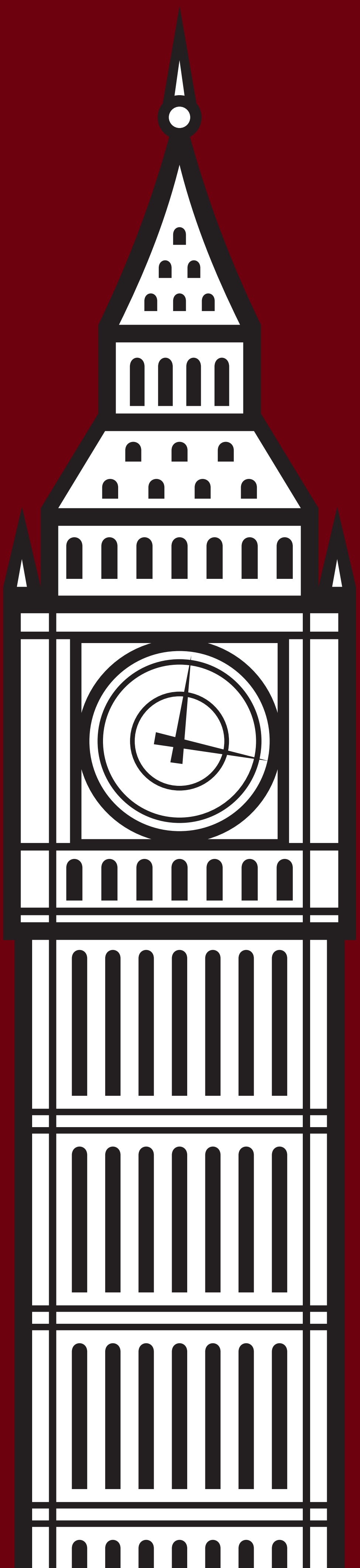


XML LONDON 2015

CONFERENCE PROCEEDINGS

UNIVERSITY COLLEGE LONDON,
LONDON, UNITED KINGDOM

JUNE 6-7, 2015

XML London 2015 – Conference Proceedings
Published by XML London
Copyright © 2015 Charles Foster

ISBN 978-0-9926471-2-4

Table of Contents

Are you stuck fixing

YESTERDAY

Or are you solving for

TOMORROW?

The world's largest banks use MarkLogic to get a 360-degree view of the enterprise, reduce risk and operational costs, and provide better customer service. What are you waiting for?

Hello, we're your
Enterprise NoSQL
database
solution.

MarkLogic®

Say hello to the new generation.

POWERFUL & SECURE | AGILE & FLEXIBLE | ENTERPRISE-READY | TRUSTED

www.marklogic.com · sales@marklogic.com

<oXygen/>®
XML Editor

XML Authoring

<oXygen/> XML Author is the most efficient solution for implementing single source publishing and content reuse.

XML Development

<oXygen/> XML Developer is specially tuned for XML development, providing the best coverage of today's XML technologies.

www.oxygenxml.com

MarkLogician

MarkLogic Consultancy Services

marklogician.com

General Information

Date

Saturday, June 6th, 2015
Sunday, June 7th, 2015

Location

University College London, London – Roberts Engineering Building, Torrington Place, London, WC1E 7JE

Produced By

XML London (http://xmllondon.com)

Sponsors

Gold Sponsor

- MarkLogic - http://www.marklogic.com

MarkLogic®

Silver Sponsor

- oXygen - http://www.oxygenxml.com

<oXygen/>® XML Editor

Bronze Sponsor

- Saxonica - http://www.saxonica.com

SAXONICA.COM XSLT AND XQUERY PROCESSING

Preface

This publication contains the papers presented during the XML London 2015 conference.

This is the third international XML conference to be held in London for XML Developers – Worldwide, Semantic Web and Linked Data enthusiasts, Managers / Decision Makers and Markup Enthusiasts.

This 2 day conference is covering everything XML, both academic as well as the applied use of XML in industries such as finance and publishing.

The conference is taking place on the 6th and 7th June 2015 at the Faculty of Engineering Sciences (Roberts Building) which is part of University College London (UCL). The conference dinner and the XML London 2015 DemoJam is being held in the UCL Marquee located on the Front Quad of UCL, London.

The conference is held annually using the same format, with XML London 2016 taking place in June next year.

— Charles Foster
Chairman, XML London

Improving Pattern Matching Performance in XSLT

John Lumley

jωL Research & Saxonica

<john@jwlresearch.com>

Michael Kay

Saxonica

<mike@saxonica.com>

Abstract

This paper discusses improving the performance of XSLT programs that use very large numbers of similar patterns in their push-mode templates. The experimentation focusses around stylesheets used for processing DITA document frameworks, where much of the document logical structure is encoded in `@class` attributes. The processing stylesheets, often defined in XSLT1.0, use string-containment tests on these attributes to describe push-template applicability. For some cases this can mean a few hundred string tests have to be performed for every element node in the input document to determine which template to evaluate, which in sometimes means up to 30% of the entire processing time is taken up with such pattern matching. This paper examines methods, within XSLT implementations, to ameliorate this situation, including using sets of pattern preconditions and pre-tokenization of the class-describing attributes. How such optimisation may be configured for an XSLT implementation is discussed.

Keywords: XSLT, Pattern matching

1. Introduction

XSLT's push mode of processing [5], where templates are invoked by matching XPath-based patterns that describe conditions on nodes to which they are applicable, is one of the really powerful features of that language. It allows very precise declarative description of the cases for which a template is considered relevant, and along with a well-defined mechanism of priority, and precedence, permits specialisation and overriding of 'libraries' to encourage significant code reuse. Whilst other features of XSLT are valuable, push-mode pattern matching is almost certainly the most important.

Consequently much effort has been expended on developing XSLT-based processing libraries, for many types of XML processing, most notably in 'document engineering', such as DocBook and DITA, which use pattern-matching templates extensively. Typically a processing step might involve the use of hundreds of templates which have to be 'checked' for applicablity against XML nodes that are being processed in a push fashion. One of the challenges for the implementor of an XSLT engine is to ensure that for most common cases, this matching process is efficient.

Various aspects of overall XSLT performance have been studied and reported [1] [3] including optimization rewriting [2]. In this paper we will examine some cases where, owing to the nature of the XML vocabularies being processed and the design of the large XSLT processing stylesheets employed, the *default matching process* in one XSLT implementation (Saxon) can be rather expensive, in some cases taking about a third of all the transform execution time. We'll discuss possible additions and modifications to the pattern-matching techniques to improve performance and show their effects. Some knowledge of XSLT/XPath is assumed.

The paper is organised as follows:

- We first describe “push-mode” processing in XSLT and what the process for matching template patterns is in detail.
- How this process is performed in the Saxon implementation is presented, along with some general remarks about the problems of many templates matching predicated generic, as opposed to named, elements.
- We discuss in detail measurements of the pattern matching performance when processing a large sample DITA document.
- Possible improvements using sets of *common preconditions* to partition applicable templates are outlined and performance measurements using a variety of these tactics in processing the sample are discussed.
- Some other approaches, involving more detailed knowledge of stylesheet expectations are discussed briefly.

doi:10.14337/XMLLondon15.Lumley01

- We speculate on methods to define and introduce such tuning features into an XSLT implementation.

2. XSLT push mode

XSLT's push-mode of processing takes a set of items (usually one or more nodes from the input document such as elements, attributes or text) and for each finds a stylesheet template declaration whose pattern matches the item (the “context item”) in question. Assuming one is found, the body of the template, which can contain both result tree fragments and XSLT instructions, is executed to generate a result which is then added to the current *result tree*. This push mode is often exploited highly recursively, descending large source input trees to accumulate result transformations, usually as modified trees. To understand the problems with large sets of such templates, which is often the case in industrial-scale document processing applications, we need to describe more closely what this pattern matching process is.

When processing a candiate item (“context item”) in XSLT via the `xsl:apply-templates` instruction the following is the effective declarative procedure:

1. All the templates that have `@match` patterns and operate in the “current” mode are considered candidates.
2. For each template so chosen the pattern (which is a modified form of an XPath expression) is tested against the context item to determine a boolean value. Only those yielding true are considered.
3. The templates with the highest *import precedence* are retained. (Templates in an imported, as opposed to included, stylesheet have precedence lower than those in the stylesheet that declares the importation, or any following “sibling” imported stylesheets.)
4. From these only those with the highest explicit or implicit priority are considered. (Patterns have an implicit priority level calculated based on a 'specificity' formula, such that more specific cases (e.g. `piece[@class = 'my.class']`) have higher priority than less specific ones (e.g. `piece`), and thus supporting a natural style for general and specific case programming. Rules can override this by declaring an explict priority.)
5. Members of the remaining set of templates are all potential candidates:
    - If the resulting set is empty, the result is an empty sequence.
    - If it has just a single template member, then the result of the `xsl:apply-templates` is the (non-error) result of executing the sequence constructor of that template with the tested item as the context item.
    - If the resulting template set has more than one member, then it is an implementation choice as to whether a dynamic error is thrown[1]. If not, then the last in “sequence” is chosen and its body executed.

What this list doesn't prescribe is how the process is to be implemented. Clearly there are a number of possibilities of improving performance, by for example examining candidate templates in a suitable order, or pre-classifying subsets of the possible templates. In this paper we examine some possibilities which look deeper into the template patterns themselves.

3. Template rules in Saxon

The algorithm used for matching template patterns in the Saxon processor has been unchanged for many years [1], and works well in common cases. In simplified form, it is as follows:

For template rules whose match pattern identifies the name of the element or attribute to be matched (for example `match="para"`, or `match="para[1]"`, or `match="section/para"`), the template rule is placed on a list of rules that match that specific element or attribute primary QName . Other rules are placed on one of a set of generic lists, arranged by type (`document-node(), text(), element(*)...`) . Both the named and generic lists are ordered by "rank", a metric that combines the notions of import precedence, template priority, and ordering of rules within the stylesheet.

When apply-templates is called to process a specific element, Saxon finds the highest-ranking matching rule on the list for that specific element name, and also the highest-ranking matching rule on the generic list[2]. It then chooses whichever of these two has higher rank. The search of the generic list is abandoned as soon as it can be established there will not be any matching rule with higher rank than a rule found on the specific list. But note that, as we'll see later in our example, once a match has been found in either the specific or the generic list, that list *must still be searched* for other matching candidates of similar rank.

In current versions of Saxon each pattern in the rule chain is examined in turn, to determine a boolean matches value. Of course the boolean match processing is

[1] In XSLT 3.0 this choice can be controlled via stylesheet declarations.

[2] A template can be referenced from multiple lists when its match conditions contains a union of patterns.

processed lazily, and in strict sequence, with falsity propagating as quickly as possible, so for example, ancestor patterns are only examined if the `self::` pattern proves true.

For very many stylesheets, this works well, because most rules specify an element name, and there are typically not many rules for each element name, so typically each element that is processed is matched against at most half a dozen rules on the specific list for its element name; usually the generic list does not need to be considered, because rules on the generic list usually have lower rank than rules on the element-specific list.

The algorithm becomes ineffective, however, when the stylesheet defines very few rules that match specific element names, and many rules that are generic. Typically such stylesheets match elements according to predicates applied to their attributes, rather than matching them by name. A worst-case example of such coding can be found in the DITA-OT family of stylesheets[1]. Consequently these have therefore been used as a test case for exploring improvements to Saxon's pattern matching algorithm. This is even more problematic when the stylesheets use a large range of import precedences, as is also true in the example, where as we'll see there are some 35 different well-populated ranks. Moreover even the named lists gain little as time is dominated totally by checking the unnamed sets.

