457
Calabash	4:42	1631
s9api + partial optimizations	3:12	360

Streaming XML processing provides good performance with a good memory footprint. However, it also complicates measurement of performance, as it becomes difficult to separate the cost of the computation from the communication. We are hoping that writing filters in Java for performance reasons will become a thing of the past, as improvements in XSLT performance or alternative streaming technologies become available.

6.1. Future possibilities

This case study has concentrated on pipelining techniques that we could easily apply to our existing code in the form of discrete pipeline components. There are other approaches that could be taken and may review in the future:

6.1.1. Linking using the saxon:next-in-chain attribute

The `saxon:next-in-chain` attribute can be used to create pipelines. One slight disadvantage is that it hardwires the pipeline structure into the filters and has slight reusability implications (a filter may be reused in different pipelines and may be followed by different filters).

6.1.2. Merging filters

Using temporary trees and modes it may be possible to merge filters together in a single transformation. This may save time, but we no longer have explicit control of the garbage collection of the intermediate trees. Another issue is that when existing filters already use modes, the design of an automatic merging process becomes slightly harder.

6.1.3. Change the pipeline/processing model

For very simple filters we are doing a lot of serialization and tree-building. Most of the time is spent running the 'identity template' in many filters. Perhaps we can instrument and quantify how much time is spent running this filter as opposed to other filters. If it is substantial, we could consider using a persistent in memory data structure (perhaps a form of DOM with XPath support) and then have Java methods for modifying and updating it. We prefer the XSLT processing model and DOM trees can consume a lot of memory; perhaps XQuery Update could be used as an alternative approach to replace one or more filter stages.

Bibliography

[1] Normam Walsh. XML Calabash. http://xmlcalabash.com/

[2] Nigel Whitaker. Filter Pipeline Performance, saxon-help email list, 21 May 2009 http://markmail.org/message/l3k5ajhcvh6gosgq

[3] Jeff Suttor, Norman Walsh (eds). JSR 206: Java API for XML Processing (JAXP) 1.3 Version 1.3, 30 September 2004. http://www.jcp.org/en/jsr/detail?id=206

[4] OASIS: Open Document Format for Office Applications (OpenDocument) Specification v1.1. OASIS Standard, 1 February 2007. http://docs.oasis-open.org/office/v1.1/OS/OpenDocument-v1.1.pdf

[5] Nigel Whitaker, Thomas Nichols. Powering Pipelines with JAXP XML 2004 (IDEAlliance) November 15-19, 2004, Washington, D.C., U.S.A. http://www.deltaxml.com/dxml/336/version/default/part/AttachmentData/data/deltaxml-paper-xml-2004.pdf

[6] David Megginson. Simple API for XML http://www.saxproject.org/

[7] Saxonica Ltd. The Saxon XSLT and XQuery Processor http://www.saxonica.com/

[8] Normam Walsh, Alex Milowski, Henry S. Thompson (eds). XProc: An XML Pipeline Language. W3C Working Draft, 5 January 2010 http://www.w3.org/TR/2010/WD-xproc-20100105/

[9] Michael Key (ed). XSL Transformations (XSLT) Version 2.0 W3C Recommendation, 23 January 2007http://www.w3.org/TR/xslt20/

NAXD

Native XML Interface for a Relational Database

Karel Piwko
FIT BUT
<xpiwko00@stud.fit.vutbr.cz>

Petr Chmelař
FIT BUT
<chmelarp@fit.vutbr.cz>

Radim Hernych
<hernest@post.cz>

Daniel Kubíček
FIT BUT
<xkubic17@stud.fit.vutbr.cz>

Abstract

XML has emerged as leading document format for exchanging data. Because of vast amounts of XML documents available and transferred among companies, there is a strong need to store and query information in these documents. However, the most companies are still using a RDBMS for their local data warehouses and often it is necessary to combine legacy data with the ones in XML format, so it might be useful to (re)evaluate storage possibilities for XML documents in a relation database.

In this paper we focused on structured and semi-structured data-based XML documents, because they are the most common when exchanging data and they can be easily validated against an XML schema. We propose a slightly modified Hybrid algorithm to shred documents into relations and we allowed redundancy to make queries faster. Our goal was not to provide an academic solution, but fully working system supporting latest standards which will beat up native XML databases both by performance and vertical scalability.

Keywords: XML, XML persistence in RDBMS, Hybrid algorithm, RELAX NG mapping, XML:DB API, XPath LL(*) parser, query performance optimalization, data redundancy

1. Introduction

The XML[1] language has emerged as the most commonly used language for data description and information exchange nowadays. It virtually replaced all proprietary solutions used before. Obviously, these amounts of data must be stored and we would like to query them.

Data are usually stored in some kind of database. Although there are various ways how to define a database, we consider database being storage and retrieval engine and we prefer properties bounded to traditional relation databases over high availability and horizontal scaling, as some of databases (Apache HBase[2] or Bigtable[7]) which follow NoSQL movement.

We are convinced that relational databases are still the best solution for managing middle size data (up to 10 GiB) within a company, although the *one-solution-fits-all* might not be ideal in general[3].

However, there are more comfortable ways to work with data than using multiple SQL commands glued together with arbitrary middleware although even in this area things are getting easier for users. By all means, using REST interface and XQuery 1.0[3] language, possibly combined even with XForms to get XRX[4], is much more comfortable to an average user[12]. However, we must often integrate legacy data with these relatively newly obtained XML documents and provide unified access to both of them.

This is why we have decided to store XML documents within a RDBMS. We are aware that there are many developers who tried to do the same before [1][2][4][9][14], so the main aim of this work is to create such system using existing or slightly modified mappings to on the one hand beat performance of native XML databases and on the other hand provide the same user comfortability as XML database are doing.

This paper is organized as follows. In section Section 2 we are shortly describing XML, XML documents and the ways how to describe content of XML documents. Further, in section Section 3 we overview possible methods of storing XML document in a RDBMS with focus on *Hybrid* method, followed by XML document retrieval in section Section 4. Then we evaluate our solution providing intermediate results in section Section 5.

[1] http://www.w3.org/XML/
[2] http://hadoop.apache.org/hbase/
[3] http://www.w3.org/TR/xquery/
[4] http://en.wikibooks.org/wiki/XRX

2. XML and XML schemas

In this section we describe shortly XML language and its validation by different XML schemas. We will show differences of the schemas and explain why we have chosen RelaxNG in our system.

The XML language was created in 1996, as a formal simplification and application of SGML language. Its aims were to be flexible, readable and independent of character encoding and language. Additionally, any XML document can be validated against a XML schema, or because of namespace support, against multiple schemas. There are different XML schema languages, we will describe the most common ones.

- *DTD* This schema defines document by a nested lists of possible elements and attributes. It does not allow further constraints on an element or an attribute (e.g. value type or length). The schema itself is not an XML document and it usually too general[5], so programmatic construction of a graph based on it is unnecessarily complex.
- *XML Schema* XSD[5] defines documents as collections of elements and attributes, including relations of elements. Moreover, it defines types of elements and attributes, their default values and accepted content. It also allows to define order and repetition of elements in a finer way.
- *RelaxNG* RNG[6] defines documents as patterns, which are compared to elements in document instance. It is based on a formal theory of tree automata. It provides both XML and non-XML (compact) syntax for defining schemas, aims to be simple for users with knowledge of regular expressions and tries to unify elements with attributes as much as possible.

In our system we have chosen RNG because of simplicity and available tooling. Nevertheless, RNG and XSD can be converted one to each other which allows us to adopt all premises found during XSD schema-driven mapping research.

3. Persistence of XML schemas and XML documents

In previous section we discussed XML and XML schemas and we decided to tight up with RNG schema description. In current section, we will describe the way how XML documents[7] can be stored in a persistence data warehouse, focusing on the relational databases and using the advantage of having an XML schema defined or generated for data based XML documents. After an introduction of the problematics and existing solutions, we will compare our solution with its competitors. [7]

[5] http://www.w3.org/TR/xmlschema-0/
[6] http://www.relaxng.org/
[7] We are not discussing storage of XML schemas in a relational database and retrieval of the schema from a relational schema created by process described in this section. Nevertheless, an XML schema in the

3.1. Methods of storage XML in a persistence storage

XML documents can be stored in multiple ways either in a specialized storage, e.g. modeled as a graph structure and stored into a graph database or using an index based approach while storing them into a specialized database, such as eXist[8] is doing. Another approach is to consider XML documents structured enough for application purposes and store them in a key-value based storage and/or use some kind of a middleware to provide XML querying functionality.

We are considering storing them in a relational databases, which provide us following advantages:

- Technologies used in relational databases are developed for a long period of time and the are considered mature,
- Relational databases provide transaction (precisely ACID transaction) support and allow multi-concurrent access with a locking scheme,
- Interface provided and programming language support by RDBMS is huge,
- Relational (legacy) and XML content can be easily mixed in one application,
- Relational databases can be used as source of data-mining much easier than XML documents, although there is a research related to information retrieval in XML data warehouses[8].

On the other hand, we have to cope with following drawbacks:

- We have to develop techniques of storing XML data in relations,
- We cannot scale horizontally easily[6] because of enormous costs, which makes our system unusable in the environment where the high-availability is a must,
- We have to establish a way of querying data in a relational database.

However, we can sacrifice some properties of RDBMS (to be precise, properties associated with the design of RDBMS-based applications, such as data redundancy) to at least try to overcome these disadvantages.

3.2. Storage of XML data in relational databases

When storing XML data in a relational database, we can basically divide available methods into three categories, for our purposes we will only provide their short description. Readers interested in further details can follow [4]. These methods are:

- *Generic methods* XML schema is not used and the relational schema must be either generated general enough to handle all types of documents or usage of this storage is restricted to a limited set of documents. These methods has generally

RNG form is just an ordinary XML document, which can be stored by the same way as the other XML documents.

[8] http://www.exist-db.org/

lower performance because of numberous joins required to retrieve XML fragment from underlying structure.

- *Schema-driven methods* Relation schema is a transformation from either the existing XML schema or a XML schema is generated from document sample(s). The relation scheme can further be fine-grained and optimized by various methods. This area will be the main focus of our work.
- *User-defined and user-driven methods* User is responsible for creation of mapping between an XML document and tables in a relational database. This is no doubt the most flexible method and furthermore the easiest one to be implemented. However, this approach is interactive and user is required to be both skilled in XML and relational databases to yield the most efficient schema mapping. The user-driven approach tries to provide user reasonable default fixed mapping and the ability to influence it, such as *Mapping Definition Framework*[9].

3.3. Schema driven methods

During description of this approach, we will focus on data type XML documents and RNG schema. It must be said, that the most of schema-driven transformations have these limitations and therefore require enhanced functionality of underlying RDBMS:

- Identity constraints of an schema are usually mapped to constraints on relational tables, but a relational table contains just a subset of whole XML document, where the schema constraint is valid,
- Wild-cards enable storing of arbitrary element at place of the definition in the schema and thus can be stored only in a general data type column of a relational database, which must obviously be XML-aware to process wild-card elements,

Methods driven by a schema traversal can be divided into two categories, the *fixed* ones or the *flexible* ones[13]. The former is based only on XML schema while the latter uses more than one XML schema for a set of XML documents and evaluates speed of sample queries performed by the relational database. Further information on how queries are evaluated can be found in section Section 4.

For both of the methods, the XML schema of a document can be further simplified and transformed up to following constrains are met [14]:

- Any document conforming to the XML schema can be stored in resulting relational schema,
- Any XQuery executable over XML document can be executed in relational database instance.

The basic idea of schema simplification is follow repeatedly three types of transformation: /a/ flattening structure (e.g. inlining elements into their parents), /b/ reduction of unary operators to single one, /c/ grouping sub-elements (e.g. optimizing number

of sub-elements possible for the parent element) [14]. In document [1] was further shown, that grouping sub-elements into more than one group (more precisely, for `1..*` ER mapping, create groups of size `1` and `*`) can leverage performance, but this depends on statistic distribution of the sub-element. Dealing with groups of elements make processing of a schema more difficult, but does not involve any limitations of getting this done.

3.4. The Hybrid method

Hybrid method is considered as the best fixed method[14]. It was originally proposed for generation of the relational scheme for DTD, but it can be used for RNG as well. This method decides whether to include child element in parent's table (*inlining*) or create distinct table and establish a relationship by providing foreign keys. It identifies common elements, which spread among document definition, storing them in one table. Moreover, it tries to solve recursion which occurs within general definitions by identifying strongly connected components of the schema's graph representation. Elements are inlined if their in-degree is greater the one even if they are shared when reachable from a general node, except recursive descent.

In our modification we are not inlining elements if they are structured (complex) and appear more then once in a parent and/or the element can occur more than three times in its parent.

Basic types are mapped to SQL types `TEXT`, `DECIMAL`, `SMALLINT`, `INTEGER`, `BIGINT`, `BOOLEAN`, `FLOAT`, `DOUBLE`, `DATE`, `TIME`, `TIMESTAMP` and `INTERVAL`. Elements with mixed content are stored as `TEXT` and XML support of underlying database is required. Although constraints can be mapped to integrity checks, this is not used because invalid documents are not accepted, as described further in the text.

Relations are created from a graph, which is basically a DOM tree of RNG scheme. The graph is traversed to find out the root element of a scheme without backward references and nodes are processed according to their type (simple, complex) and position recursively. Encoding of nodes is preserved. Metadata are stored in tables `xcollection`, `xdocument` and `xtable`. More details can be found in [10].