3.1. Generic match patterns

As hinted above, the presence of a lot of patterns of the form `*[ predicate ]` mean that measures such as indexing patterns on primary element name become ineffective. Are there other indexation schemes that can be employed? Clearly if we have many match patterns of the form:

```
*[@x = 'a']
*[@x = 'b']
*[@x = 'c']
```

and it is possible to determine statically that these predicates are mutually exclusive, then should be possible to construct a hash index whereby we can lookup the value of attribute `@x`, and directly determine which of the patterns applies.

Unfortunately real life is more complicated. A framework where almost all stylesheet template patterns match generic rather than named elements is DITA-OT, where the patterns take the form

```
*[contains(@x, 'a')]
*[contains(@x, 'b')]
*[contains(@x, 'c')]
```

As it happens, in DITA these patterns are designed to be mutually exclusive, so only one of them will ever match. But there is no way an optimizer can know that and we cannot thus meaningfully generate indices. So if we are going to avoid a sequential search of all the patterns, we need a different approach. The rest of this paper examines what this might be, after discussing a specific DITA document processing example in detail.

4. Processing a DITA document

The DITA-OT framework is a set of mainly XSLT (1.0/2.0) tools for processing DITA documents, used extensively in automatic generation of technical documentation. DITA itself describes document components in XML trees, using the `@class` attribute extensively, with class membership described through a whitespace-separated set of class names. (For a fuller description, see Class attribute syntax in the DITA documentation.)

One consequence of this is that many of the processing templates within the DITA-OT framework describe applicability through a match “is this element in class *xxx* ”. Unfortunately, within an XSLT1.0 context, where tokenization isn't present, this is typically described as a predicated match pattern

```
*[contains(@class,' classname  ')]
```

(Additional leading and trailing spaces are added to the class attribute value to support this generic `contains()` match.) A little thought would suggest that when a large number of such patterns all compete to examine the `@class` attributes of pretty much every element in a document then the pattern-matching process might be very expensive. And so it turns out.

Note

This predicated match pattern appears so frequently throughout this paper, that an abbreviation `@C{ classname }` will often be used in tables and figures to replace this construct.

While examining a (different) DITA processing performance issue for a client, Saxonica carried out some measurements on the size of this pattern-matching problem. The chosen situation was one of the stages of expansion of a DITA document into a PDF result, via an XSL-FO route. In particular we examined the conversion of a fully formed DITA source into XEP-specific XSL-

[1] By contrast the Docbook https://github.com/docbook/docbook toolsets, of similar size, use almost entirely named element pattern matches.

FO source (target `transform.topic2fo.main` in the DITA-OT build script architecture). Through most of the rest of this paper, we'll study in detail the processing of a particular source document through this XSLT transformation.

4.1. Source document and transform

The processed document had the following characteristics:

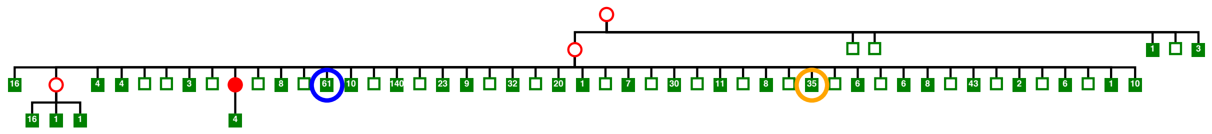
- An 80 page specification for a electronic component, involving lots of tabular descriptions of bit-significances etc.
- Source file: 2.66 MB (246kB of redundant whitespace)
- Source XML tree: 13,066 elements, 46,831 attributes, 6,093 non-whitespace text nodes – total 65,990 significant nodes; tree depth: maximum: 13, average: ~10; sibling width: maximum: 57, average: ~2.
- Result XML tree: 19,441 elements, 91,048 attributes, 6,140 non-whitespace text nodes
- Final PDF document: ~80 pages.
- The document is very table-heavy – 262 tables with 1,309 rows and 4,863 cells, i.e. almost 50% of all the document elements describe table components.
- All bar two elements of the source document contain a `@class` attribute, and there are 43 different values for its text value, the most frequent of which is used 3,683 times.

The processing XSLT transformation had the following characteristics:

- 58 source files.
- 70 pattern-matching modes.
- 418 pattern-matching templates, 155 named templates. (258 of the matching templates are in the `#default` mode.)
- 5 user-defined functions.
- Of the 58 source files, only 33 contain templates. Those that don't either act as importation expanders or contain global parameters or attribute sets.
- All stylesheet connection is via `xsl:import`; there is no use of inclusion and no multiple importation. The importation tree is relatively shallow, at most with a depth of 4 and looks like Figure 1. DITA-OT stylesheet importation tree for DITA→FO.

  where solid nodes indicate stylesheets that contain templates and circles denote stylesheets that import other stylesheets. (The two leaf nodes circled are stylesheets containing the most frequently used templates. The consequences of this are discussed in Section 6.1, " "Un-disambiguating" rules".)

Figure 1. DITA-OT stylesheet importation tree for DITA→FO



4.2. Processing characteristics

Using Saxon 9.6EE on a quad-core i7 1.6 GHz laptop running 64-bit Windows7 with 4GB of RAM, the processing took around 15 seconds. But of more interest are some of the internal statistics. In processing this document to completion, 75,950 template rules 'executed', i.e. their patterns matched and their sequence constructor bodies were processed further[1]. (This is comensurate with a model where most nodes are only 'touched' once during processing.) Determining which rule to execute at each stage took approximately 4.5 seconds, i.e. *~ 30% of all processing involved template pattern matching.*

Of the 70 pattern matching modes in the source XSLT, only 35 were active on this document and only three have significant performance impact, accounting for 96% of all the calls and 99.7% of all the time taken matching template patterns.

Table 1. Significant Modes

Mode	Purpose	# invocations	% invocations	time / ms	%time
`#default`	General	13,095	17.2	4,330	97.8
`toc`	Table of Contents	22,088	29.1	51	1.1
`bookmark`	Bookmarks	37,752	49.7	33	0.8

Whilst the proportion of the number of invocations depends upon characteristics of the document and the DITA-OT framework, the performance costs per node depend upon the complexity of the patterns involved. Thus if we examine the patterns for the `#default` mode, it immediately becomes apparent why there is such disparity.

Table 2. Mode Patterns

Mode	Purpose	# template patterns in mode			#templates matched
		element(*)	element(*named*)	attribute(*named*)	
`#default`	General	240	19	8	39
`toc`	Table of Contents	2	4	0	3
`bookmark`	Bookmarks	2	5	0	3

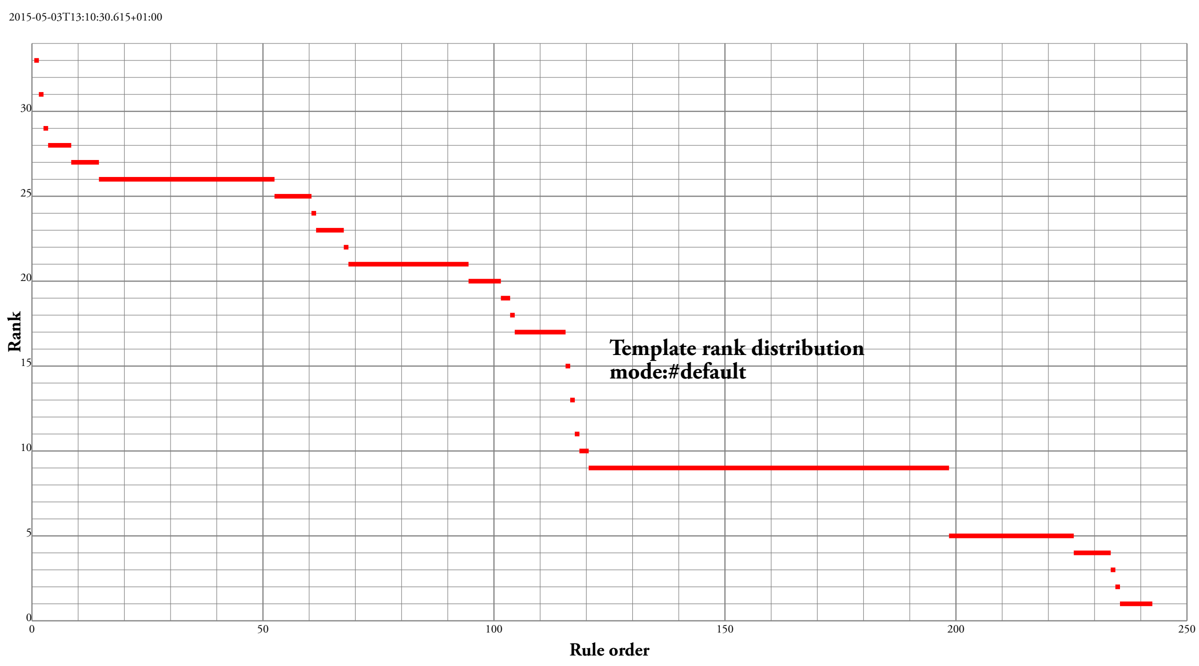
Clearly the number of the template rules that need to be checked for unnamed elements (`*`) in the `#default` mode dominates. How do these 240 templates differ? Firstly as the DITA -OT framework uses a large number of files via `xsl:import` inclusions, they have strongly differing precedences [2]. Template match patterns also have differing implicit or explicit priorities. Together these two properties constitute a *rank*, precedence before priority - matching higher rank patterns are chosen over lower. When multiple patterns match at the same rank optionally either the last in sequence is chosen or a dynamic error is thrown.

In this case, the 240 templates are spread across 25 different ranks, with the sequence order distribution shown in Figure 2, "Template rule order and precedence ranking".

[1] A union pattern (`pattern1 | pattern2`) is considered to be a set of separate matches for this analysis - one for each pattern.

[2] There is no use of the `xsl:apply-imports` instruction within the framework, implying a simple overriding model. `xsl:next-match` is not used either, though that wasn't present in XSLT1.0

Figure 2. Template rule order and precedence ranking



Missing ranks involve templates matching named elements and attributes. The overall total of 35 ranks is in line with the approximately 33 imported stylesheets of the framework. Within a rank rules are ordered in reverse document order as when ambiguous rules are permitted *later* rules are chosen; hence they are placed earlier within order within a rank. Details for the most heavily used ranks are given in the following table:

Table 3. Heavily populated pattern ranks in mode #default

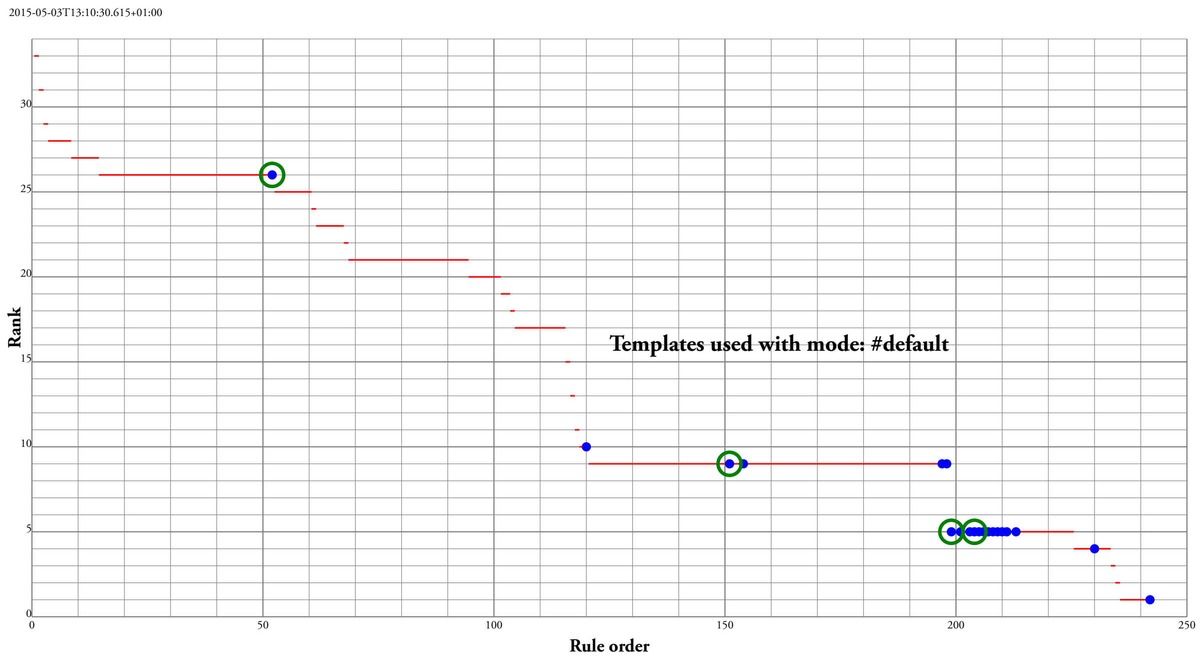
Rank	27	26	25	23	21	20	17	9	5	4	1
start	9	15	53	62	69	95	105	121	199	226	236
end	14	52	60	67	94	101	115	198	225	233	242
size	6	38	8	6	26	7	11	78	27	8	7

Rank 5 contains mostly templates associated with tables, from the stylesheet `tables.xsl` and these table-matching templates appear to be unique, i.e. no templates in other stylesheets would be anticipated to match. Rank 26 (from `pr-domain.xsl`) contains templates for the programming domain.

As already remarked, all templates of a given rank are candidates to match in preference to those of a lower rank. Thus for example, when processing a node whose correctly matching template is that with order number 150 and rank 9, all the 122 templates of higher rank must be eliminated, *and all the 78 templates of equal rank tested for possible conflict*, i.e. a total of just under 200 template match conditions must be examined.

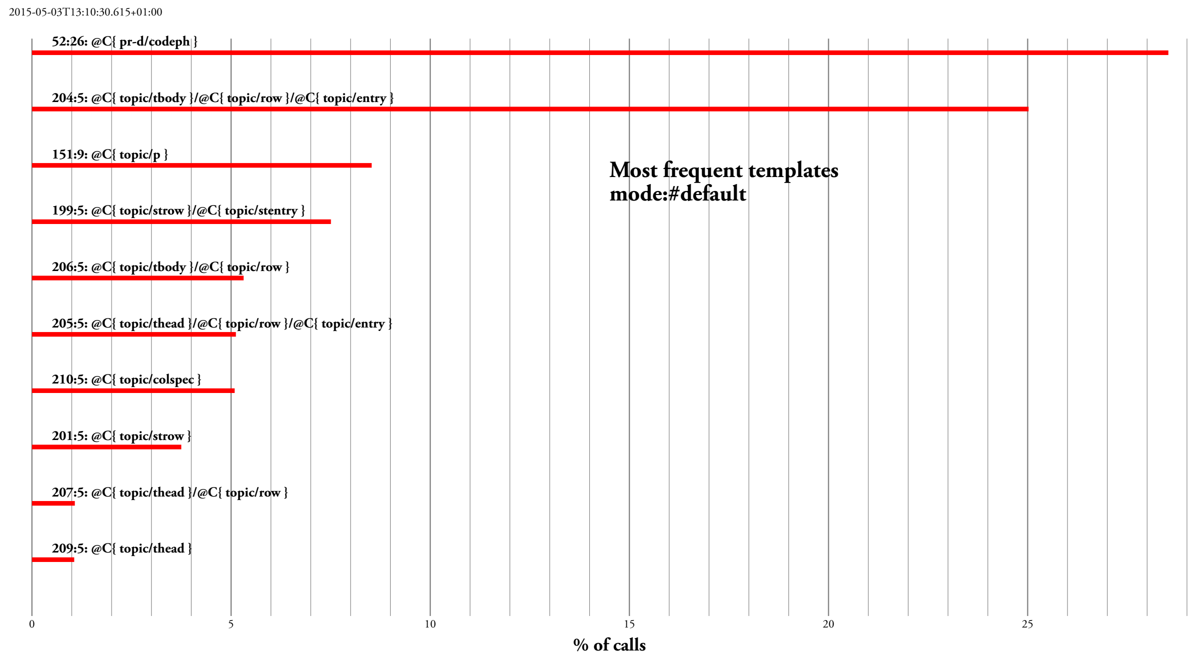
We've examined the rank ordering of all the templates used in the `#default` mode, but which are actually matched within the processing of this sample document? Figure 3, "Rule order and precedence of matched templates" shows the distribution of the 39 templates that were invoked.

Figure 3. Rule order and precedence of matched templates



The blue dots indicate order/rank of matched templates, the green circles surround the four most frequently matched of these, the implications of which are discussed below. What are the most frequently matched templates? Figure 4, "Most frequently matched templates" shows the percentage of all matches taken by the ten most significant, labelled with rule order, rank and (abbreviated) match pattern.

Figure 4. Most frequently matched templates



For this document the most commonly matched template in the `#default` mode, accounting for 28% of unnamed element matches, has order number 52 (it matches `pr-d-codeph`) but the next most called (25%, matching topic table entries) has order number 204 and rank 5. (Their positions are circled in green in Figure 3, “Rule order and precedence of matched templates”.) The next 6 most commonly matched templates account for 35% of calls collectively and are in ranks 9 and 5.

So we have the situation that whilst for 28% of the successful element matches 50 patterns must be checked each time (i.e. the end of rank 26), for more than 60% of the matches, either 200 or 225 patterns must be checked.

Thus far we haven't looked at what these patterns are, merely their required order of checking. Let's examine the top seven unnamed element patterns in the `#default` mode:

Table 4. Most frequent patterns in mode #default

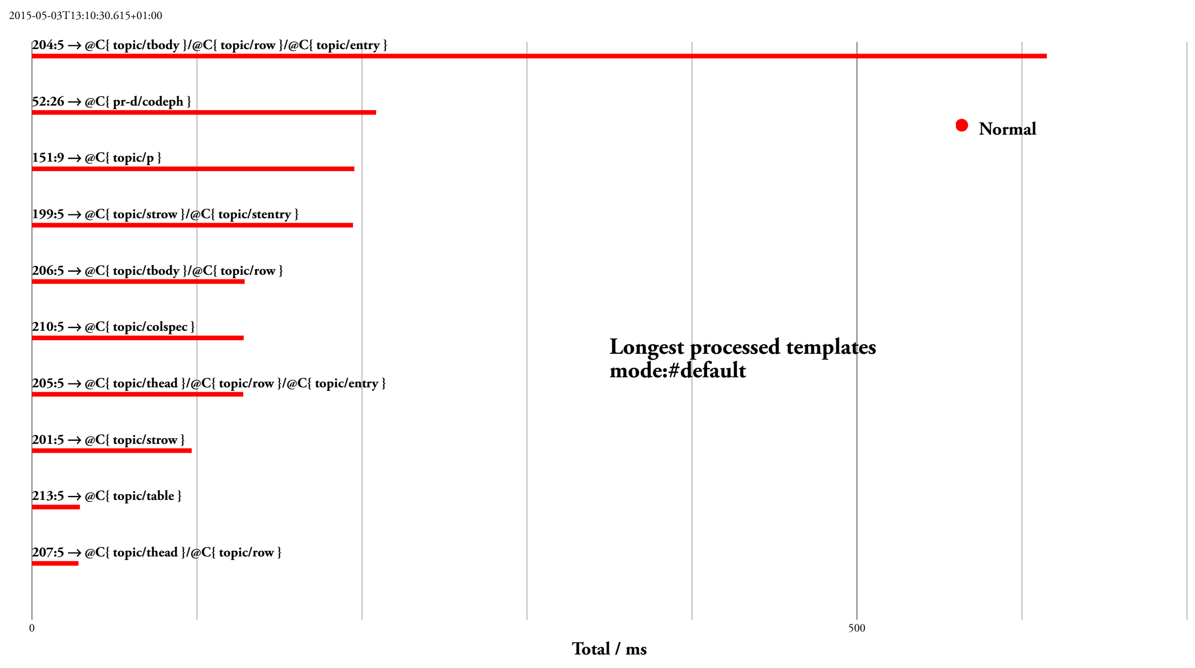
Order	Rank	% calls in mode	Pattern
52	26	28.5	`@C{ pr-d/codeph }`
204	5	25.0	`@C{ topic/tbody }/@C{ topic/row }/@C{ topic/entry }`
151	9	8.5	`@C{ topic/p }`
199	5	7.5	`@C{ topic/strow }/@C{ topic/stentry }`
206	5	5.3	`@C{ topic/tbody }/@C{ topic/row }`
205	5	5.1	`@C{ topic/thead }/@C{ topic/row }/@C{ topic/entry }`
210	5	5.1	`@C{ topic/colspec }`

(`@C{ xxx }` is an abbreviation for `*[contains(@class,' xxx ')]`) Here we can see the problem - each pattern must perform 'free-position' string matching on an attribute of at least the element node under test and sometimes within one or even two ancestors. And it turns out that of these 240 template rules, *all bar one* of them have a similar form[1]. So for templates whose order is 200+, more than 200 other string matches of very similar form have been performed, on *every element* processed through `xsl:apply-templates`

When we look at the amount of time consumed the picture is similar.

[1] We are somewhat puzzled by the redundancy in representation present – table entries are represented both by an element `entry` and a token within `@class`. Are there circumstances where a table cell is *not* represented by an element `entry`? If this is not the case, then why use the `*[....]` pattern rather than one keyed on the element QName? We understand that support for extensibility of element-vocabulary was one reason. Far be it for the authors to criticise the design choices of DITA in its representation of class membership, or the DITA-OT framework for the processing architecture, but *this is no way to run a railway*.

Figure 5. Pattern matching time for the most frequently matched templates



Given the architecture where class membership is described in the `@class` attribute, an obvious question is whether in practice elements can be members of multiple classes. The input clearly shows this to be so – 3,864 of the 13,066 elements have multiple class "tokens", the vast majority being - `topic/ph pr-d/codeph`, which according to the DITA reference, declares that the given element is a structural element equivalent to both a phrase in a generic topic and a code-phrase in a programming domain.

Clearly for this type of stylesheet the issue is that very many of the templates, being processed independently, have to carry out the same sort of operation multiple times on the same node. What methods might be available to reduce the processing, preferably to a minimum? In the rest of this paper we discuss some possible improvements of the following general types:

- Rule preconditions: determining common boolean preconditions that must be satisfied for a (large) number of rules to possibly match – the results for a specific node can be cached and rules that are bound to fail can be excluded rapidly. These approaches have the property that they are *heuristic* in performance improvement but retain correct stylesheet behaviour.
- Other methods that require *oracle* guarantees about the stylesheet behaviour, effectively allowing short-cuts which are not generally applicable to all stylesheets. The cases include:
  - Suspending rule ambiguity checking, and potential template/stylesheet reordering.
  - Pre-tokenizing suitable properties: exploiting higher-level knowledge in replacing patterns with access to deeper structure.
  - Using `key()` structures.