3.5. Shredding XML document into relations

The question is whether to allow storage of documents not conforming to given schema. We have basically four possibilities:

1. Allowing storage of these documents by dynamically creating database relations based on their schema,
2. Store parts of documents not conforming to schema using general tables (we call them *junk* tables),
3. Reject the documents as not being valid for our storage schema,

4. Transform documents on-the-fly to the schema in our database.

On the one hand accepting them can allow us being more general but on the other hand will greatly augment complexity of queries because unions of results gained from different schema mapping.

Furthermore is difficult for user to query data which are not conforming to schema, because user is not aware of the existence of additional XML elements. When queried data are an input for another processing tool, elements not defined in the XML schema unnecessarily leverage complexity of the tool. We evaluated junk table approach in [10], where we found out that junk table is used mostly when a part (or whole) document is to be retrieved. The most appropriate approach would be to let user define if the not conforming part of document can be either transformed or cut off and the rest of document stored in database. Since insertion of data into DBMS should be non-interactive, this part of data preprocessing is not performed and user is recommended to clean data himself.

Because we expect data stored in database not to change frequently and reasons stated above, we decided to reject all documents not conforming to the given schema.

3.6. Modifying persistent data

The real performance of our solution depends on the ratio between select and modification queries. No doubt, because we allowed redundancy of the data, we preferred faster selection. We expect our storage system to be mostly used for retrieval of useful information, which are rather static and do not modify over time. We have decided to limit update of the data to document level, because most common usage of our system aims to be a storage with fast retrieval of either XML chunks or whole XML documents and finer grained query evaluation optimizations would raise complexity significantly.

4. Retrieval of XML Data from Persistence Layer

In the XML world, there are specialized languages designed to query XML documents and collections. The most commonly used are XPath 2.0[9] for navigation in the XML tree and XQuery. In this section we will describe how document is retrieved from a database and how this impact performance.

Obviously, when XML document is shredded into relations, retrieving it in the form how it was stored is the most difficult and time consuming operation. We overcame this problem by allowing data redundancy, so instance of the document is stored in RDBMS as a BLOB object. This makes the query much faster. It would be nice to be able to provide the same data redundancy for other frequently retrieved and relatively large XML chunks.

[9] http://www.w3.org/TR/xpath20/

When an XML fragment is to be pulled out from the database, we have to read all the metadata (see section Section 3.4). Additionally, it is necessary to traverse all tables with children of the desired element. Our system descends from the root element and consecutively call SQL statements without creating exhaustive join operations. The fragment can be retrieved both as DOM tree or in its textual form. Since evaluation of XPath is being upgraded, it is not mentioned in this paper.

5. Evaluation

When measuring performance, we focused on lookup and publish queries because they exercise the use cases expected for our system. The same XPath queries were executed on both our system and eXist, version 1.2.5. The benchmark consisted from XPath queries with up to six steps and simple node and attribute predicates over a collection of approximately 1000 documents retrieved from weather[10] of total size 6 MiB. Results were measured on Intel Celeron M 1.6 GHz with 2 GiB RAM. For imitating access to data, we let XML:DB API generate DOM tree, which was traversed for each returned `XResource`. The results were already published in [10]. Our system showed up significantly superior performance in compared to eXist, both in result retrieval and DOM tree traversal. Extract of results is shown in table Table 1.

Table 1. Sample of performed queries, times are in milliseconds

XPath query	Total results	NAXD [ms]	eXist [ms]
`/weather/head/locale`	1,000	120	146
`//wind`	11,000	145	547
`//part/wind/*`	40,000	141	2,156

However, since then the evaluation core was rewritten to conform XPath 2.0 constraints, we would like to provide more data intensive application as a testsuite. Testing scenario is an XRX based Czech Wikipedia portal created when comparing eXist and relational database performance[12] or its modification. This work is currently in progress.

6. Conclusion

In this work we have shown that data-centric XML documents which conform to RelaxNG schema can be stored in the relational database using modified *Hybrid* algorithm with very satisfiable result. Because we allowed redundancy, we are able

[10] http://www.weather.com/

to pull out original document even with processing instruction, comments and other elements which are not stored in relation tables.

Since whole XPath 2.0 is supported, resulting system does not suffer from limited functionality and proves our concept viable.

Our approach leverages XML:DB API. In the future we would like to provide provide XQJ implementation in hand with extending XPath 2.0 to XQuery 1.0 support. XQJ, JDBC equivalent for XML databases would enrich expression power while providing additional performance because of better expression caching.

Bibliography

[1] Bohannon Philip - Freire Juliana - Roy Prasan - and and Siméon Jérôme: From XML Schema to Relations: A Cost-Based Approach to XML Storage

[2] Bohannon, Philip - Freire, Juliana, Haritsa, Jayant R. - Roy, Prasan - Siméon, Jérôme - Ramanath, Maya: LegoDB: Customizing Relational Storage for XML Documents 2002

[3] Stonebraker, Michael - Madden, Samuel - Abadi, Daniel J. - Harizopoulos, Stavros - Hachem, Nabil - Helland, Pat : The end of an architectural era: (it's time for a complete rewrite) 2007 VLDB Endowment

[4] Mlýnková, Irena - Pokorný, Jaroslav: XML in the World of (Object-)Relational Database Systems 2005

[5] Malloy, Mary Ann - Mlýnková, Irena: Closing the Gap between XML and Relational Database Technologies: State-of-the-Practice, State-of-the-Art and Future Directions

[6] Wei, Zhou - Pierre, Guillaume - Chi, and Chi-Hung: Scalable Transactions for Web Applications in the Cloud Delft, The Netherlands 2009

[7] Chang, Fay - Dean, Jeffrey - Ghemawat, Sanjay - Hsieh, Wilson C. - Wallach, Deborah A. - Burrows, Mike - Chandra, Tushar - Fikes, Andrew - Gruber, Robert E.:Bigtable: A distributed storage system for structured data 2006

[8] Pérez, Juan M. - Pedersen, Torben Bach - Berlanga, Rafael - Aramburu, and María J.:IR and OLAP in XML Document Warehouses 2005 Springer Berlin / Heidelberg

[9] Amer-Yahia, Sihem - Du, Fang - Freire, Juliana: A Comprehensive Solution to the XML-to-Relational Mapping Problem 2004

[10] Radim Hernych: Transformace a persistence XML dat v relační databázi 2009

[11] Mlýnková, Irena: XML Data in (Object-)Relational Databases 2007

[12] Karel Piwko: Nativní XML databáze 2008

[13] Amer-Yahia, Sihem: Storage Techniques and Mapping Schemas for XML 2003

[14] Shanmugasundaram, Jayavel - Shekita, Eugene - Kiernan, Jerry - Krishnamurthy, Rajasekar - Viglas, Efstratios - Naughton, Jeffrey - Tatarinov, Igor: A general technique for querying XML documents using a relational database system 2001

A [insert XML Format] Database for [insert cool application]

Vyacheslav Zholudev
Jacobs University Bremen
<v.zholudev@jacobs-university.de>

Michael Kohlhase
Jacobs University Bremen
<m.kohlhase@jacobs-university.de>

Florian Rabe
Jacobs University Bremen
<f.rabe@jacobs-university.de>

Abstract

TNTBase is a versioned XML database; it combines XML fragment access techniques like XQuery with file system functionality and versioning features a la Subversion. We present an infrastructure how the generic TNTBase system can be specialized to include document format-specific services, such as validation, format-specific virtual files and human-oriented presentation.

1. Introduction and Related Work

In [18] we have presented the TNTBase system, a versioned XML database, which combines XML fragment access techniques like XQuery [16],[17] with the file system functionality and versioning features of Subversion [11]. This system is intended as a generic storage solution for web applications based on collections of XML documents. However, most such applications are tailored to specific features of their respective document formats. For instance, DocBook [15] manuals rely on format-specific style sheets for web presentation and printing, which only work if all documents are valid instances of the DocBook format. The situation is similar for other applications; some even combine multiple document formats and thus need to support multiple validation schemata and presentation pipelines. Furthermore, even homogeneous collections may contain documents in various versions of the underlying formats: indeed the versioning capabilities of the TNTBase system should enable the management of long tails of legacy materials for which format conversion is not cost-effective.

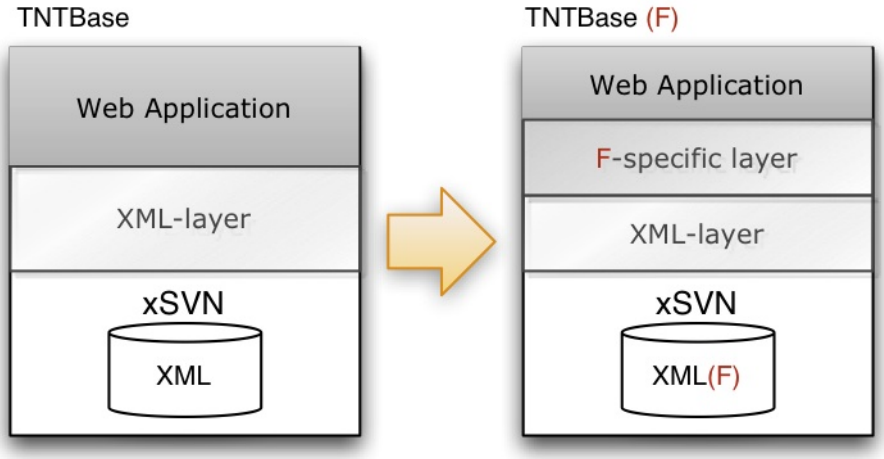


Figure 1. TNTBase(*F*): A Basis for Web Applications

In our experiments with the TNTBase system, it has turned out that many of the format-specific functionalities of the web application layer can be performed by the TNTBase system if we add format-specific interfaces for validation, aggregation, and presentation. We will call this layer TNTBase(*F*), since the services are parametric in the document format *F*. This extension leads to a refined information architecture of TNTBase-supported web applications; see Figure A Basis for Web Applications, where the generic TNTBase(*F*) layer takes over format-specific functionalities and thus decreases the effort for developing web applications.

In this paper we present the TNTBase(*F*) infrastructure and its APIs for validation (Section 3), format-specific virtual files (Section 4), and human-oriented presentation (Section 5). We will present much of the functionality of TNTBase(*F*) by using our OMDoc format [5] (*F* = *OMDoc*) as a running example, since this drives the particular development and integrates many of the structural prerequisites for virtual documents. The work presented in this paper is based on the experience of by managing a collection of more than 2000 OMDoc documents (ranging from formal representations of logics to course materials in a first-year computer science course) in TNTBase(*OMDoc*).

This paper is a short version of [19], which should be consulted for details we had to omit for space reasons.

2. Document-Format-Specific Features for TNTBase

To make this paper self-contained we briefly recap TNTBase (see [18],[14] for details). Then we take a closer look at the document-specific workflows in document-collection-centered applications that can be generically supported by a versioned XML storage system like TNTBase. This analysis serves as a motivation and basis for the

implemented TNTBase(*F*) layer, which we will present in detail in the following sections.

2.1. TNTBase State of the Art

The core of TNTBase consists of the xSVN module, which integrates Berkeley DB XML [2] into a Subversion server. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or LaTeX source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage. Keeping XML documents in DB XML allows us to access those files not only via any SVN client, but also through the DB XML API that supports efficient querying of XML content via XQuery and modifying that content via XQuery Update. As many XML-native databases, DB XML also supports indexing, which improves performance of certain queries.

The TNTBase system is realized as a web-application that provides two different interfaces to communicate with: an xSVN interface and a RESTful interface for XML-related tasks. The xSVN interface behaves like the normal SVN interface — the mod_dav_svn Apache module serves requests from remote clients — with one exception: If one of the committed XML files is ill-formed, then xSVN will abort the commit transaction. The RESTful interface provides XML fragment access to the versioned collection of documents:

Querying:	xSVN extends DB XML XQuery by a notion of file system path to address path-based collections of documents. For instance, a part of a query **collection(//papers*/*.xsl)** will address all XSLT stylesheets in the child folders of directories having the **papers** prefix.
Modifying:	It is also possible to modify XML documents via XQuery Update. In contrast to pure DB XML, modified documents are versioned, i.e., a new revision is committed to xSVN.
Querying of previous revisions:	Although xSVN's DB XML back-end by default holds only HEAD revisions of XML documents, it is also possible to access and query previous versions by providing a revision number if they have been *cached* (by user request). Previous versions cannot be modified since once a revision is committed to an xSVN repository, it becomes persistent.
Virtual Files:	These are essentially "XML database views" analogous to views in relational databases; these

are tables that are virtual in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, TNTBase virtual files (VF) are the results of XQueries computed on demand from the XML files explicitly represented in TNTBase, presented to the user as entities (files) in the TNTBase file system API. Like views in relational databases TNTBase virtual files are editable, and the TNTBase system transparently patches the differences into the original files in its underlying versioning system. Again, like relational database views, virtual files become very useful abstractions in the interaction with versioned XML storage.

2.2. Validation and Interface Extraction

One of the salient features of XML as a representation format are its well-established methods for document validation. Indeed most document management workflows take advantage of this and check grammatical constraints on documents by schema validation to simplify further document processing. TNTBase(*F*) supports this practice by validating documents upon commit by default. However, sometimes schema validity is too strong a restriction in practice (e.g., during document authoring or format migration), so the level of validity is configurable. Moreover, many document format constraints — e.g., inter-document link consistency or the absence of link cycles — cannot be checked even by modern schema languages; therefore TNTBase(*F*) provides an API for custom validation modules that can verify that high-level document (collection) integrity constraints are met, if later processing steps require them.

Both high-level validation and further document processing often rely on specific information about other parts of the document collection. To support these processes incrementally, TNTBase(*F*) supports the extraction and indexing of such information upon commit. This approach is analogous to the "separate compilation" paradigm in programming, where declared methods and their types are extracted at compile-time into "signature" files that are used when compiling other files. A prominent example in the XML arena is the extraction of RDF triples for the use in query engines from RDFa-annotated documents. As the specific information is format and application specific, we speak of *interface extraction* for the purposes of this paper. TNTBase(*F*) enables the user to configure interface extraction at commit time; in our experience, this is a crucial ability to make high-level validation tractable. For details of the realization we refer to Section 3.

2.3. Compilation, Browsing and Cross-Referencing

Eventually, the purpose of most document collection applications is to enable humans and machines access to (processed forms) of the document collection. Technically, this usually involves the transformation into specialized presentation-oriented formats for machine (e.g., programming languages) or human (e.g., PDF generation) consumption. Depending on locality properties of the transformation and the rate of change of the document collection, it can be more efficient to execute the transformations when a document is committed to the collection (then we can employ a process analogous to interface extraction) or when the document is requested. Both workflows need to be supported in TNTBase(*F*), and in both cases these documents are automatically served in addition to their sources. If a post-processing of documents for human consumption is defined, TNTBase can provide an enhanced browsing interface. In the future, the TNTBase architecture will make it possible to provide higher-level presentational services such as navigation bar, cross-referencing, and in-place expansion of links, if it is told, which language features constitute document fragments and links to documents or document fragments (see Section 4.2). For details of the realization in TNTBase(*F*) we refer to Section 5.

2.4. Virtual Documents

TNTBase virtual files as introduced above have one great disadvantage: They are not instances of one of the XML formats in the application. For instance the example of a virtual file of section headings of the SVN Book [11] used in [18] groups the **title** elements in special TNTBase elements, giving a mixed-namespace format specific to the TNTBase system. One generally wants to have virtual files that are "virtual documents", i.e. valid instances of a given format *F*. This makes *virtual documents* into first-class citizens at the web application level. In theory, if a document format permits to be split into fragments, it becomes possible to recombine the same content fragments into multiple (virtual) documents. If virtual documents make their composition explicit, edited versions can be decomposed into edited fragments, and the changes can be propagated to the original document and other virtual documents using said fragments. But achieving this is surprisingly difficult; both in theory and in practice. In Section 4.2 and Section 4.3, we discuss the properties the target format *F* must have to allow this, and the TNTBase(*F*) implementation.

2.5. Realizing TNTBase(*F*)

As described in [18], TNTBase has two major components: xSVN and Java-based Web-application that is built on top of the Java library for accessing xSVN's DB XML directly. Most of the material described in this paper involves both parts and interactions between them. For instance, when one commits a new version of some

XML documents and wants their presentation to be cached, this involves committing files in xSVN, figuring out what files the presentation should be generated for, sending the corresponding request to the Java part of TNTBase, generating presentation and finally saving it into DB XML. We omit the technical details about the interactions, since they mainly concern the (painful) integration of C++ and Java. For our initial implementation, we utilize the standard SVN hook mechanisms for pre-commit and post-commit processing. In a nutshell, we use a pre-commit or post commit hook (depending on the processing purpose, e.g. validation or generation presentation) for figuring out what subset of committed files is subject to further processing.

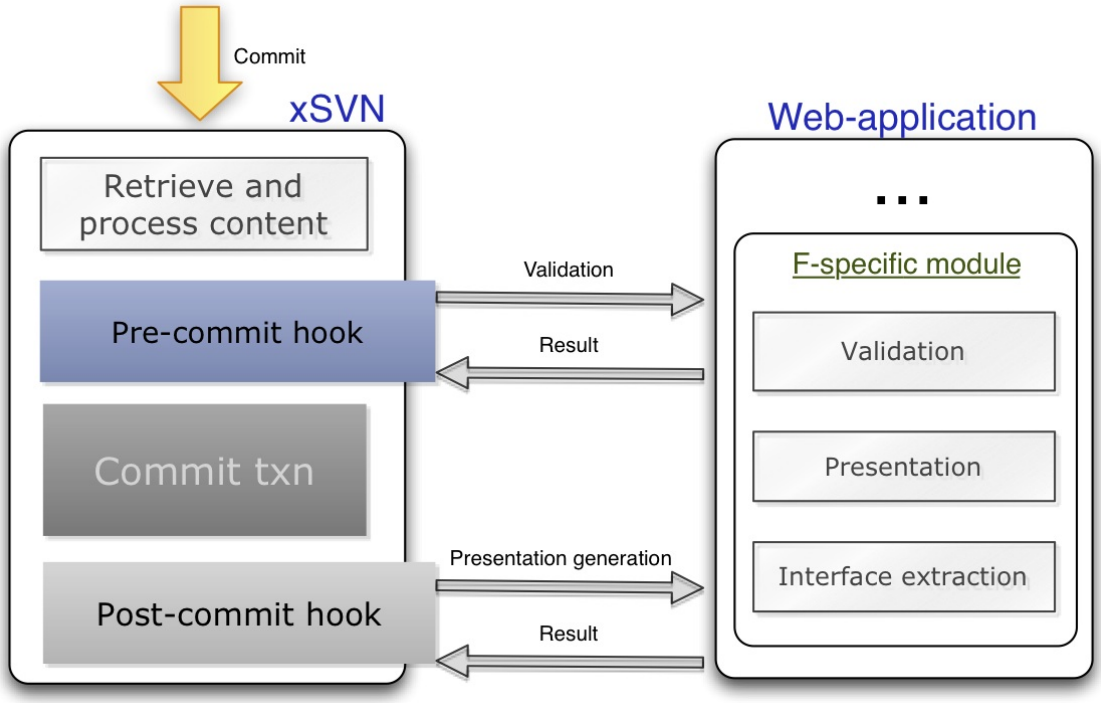


Figure 2. Interactions between xSVN and Web-application

Once this is done, a script sends requests to the TNTBase RESTful interface, where the actual processing is executed. The return codes and error stream are used to notify the committer about the result (e.g., in case of validation errors). For an example see Figure Interactions between xSVN and Web-application. This mechanism is surprisingly flexible and naturally fits into the xSVN approach since it is inherited from SVN. In the future, a tighter integration will add additional scalability and manageability, but our approach shows that that the workflows described in this paper naturally work inside the TNTBase architecture.

3. Validation and Interface Extraction

TNTBase provides a powerful schema and format-specific validation infrastructure. When committing a set of changes to TNTBase, a validation method is selected for every affected file, that validation is performed, and the xSVN transaction is rejected if validation fails. That ensures that only well-formed documents are stored in TNTBase. The validation method is selected per file via the **tntbase:validate** property (and thus can easily be switched off temporarily). Successful validation may also return extracted information about the committed file and its dependencies, e.g., a document format-specific index, to be stored by TNTBase for later querying.

An important property of TNTBase is that committing a change to a **tntbase:validate** property will result in automatic revalidation of all affected files. Thus, TNTBase guarantees that files with a certain **tntbase:validate** property are well-formed.

3.1. Selecting a Validation Method

Every directory or file may have the **tntbase:validate** property whose value is a string. If the property is absent, it is assumed to be empty. For a file its value is a string that defines the validation methods to be applied. For a folder the **tntbase:validate** property is a whitespace-separated list of strings.

If this list is of the form e_1, m_1, …, e_n, m_n, the every e_i is treated as a file name extension and every m_i as a validation method. The semantics is that files with the extension e_i are to be validated using method m_i. If the length of the list is odd, the last entry is treated as a default validation method that applies to any file with an extension that was not met before in the list.

When validating a file, the entry in its **tntbase:validate** property determines the validation method (or methods). If there is no matching entry, one is searched in the corresponding property of the parent directory, and so on until a matching entry is found. If the root directory is reached without finding an entry, the predefined validation method **none** is chosen, which does nothing and always succeeds.

3.2. Defining Validation Methods

Every TNTBase repository comes with a special top-level directory called **admin**. This folder contains all configuration files for that repository so that configurations can be easily made via SVN (and are automatically versioned themselves!). The configuration of validation behavior is done in the **process** subfolder. It may contain a file **methods.xml**, which defines validation methods. If it is not present, no methods are defined.

The content of this file is of the form

<methods>$M_1 \ldots M_n$</methods>

where every M_i is of one of the following forms:

```
<schema name="n" type="t" location="l"/>
<java name="n" class="c"/>
```

In the first case, validation is performed by schema validation. l is a URL giving the location of the schema, and t is its type: Currently, a user can choose between the **rnc** and **rng** for the text and XML syntaxes of RelaxNG schemata correspondingly. Other XML schema language are planned to be added in the future, e.g. Schematron [13]. Also, DTD and XML Schema validation is supported automatically by DB XML, and the schemas should be provided as XML files themselves. n is the name of the validation method that occurs in the **tntbase:validate** property.

While the first case, already permits simple validation through different XML schemata, the second case offers full customizability. Here, n is as for schema validation, and c gives the qualified name of the Java class implementing the validation interface. (See the project website http://tntbase.mathweb.org for documentation of the interface.) These implementation classes must be in the Java **classpath** provided when starting TNTBase.

3.3. Applying Validation and Interface Extraction

Every m_i in the **tntbase:validate** property is of the form $s_i?(+v_i)?$, where s_i is the name of an XML schema validation method and v_i is the name of a Java validation method. The latter may generate any serializable content. The intuition here is that the validation automatically produces information that is desirable to store for later use. These can be transformations of the committed file into human- or machine-readable formats such as XHTML or RDF.

These results are stored as XML documents in TNTBase and every such document is associated with a real XML file in TNTBase. They are not shown in the TNTBase file system and can be queried via a separate TNTBase interface. For that purpose, the **method** element from above receives one additional optional attribute **output**. If the value of this attribute is *true* TNTBase will cache the files containing the extracted information and will make them accessible via dedicated RESTful methods.

In particular, the output of validating the file **PATH** is available at http://tntbase.your.host.org/restful/integration/output/METHOD/PATH, and XQuery access to all output files is possible via http://tntbase.your.host.org/restful/integration/query/METHOD?query=XQUERY_EXPRESSION. In both cases, **METHOD** is the name of the validation method.

As an example, we give the current setup of the OMDoc repository used in the Latin project [8]. Here the top-level folder of the repository carries the property **tntbase:validate=omdoc mmt-rng+mmt**. This means that all files with the **omdoc**

extension are to be validated using first the **mmt-rng** and then the **mmt** method. A subfolder **inprogress** contains work in progress and carries the property **tntbase:validate=none** to avoid validation.

The **methods.xml** file looks as follows:

```
<methods>
  <schema name="mmt-rng" type="rnc"
          location="/var/www/tntbase/schemata/mmt.rnc" type="rnc"/>
  <java name="mmt"
          class="info.kwarc.jomdoc.frontend.TNTValidate" output="true"/>
</methods>
```

Thus, the method **mmt-rng+mmt** consists of calling first the schema validation using the schema **mmt.rnc**, followed by a custom java implementation in the class *TNTValidate*.

The extraction method of the class *TNTValidate* returns a set of RDF triples that represent the **omdoc** file according to the **omdoc** ontology. Because the value of the **output** attribute is **true**, the generated RDF theory (ABox) is accessible via the RESTful TNTBase interface. For example, the RDF triples of the file **math/algebra/algebra1.omdoc** are accessible as http://tntbase.your.host.org/restful/integration/output/mmt/math/algebra/algebra1.omdoc.

4. Virtual Documents

In this section, we will take a closer look at virtual documents: As they are surprisingly complex to understand, we introduce the concept and its realization by an example in Section 4.1 before we develop a precise terminology to capture all aspects of the concept: Section 4.2 and Section 4.3 develop a set of prerequisites for the document format to make use of virtual documents and show how they can be achieved generically.

4.1. Virtual Documents by Example

To fortify our intuition about virtual documents let us consider the following situation: We have a set of exercises with problem statements and solutions in our document collection. Say they are marked up in the following way:

```
<exercise>
  <problem>P</problem>
  <solution>S</solution>
</exercise>
```

Figure 3. An Exercise with Solution

Now we want to make them available in two forms: without solutions to students and with (master) solutions to the teaching assistants. In this situation, virtual documents are ideal. Generally, for a virtual document we specify an XML file like the one in Listing A Virtual Document for Practice Exercises. It consists of a skeleton and a list of external queries.

```
 1 <tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
 2   <tnt:skeleton xml:id="exercises">
 3     <omdoc xmlns="http://omdoc.org/ns" ▶
xmlns:dc="http://purl.org/DublinCore">
 4       <dc:title>Exercises for GenCS</dc:title>
 5       <dc:creator>Michael Kohlhase</dc:creator>
 6       <omdoc>
 7         <dc:title>Acknowledgements</dc:title>
 8         <omtext>The following individuals have contributed material to ▶
this document</omtext>
 9         <tnt:xqinclude ▶
query="distinct-values(collection(/gencs/*/*.omdoc)//dc:author)">
10           <tnt:return><omtext><tnt:result/></omtext></tnt:return>
11         </tnt:xqinclude>
12       </omdoc>
13       <omdoc>
14         <dc:title>SML Exercises</dc:title>
15         <tnt:xqinclude>
16           <tnt:query name="xq.SMLex"/>
17           <tnt:return><exercise><tnt:result/></exercise></tnt:return>    ▶

18         </tnt:xqinclude>
19       </omdoc>
20       <omdoc> …</omdoc>
21     </omdoc>
22   </tnt:skeleton>
23   <tnt:query name="xq.SMLex>
24     for $i in collection(/gencs/SML/*.omdoc)//exercise
25       return $i/*[local-name() ne 'solution']
26   </tnt:query>
27   …
28 </tnt:virtualdocument>
```

Figure 4. A Virtual Document for Practice Exercises

Conceptually, the skeleton consists of the top-level parts of the intended exercises document, where some document fragments have been replaced by embedded XQueries that generate them. In general, queries have the form

```
<tnt:xqinclude>
  <tnt:query>q</tnt:query>
  <tnt:return>R</tnt:result>
</tnt:xqinlcude>
```

Figure 5. A XQuery Fragment Reference

where q is an XQuery and R consists of a result expression where the **tnt:result** element will be replaced with the results of q. In line 7-9 we have such a query in a syntactic variant where

<tnt:xqinclude query="q">R</tnt:xqinclude>

was used to abbreviate the primary query form above for queries that do not contain embedded elements. The result of this particular query would be a list of author names wrapped in an **omtext** element. The query in lines 15-18 uses another useful syntactic feature: it refers to the query by reference to enable reuse and sharing for virtual documents (see below). [1] The result of this query would be a list of exercises of the form

```
<exercise>
  <problem>P</problem>
</exercise>
```

Note that the default and the **dc** namespace within the query are inherited from the dominating **omdoc** node.