5. Preconditions

With long sequences of patterns, many of which have some similarities, one possibility is to detemine a smaller set of precondition patterns that can be tested as required for a node before the "main" pattern is checked. The value of such a precondition for a given node can be computed only when required (i.e. the first time when a pattern that uses that precondition has to be checked) and stored to avoid subsequent recomputation. The hope of course is to eliminate quickly higher-rank patterns that cannot match as one or more of their preconditions has already been determined to fail. For example if there were a large set of templates with matches of the form `chapter/` *`node-condition`* , such as:

```
chapter/title[condition1],
chapter/title[condition2],
chapter/para,  chapter/section ...
```

then they all share the requirement that `exists(parent::chapter)` must be true for the pattern to match. Thus computing whether this precondition is satisfied for a given node *once* may aid performance in several possible ways:

- If a node does *not* have a `chapter` parent, then this need be determined only *once* for the node and each of these templates can be ruled out immediately.
- The pattern might be partially-evaluated within the context that the precondition is true, e.g. precondition(`exists(parent::chapter)`) reduces the patterns to

```
precondition:exists(parent::chapter) :
    title[condition1], title[condition2],
    para,  section ...
```

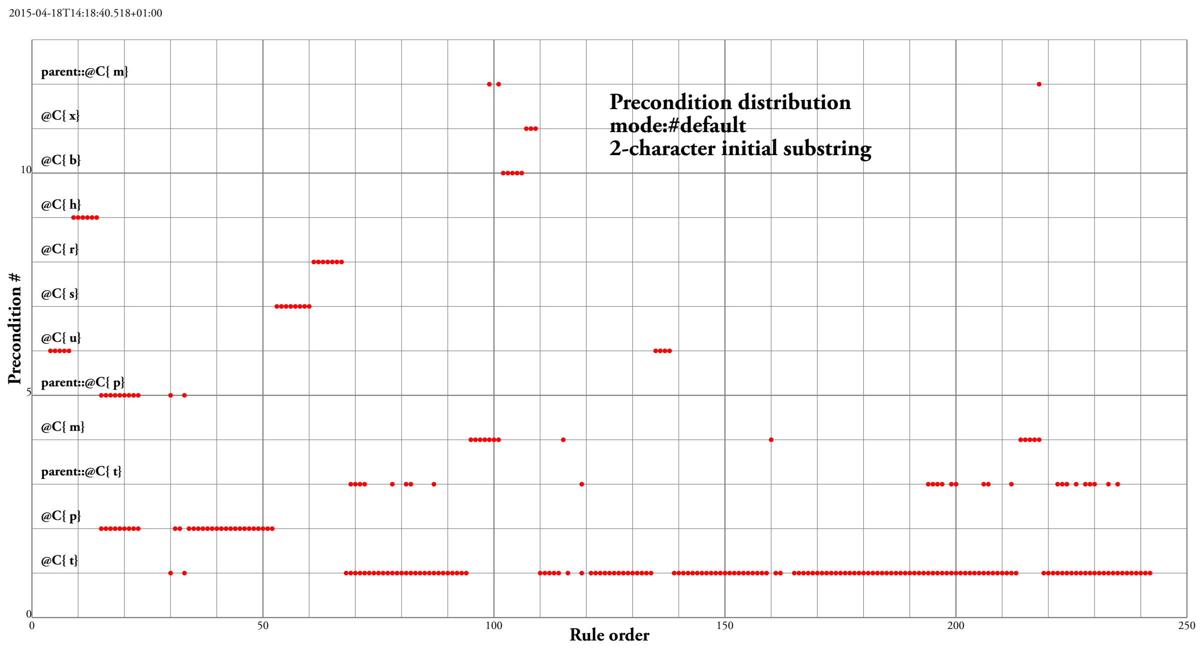
In our DITA example, using `exists(@class)` will gain us little – almost every element in a DITA document has a `@class` attribute and 90% of all element matches predicate upon it. We could choose to use the `contains(@class,...)` as a more discriminating condition. Of the 239 template rules that share that test for the context item (the node under test within `xsl:apply-templates`) there are 204 distinct values for the check (the largest common set has just 7 members, most have of course 1).

In some cases these preconditions might be common, especially when tested on ancestor nodes. For example rules #204 and #205 both test for `topic/entry` on the element and `topic/row` on the parent, differing only whether the grandparent is a table body or head. In this case, and especially with our sample document which is *very* table heavy, the precondition "pair" `contains(@class,' topic/entry ') and contains(../@class, ' table/row ')` might be beneficial (tested on #204, but result available for #205), but it is admittedly a very special case. What other more general preconditions might be useful?

A necessary precondition for `contains(@class,string)` is `contains(@class,any-substring-of(string))` so some expression of this form might be useful. Choosing to use just the first character of the comparator string, which might normally be expected to be of some use in many cases, fails miserably here, as due to the class representation model being used within DITA-OT, a leading space is appended to the class token comparand (effectively implementing an equivalent of `tokenize(@class,'\s+') = 'entity-class'`) – no gain there then.

If we choose to use the first two characters, we get 12 different preconditions, three of which apply to a `parent::*` context. The distribution of the use of these preconditions across the rule order is shown in Figure 6, "Distribution of 2 character initial substring preconditions":

Figure 6. Distribution of 2 character initial substring preconditions



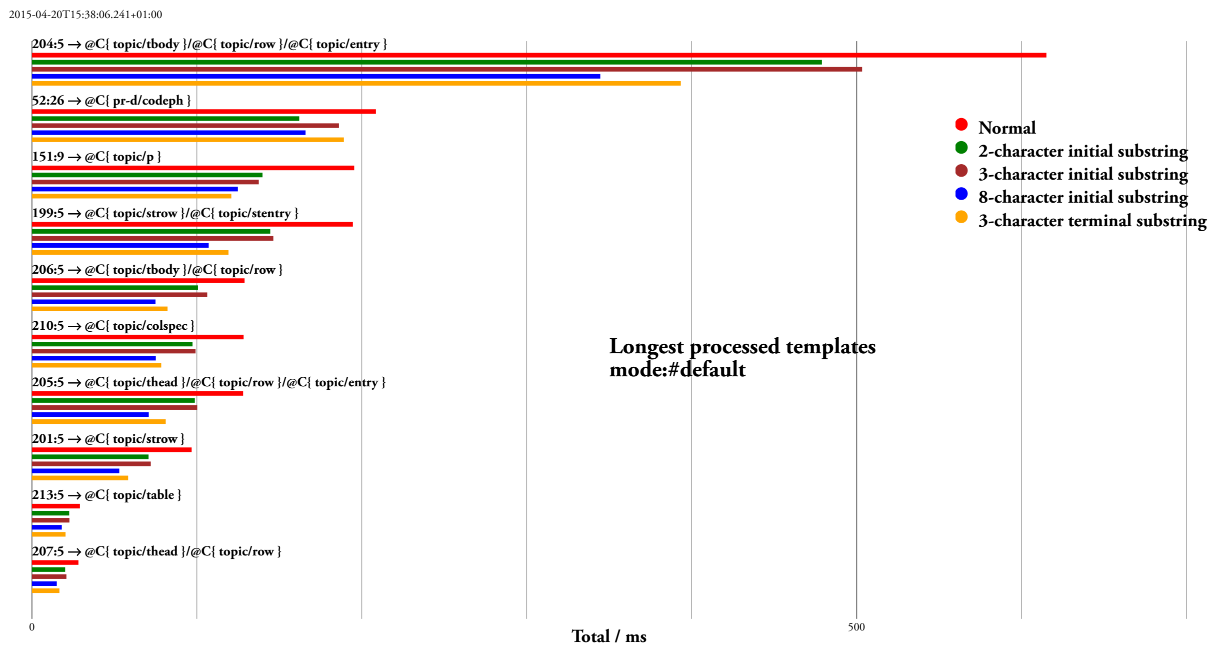
Note that these preconditions are not mutually exclusive – some of the compound cases involve two conditions, one on the context element and the other on its parent. This is the case for 43% of the element template matches on the sample document. By changing the fixed-length initial substrings used we get a variety of balances between precondition group sizes:

Table 5. Precondition group sizes as a function of initial substring discriminant

Substring length	# preconditions	Largest reference set
2	12	146
3 - 5	14	121
6	16	121
7	46	121
8	75	17

We can modify our applicable rule search such that each rule has a set of required preconditions (described by a list of indices into a cache of expressions and boolean values) which are tested before the main match is then processed. The rest of the rule list processing machinery is unaltered. The effect on the performance is shown in Figure 7, “Effect of differing substring preconditions”:

Figure 7. Effect of differing substring preconditions



Note that when we are using these substring preconditions, there is little we can do to “pre-evaluate” the patterns themselves. Just because `contains(.,'abc')` is a precondition for `contains(.,'abcdef')`, does not of course imply `contains(.,'def')` is now sufficent, unlike in other cases where a “true” condition reduces the expression.

Now of course in this example there is some implicit structure in the class token using the solidus (/)[1]. Unfortunately this insight gains us little - using the substring before we get 14 groups, using the substring after we get 197! Obviously a better approach is to use a more infomation-theoretic partitioning. For example 121 template rules match a class substring `' topic/...` but adding one more character to the discriminant for this case replaces this group with 19 subgroups, the largest being 12-15 in size.

There is nothing that says we are restricted to initial substrings. Choosing the last 3 characters (which of course end with a space) gives us 53 different preconditions, with the largest group being 19 in size. The effect is similar, as is shown in Figure 7, “Effect of differing substring preconditions”.

A recursive approach (extending the length of the substring for a particular group until subsequent subgroups are smaller than a proportion of the size of the

[1] It denotes DITA element equivalence to a given base element within a given topic.

original set or some minimum size) can give us a suitable partitioning. Doing this for this for 5% or a minimum of 30 gives us 41 preconditions, with a largest reference group of 26. A general rule of thumb suggests that when the number of groups is similar to the size of each group, the testing workload should be minimised.

On a more information-theoretic basis, we could generate some optimal partioning tree. However we have an issue that currently this would be done at compile-time, when, whilst we know the frequency of pattern components spread across the template spaces, we don't know the relative frequencies of execution on documents at run time. A possibility might be to collect statistics from representative training runs, which are then used to tune a subsequent compilation[1].

The performance figures above use Saxon's default rule list representation with precondition references added to the rules. Another possibility is to split the rule list into a number of separate lists where within each sublist rules which would *fail* a common precondition have been eliminated, then choose between the different lists at search start, based on checking a small number of those preconditions. For example, consider the two most frequent preconditions in Figure 6, “Distribution of 2 character initial substring preconditions” – `@C{ t}`, which qualifies 146 of the 240 rules and `@C{ p}` which qualifies 30. Satisfying the first condition does not restrict matches to just those rules which have that precondition; given the nature of the `contains()` function (and DITA's possibility of multiple class tag values) it would be entirely possible for one of the rules having the second precondition to match also. Rather the *failure* of `@C{ t}` rules out all those 146 rules, leaving a list of just 94 to be checked.

So now we have to look at the inverse of the problem. Table 4, “Most frequent patterns in mode #default” shows that just six patterns, all predicated on `@C{ t}`, account for at least 56% of all the template matches. Hence failure of this condition (which as we've seen limits the rules to be checked to 94) would only be invoked for a maximum of 45% of all calls. Failing `@C{ p}` will be frequent of course (for 70% of the elements), but it only eliminates 30 rules, albeit at high rank order.