If we want to edit a virtual document, then we should tell TNTBase to send it to us in a special editing mode. Then the relevant parts of our virtual document will be of the form

```
<exercise>
  <problem tnt:srcfile="/gencs/SML/prob1.omdoc"
           tnt:xpath="(omdoc/exercise/problem)[2]">P</problem>
</exercise>
```

Note that the result fragments have been decorated with **tnt:srcfile** and **tnt:xpath** attributes that specify the origin of the text fragments. These attributes are syntactical variants of the source references in virtual files (see [18]) and enable editability and transparent commit in the same way.

Thus, the virtual document in Listing A Virtual Document for Practice Exercises indeed expands to the desired exercises document after XQuery processing. In TNTBase(F), virtual documents are created simply by committing a *virtual document specification* (the XML file in Listing A Virtual Document for Practice Exercises) to

[1] In the future, we intend to integrate the XQuery module system into virtual documents so that individual XQuery function and variable declarations can be shared between queries.

the xSVN repository and executing an additional method of TNTBase that takes as an input the path of a virtual document specification and a path of a new virtual document itself. Thus, we can create multiple virtual documents based on a single virtual document specification. For instance, the above virtual document specification's XQueries can be changed so that they select problems in the directory where virtual document resides. This is done by *for $i in collection(./*.omdoc)//exercise return $i/*[local-name() ne 'solution']*, where the first "." in the query stands for the *current directory*. Thus we write a specification once, and may use it for creating virtual documents without solutions in different folders separated by topic.

The reference-based setup caters for a wide variety of reuse scenarios. For instance the document for the GenCS teaching assistants could be specified by the following virtual document specification, which reuses the skeleton from Listing A Virtual Document for Practice Exercises:

```
<tnt:virtualfile xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton href="../exercises.vf#exercises"/>
  <tnt:query name="xq.SMLex>
    for $i in collection(/gencs/SML/*.omdoc)//exercise return $i
  </tnt:query>
  …
</tnt:virtualfile>
```

Note that the new query does not remove the solutions. It is lexically captured by the named reference in the skeleton. One may think of method overloading in Java or C++. It is also possible to have an empty **tnt:query** element is the specification. Then it becomes an *abstract* specification, and can be compared to abstract methods in Java or pure virtual functions in C++.

4.2. Document Formats and Document Fragments

In the example above we have been very informal; to fully understand the concept of virtual documents we have to work a little harder and pin down the relevant concepts. We will provide the relevant definitions and explain them by examples that show the problems we have glossed over above.

Definition 1 (Document Format) . A *document format* consists of

- a formal language, whose words we call *documents*,
- a formal language of *fragments* over the same alphabet,
- for every document D a set of pairs (p, F), where F is a fragment occurring as a sub-word of D and p is some expression, called the *position* of F in D.

We write $D(p) = F$ to retrieve the fragment at position p.

The intuition of fragments is that fragments occur as components of documents. Document formats are typically given as a context-free grammar that subsumes the

language with some non-terminal symbols designated as the fragment-producing ones. For example, we define the format XML/XPath as follows: The documents are the XML documents, and the fragments of an XML document D are the element nodes occurring in D; the fragment positions are given by XPath expressions [1]. Another XML-based document format is XML/id: Here, only the nodes with **xml:id** attributes [10] are fragments, they are addressed by the values of of **xml:id** attributes which serve as positions. For text documents, we can use substrings as fragments together with line and column numbers as positions (or paragraphs and paragraph numbers if we use empty lines as paragraph separators). Furthermore, any formal language trivially becomes a document format if every document has itself as the only fragment.

Any particular XML language becomes a document format in the sense of Definition 1 (Document Format) by treating it as a special case of XML/XPath. However, it is often useful to define fragments differently, in particular to restrict the fragments to ones that carry self-contained meaning in the document format. After all, documents are composed of fragments that represent logical structuring units.

The OMDoc1.6 [2] format has been designed around the notion of "semantic fragment". These are represented by tags such as *omdoc* (used both for documents and document sections), *theory*, *symbol*, *axiom*, *notation*. Other element nodes, such as the ones representing individual mathematical expressions, are not considered fragments. In the OMDoc1.6 design the fragment nodes all have **name** attributes: Any element with a name yields a fragment, and any tag that is worthy of being extracted as a semantic fragment gets a name. Contrary to **xml:id**s, the **name** attributes have to be unique only within the scope given by their parent fragment, which gives rise to a hierarchical naming scheme that is more adequate for mathematical practice. For example, the relative URI **algebra.omdoc/groupoids?Monoid?comp** [3] refers to the declaration of the composition operation in the theory of monoids occurring in the section on groupoids in the document **algebra1.omdoc**. The general form of an OMDoc fragment position is $sec_1/.../sec_m?mod_1/.../mod_m/sym$, where the sec_i are section names, the mod_i are the names of nested modules in the OMDoc module system [12], and *sym* is a symbol name. Clearly, the set of fragments of an OMDoc document can be naturally included into the set of fragments of the same document considered as an XML/XPath document.

We use names instead of **xml:id**s as fragment positions because the latter have to be document-unique, whereas in virtual documents, we want to be able to flexibly

[2] The OMDoc1.6 format is the radically redesigned successor of OMDoc1.2 [5] which is the first draft of the OMDoc2 format. It is based on the semantic data and referencing model of [12], which makes it an interesting case study for our purposes here

[3] The use of an additional ? in the query is unusual but legal. It leads to a very compact notation, which is important in OMDoc.

recombine existing fragments into new documents. Already the next definition could lead to name clashes if we used **xml:id**s to identify fragments.

Definition 2 . We say that a document format F supports *reference expansion* if it provides

- for every document D in F, a distinguished set of fragments called *fragment references*,
- for every fragment reference R in F, a list $(d_i, p_i)\,|_{i=1}^{n}$, called *results*, where d_i is a document URI and p_i is a fragment position in the document referenced by d_i,
- an operation $exp(d, p)$ that given a URI d of an F document and a position p in it returns a document fragment in F,

such that D is semantically equivalent to the document that arises from replacing a fragment reference R with the concatenation $exp(d_i, p_i)\,|_{i=1}^{n}$. A document is called *fully expanded* if it contains no fragment references.

Technically, the precise definition of semantic equivalence must be given by the document format as an equivalence relation between documents. The right choice of the semantic equivalence relation can be quite difficult, and often different notions of equivalence must be employed in different contexts. For example, replacing relative URIs in an XHTML document with appropriately resolved absolute ones will usually not change the semantics of the document. However, in certain situations, it is still desirable to forbid such a transformation, e.g., when a directory of interlinked documents is often moved to different locations. For simplicity, we will not go into details and simply assume such an equivalence relation to be present.

The intuition behind reference expansion becomes clear if we define it for the document format XML/XPath from above: It becomes a document format XML/XPath/XInclude with reference expansion by using XML inclusions via XInclude [9], i.e., the fragment references are the **xi:include** elements. XInclude (essentially) requires that processors obtain the document fragment(s) referenced by the **href** and **xpointer** attributes of the **xi:include** element and replace the **xi:include** element with them, which we take as semantic equivalence. Thus, we can define that for every **xi:include** element, the result list is $(d, p_1), \ldots, (d, p_n)$ where d is given by the **href** attribute, $p_1, \ldots, p_n$ are the normalized XPath expressions identifying the nodes matched by the **xpointer** attribute.

At first it might seem that one should always put $exp(d, p) := D'(p)$ where D' is the document at URI d. However, it is useful to permit the document format to perform an arbitrary operation instead – typically this operation consists of computing $D'(p)$ and then applying some post-processing to it. For example, **xi:include** computes $exp(d, p)$ by adding an attribute **xml:base** to $D'(p)$ because $D'(p)$ might

inherit its **xml:base** from an ancestor node within D'. Similarly, XML namespace attributes inherited from an ancestor may have to be added to $D'(p)$.[4]

OMDoc1.6 integrates a custom syntax for reference expansion. Every tag designating a fragment may alternatively occur as an empty element node with an **href** attribute instead of a **name** attribute. The value of this attribute is of the form d/p where d is a URI without query or fragment and p is an OMDoc fragment position. The list of results always has length 1 and is given by (d, p). (Note that since both d and p may contain slashes, the computation of (d, p) has to be carried out by a server providing the resource d). To define reference expansion, let D' be the document at d. Then $exp(d, p)$ returns $D'(p)$ with an added attribute **base** whose value references the parent fragment of $D'(p)$ in D'. For example, assume the following OMDoc document D' at URI d:

```
<omdoc>
  <omdoc name="div_1">
    <theory name="th_1"><constant name="c_1"/></theory>
  </omdoc>
 </omdoc>
```

A document D may contain a fragment reference to the theory **th_1** in the section **div_1** of the form *<omdoc href="d/div_1?th_1"/>*. Then $exp(d, \texttt{div}_1 \,?\, \texttt{th}_1)$ yields

```
<theory name="th_1" base="d/div_1"><constant name="c_1"/></theory>
```

The added **base** attribute serves two purposes. Firstly, it functions as a base address relative to which all semantic references occurring inside **th_1** are resolved. (There are none in this example, but practical theories contain large numbers of mathematical expressions containing semantic references to theories and constants.) Secondly, it remembers the origin of the theory **th_1** so that a link between the theory defined in D' and its copy in the fully expanded version of D is maintained. This link is important for several applications, and below we will use it in particular to propagate changes made to **th_1** from D to D'.

Definition 3 (Document Contraction) . In a document format that supports reference expansion, we speak of *document contraction* if a document D is transformed into a semantically equivalent document D' by replacing a fragment of D with a fragment reference. A fragment that is not simply a fragment reference is called *fully contracted* if no further contraction is possible. A fully contracted fragment is called *atomic* if it is also fully expanded. S is called the *skeleton* of a document D if S

[4]The latter is usually not necessary for XInclude, which technically acts on info sets rather than documents.

arises from D by contracting exactly the atomic fragments. Finally, we say that a document format supports document contraction if every document has a skeleton. Both XML/XPath/XInclude and OMDoc support document contraction. However, for XML/XPath/XInclude, the atomic fragments are the empty elements, and thus the skeleton is not interesting because it is isomorphic to the original XML document. But for document formats with custom definitions of fragments such as OMDoc, the skeleton yields an interesting decomposition into medium-sized chunks.

A good example of a skeleton is the table of contents with every node representing one entry. This can also serve as a guiding intuition to define document formats: An XML node should count as a fragment if it could occur in a table of contents. This motivated our choice of fragments for a document format for DocBook: The skeleton of a DocBook document is just its table of contents. More precisely, we can say the following.

Theorem 1. *Let* F *be a document format arising from XML/XPath/XInclude by restricting the documents to a sub-language of XML and by restricting the fragments of a document to element nodes with certain tags. Further assume that the* F*-documents may contain* ***xi:include*** *fragments and that the specification of* F *imposes semantic equivalence under reference expansion as defined above for XML/XPath/XInclude. Then* F *supports reference expansion and document contraction.*

As an example, consider the left OMDoc document in Figure Contracting Documents with URI d. It can be contracted to the one on the right: Here all fragment references are relative to the base URI d, which receives a trailing slash so that, e.g., *div_1/div_1?theory_1* resolves to *d/div_1/div_1?theory_1*. Note how the contraction status is different between the fragments at position *div_1/div_1* and *div_1/div_2*. The former is skeletal, the latter is fully contracted. Such an OMDoc document typically arises after a fully contracted document has been served and the user has interactively expanded some of the fragment references.

4.3. Virtual Documents

Contraction permits to decompose a document into its fragments, and by expanding fragment references this process can be inverted. This is useful in itself, but it also opens the door to a powerful application: New documents can be composed partially or completely out of fragments from existing documents. If we follow this idea systematically, we can conceptually consider all documents to be skeletons that pick fragments from a shared reservoir. In particular, there is no conceptual distinction needed between the document in which a fragment is created and the documents, from which that fragments is referenced.

Virtual documents are simple as long as they are created or read but not changed. But assume a document D' that contains a reference to a fragment F of D, and assume

```
<omdoc base="d/">
  <omdoc name="div_1">
    <omdoc name="div_1>
      <theory name="th_1">
        <constant name="c_1">
          <type>T</type>
          <definition>D</definition>
        </constant>
      </theory>
    </omdoc>
    <omdoc name="div_2">
      <theory name="th_1">
        ...
      </theory>
      <theory name="th_2">
        ...
      </theory>
    </omdoc>
  </omdoc>
</omdoc>
```

```
<omdoc base="d/">
  <omdoc name="div_1">
    <omdoc name="div_1>
      <theory name="th_1">
        <constant
          ▶
href="div_1/div_1?th_1?c_1"/>


      </theory>
    </omdoc>
    <omdoc name="div_2">
      <theory ▶
href="div_1/div_2?th_1"/>


      <theory ▶
href="div_1/div_2?th_2"/>


    </omdoc>
  </omdoc>
</omdoc>
```

Figure 6. Contracting Documents

that the reference has been expanded. Further assume, we changed F to F' within this expanded version of D'. When these changes are committed, we have two options:

- The changes to F are propagated to D. D' is contracted to its original form, which means D' did not change at all.
- D' is stored in its expanded form, which means that a new atomic fragment is stored for F', and a new revision of D' is stored where the fragment reference now points to F'.

Clearly, both options are desirable under different circumstances. The following definition is strong enough to handle both cases.

Definition 4 (Virtual Documents) . *Assume a document format* F *such that 1) all documents are XML documents, 2) all fragments are XML element nodes, 3) reference expansion and document contraction are supported. We say that the format supports* virtual documents *if it provides a partial function contr(F') that takes a document fragment F'*

and (if defined) returns $(d,\ p,\ F)$ *such that* $contr(exp(d,\ p)) = (d,\ p,\ D(p))$ *where D is the document at URI d.*

The intuition is that $contr(F') = (d,\ p,\ F)$ holds iff F' arose from a reference expansion where $(d,\ p)$ is the URI-position pair used to identify the referenced fragment. In that case, F arises by inverting whatever modification $exp(d,\ p)$ made to $D(p)$. Then F represents a new version of F' that can be propagated to the document at URI d.

To see more clearly how this works, we give a document format VXML for virtual XML documents. We define it as follows:

- The documents are XML documents.
- The fragments are the XML element nodes with XPath expressions as their positions.
- The fragment references are of the form of Listing A XQuery Fragment Reference,

  where Q is an XQuery returning a list $(d_i,\ p_i)\,|_{i=1}^{n}$ of results, and R is arbitrary XML in particular using an element node <*tnt:result*/>. The namespaces of the query Q are inherited from the *tnt:query* node.
- For every result, the node $exp(d_i,\ p_i)$ is defined as follows:
  - Take the node list R and add to all top-level nodes an attribute <*tnt:virtual=″q″*> where q is the XPath expression of the *tnt:xqinclude* element. Let this yield L.
  - Let N_i be the node list arising from replacing the node <*tnt:result*/> in L with: the node $D_i(p_i)$ (including its in-scope namespace attributes) where D_i is the document at URI d_i with the following additional attributes:
    - an attribute *xml:base* as in the XInclude specification,
    - an attribute *tnt:added=″xml:base″* if an *xml:base* attribute was added,
    - the attributes *tnt:srcfile=″*d_i*″* and *tnt:xpath=″*p_i*″*.
  - Return the concatenation $N_1 \ldots N_n$.
- For every fragment F' the function $contr(F')$ is defined whenever F' has exactly one descendant node N with an attribute of the form *tnt:origin=″d#xpointer(p)″*; and in that case it returns $(d,\ p,\ F)$ where F arises from N by removing *tnt:origin* attribute and if applicable the *xml:base* attribute.

After retrieving VXML documents from TNTBase they can be dynamically expanded – e.g., by using an interactive browser or a TNTBase-supporting editor – in which case they will have *tnt:virtual=″q″* attributes. When committing the document, TNTBase (i) replaces all elements with such attributes with the original fragment

reference, which can be retrieved from the database using the XPath expression q, and (ii) propagates the respective changes to the origin document using the $contr(-)$ operation. This corresponds to option A above. The user can choose option B by removing the **tnt:virtual** attribute.

The above definition of VXML assumes that all results of the XQuery correspond to physical nodes in some documents rather than being constructed during the XQuery. This is a necessary prerequisite for the propagation of changes. In general, this may only hold for some result nodes, in which case only changes to those nodes can be propagated. TNTBase is able to handle that situation as well, but we avoid the description here for simplicity. Note that in our example above, the OMDoc format supports virtual documents because the **base** attribute mentioned in the definition of the OMDoc reference expansion determines the origin of a fragment.

5. Presentation: Compilation and Browsing

The presentation support of TNTBase is targeted at providing document format-specific browsing interfaces. We use the term *rendering* for the process of transforming documents into human-oriented presentation formats such as XHTML.

Rendering can be done at commit or at view time. Commit-time rendering is generally preferable because it makes viewing faster. However, it is not suitable for interactive editing of documents and for browsing virtual documents which are created on the fly using dynamically executed XQueries. Furthermore, the presentation of a document can be affected by changes in other documents, e.g., if a document contains fragment references that are expanded before rendering. Therefore, TNTBase supports both commit- and view-time rendering.

Both types of *rendering* are controlled by a configuration file **browsers.xml** in the folder **admin/viewing**. Its content is of the form

<browsers>$B_1 \ldots B_n$</browsers>

where every B_i is of one of the following forms:

<xslt name="n" location="l"/>

<java name="n" class="c"/>

The semantics is very similar to the one of the corresponding entries of the **methods.xml** file. The main difference is that n is not referenced in a subversion property but directly in a URI.

For *view-time rendering*, if *PATH* is a path in a repository with the URL *REPOS*, then the URL *REPOS*/*restful*/*presentation*/*render*/*n*/*PATH* yields the version of that file rendered using the method n. In particular, *PATH* need not refer to a file but can also contain queries that generate virtual documents on the fly. Currently two

types of rendering methods are supported: XSLT transformations and custom Java rendering. In the latter case, a user has to provide an implementation of certain TNTBase interfaces that return rendered documents.

For *commit-time rendering*, the situation is very similar. The main difference is that a user has to generate (via an ad-hoc TNTBase script) the post-commit hook that internally will invoke the necessary presentational caching method via the RESTful interface. When creating such a hook, the user has to provide the name of a browser, say *n*, that is declared in **browsers.xml**. Once this is set up and rendered documents have been generated at commit time, one uses the URL *REPOS/restful/presentation/view/n/PATH* to view the cached presentation. In this case not every *PATH* will yield a document because the commit could have occurred before the commit-time rendering was configured. Furthermore, it is possible to force a refresh of the cached presentation by retrieving the URL *REPOS/restful/presentation/cache/n/PATH*. (In fact, this is what the above-mentioned post-commit hook does.)

6. Conclusion and Future Work

We have presented TNTBase(*F*), a format-specific extension of the TNTBase system that allows the (web) application developer to simply configure common format-specific workflows like validation, compilation, and presentation within the TNTBase system. This considerably reduces the application logic of (web) applications that manage large, changing XML-based document collections. The TNTBase(*F*) framework is implemented as an extension of TNTBase and can be obtained from the project website http://trac.mathweb.org/tntbase. TNTBase(*F*) is under active development and the project website should be consulted for details and the evolving state of implementation.

To get a better intuition for the power of the TNTBase(*F*) framework, let us look at the presentation of an ontology in Figure A complex presentation of a documented Ontology. This example from [7] shows a heavily cross-referenced XHTML+MathML rendering of the underlying OMDoc sources, where much of the structural relations have been annotated in RDFa. The rendering and RDFa annotation process made use of the extracted interface information (see Section 3), which is then picked up by JOBAD [3], a JavaScript library that instruments the XHTML for interactivity. For instance, JOBAD can use CSS properties to collapse a proof or embedded MathML **maction** elements or pick a different (pre-generated) notation variant. But JOBAD can also use AJAX-style callbacks to the TNTBase(*OMDoc*) system for definition lookup: For any symbol a right-click menu item will query TNTBase for the definition of the corresponding constant, which will then be served by TNTBase(*OMDoc*) and displayed in a popup by JOBAD.

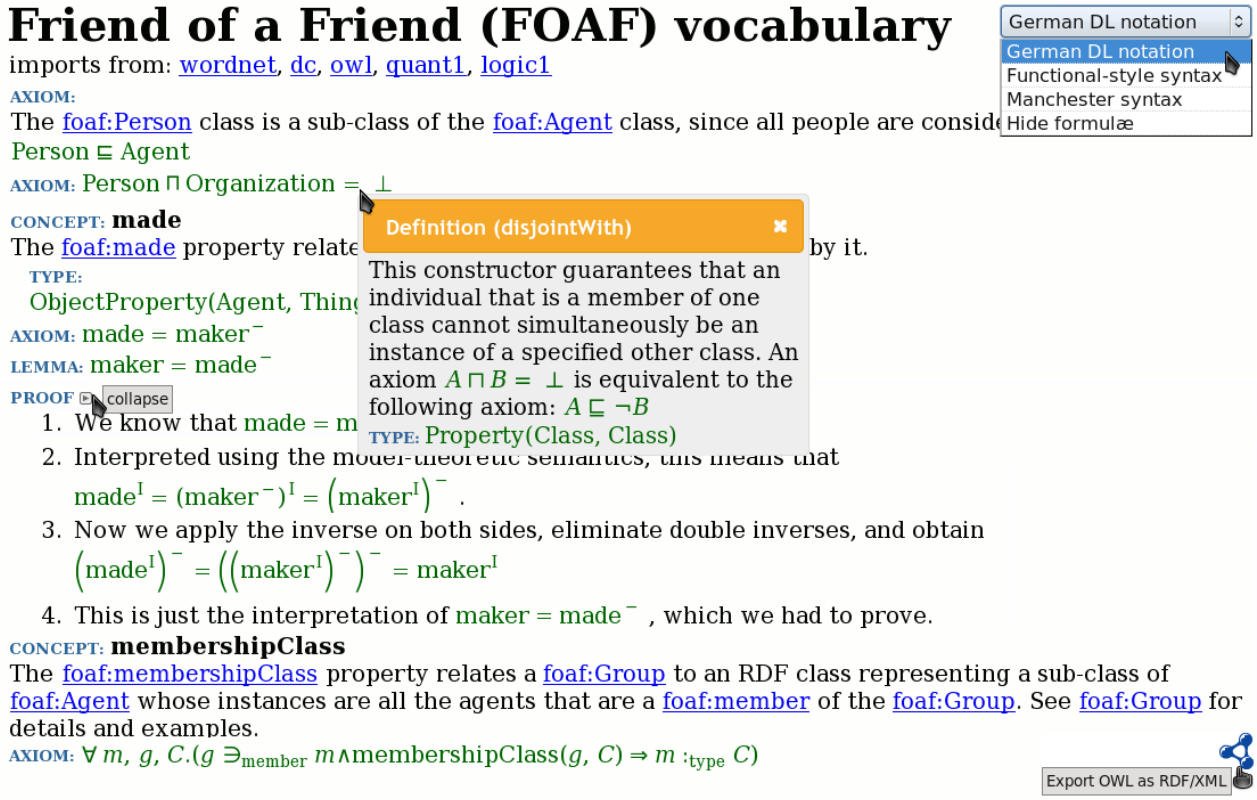


Figure 7. A complex presentation of a documented Ontology

This example uses only the techniques from Section 3 and Section 5: Pre-existing validation, annotation, rendering, and interaction functionalities were integrated into TNTBase(*F*), which also serves as a web application platform [4]. The work reported here was influenced by discussions of how to integrate TNTBase into the systems of the KWARC research group, in particular the SWiM system. The latter is an OMDoc-based semantic Wiki for scientific/technical documents [6],[7], in which workflows similar to the ones described in this paper had to be established for the integration of the OMDoc format into the underlying IkeWiki platform.

We are using TNTBase(*OMDoc*) in daily practice exploring the possibilities of TNTBase(*F*) for a semantic document format that is designed to support document aggregation and to stretch the limits of document modularization. The newly implemented virtual documents allow to aggregate fragments of the existing document base and access, edit, and commit them, much like regular documents, if the XML format supports document aggregation in the sense we have defined in this paper. We are also looking into the use of TNTBase(*OWL*) for ontologies, which is semantically structured document format, but does not (directly) support document aggregation with the intention of developing a suitable generic XML extension that generalizes XInclude [9] to queries.

Even in our limited experience with virtual files, they have turned out to be an enabling technology. Take for instance the example in Section 4.1. There we can

use the virtual document in Listing A Virtual Document for Practice Exercises to give students access to fragments of the original XML documents (the problems) while hiding others (the solutions[5]). Here, virtual documents promise to enable (some) fine-grained access control in TNTBase; we plan to study the consequences of this idea in the near future.

The next larger step in the development of TNTBase will be the introduction of distribution facilities for versioned XML document storage supporting both push and pull-based workflows. We hope to gain not only distributed document management functionalities for TNTBase, but also to offer offline capabilities for web applications, which can then simply integrate the TNTBase library for transparent caching. In such applications, the TNTBase content would take the function of a subversion working copy with the additional ability of offline commits.

References

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C recommendation, World Wide Web Consortium (W3C), January 2007.

[2] Berkeley DB XML, seen January 2009. available at http://www.oracle.com/database/berkeley-db/xml/index.html.

[3] Jana Giceva, Christoph Lange, and Florian Rabe. Integrating web services into active mathematical documents. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *MKM/Calculemus 2009 Proceedings*, number 5625 in LNAI, pages 279–293. Springer Verlag, 2009.

[4] Michael Kohlhase, Jana Giceva, Christoph Lange, and Vyacheslav Zholudev. Jobad – interactive mathematical documents. In Brigitte Endres-Niggemeyer, Valentin Zacharias, and Pascal Hitzler, editors, *AI Mashup Challenge 2009, KI Conference*, 2009.

[5] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[6] Christoph Lange. SWiM – a semantic wiki for mathematical knowledge management. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 832–837. Springer, 2008.

[7] Christoph Lange. *Semantic Web Collaboration on Semiformal Mathematical Knowledge*. PhD thesis, Jacobs University Bremen, 2010. submission expected in January 2010.