6. Other possibilities

Using preconditions, as described above, does not change the correctness of the stylesheet behaviour under any circumstances. However, if the stylesheet designer can make certain guarantees about the overall stylesheet operation, then there are a number of other possibilities we might consider

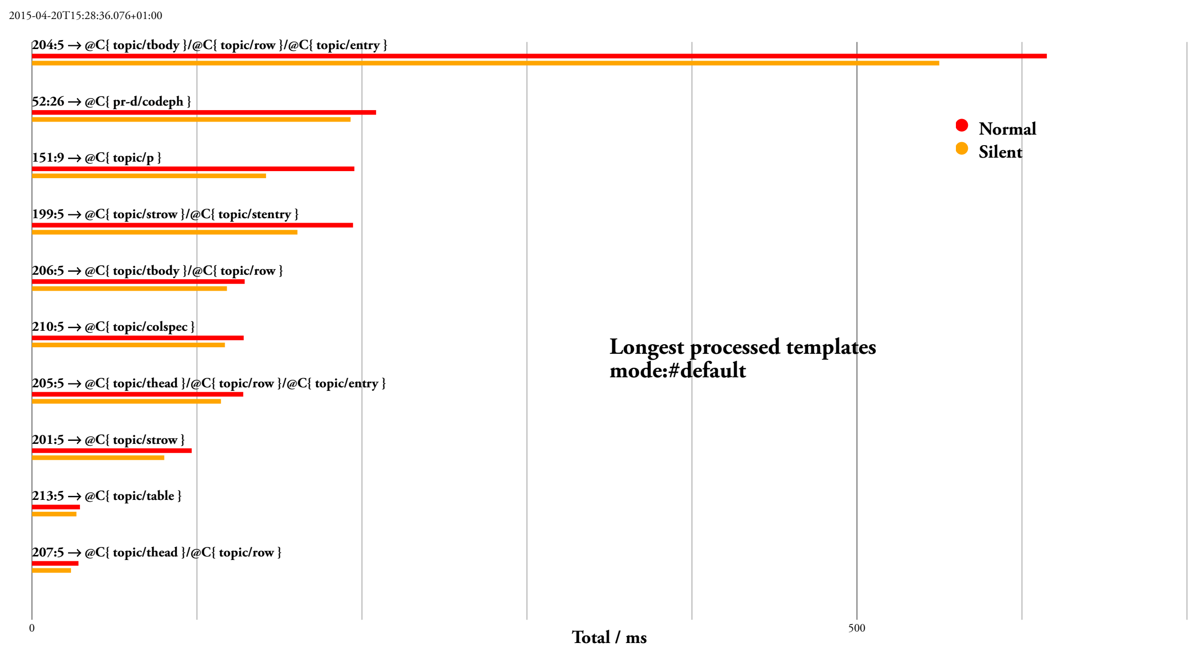
6.1. “Un-disambiguating” rules

We have noticed that in the execution of this stylesheet on the example source document, there actually is no ambiguity in applicable rules – all template patterns for a given precedence/priority rank have mutually exclusive patterns *on the nodes present in the example DITA document*. Hence we can assume that once a pattern for a template matches, all others of similar rank can be discarded. In effect we are suspending the checking of rule ambiguity, and provided that the mutual exclusivity is true for all practical purposes, which cannot be determined statically, the stylesheet will still continue to function correctly.

In our example, where there are many templates sharing the same rank (e.g. the 78 of rank 9, or 25 of rank 5) we can eliminate further search to “rank end”. The effect may be slight but can be worth exploring. For our example we get the following:

[1] XSLT's package delivery mechanism might be a useful aid to this.

Figure 8. Effect of removing rule ambiguity



The effects are most marked on rules #151, where the number of rank 9 rules to be tested drops from 78 to 31 (total rules checked 198 → 151) and #204, where only 5 of the rank 5 rules need checking, as against 25 (total rules checked 225 → 204). Interestingly, as we might expect, rule #52 (which is the most frequently called) gains no benefit, as it is the last member of rank 26 as shown in Figure 3, "Rule order and precedence of matched templates".

We might also speculate on the effect if the 38 rules of rank 26 are re-ordered so that rule #52 is tested *first* and hence the number of rules checked for such nodes drops 52 → 16, suggesting time taken might drop to a third of its previous level. This would of course be predicated on user-provided guarantees of mutually exclusive applicability of rules[1]. In the case of rule #52 simple movement of the template from the beginning to the end of its source file might have similar effect!

In a similar manner for rule #204, it is imported through the `tables.xsl` stylesheet, whose position is circled in blue in Figure 1, "DITA-OT stylesheet importation tree for DITA→FO". If these importations are mutually exclusive, and they may well be, moving the importation of that file to the *end* of the importation list in its parent stylesheet increases the precedence of its templates and hence rank. The orange circled stylesheet contains rule #52. Such rearrangement of source declaration orders may be possible by collecting statistics from representative training runs to detect such conditions.

Within XSLT2.0 it is an implementation choice as to whether conflict raises an error or the last applicable rule in declaration order is chosen. In Saxon, when warnings are silent, the last is always chosen regardless, other rules not being checked. In XSLT3.0 (which this DITA framework predates) this behaviour can be controlled by a `@on-multiple-match="use-last|fail"` property on a suitable `xsl:mode` declaration and issuing of warnings (which imply other rules must be checked) similarly.

6.2. Pretokenizing

The DITA architecture is effectively embedding structure (multiple class membership) within the `@class` attribute. If we can be guaranteed that this is the case, then an option might be to generate preconditions that operate on those implicit tokens whilst tokenising the appropriate accessor once for each node. So for example `*[contains(@class,' topic/entry ')]` would be considered equivalent to the pattern

[1] We would be interested in situations within DITA-OT where there might be some expectation of non-exclusive rules at the same import precedence being written.

`*[tokenize(@class,'\s+') = 'topic/entry')]` [1] which can then be further converted into a pair:

```
$tokens.class := tokenize(@class,'\s+')
         → ('foo','topic/entry')
test:
   $tokens.class = 'topic/entry'
```

where the node would be tokenized exactly once for each containment-tested attribute (when the condition is first required) and then need only be tested for value membership against the token set in further rules examining the same attribute properties. In most cases the `@class` attribute contains three tags (of which the first is either '+' or '-' which is never tested by templates, at least in the current test, so the string literal sequences to be tested are very short.

Now we define these tests as specialist tokenisation preconditions (they may of course be shared between rules) and index into the collection from the rules. And unlike with the substrings, we *can* project the effect of a true precondition into the pattern viz:

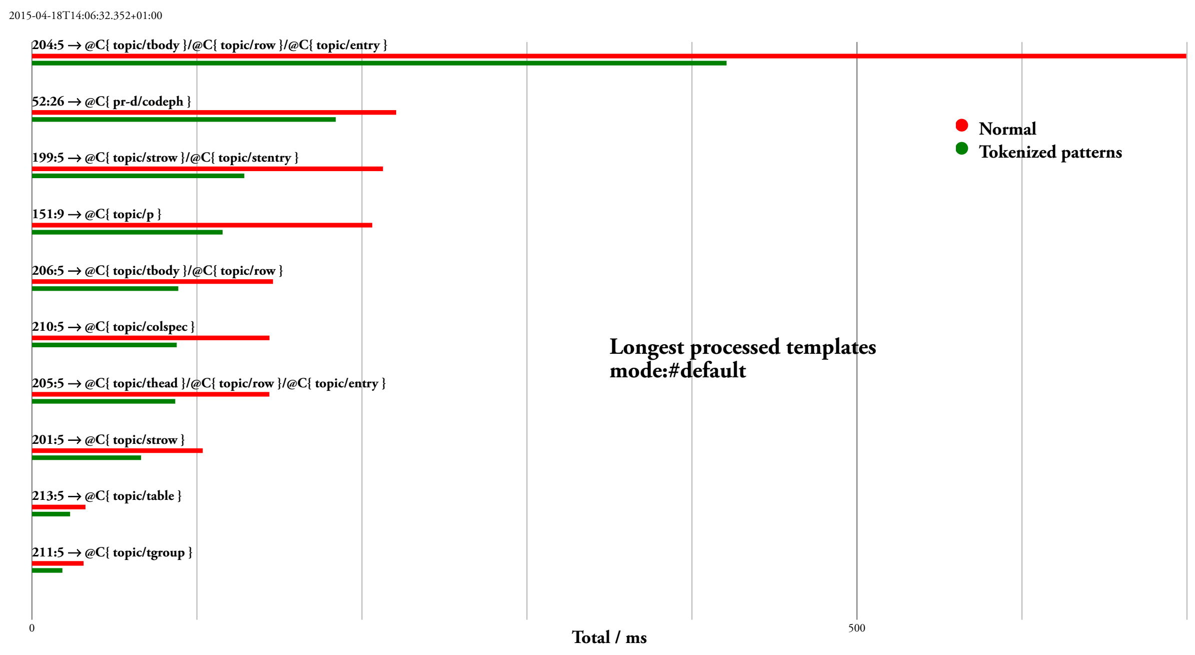
```
R1: *[contains(@class,' topic/entry ')]
R2: *[contains(@class,' topic/row ')]
R3: *[contains(@class,' topic/row ')]/
       *[contains(@class,' topic/entry ')]
→
R1: *[tokenize(@class,'\s+') = 'topic/entry')]
R2: *[tokenize(@class,'\s+') = 'topic/row')]
R3: *[tokenize(@class,'\s+') = 'topic/row')]/
       *[tokenize(@class,'\s+') = 'topic/entry')]
→
$tokens.class :=
    tokenize(@class,'\s+')
$tokens.parent.class :=
    tokenize(parent::*/@class,'\s+')
$preconditionM := $tokens.class = 'topic/entry'
$preconditionN := $tokens.class = 'topic/row'
$preconditionP := $tokens.parent.class = 'topic/row'

R1:  $preconditionM && *
R2:  $preconditionN && *
R3:  $preconditionP && $preconditionM && *
```

where the token variables and precondition references are held within the rule-processing structure. Within Saxon these element match rules (`*`) would be indexed on the “unnamed element” list, so the last part of each of the final rule patterns would always yield true. In this case these rules have been reduced to just a conjunction of their preconditions.

The preconditions only call for the appropriate tokenisation when it is first needed (they are effectively single-assignment local variables with a scope for the pattern match for a single node, and evaluated lazily) – so that other preconditions involving differing values only need to check within their own sequence comparison. Obviously for a predicate which already uses explicit tokenisation mechanisms (such as processing semicolon-separated `@style` descriptions on SVG and the like) then this technique can be used similarly. For our example document, we get the following performance improvements:

[1] It is only guaranteed equivalent if the `@class` value string starts and finishes with at least a single space.

Figure 9. Effect of pre-tokenizing pattern test inputs



The number of preconditions is now large (~200, corresponding to each possible tag value mentioned in the stylesheet) but most are referenced only once. However they all share a single tokenisation of the `@class` attribute (or that of the parent's in some cases)

Note

A generalisation of this technique into evaluating and then crossreferring to common subexpressions is a possibility. In the case here the binding between preconditions and the evaluated variable is very tight (the variable value for a given node is merely a (small) finite list of strings). Extension to a more generic sequenced-value approach would probably be considerably more complex.

6.3. Using key() mechanisms

From early on XSLT has defined a `key()` mechanism to speed searching for applicable nodes within an XML tree. Using the `xsl:key` declaration a set of nodes can be classified into a number of subsets dependent upon an expression evaluated for each node[1]. Usually support for this within an XSLT implementation is efficient, the key being computed only once. Thus it is tempting to see whether a suitable set of keys can be generated and used within modified patterns. The approach is basically:

```
*[contains(@attr,'string')]
      → *[key('attr','string',.)[1] is .]
```

where effectively the key has been defined by:

```
<xsl:key name="attr"
   select="*"
   use="let $e := . return
    (string1, ... stringN)[contains($e/@attr,.)]"/>
```

The key has indexed all the nodes whose `@attr` contains any of the substrings mentioned within the templates, based on that substring. The pattern uses this key, subsetting the nodes to just those which are `descendant-or-self::*` of the element being tested – if the first node is the target node then there is a match.