[5] Assuming of course that we have configured TNTBase's file-level authentication and authorization to restrict access to the course materials to instructors.

[8] Latin: Logic atlas and integrator. https://trac.omdoc.org/latin/.

[9] Jonathan Marsh, David Orchard, and Daniel Veillard. XML inclusions (XInclude) version 1.0 (second edition). W3C Recommendation, World Wide Web Consortium (W3C), November 2006.

[10] Jonathan Marsh, Daniel Veillard, and Norman Walsh. xml:id version 1.0. W3C recommendation, World Wide Web Consortium (W3C), September 2005.

[11] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2008.

[12] Florian Rabe and Michael Kohlhase. A web-scalable module system for mathematical theories. Manuscript, to be submitted to the Journal of Symbolic Computation, 2009.

[13] schematron. http://schematron.com.

[14] Tntbase trac. https://trac.mathweb.org/tntbase.

[15] Norman Walsh and Leonard Muellner. *DocBook 5.0: The Definitive Guide*. O'Reilly, 2008.

[16] XQuery: An XML Query Language, seen December 2007. available at http://www.w3.org/TR/xquery/.

[17] XQUpdate: XQuery Update Facility 1.0, seen February 2008. available at http://www.w3.org/TR/xquery-update-10/.

[18] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: a versioned storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

[19] Vyacheslav Zholudev, Michael Kohlhase, and Florian Rabe. A [insert xml format] database for [insert cool application] (extended version). Technical report, Jacobs University Bremen, 2010.

XQBench – A XQuery Benchmarking Service

Peter M. Fischer
ETH Zurich
<peter.fischer@inf.ethz.ch>

Abstract

Benchmarks have been a driving factor for acceptance and progress in the relational database area, as they gave researchers and engineers directions on the issues to tackle, and marketers the leverage to sell progress on these issues to customers. For more than two decades, standard benchmarks covering most application area of relational database have existed, with the most prominent example being the TPC suite.

XQuery has not (yet) reached this level of maturity. Benchmarks do exist for particular application scenarios (XMark, TPoX), microbenchmarking (MeMBer, The Michigan Benchmark), but most of the results shown by vendors and academics alike are specific to customers, implementations or usage scenarios, with little attempts to generalize them and compare over a wider range of implementations. In the first half of this decade, there was a short period in which academia showed interest in XQuery benchmarks, leading to series of proposals (XMark, XMach-1, XOO7, XBench). This interest has waned rather quickly, and industry has not really picked up the challenge (TPoX being the sole exception).

Instead of proposing another benchmark workload, we are working to provide a public service to run XQuery benchmarking workloads on a well-defined environment, including an installation of the most commonly used (open-source) XQuery implementation on a stable hardware and OS setting. By doing so, two main goals can be achieved: (1) Workloads can easily be compared on multiple implementations, leading to a broader coverage and applicability (2) A comprehensive collection of workloads can be established, highlighting performance aspects in particular areas and possibly leading to a general benchmark suite.

Currently, nearly all of the features of the benchmarking service are implemented. The service undergoes internal testing to ensure stability and correctness, and also some more work is needed on documentation. We expect the service to become publicly available around the time of the XML Prague workshop. Preliminary result have been established running XMark on Saxon B/HE, MonetDB, eXist, BerkeleyDB XML, Sedna, xQuilla and Zorba; we

also plan to run TPoX, MeMBer and some custom benchmarks until the general release.

The benchmarking service will be available on http://xqbench.org

Keywords: XQuery, Benchmark, Service

1. Benchmarking Service Requirements

The XQuery benchmarking service requires a system on which users can tests a set of queries and documents against a set of XQuery engines. Such as system needs to provide means for managing the queries and documents, specifying experiments, executing them reliably and reproducably and finally presenting and evaluating the results.

2. Data and Metadata Model

2.1. Overall Model

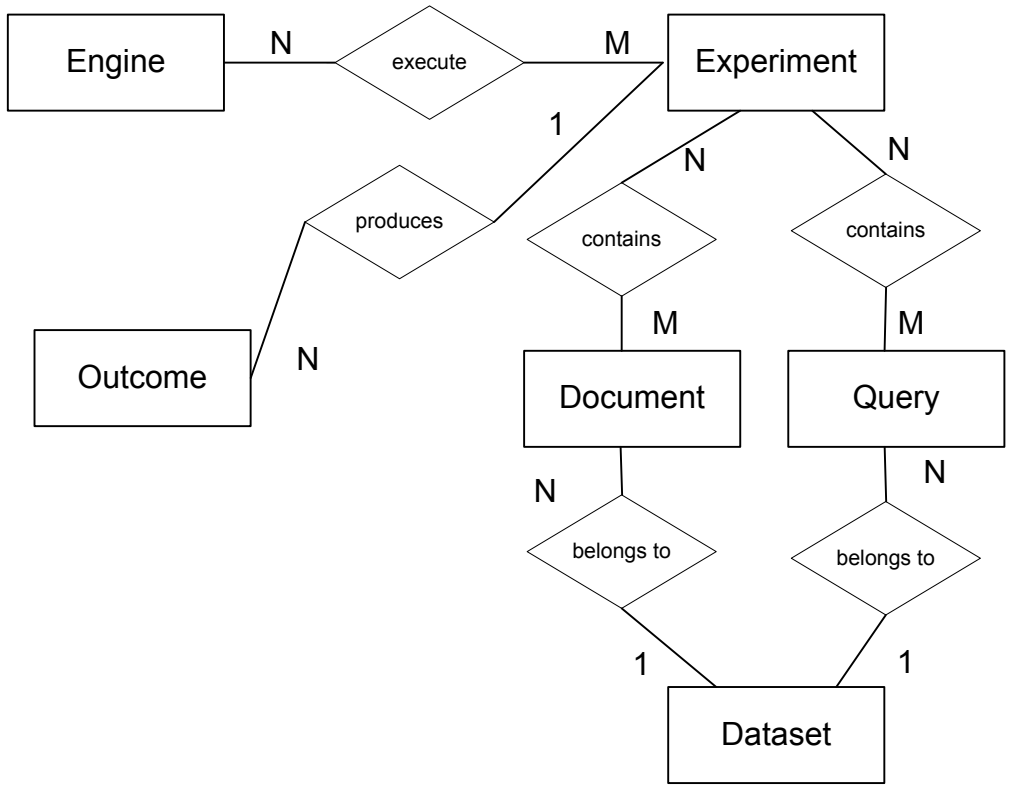


Figure 1. Overall Data Model of the Benchmarking Service

In order to maintain the data sets, specify experiments and evaluate the experiment results, extensive data and metadata is collected. Figure 1 shows the relationships in the data model behind the benchmarking service, based on the data model of XCheck[11]. An experiment contains documents and queries, which are executed on the documents. Documents and Queries belong to a Dataset specifation, (e.g. XMark), so that queries are only executed on matching documents. Experiments are executed on a set of XQuery engines, for every execution an outcome is produced.

In the following, the individual entities are explained on examples, the accompanying schemas are available on the benchmarking service.

2.2. Documents and Data Sets

Example 1. Sample document metadata

```
<documentEntries>
 <document>
   <docId>XMark_10.0</docId>
   <name>XMark 0.1 (1MB)</name>
   <dataset>XMark</dataset>
   <description>XMark , Scale Factor 0.1</description>
   <created>2008-06-23</created>
   <author>XMark</author>
   <generatorInfo>xmlgen.Linux -f 0.1 -o XMark_0.1.xml</generatorInfo>
   <file>XMark_0.1.xml</file>
   <size>10.0MB</size>
  </document>
 <document>
 ...
</documentEntries>
```

For a document, an identifier, a name and a description are saved. In addition, each document carries its dataset identifier, the date when the document was created and its author, the document generator information, the XML file name and the file size.

2.3. Queries

Example 2. Sample query metadata

```
<query>
    <name>XMark query 1</name>
    <id>XMark1</id>
    <descr>Return the name of the person with ID `person0'</descr>
    <dataset>XMark</dataset>
```

```
<created>2008-06-23</created>
<author>XMark</author>
<language>XQuery 1.0</language>
<categories>
    <cat>xpath</cat>
    <cat>flwor</cat>
    <cat>join</cat>
</categories>
<query file='query1.xq'>
    <![CDATA[
    let $auction := doc() return
    for $b in $auction/site/people/person[@id = "person0"]
     return $b/name/text()
    ]]>
    </query>
      <expectedresult document='XMark_0.01'>
        query1.xdm
      </expectedresult>
      </query>
```

For queries, again a name, an identifier and a description are stored, as well as the a creation date and the author. To further describe the properties of the query, the language requirements (XQuery 1.0) and the categorization in terms of the operations can be provided. Finally, the query text (both inline and as file reference) and a expected result (based on a given document) are to be provided.

2.4. Engines

Example 3. Sample engine metadata

```
<engines>
         <engine id="Zorba" type="xquery">
                <name>Zorba</name>
                <version>0.9.8</version>
                <language>XQuery 1.0</language>
                <language>XQuery 1.1</language>
                <language>XQuery Update 1.0</language>
                <language>XQuery Scripting 1.0</language>
                <homepage>http://www.zorba-xquery.com</homepage>
                <description>Zorba is a general purpose XQuery
                processor implementing in C++
                the W3C family of specifications.</description>
                <adapter>bin/CLAdapter.pl -e zorba-0.9.8.xml</adapter>
                <path>/local/zorba-0.9.8/build/bin</path>
                <cpu_time>n</cpu_time>
        </engine>
```

```
        ...
</engines>
```

For every XQuery implementation, there is an entry (with a unique identifier) that provides the name, the version, the supported languages, a home page and a description. To facilitate the actual execution of this "engine", information about the wrapper with the actual invocation information (called "adapter", see Example 7), and the install path in the file system are provided.

2.5. Experiments

Example 4. Sample Experiment Metadata

```
<experiment>
 <name>2010-01-28T12_45_59</name>
  <description>XCheck 2010-01-28T12_45_59</description>
 <engines>
 <engine>Zorba</engine><engine>XQuilla</engine></engines>
 <documents>
    <document id="XMark_10.0">
      <description>XMark , Scale Factor 0.1</description>
      <file>XMark0_1.xml</file>
    </document>
 </documents>
 <queries>
    <query id="q1">
      <description>Return the name of the
         person with ID `person0'</description>
      <filequery engine="all">
          XMark/query1.xq
      </filequery>
    </query>
 </queries>
 <groups>
 <group id="g0"><enginelist refs="all"/>
   <documentlist refs="XMark_10.0"/>
   <querylist refs="q1"/></group>
 </groups>
<customName>PaperExample</customName>
</experiment>
```

An experiment contains the internal name, a description, the list of engines (identified by the engine, see Section 2.4), the list of documents (identified by the docId, see Section 2.2) and the list of queries (identified by the query id, see Section 2.3). In addition to these identifers, explicit descriptions and file paths are given, in order

to make the evaluation more robust and transparent. Since slight variations in the syntax or semantics supported by various engines might require different versions of a query, alternative versions can be given, annotated with the engines on which they should be used. The total set of combinations of datasets, queries and engines can partitioned into groups, which are executed separately. This is useful for graphical evaluations (as to reduce the number of dimensions) and separating workloads containing different datasets.

2.6. Experiment Results

Example 5. Sample experiment result

```
<outcomes>
      <engine id="Zorba">
        <document id="XMark_10.0" size="11669705">
          <query id="q1">
            <result size="17171"></result>
            <doc_processing_time sd="0.000">0.000</doc_processing_time>
            <query_compile_time sd="0.001">0.015</query_compile_time>
            <query_exec_time sd="0.283">36.109</query_exec_time>
            <total_time sd="0.280">36.503</total_time>
          </query>
        </document>
      </engine>
      <engine id="XQilla">
        <document id="XMark_10.0" size="11669705">
          <query id="q1">
            <result size="17607"></result>
            <total_time sd="0.055">16.503</total_time>
          </query>
        </document>
      </engine>
      <info>
        <start>Thu Jan  28 12:46:44 2010</start>
        <finish>Thu Jan  28 13:20:32 2010</finish>
        <duration>2028</duration>
        <iterations>3</iterations>
        <cpu>AMD Opteron(tm) Processor 248</cpu>
        <ram>8240816 kB</ram>
        <system>Linux 2.6.18</system>
        <file_bench>/projects/xqbench/XCheck-0.2.0/experiments/
            2010-01-28T12_45_59/experiment.xml</file_bench>
        <save_results>false</save_results>
      </info>
    </outcomes>
```

Each execution of an experiments generates an outcome, either by successfully completing or by an unplanned termination. On success, the captured information is expressed as XML, HTML and a series of graphical plots. The XML file contains the complete information in a low-level, detailed form, whereas the other formats build on it and show a more human-readable and easy to understand presentation.

Given the execution strategy of XCheck, the results are grouped by engine, and then by document, each identified in the same way as in the experiment description. Document sizes, result sizes and total execution time will always be recorded, more fine-grained information like document loading time, query compilation time or query execution time only if the engine exposed it. In the example above, these times are available for Zorba, but not for XQuilla.

In addition to the measured outcomes, additional information on the execution circumstances are recorded, such as start and time, the number of iterations over which the results have been averaged, CPU, memory and operating system information and the file path of the experiment.

3. Benchmark Service Architecture

3.1. Overall Architecture

Given the long-running nature of many benchmarks, scalability is an import concern for the benchmarking service. As a result, the benchmarking service is split into a management frontend, repositories for both workloads and experiment results and a collection of backend nodes that perform the actual execution of the experiments. Figure 2 shows this architecture and the interactions of the components. The repositories and the frontend are fairly tightly coupled, whereas there is a more loose coupling among the backend nodes and among the backend and the rest of the system:

- Each node contains a (partial) replica of the XML documents and the queries needed for the experiments. This eliminites access delay and contention when accessing the main repository, thus making the experiments more predictable. In addition, changes in the repository (such new documents or modified queries) can be carried out from the frontend immediately. Before a new experiment is running on a backend node, synchronization will to reflect the updates on the repository.
- On submission, experiments are queued until a suitable execution backend becomes available. There are several reasons for that; (1) Since measuring the execution time should be as precise as possible, all competing accesses on a backend should be blocked until the currently running experiment ends, either by completion or timeout. (2) If failures like machine crashes occur, an experiment can easily be restarted, as all the relevant information is in the queue.

- On completion of an experiment (whether successful or not), the results are propagated to the repositoy of results. As a consequence, backend nodes can be removed without losing the experiments run on them. Search operations on the results will also not affect the backend

Before describing each of the components in more detail, the role of monitoring and notification should be clarified: To provide more control and transparency over the execution state of experiments, system performance parameters and service availability are monitored: collectd [9] records performance parameters such CPU or RAM usage, while Nagios [10] is used to check node and process activity.

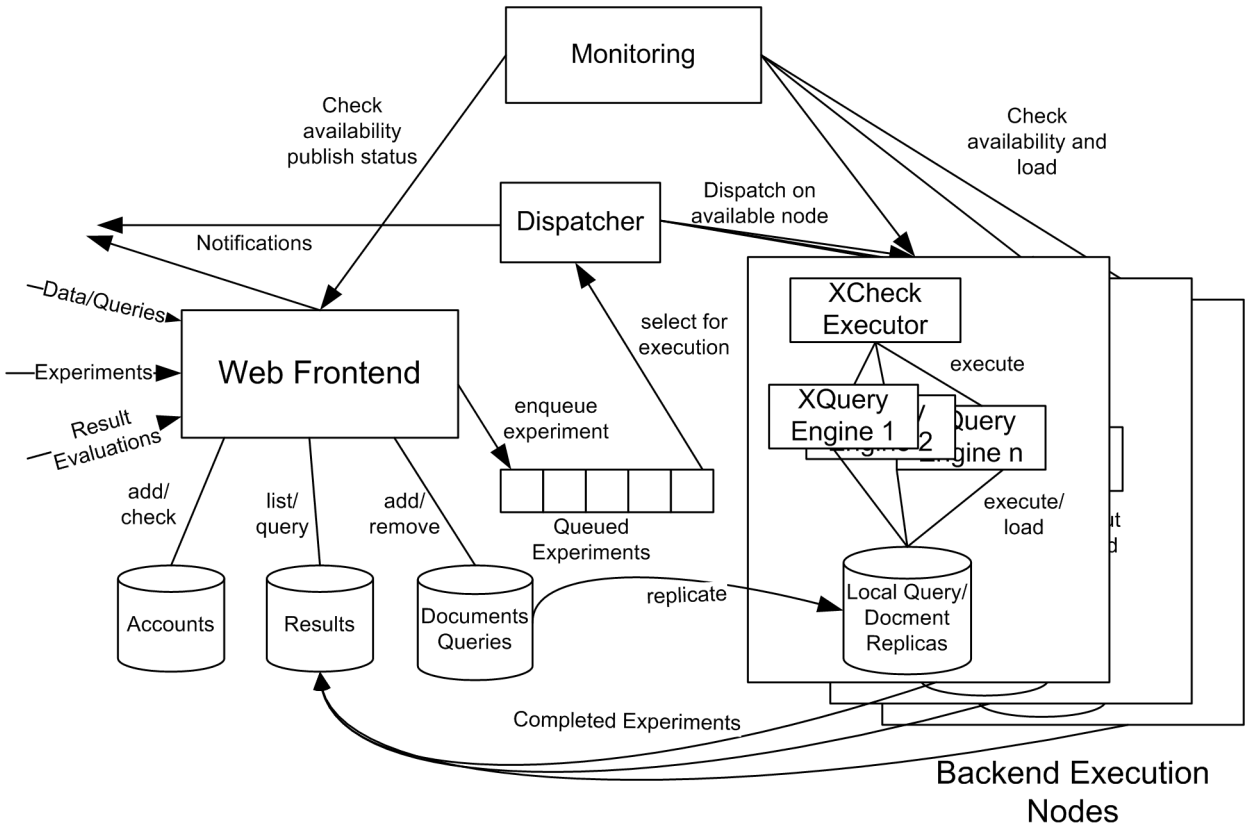


Figure 2. Benchmarking Service Architecture

3.2. Web-Based Frontend

The benchmarking service provides a web frontend to give users a convenient way of interaction. It provides features to

- manage user registrations and permissions
- upload and manage datasets and queries
- submit experiments
- show, search and evaluate the outcomes

A more detailed overview on the interaction and usage is presented in Section 4. Technically, the frontend is realized as a JSP/Servlet invoking an XQuery engine (MXQuery) for the data management and UI rendering. Authentication and user managment are handled by the JSP/Servlet container, storing the account data and user roles in a relational database.

3.3. Data and Results Repository

Since all data, metadata and result have an XML format, they can be stored as such. Queries and expected results may not be in XML format, but have at least a textual representation. To provide easy storage scalability, backup and replication, all data items are stored in the file system, using a folder hierarchy. All modifications, searches and transformations are performed using the file-based XQuery operations of MXQuery; replication and result consolidation using rsync and a set of shell scripts. In the long run, it might be suitable to load metadata as well as experiment and result data into a native XML database, as to benefit from indexing for faster search.

3.4. Workload Distribution and Monitoring

For a small set of backend execution nodes, a simple dispatching approach has been chosen: All pending experiments are written into a shared filesystem. Each backend executor checks these entries, and picks the first experiment that has not been picked up by another node by placing a locking. After some waiting period (as to prevent other node to also put a lock there), it will actually start the execution.

Monitoring utilizes the existing host monitoring infrastructure already in place at ETH. On the backend execution nodes and also the the web frontend node, data collection and status checking processes are run, which deliver their information into the monitoring and notification backends.

3.5. XCheck Experiment Executor

The benchmarking service builds on XCheck [11] as execution backend, which provides the functionality of running a set of queries on multiple implementations, collecting the results and providing graphical plots. XCheck is an open-source Perl script collection developed at the University of Amsterdam. It takes an experiment file as input, and executes the workloads as follows

Example 6. XCheck execution strategy

```
for $engine in ENGINES
   for $doc in DOCUMENT
       for $query in QUERIES
```

```
let $res := (for numrepetitions
  execute($engine,$doc,$query))
return average-values(fn:subsequence($res,2))
```

The averaged values and execution information are then put into an XML files, rendered into HTML with some simpler graphs and also into some more complex, standalone graphs.

By repeating the execution, and leaving out the first run, the effect of a "cold" system is minimized, and more reliable results are provided. In addition to averaging, XCheck provides a number of features to make experiment execution more stable, including timeouts to stop "runaway" experiments. For an XQuery engine to be executed from XCheck it requires a wrapper, called "adapter" that indicates how to execute the engine, how to interpret the results, etc.

Example 7. Example XCheck Adapter for Zorba

```
<adapter>
  <engine id="Zorba">
    <command>
      <executable><![CDATA[zorba -o #result -t #query >& #times]]>
      </executable>
      <file_query>y</file_query>
      <fullpath_doc>y</fullpath_doc>
    </command>
          <times>
            <factor_time>0.001</factor_time>
            <time id="t1">
              <line>Compilation time: (\d+) milliseconds</line>
            </time>
            <time id="t2">
              <line/>
            </time>
            <time id="t3">
              <line>Execution time: (\d+) milliseconds</line>
            </time>
            <doc_processing_time>0</doc_processing_time>
            <query_compile_time>#t1</query_compile_time>
            <query_exec_time>#t3</query_exec_time>
          </times>
      <error>Error on|Fatal error</error>
   </engine>
</adapter>
```

Adapters are again XML files, describing the command line with the parameters, possible setup and teardown operations, pattern matching and scaling factors for

different times, and patterns to detect errors in the output. XCheck will call the command line tool of the XQuery implementation according to the parameters given in the adapter, redirect the output, and parse relevant time and error information.

4. Workflow and Usage Guide

4.1. Public and Private Operations

Datasets and queries for common benchmarks are preinstalled, and benchmark "experiment" results on them are made publicly available. Users can upload their own datasets and queries, run experiments on both their own datasets and queries as well the publicly available ones. By default the data sets, queries and results contributed by users are private, but they can easily be made public, thus becoming usable and searchable by everybody. By making this distinction, it allows users to perform the experiments without disclosing them to other users or the general public; obviously they need to disclose their data and queries to the benchmarking service. From a benchmarking service point of view, knowing the users simplifies the operations, since every uploaded data and query can be traced back to an owner.

4.2. Experiment Creation and Submission

As a first step to test XQuery engines for a set of queries and documents, the user needs to create an experiment. The GUI provides a choice of XQuery engines, datasets with documents and queries. For all of these, the metadata is directly available; queries and document can also be completely downloaded. The user can choose documents, queries and engines, and provide a name for the experiment. In addition, he/she can specify at which stages of the experiment execution notifications should be sent:

- On Submission
- On Execution Start
- On Execution Completion

A registered user can upload his/her own data sets and queries, non-registered users are restricted to only choose the publicly available ones. After the user submits his/her workload, an experiment data set according to Section 2.4 is created, with a timestamp-based identifier in addition to the user-defined name. If the experiment contains multiple datasets (e.g. XMark and TPoX), execution groups for each dataset are created, so that XMark queries are only run on XMark documents, etc.

Since the execution of an experiment is the cross-product of the number of engines, number of documents, number of queries and number of repetitions, a single

experiment could easily run for weeks or months, if specified too broadly. In order not block the execution backend nodes for too long, the experiment is checked for its maximum runtime, based on the cross-product mentioned before and the timeout specified in XCheck for a single query execution. If this maximum runtime exceeds a threshold, the experiment is rejected. This threshold is set higher for registered users, as to provide more freedom for them, whereas "public" experiments should have lower priority. In addition to this limit on a single experiment, there is also a limit on the number of outstanding experiments for each registered users or for all non-registered - again to prevent overloading the service.

4.3. Result Presentation and Search

The execuction of an experiment in XCheck captures the run times of the individual engines (split into document loading, query compilation and query execution, where available) as well as result sizes. These results represented as HTML, XML and in a set of graphs. This result is transferred into the result repository, from which a user can access or view the experiment results. A notification is sent to the user at the end of the experiment, providing a direct link to these results. In addition, a user can view all his/her experiment results, and also can view the public experiments. The GUI lists all experiment and also provides a filter option on this list along with a general XQuery support.

- List of Experiments, completed and outstanding: The GUI lists all the experiments that have been submitted by the user in the past. Hence, an experiment is added to the list of all experiments, after its submission by the user. However, experiment results are only available once the experiment is completely executed and the results have been transferred to the repository.
- The list of all experiments can be filtered by expressing simple constraints using the GUI, which are treated as a conjunction. This selection is based on the user selection of XQuery engines, documents and queries, similar to the way how an experiment is built. If, for example, a user selects the Zorba and XQilla engines, the XMark 10MB document and the XMark 1 query, all those experiments would be listed that contain the Zorba and XQilla engines, the XMark 10MB document and the XMark 1 query. Additional engines, documents and queries are possible, but the listed ones need to be present.
- General query over the experiments and results: Since the simple filter above is not sufficient for more complex evaluations, the benchmarking service provides the possibility to search the experiment descriptions and results using XQuery. An example for such a complex evaluation would be searching for all queries in which Zorba beats XQuilla on one document, but loses on another. Since the experiment description and the results are both in XML format, the GUI provides a method to express full XQuery statements over the collections of experiments

and outcomes (limited to the user's and the public ones). A join between descriptions and outcomes is already provided, so that the query writer can focus on the actually relevant conditions. The results is provided as a direct XML download, which can be further evaluated locally.

4.4. Managing Documents and Queries

The Benchmarking service provides a number of preloaded public datasets and queries, among them XMark and TPoX. However, the more interesting use case is to add custom documents and queries, as to test them over a variety of implementation under controlled circumstances. Several different actions may need to be performed:

- Adding schemas for new XML datasets: For every document uploaded onto the benchmarking service, it is required to provide a data set, which is expressed as an identifier and a schema. The schema helps in checking the uploading XML document conformance with the schema, and also it prevents query belonging to one schema to run on documents having different structure.
- Adding a new document: To add a new document to the benchmarking service, the user needs to upload the document from the GUI, along with providing the details about the XML document, like schema (or dataset), name, description, etc. Only after validation it will be added to the list of documents that can be used for benchmarking. Due to the limitations of current browsers, files bigger than 2GB cannot be reliably uploaded. In such cases, manual administrator intervention is needed. For the long run, it is planned to include a document generation service, e.g. based on ToXGene[8], in which just the document generator input is uploaded.
- Adding new queries: When adding a new query to the benchmarking service, user need to provide a data set to which this query belongs, metadata information like the language features the query requires (update, schema), metadata on the operations used (path steps, joins), expected results, and finally, the actual query.

4.5. Providing Reproducability Information

An important factor in benchmarking is traceability and reproducibility. Several steps are taken to ensure them in XQBench: (1) the installation and configuration logs for all XQuery implementations on XQBench can be downloaded. (2) "Adapter" files to integrate the implementations into XCheck are linked from the implementation descriptions. (3) Data set and queries for particular experiments can be downloaded directly from the results; in the same way error messages. (4) Users are notified on the execution and completion of their benchmark experiment, and can follow the benchmark system load behavior, when required.