We implemented this scheme, but unsurprisingly rather than improve, matters deteriorate significantly. In computing the key (which Saxon does on the first request via the `key()` call) *every* document element is processed, for *every possible contained substring* mentioned in the template sets. Equally well, the predicated key lookup starting at a given node (`[key('attr','string',.)[1] is .]`) involves searching through all the document-ordered nodes already computed for the key (which in our case of course means pretty much all the elements in the entire document) to find the current focus. A

[1] A node can be a member of several subsets, as the key determination can produce several values (e.g. `xsl:key match="car" use="@year,@colour"`).

moment's thought suggests that will have *$O(n^2)$* performance. (If the templates were of the form `*[@attr='string']` a key approach might work – certainly `<xsl:key name="attr" select="*" use="@attr"/>` will be very much cheaper.)

7. Generalisation?

In the introduction we mentioned both DITA and Docbook being significant large document-engineering frameworks. Our experimentation has focussed on DITA given the expensive nature of processing its class representation. We were curious to see if similar issues might appear in processing Docbook documents – we believe this not to be the case. A survey of one of the steps (conversion of Docbook into HTML[1]) which is of similar "size" to those within DITA-OT, shows that of the 1500 pattern matching templates within 59 files, only 113 are against unnamed elements or attributes, and none of the 190 modes has more than two. The vast majority of patterns are described for named elements and thus would be fully indexed within Saxon. Hence we anticipate the methods discussed in this paper would not be necessary for that framework.

We have shown that extracting a set of preconditions, where evaluating one precondition for a particular node can eliminate many match patterns, is an effective strategy for the DITA stylesheets we have been studying. This then raises the next question: can the technique be generalized so that it is suitable for inclusion in a general-purpose XSLT processor, producing performance benefits for a sufficiently large set of stylesheets to justify its existence?

This divides into two sub-questions: firstly, is the general strategy of extracting preconditions general-purpose enough? We think it is. Secondly, what about the specific rules that we have found to work well on the DITA stylesheets? Here, we are not convinced – they are intimately tied up with the way DITA-OT decomposes the class representation tags.

We believe that for the same general approach to work with different kinds of stylesheets, we may need to make the rules for extracting preconditions in some way configurable. So we might consider shipping the product with a set of rules that work well for DITA, and other sets of rules that work well for other XML vocabularies. We could consider defining a vocabulary allowing the rules to be written declaratively (see John Snelson's paper on declarative XQuery rewrite rules [4]) for some possibilities. The designers of an XML vocabulary could then perhaps ship a Saxon optimizer plug-in that applies rules appropriate to the specific vocabulary. Saxon could perhaps select an appropriate plug-in from the repertoire available based on the namespaces in use in the particular stylesheet.

8. Conclusions

In this paper we have examined performance issues in processing a relatively large document with an XSLT transform containing a large number of generic templates whose match computation can be expensive, and where large numbers of pattern matches occur very late in "rank order". We've shown that by choosing suitable shared preconditions for rules, which need only be computed once for a node under test, we can ameliorate the effect of such long rank sequences in pattern sets. Alternatively, by choosing to add some "higher-level" knowledge, declaring that a given set of patterns is in effect implementing a tokenisation, we can also improve pattern matching.

As implementors of a major XSLT processor, our next step is to examine ways that such heuristics can be added and configured in the product. Some of these may be very specific declarations within a configuration. Others might be associated with running training sets, collecting statistics and proposing specific tunings. Watch this space...

Saxonica would like to thank its (anonymous) client who was very willing to let us study the processing of one of his real DITA documents in detail. Hopefully we'll be able to repay him soon with some welcome "tune-up".

References

[1] Michael Kay. *Saxon: Anatomy of an XSLT processor*. 2005. http://www.ibm.com/developerworks/library/x-xslt2/

[2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages. 2007. http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html

[1] `docbook/xsl/html/docbook_custom.xsl` in the Oxygen 16.1 implementation

[3] Michael Kay and Debbie Lockett. *Benchmarking XSLT Performance*. XML London 2014. June 2014. doi:10.14337/XMLLondon14.Kay01

[4] John Snelson. *Declarative XQuery Rewrites for Profit or Pleasure*. XML Prague 2011. March 2011. 211-225. http://archive.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf

[5] *XSL Transformations (XSLT) Version 3.0*. 2014. World Wide Web Consortium (W3C). http://www.w3.org/TR/xslt-30/

It's the little things that matter

How certain small changes in XSLT 3.0 can improve your programming experience drastically

Abel Braaksma

Abrasoft

<abel@abrasoft.net>

Abstract

Some larger features of XSLT 3.0 and by extension XPath 3.0, like higher order functions, packages and streaming, have been covered extensively in previous papers. This paper aims to fill the gap of knowledge and shows you how several seemingly smaller changes improve your life as a programmer, how they make common idioms easier to implement or how they create a cleaner programming experience. Features covered in this paper include try/catch and structured error handling, memoization of functions, iteration, merging, text value templates, assertions, modes and enforcing mode declarations, shadow attributes, forking and how it supports multi-threading, applying templates on atomic values, maps, 2.0 backwards compatibility and processing JSON input.

After reading this paper, you should have a firm grasp of what to expect from switching from XSLT 2.0 to XSLT 3.0, if packages and streaming are not your primary concerns.

Keywords: XML, XSLT, XPath

1. Disclaimer

This paper discusses new features defined in XSLT 3.0 and XPath 3.0. The XSLT 3.0 specification [1] is a Last Call Working Draft [2] but the information in this paper may be extended to features added after this public version, based on publicly available bug reports [3]. XPath 3.0 [4] is a W3C Recommendation, this paper focuses on XPath 3.0 and not on new features introduced in XPath 3.1.

This paper is based on the publicly available versions of XPath 3.0, XSLT 3.0 and XDM 3.0 as of March 12, 2015, see [2] [4] [5]. Since XSLT 3.0 is not yet final, it is possible that references and details change before the final specification receives Recommendation status.

2. An introduction

The XSLT 3.0 specification is long underway, in fact, since the XSLT 2.1 [6] document was published, almost 5 years have passed. During the cause of its development, the main focus has naturally been on major new features like streaming and packages. However, both from external and internal input, many smaller changes have been introduced and have made their way normatively into the new specification. With the specification being close to Candidate Recommendation status, it is a good moment to reflect on the past half decade of specification development and review how XSLT 3.0 can improve the lives of us programmers.

The *bigger* new features have received ample attention in recent and less-recent talks and papers, with among others, [7] on packaging[1], [9] on streaming analysis, [10] on the internals of a streaming processors, [11] on streaming in XSLT 2.1 and from myself, [12] on higher-order functions and [13] on streamable functions.

The *smaller* features have been lightly touched during standards update talks during conference time, but have received little attention in recent papers or talks. This paper will introduce the interested reader to two handfuls of new features introduced in XSLT 3.0.

Each of the following sections briefly describes a new feature, gives an introduction to the main syntax and use, shows how it can improve your current, typically XSLT 2.0-style of programming experience, summarizes some of the caveats you may encounter when using this new feature and finally listing limitations imposed by the official specification text, or by practical concerns from a programming standpoint.

[1] Florent's paper and talk were about the packaging extensions introduced by [8], but these features have since made it into the specification and have become a major new feature of the language.

doi:10.14337/XMLLondon15.Braaksma01

3. Structured error handling with try/catch

Availability in XSLT 2.0: Not available, unless through extension mechanisms, like the one proposed in [14].

The instructions `xsl:try` and `xsl:catch` provide a means for catching and handling dynamic errors in XSLT 3.0. It is only possible to catch *dynamic* errors, that is, errors that occur after compilation of the stylesheet and after any static errors have been reported back by the processor, such as incorrect XPath expressions, missing or wrong values of attributes etc.

3.1. Syntax and use

```
<xsl:try>
   <!-- Content, must include an xsl:catch -->
 </xsl:try>
```

```
<xsl:catch>
   <!-- Content -->
 </xsl:catch>
```

The `xsl:try` instruction wraps a piece of code for which you expect a possible error to be raised. This is similar to the scoping rules in C++ style languages where the `try{...}` block is used to scope a part of the code for which you want to capture errors. It is allowed to nest `xsl:try` inside itself or indirectly through templates or function calls, when an error is raised, the innermost `xsl:try` block and its accompanying `xsl:catch` that matches the error will be used for catching the error.

The `xsl:catch` element is a required part of each `xsl:try` block and must be the last element in the sequence constructor (it cannot be followed by anything else than whitespace or XML comments). You can have more than one `xsl:catch` element, they are each other siblings. The first `xsl:catch` element for which the `errors` attribute matches the thrown error (which is an EQName, that is, both the namespace and the local-name part must match) will be evaluated and the result of its sequence constructor will be returned instead. The body of the `xsl:try` will be discarded and other `xsl:catch` elements will not be processed. The value of the `errors` attribute is a space-separated sequence of NameTests, as used in XPath. If absent or `*` it matches all errors. You can select all errors in one namespace with `errors="err:*"` or all errors with a specific name in all namespaces with `errors="*:ERR1234"`. A specific error is caught with a full QName, as in `errors="err:FOAR0001"`.

Once an error is caught, its properties can be examined using special variables, such as `$err:code`, `$err:description` and `$err:value`, the latter being a user-supplied value with the `fn:error` function. Apart from the error code, all these values are implementation dependent. A processor can further give information on the location of the error by providing values for the `$err:module`, `$err:line-number` and `$err:column-number` variables. The namespace for `err` is the standard error namespace `http://www.w3.org/2005/xqt-errors`, which must be in scope to be able to use these special variables. The variables are only available within the scope of `xsl:catch`.

If the `rollback-ouput` attribute is set, which is the default, any output generated from the body of the `xsl:try` prior to catching the error, will be rolled back. This attribute is available primarily for situations where memory is a constraint, as with streaming, where keeping the entire output in memory to be able to perform a rollback afterwards can mean that the processor will run out of memory constraints. Another use-case is `xsl:result-document`, specifically if the target does not allow a rollback, for instance if the target is a mail address, an online resource or a database without checkpoint capabilities.

The effect of using the attribute may be slightly different than one might expect. If the attribute is set to `no`, and the processor is not capable of rolling back, a new dynamic error is raised, `XTDE3530` and further processing fails (unless that error itself is wrapped in another try/catch). The reasoning behind this behavior and raising an error instead of simply continuing processing is that if no rollback is possible, the state of the result document will be in an undetermined state and further writing to the same result document may yield unexpected results.

In conjunction with this new instruction, it is possible to raise your own errors in XPath with the new `fn:error` function and in XSLT with the `xsl:assert` and `xsl:message` instructions.

You can rethrow an error only by raising a new error. This will lose the context, as inner errors are not maintained[1]. A typical function call to rethrow an error is `fn:error($err:code, $err:description, $err:value)`. It is not possible to set line info for a newly created error.