5. Preliminary Results and Status

Currently, nearly all of the features of the benchmarking service are implemented. The service undergoes internal testing to ensure stability and correctness, and also some more work is needed on documentation. We expect the service to become publicly available around the time of the XML Prague workshop.

Preliminary result have been established running XMark on Saxon B, MonetDB, eXist, BerkeleyDB XML, Sedna, xQuilla and Zorba. Several commercial XML database have also been tested, but since the licensing terms disallow publishing any results without permissions, we are currently not making these implementations available.

We also plan to run TPoX, MeMBer and some custom benchmarks until the general release.

6. Conclusion and Future Work

XQBench provides a general XQuery benchmarking service, allowing to execute standard and custom workloads over a well-defined set of XQuery implementations on controlled hardware. The results can be searched and downloaded as needed. The goal is to make XQuery benchmarking a much easier undetaking, thereby enabling vendors and users alike to understanding performance tradeoffs and advance the field.

Future work will focus on integrating more XQuery engines, broadening the base of benchmarks, and enciting users to contribute their own workloads. An important conceptual issue is to address the limits of XCheck, as it does not really deal will with the different interaction models of different XQuery implementations: the repeated invocation of a command line tool does not really cater for virtual machine startup cost or just-in-time compilation benefits, neglecting the benefits of repetition to such cases. Similarly, preloading or not preloading XML data into database-style implementations make give a huge performance difference, and needs to be expressed properly when running measurements.

Bibliography

[1] Afanasiev L., Manolescu I., and Michiels P. MemBeR: A Micro-benchmark Repository for XQuery Proceedings of the Third International XML Database Symposium, XSym 2005, Trondheim, Norway, August 28-29, 2005

[2] Böhme T. and Rahm E. XMach-1: a benchmark for XML data management In Proceedings of the German Database Conference BTW2001. Springer, Berlin, 2001, pp. 264–273

[3] Bressan S., Lee M, Li Y., Lacroix Z., and Nambiar U. The XOO7 Benchmark Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop

DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers

[4] Nicola M., Kogan I., and Schiefer B. An XML transaction processing benchmark In Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, 2007, pp. 937–948

[5] Runapongsa K., Patel J.M., Jagadish H.V., Chen Y., and Al-Khalifa S. The Michigan benchmark: towards XML query performance diagnostics Inf. Syst., 31(2):73–97 (2006)

[6] Schmidt A., Waas F., Kersten M.L., Carey M.J., Manolescu I.,Busse R.: XMark: a benchmark for XML data management In Proceedings of the Very Large Database Conference: 974–985 (2002)

[7] Yao B.B., Özsu M.T., and Khandelwal N.: XBench Benchmark and Performance Testing of XML DBMSs ICDE 2004:621–633

[8] Denilson Barbosa, Alberto Mendelzon, John Keenleyside and Kelly Lyons. ToXgene: A template-based data generator for XML SIGMOD 2002 http://www.cs.toronto.edu/tox/toxgene/

[9] collectd – The system statistics collection daemon http://collectd.org/

[10] Nagios http://www.nagios.org/

[11] XCheck - A Platform for Benchmarking XML Query Processors http://ilps.science.uva.nl/Resources/XCheck

Demos

XQuery in the Cloud

Matthias Brantner
28msec Inc.
<matthias.brantner@28msec.com>

Daniela Florescu
Oracle
<dana.florescu@oracle.com>

Donald Kossmann
28msec & ETH Zurich
<donaldk@inf.ethz.ch>

This demo (or talk) shows how two key technologies can be combined in order to provide a novel breed of scalable information processing systems and architectures: (a) XQuery and (b) Cloud Computing. Using examples, we show how several classes of applications can be supported well in this way:

- (Enterprise) Web Mashups; e.g.,
- Workflow management; e.g., a reviewing / due dilligence support system
- Analytics; e.g., correlating Twitter and Facebook events with stock market time series
- Data integration; e.g., integrating internal data from a Human Resource database with open social data from Twitter, Facebook, LinkedIn etc.

Lately, a series of different proposals have been made in order to carry out information processing in the cloud: Microsoft Azure relies on the .NET languages, Google AppEngine supports Python and a simplified sub-set of Java and SQL, and Force.com provides a proprietary language. We will show why XQuery is an ideal candidate to develop and deploy applications in the cloud. First, XQuery is a family of standards of the W3C. Second XQuery is extremely powerful with support for database queries, application logic, full-text search, and web services. Third, XQuery has a simple and yet flexible data model: It supports structured, semi-structured, and unstructured data. Furthermore, instances of the XQuery data model can be implemented efficiently in a cloud computing infrastructure (e.g., S3 or any other key-value store). Fourth, XQuery is a functional programming language which can automatically be optimized and parallelized. This property is particular important in cloud computing infrastructures. Last but not least, XQuery is unified framework for all tiers; database and application logic. This way, duplicate work at different levels (e.g., caching, logging, security) is avoided and wasted work for data marshalling is avoided, too. This property allows to provide a single-tiered application

and database architecture. Such an architecture is again particularly important in cloud computing scenarios because virtualization amplifies inefficiencies in the exsting multi-tier architectures.

The demo (or talk) will detail a single-tiered cloud computing architecture based on XQuery and will show this architecture at work with the example application listed above. To this end, we will use the open-source Zorba XQuery engine installed on top of the Amazon Web Services cloud computing infrastructure as provided by 28msec Inc.

XQuery Development Tools (XQDT)

Adding an important missing piece in the Eclipse XML support

Gabriel Petrovay
ETH Zurich
<gabriel.petrovay@inf.ethz.ch>

William Candillon
ETH Zurich
<william.candillon@inf.ethz.ch>

1. Introduction

This short article briefly presents the XQuery Development Tools (XQDT), the open-source Eclipse plug-ins for XQuery/XPath development. XQuery is the standard W3C data processing language built specially for XML and XML Schema, and is a superset of the first XML simple query language, XPath. As a result of the ubiquitous presence of XML in the Web 2.0 infrastructure (e.g. government, military, financial, digital content, media), XQuery gained recently significant attention in the industry, being implemented by open source consortia, startups, and large database vendors alike. Moreover it is being used by customers in various scenarios ranging (among others) from document search, content re-purposing, analytics, message processing, queries on variable structure datasets, RSS filtering, to building entire Web 2.0 applications using XQuery. The latter scenario is currently called "the XML end-to-end" architecture.

2. Motivation

Until 2009, the open-source comunity was lacking support for XQuery tools. XQDT, an Eclipse-based implementation, came to fill this slot. Moreover, XQDT has recently joined Eclipse. Thus, XQDT is adding another important missing piece in the Eclipse XML puzzle as part of the Web Tools Platform Project.

Most of the existing XQuery editors support only the XQuery 1.0 W3C standard. Both early adopters and the research community need also support for the emerging complementary language extensions, currently in the process of standardization by the W3C (XQuery Update Facility 1.0, XQuery Scripting Extensions 1.0, and XQuery 1.1). XQDT not only provides support for these extensions, but also stays upto-date with the most recent changes in these W3C working drafts.

3. XQDT Architecture

XQDT is built as a set of Eclipse plug-ins that provide both basic XQuery functionality and extension points for different vendor specific implementations. XQDT currently comes with built-in support for Xquery processors like Zorba XQuery processor (FLWOR Foundation), Sausalito XQuery Web Application Framework (28msec), or MarkLogic Server (MarkLogic).

The XQDT UI plug-ins provide the user interface components. One of the goals of XQDT is to make the user experience as consistent as possible with the other Eclipse-based language tools. The Eclipse workspace is divided into projects. XQDT provides two XQuery-based project types:

- XQuery Project: for generic XQuery editing and running;
- Sausalito Project: for projects run with the Sauslito XQuery Web Application Framework.

Currently, the full XQuery functionality is only available inside projects having one of the two types listed above.

The main UI components of XQDT are a rich-feature XQuery Editor, an outline view, XQuery preference pages, and UI support for XQuery debugging. All the XQDT UI component are gathered together for easy access in the XQuery Perspective.

The UI plug-ins are built on top of the XQDT Core plug-ins that provide among others, URI resolvers, the XQuery parser. The abstract syntax tree generated after the code parsing is used to build the internal XQuery model. This is used for many core operations like pletion or syntax highlighting.

The whole XQDT plug-in stack is built on top of the Dynamic Languages Toolkit (DLTK) that is shipped with Eclipse. DLTK provides template components that can be used for building programming language support plug-ins in Eclipse.

Besides this plug-in dependency stack, XQDT is also integrating tools available from the WTP project (see Section 5). These tools are use both to leverage existing functionality inside the WTP project or to build the infrastructure for future developments of XQDT.

4. XQDT features

Below we briefly describe some of the most important features of XQDT. Several attached screenshots also depict some of these features.

4.1. Syntax Highlighting

Because XQuery is not a keyword language, syntax highlighting is not a trivial problem. XQDT solves it using a mixtures of static and semantic rules. The static rules determine an approximate local highlighting as you type. An asynchronous

process that parses the source code provide input for the semantic rules. After the semantic rules are applied, the syntax highlighting is complete and correct. The syntax highlighting can be configured to work with different XQuery language extensions.

4.2. Code Completion

The code completion in XQDT can be split into several categories:

- "keyword" completion,
- code templates,
- function completion.

4.2.1. "Keyword" Completion

The words that might have a special syntactical meaning are proposed in XQDT as "keywords". They are also adaptable to the XQuery language extension used.

4.2.2. Code Templates

The complexity of the XQuery language can be overcome by using the extensible set of code templates for both standards and language extensions that XQDT provides. The code templates are also context aware and provide completion proposals for several parts of the template. This is needed for example when a template needs to be populated with the in-scope variables, or a module import statement that provides suggestion about the available built-in or other user library modules.

4.2.3. Function Completion

XQDT provides a complex mechanism for function completion. Several categories of functions are taken into account for this process:

- local functions - these are function defined in the same module where the user triggers the function completion;
- imported user functions - these are function imported from other user library modules;
- built-in functions - these are functions provided by a specific XQuery processor. We have built a flexible extension mechanism to allow plugging in different XQuery processors.

4.3. Error Reporting

4.3.1. Syntax Errors

As-you-type syntax validation is performed and the problems are immediately reported both in the XQuery Editor and the Problems View. Using an ANTLR- based parser, XQDT is able to report more than one error in the same XQuery source code file.

4.3.2. Semantic Errors

XQDT provides the interfaces for vendors to enhance the editing experience. By implementing a semantic code checker, XQuery processors can perform advanced checks, like: in-scope variables and functions, invalid URI syntax, etc.

4.4. XQuery Interpreters

The XQuery interpreters are XQuery processors that can be plugged in XQDT to provide running and debugging capabilities. The XQuery interpreters can also provide extended functionality, like the semantic code checker for the XQuery editor, or specific URI resolving for things like code completion. Currently, XQDT has support for the following processors:

- Zorba XQuery processor;
- Sausalito Web application framework;
- MarkLogic Server;
- Java-based XQuery processors like MXQuery, Saxon, etc.

4.5. Support for Language Extensions

XQDT provides support for both XQuery W3C standards (XQuery 1.0) and working drafts for different language extensions (XQuery Update Facility 1.0, XQuery Scripting Extensions 1.0, and XQuery 1.1). Moreover, XQDT supports also vendor specific language extensions like Zorba's "eval" expression, for example.

4.6. XQDoc Generation

Vendors of specific XQuery interpreters are invited to contribute "XQDoc generators". One such generator is provided for the Zorba interpreter type. Having well formated and proper documented XQuery code, the user can With just a few clicks generate XHTML documentation for the selected modules.

4.7. XQuery Running and Debugging

XQDT provides support for running and debugging XQuery code. While the running support is already built into the interpreters and is available out-of- the-box, the debugging support needs some programming effort from vendors to support their XQuery processors in XQDT. There are a couple of debugging protocols that can be used:

- DBGP (Debug Protocol): a generic debugging protocol with built in support in DLTK;
- Zorba Debug Protocol: a vendor specific protocol that can be used through code reuse for other processors as well.

The debugging tools allow users to use breakpoint for stopping the execution at certain locations, to inspect the in-scope variables and to evaluate random XQuery expresions.

4.8. Other Features

There are plenty of other features built in XQDT, but a complete list is out of the scope of this short article. These include things like, supports for XQDoc Comments, XQDoc documentation generation, code folding, template source code files, etc.

5. Project Status

XQDT is an active incubating project inside the Eclipse Web Tools Platform (WTP) Project. Having recently joined the WTP project our main focus is to integrate our work with the existing infrastructure WTP provides. Therefore, we are currently focused to integrate the following components into XQDT:

- XML editor
- XML Schema Editor
- XML code completion
- XML views

In parallel, we are also working on exporting more functionalities provided by the DLTK infrastructure like:

- code search and refactoring
- code navigation

For all these issues XQDT would benefit a lot from external contributions. Those can be made either by submitting patches through the Eclipse Bugzilla tracking system, or by becoming a committer in the XQDT project. The committer status can be taken into consideration for large contributions.

6. Conclusions

XQDT brought the first open-source XQuery editor into the Eclipse Platform. But XQDT is more than just an editor. It provides tools for managing XQuery projects, for running and debugging XQuery code, or even for developing full-fledged XQuery web application. Moreover, XQDT provides support for most of the W3C XQuery language extensions and also for vendor specific extensions. All these make XQDT suitable for a large number of users, like those learning Xquery, early adopters of various W3C specifications, or even companies that want to build complex XQuery based projects.

Jiří Kosek (ed.)

XML Prague 2010
Conference Proceedings

Vydal
MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 309. publikaci

Obálku navrhl prof. Jaroslav 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 2010

ISBN 978-80-7378-115-6