3.2. Improving your code

Since structured error handling was virtually absent in XSLT 2.0, this feature does not have comparable examples in XSLT 2.0, except where external input is checked with code. For instance, in the past you may

[1] Some languages provide a means for a field containing the inner exception, which maintains thread information and location of an exception that was thrown previously. Such mechanism is not available in XSLT 3.0.

have written something like the following to calculate the average price of an article in an order:

```
<xsl:choose>
  <xsl:when test="items = 0">
    <xsl:message select="'Invalid input zero'"
      terminate="no"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:sequence
      select="total-of-order div items"/>
  </xsl:otherwise>
</xsl:choose>
```

But in XSLT 3.0, you can write this also (and opinions may vary whether this is clearer or not), as follows:

```
<xsl:try>
  <xsl:sequence select="total-of-order div items"/>
  <xsl:catch errors="err:FOAR0001">
    <xsl:message select="'Invalid input zero'"
      terminate="no"/>
  </xsl:catch>
</xsl:try>
```

Another example is with `fn:doc-available`, which can raise an error if the argument is not a valid URI. In XSLT 2.0, you probably wrote something like the following:

```
<xsl:choose>
  <xsl:when test="doc-available(settings-doc)">
    <xsl:apply-templates
      select="doc(settings-doc)"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:copy-of select="$default-settings"/>
  </xsl:otherwise>
</xsl:choose>
```

There are three issues with this code. First is that the `fn:doc-available` function itself can raise an error and in XSLT 2.0 this would mean that the transformation would fail. The second is that using the tandem of `fn"doc-available` with `fn:doc` always leads to duplicate code. Lastly, there is no extra information you can report back to the user, because you can only succeed or fail, but you cannot find out in 2.0 what the reason of failure is. Using try/catch you can rewrite this as follows and extend it for other error scenarios you would like to report on:

```
<xsl:try>
  <xsl:apply-templates select="doc(settings-doc)"/>
  <xsl:catch errors="err:FODC0005">
    <xsl:message select="'The settings doc cannot '
|| 'be retrieved because it is not a valid URI.'"/>
  </xsl:catch>
  <xsl:catch errors="proc:ERR0005">
    <xsl:message
      select="'Access denied while trying to ' ||
      'retrieve settings document.'"/>
  </xsl:catch>
  <xsl:catch errors="err:FODC0003">
    <xsl:message
      select="'The settings doc cannot be ' ||
      'retrieved deterministically, possibly ' ||
      'it is too big to fit in memory.'"/>
  </xsl:catch>
  <xsl:catch errors="err:FODC0002">
    <xsl:message select="if(contains(
      $err:description, 'available documents')
      then 'Settings document not available in ' ||
           'available documents, reason: ' ||
           $err:description
      else 'Settings document found but is not ' ||
           'valid XML, reason: ' ||
           $err:description"/>
  </xsl:catch>
  <xsl:catch errors="*">
    <xsl:message select="'Unknown error raised ' ||
    'while trying to retrieve settings document: '
    || $err:description"/>
  </xsl:catch>
</xsl:try>
```

This code shows several advantages over the previous example:

- The programmer can retrieve and return better information based on the error raised
- You can catch errors raised by the processor that are not also errors specified by the specification. Note that in this example `proc:ERR0005` is used but that essentially a processor should raise one of the predefined errors in such case. However, the `fn:doc` function is allowed to return a document with error information to prevent the transformation to blow up in XSLT 2.0. If such a processor runs in XSLT 3.0 mode, it is likely that it will raise a proper error instead and that you can catch that specific error. This also touches on a limitation of the error catching mechanism: processors are typically not allowed to raise error codes other than the ones specified in the specification, unless they cover a situation not covered by the specification[1].

[1] It is possible that this may change and that processors may define more specific errors than the ones available in the specification, but whether this proposal will make it into the final Recommendation is unclear at this moment.

- Repeated code with the same arguments for `fn:doc` and `fn:doc-available`, which, as any repeated code, is a cause of concern and maintainability, is not required anymore in scenarios such as this.

3.3. Caveats

The biggest caveat that may come as a surprise to the unaware is when and how an error is raised. Stylesheets can be processed in any order and instructions can be rearranged as a processor sees fit as long as it doesn't change the semantics or the result. In cases of try/catch, a processor must keep the instructions inside the try/catch wrapper, it cannot rearrange it such that an instruction falls outside the try/catch. However, lazy initialization of variables may mean that if such a variable contains an error, and it is used only inside a try/catch block, you might expect the error to be raised and catchable within that block, but they are not catchable in that way as it would pose a serious processor dependency and it would violate the scoping rules.

To catch errors that are raised inside a variable, the body of the variable should be wrapped in a try/catch instead and the handling of the error conditions should take place inside the variable itself.

This can still mean that the error is not raised upon priming the stylesheet, but only when the variable is actually used, which makes any reliance on the order in which errors are raised futile.

This behavior is illustrated in the following example:

```
<xsl:variable name="haserror" select="1 div 0"/>
<xsl:template match="/">
  <xsl:try>
    <xsl:value-of select="$haserror"/>
    <xsl:catch>
      <!-- never triggered -->
      <xsl:text>Error raised</xsl:text>
    </xsl:catch>
  </xsl:try>
</xsl:template>
```

The variable `$haserror` does a division by zero. While this can be statically detected, if the variable is never used, the processor does not need to raise the error and furthermore, it is not required to statically raise the error. Most processors will raise the error once it is used, in this case in the matching template. However, the `xsl:catch`, here with an absent `errors` attribute to catch all errors, will never be entered, because the error is lexically in the declaration of the variable and even though it is raised within the try/catch block, it will not be catchable from there.

Rewriting this code as follows would catch the error inside the variable, either upon priming the stylesheet, or upon using the variable, but in both cases, the error is thrown and caught from within the scope of the variable declaration. An extra try/catch block within the termplate would therefore be meaningless:

```
<xsl:variable name="haserror">
  <xsl:try>
    <xsl:value-of select="1 div 0"/>
    <xsl:catch>
      <xsl:text>Error raised</xsl:text>
    </xsl:catch>
  </xsl:try>
</xsl:variable>
<xsl:template match="/">
  <!-- will select 'Error raised' -->
  <xsl:value-of select="$haserror"/>
</xsl:template>
```

This last example also emphasizes that the processor cannot raise the error statically anymore, because it is inside a try/catch block and because the error itself is marked as dynamic. Of course, as with anything, if a processor can evaluate the whole sequence constructor statically, including catching the error and setting its value to the value of the catch block, it is allowed to do so.

3.4. Limitations

The difference between dynamic and static errors is not always as clear-cut as one might expect. For instance processors may detect certain type errors statically, in which case catching these errors dynamically is not possible. Whether a processor chooses to raise such errors dynamically or not is implementation dependent.

Another limitation is related to rolling back the output. If it is not possible to rollback the output, a dynamic error will be raised from the `xsl:try` instruction and any containing `xsl:try` instruction will raise this error as well. This only occurs when the setting for `rollback-output` is `no`. If it is `yes`, the processor will cache the result of the `xsl:try` body and will always be capable of rolling back.

As mentioned in the example above, it may be cumbersome to catch more specific errors than the rather generic errors that are defined in the specification. If such a mechanism is not going to be provided, catching more specific errors can only be done by investigating the information in the `$err:value` or the `$err:description` variables.

It is not possible to catch errors in declarations that do not have a sequence constructor (such as the select expression of a parameter), or to catch dynamic errors caused by serializing the output to the principal result document. It is possible, however, to catch errors raised by serializing to secondary result documents, by wrapping `xsl:result-document` inside a try/catch block. An alternative for catching errors from within expressions is to wrap the expression in a function that itself has a try/catch block. This is not very flexible however unless you would resort to dynamic evaluation using `xsl:evaluate`, which has its own drawbacks. A mechanism for catching errors purely within XPath is not (yet) available[1].

4. Forcing statically declared modes to prevent type errors

Availability in XSLT 2.0: no related mechanism, known extensions or workaround exists.

In all XSLT versions, including XSLT 3.0, when you apply templates to a mode, it magically exists. There is no type- or name-checking of any kind. In XSLT 3.0, a new feature was introduced on packages to force declaration of modes through `xsl:mode`, which makes it a static error if you try to use a mode that is not declared.

In a way, this feature is the same as for statically typed OO and other languages, where misstyping the name of a method, class or property would raise a static error.

4.1. Syntax and use

The new `declared-modes`[2] attribute is only available on the `xsl:package` element. To use this feature, you do not need to create a packages hierarchy. A package, if it is the principal package, behaves the same as the principal stylesheet: you can replace the `xsl:stylesheet` or `xsl:transform` root element by `xsl:package`.

The default of this attribute is `true`, which means that even if you do not specify this attribute, you will need to declare all modes that you are intending to use, as soon as you write packages. If you prefer the behavior of implicit modes, you can get the classical XSLT 2.0 behavior back by setting this value to `false`.

This attribute, and its default, applies equally to named and unnamed modes.

Example of its use:

```
<xsl:package
  version="3.0"
  declared-modes="true"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:mode on-no-match="shallow-copy"/>

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

This example just sets the default of `declared-modes`, to make it explicit and has a single template that changes a misspelled element from `<naam>` to `<name>` in the default mode, which is declared in the only (unnamed) `xsl:mode` declaration. Furthermore, this mode is declared to shallow-copy any nodes not matched, meaning that it is not required to extend this stylesheet with a typical copy-idiom.

If the (default) mode was not declared, static error XTSE3085 would be raised.

4.2. Improving your code

There are two typical cases that often cause frustration, especially in moderate to larger XSLT projects. Consider the following XSLT snippet:

```
<xsl:stylesheet version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="para">
    <xsl:apply-templates select="meta"
      mode="processmeta"/>
  </xsl:template>

  <xsl:template match="meta" mode="process-meta">
    <xsl:text>Changes made by: </xsl:text>
    <xsl:value-of select="change/author/@name"/>
  </xsl:template>
</xsl:stylesheet>
```

This, and many variants thereof, are common causes of frustration, because there is nothing wrong with the stylesheet at first sight, until you find that you made a typo in the `mode` attribute of `xsl:apply-templates`. The output here would be the text value of the nodes, because it will trigger the default templates that always exist. Quite possibly, the programmer would then try to find out why none of the templates match (a typical cause of

[1] Also the new XPath 3.1, now in Candidate Recommendation phase, does not include a mechanism for catching errors.

[2] The current text of the specification contains both `declared-modes` and `declare-modes`, either of which is a typo, my assumption is that the attribute name will be `declared-modes` in the final specification. See also Bug 28232.

this behavior is the wrong or missing namespace) only to find out much later that you incorrectly wrote `mode="processmeta"`.

If instead the processor would have raised an error that `processmeta` did not exist, you would much quicker have found the issue.

In XSLT 3.0 you should write your stylesheets starting with `xsl:package` instead of `xsl:stylesheet` or `xsl:transform`, which will automatically add the need to declare modes before their use. In most if not all cases, it means simply replacing the root element of your stylesheet, like so (it is not necessary to include `declared-modes="true"` because that is the default):

```
<xsl:package version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="para">
    <xsl:apply-templates select="meta"
      mode="processmeta"/>
  </xsl:template>

  <xsl:template match="meta" mode="process-meta">
    <xsl:text>Changes made by: </xsl:text>
    <xsl:value-of select="change/author/@name"/>
  </xsl:template>
</xsl:package>
```

The result is now error `XTSE3085` on the lines containing `xsl:apply-templates` and the declaration `xsl:template`, because the modes have not been declared. To fix that, we must add the necessary `xsl:mode` declarations, for instance as follows:

```
<xsl:package version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:mode/>
  <xsl:mode name="process-meta"
    on-no-match="shallow-skip"/>

  <xsl:template match="para">
    <xsl:apply-templates select="meta"
      mode="processmeta"/>
  </xsl:template>

  <xsl:template match="meta" mode="process-meta">
    <xsl:text>Changes made by: </xsl:text>
    <xsl:value-of select="change/author/@name"/>
  </xsl:template>
</xsl:package>
```

Note the empty `xsl:mode` declaration, which is required for the unnamed mode (the one used in the first template). After this change, the processor will still raise `XTSE3085`, this time pointing to the line containing `mode="processmeta"`. The typo is now detected and changing it to the appropriate spelling solves the problem.

We can improve on this further, though. An error that I still make myself quite often and I like to think I am not the only one, is forgetting to select the current mode:

```
<xsl:package version="3.0" declared-modes="true"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:mode/>
  <xsl:mode name="process-meta"
    on-no-match="shallow-skip"/>

  <xsl:template match="para">
    <xsl:apply-templates select="meta"
      mode="process-meta"/>
  </xsl:template>

  <xsl:template match="meta" mode="process-meta">
    <xsl:apply-templates select="change/author"/>
  </xsl:template>

  <xsl:template match="author" mode="process-meta">
    <xsl:text>Changes made by: </xsl:text>
    <xsl:value-of select="@name"/>
  </xsl:template>
</xsl:package>
```

There is nothing inherently wrong with this example from the point of view of the processor. All modes have been declared, but it is still not working. Why? Because in the first template matching `meta`, I have forgotten to include `mode="#current"` or `mode="process-meta"` on `xsl:apply-templates`, resulting in switching to the default unnamed mode.

It is a common mistake. To remedy this, you should always remove the unnamed mode from the declarations and *only use named modes*. This may seem like a big extra effort, but once you get used to it, you will get clearer and easier to read code and the processor will be able to more easily catch coding errors, resulting in quicker and better development overall. The resulting stylesheet (or better: stylesheet package) now becomes:

```
<xsl:package version="3.0" declared-modes="true"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:mode name="process-meta"
    on-no-match="shallow-skip"/>

  <xsl:template match="para" mode="main">
    <xsl:apply-templates select="meta"
      mode="process-meta"/>
  </xsl:template>

  <xsl:template match="meta" mode="process-meta">
    <xsl:apply-templates select="change/author"
      mode="#current"/>
  </xsl:template>

  <xsl:template match="author" mode="process-meta">
    <xsl:text>Changes made by: </xsl:text>
    <xsl:value-of select="@name"/>
  </xsl:template>
</xsl:package>
```

This change also requires initiating the processor with a different default mode. All processors support this. Using a common name for the named default mode and fixing all your invocation scripts, code or command lines, will fix this. A simpler solution is perhaps using a new XSLT 3.0 feature to initiate processing by a default mode, by setting the `default-mode` attribute on the outermost element, in this case `xsl:package`. This will force the processor to use that mode as the initial mode:

```
<xsl:package
  version="3.0"
  declared-modes="true"
  default-mode="main"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  .....
```

This idiom, with an initial default mode and an absent unnamed template declaration, forms a good basis for a cleaner programming style where the entry point is obvious and errors resulting from incorrect mode names becomes a thing of the past. If you prefer the (declared) default mode to be the unnamed mode, change this to have `default-mode="#unnamed"`. See also the next section, which goes deeper into the `default-mode` attribute.

4.3. Caveats

There are not many caveats, in fact, this feature removes many caveats you may have had with typical XSLT programming. The only thing to look out for is one of education: everyone in your team should stick to the new programming style.

Another thing to be aware of is that this feature is not available on `xsl:stylesheet`, as mentioned above. That means that once you have changed your principal stylesheet into a package, any imported and included modules (with `xsl:import` or `xsl:include`), directly or indirectly, will have to use declared modes as well, regardless the fact that those may still be rooted at an `xsl:stylesheet` element. In fact, it is not even possible to simply rename those into `xsl:package` for consistency, because it is not allowed to include or import packages. This can cause confusion during development, because while you are developing the individual imported modules, no static errors will be raised when compiling or testing them. Only after importing them in the main package, missing mode declarations will be found and trigger an error.

One way around this is to change the imported modules into packages, giving them a name, making all declarations public using a single `<xsl:expose names="*" visibility="public" />` and changing the `xsl:import` into an `xsl:use-package` referencing the new name.

4.4. Limitations

There are several limitations you should be aware of when using this feature, most have already been mentioned in previous sections:

- The feature is not backward compatible and cannot be applied on `xsl:stylesheet` or `xsl:transform`.
- Once introduced in the main stylesheet by changing it into a package, its influence extends to any imported stylesheets, which may cause confusion because they start with `xsl:stylesheet` or `xsl:transform`, which on itself does not have this option.
- It does not extend to used packages, if a used package, imported through `xsl:use-package` has `declared-modes="no"`, that package will not need to have declared modes.
- The default is that modes must be declared. Changing your stylesheet into a package requires adding an `xsl:mode` declaration for every mode you use, or if you don't want that, you should add `declared-modes="no"`.
- If you only use named modes, the standard default mode, which is the unnamed mode, will no longer be available and will cause errors when invoking your stylesheet. You can fix this by setting the default mode on the `xsl:package` by using `default-mode="yourdefaultmode"`.

5. Setting an entry point for your XSLT stylesheet

Availability in XSLT 2.0: In previous versions, only the default mode could be set by the invocation API of the processor, which defaulted to the unnamed mode if not set. The initial template did not have a default and could only specifically be set by the invocation API of the processor.

The working group considered that there was no common way to invoke a stylesheet without any arguments, or with only an initial match selection set, unless you would invoke the unnamed default mode. To fix this scenario, it is now possible to set the default mode on the outermost element (or any other element) and you can use a default initial named template as default entry point of your transformation.

5.1. Syntax and use

You can set the default mode on the outermost element of your principal stylesheet or the top level package[1] by using the new `[xsl:]default-mode` attribute, which is also available on all declarations, instructions and literal result elements. When set on the outermost element, it will set the default mode for invoking the stylesheet, unless it is overridden by the API or commandline arguments of the processor. You can set the default mode to the special values `#unnamed` for the unnamed mode and `#default` for the default mode, which is typically the same as the unnamed mode, unless an ancestor element has another default mode set, or if the API of your processor has the ability to override the default mode.

You can set the default initial template by using the special name `xsl:initial-template`. Its intend is that in the absence of a name for the initial template, and if the stylesheet is supposed to be invoked with an initial named template, that it defaults to this name. In practice, this means that if you invoke your processor with only the stylesheet as its argument, it will automatically select this initial template as its starting point.

Example of its use:

```
<xsl:template name="xsl:initial-template">
  <xsl:text>Main entry-point</xsl:text>
</xsl:template>
```

5.2. Improving your code

It is a common scenario in stylesheet programming to switch to different modes, especially when processing a certain input multiple times. To forget to continue in the current mode is a common mistake. For instance, consider the following snippet:

```
<xsl:template match="/">
  <xsl:apply-templates mode="invoices"/>
</xsl:template>

<xsl:template match="invoice" mode="invoices">
  <xsl:value-of select="product"/>
  <!-- unintended switching back to
       the unnamed mode -->
  <xsl:apply-templates select="price"/>
</xsl:template>
```

In this example, applying templates on `price` ends up in using the unnamed mode, because the programmer forgot to specify a new mode. To prevent this from happening, you can use the new `default-mode` attribute, which is specifically helpful if the code is deeper nested or contains a larger number of `xsl:apply-templates` instructions. It is not required to specify both the `mode` and the `default-mode`, unless you want the default mode for the containing instructions to be different from the mode specified in the `mode` attribute:

```
<xsl:template match="invoice"
  default-mode="invoices">
  <xsl:value-of select="product"/>
  <!-- this will remain inside mode "invoices" -->
  <xsl:apply-templates select="price"/>
</xsl:template>
```

Another handy use of setting the default mode is setting it on the outermost element, i.e. `xsl:stylesheet` or `xsl:package`, which will instruct the processor that, in absence of an initial mode specified on the command line or by the API, to use the default mode specified in the outermost element.

A similar method now exists for the initial named template. Suppose you have a transformation that is independent of an input document, for instance because it gets its input elsewhere, a simplified way of coding your stylesheet, and of making it very clear where the

[1] Essentially, in the absence of `xsl:package`, a stylesheet that starts with `xsl:stylesheet` or `xsl:transform` is still considered a (nameless) implicit package, where the *principal stylesheet module* is the subtree rooted at the at the `xsl:stylesheet` or `xsl:transform` element. In the case of a package, the principal stylesheet module is the *package manifest*, which is essentially the body of the `xsl:package`. In previous versions of 3.0, the body of a package contained an `xsl:stylesheet` or `xsl:transform` element, but this is no longer true, the body *is* the principal stylesheet, the same way it used to be in XSLT 2.0 (except that the outermost element is now `xsl:package`). The package containing the principal stylesheet module is called the *top level package*, any other used packages are called *library packages*.

stylesheet starts, is to use the new `xsl:initial-template` name for the initial named template:

```
<xsl:stylesheet version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template name="xsl:initial-template">
    <xsl:apply-templates
      select="doc('settings.xml')
      /settings/input-doc"/>
  </xsl:template>

</xsl:stylesheet>
```

The `xsl:initial-template` takes the role of what `int main(int argc, char *argv[])` is for C and C++, and variants thereof for C#, Java and other languages: it is the entry point of your stylesheet and your processor will select it automatically in absence of a default mode and a default template name (though processors may vary and may require you to specifically select the initial template, however, its intent is that it is the default in the *absence* of an initial template name). For instance, in the case of Exselt, invoking the stylesheet with only an argument for the XML input tree and no other arguments, will invoke this stylesheet with an *initial match selection*[1] of the root of the input tree and an initial named template of `xsl:initial-template`. If there is also a default mode, the initial template must be in the default mode, or an error is raised[2].

5.3. Caveats

Typically, this new feature on default modes and initial template will work as expected. There are, however, a few things to keep in mind:

It is processor dependent how a processor invokes a stylesheet. As a result, it is also processor dependent how it behaves when there is not enough information to invoke a stylesheet. The `default-mode` and the `xsl:initial-template` have been introduced to define defaults in the absence of such information. But there is no way to tell a processor to be invoked with call-template invocation or apply-templates invocation from within the stylesheet itself.

If you invoke a stylesheet without an initial match selection, it can be expected that it will default to call-template invocation, as apply-templates invocation will immediately raise an error (the initial match selection is required for apply-templates invocation, as otherwise there is nothing to match on). But when there is an initial match selection, but no specific default mode, and there is an `xsl:initial-template` inside the stylesheet, the processor may choose to invoke the stylesheet in either call-template or apply-templates mode.

If you want to prevent your stylesheet to be called in either way, you can create a template in the default mode that raises an error. For instance:

```
<xsl:template name="xsl:initial-template">
  <xsl:text>Correct invocation</xsl:text>
</xsl:template>

<xsl:template match="." default-mode="#default">
  <xsl:message terminate="yes">
    Please invoke this stylesheet
    in call-template mode!
  </xsl:message>
</xsl:template>
```

5.4. Limitations

The newly introduced difference between the initial match selection and the global context item could have benefits for this feature in the sense that you could create a stylesheet with global variables that depend on the global context item, and an initial template that requires the initial match selection to be absent. However, at this moment, both Saxon and Exselt do not differentiate between the global context item and the initial match selection, but this may change in the future.

[1] In XSLT 3.0, the initial context and the initial match selection do not need to be the same anymore, though most processors are likely to default them to the same node. Also, it is not a requirement that this is a node, the initial match selection and the initial context item can both be atomic items as well.

[2] Whether or not an error is raised depends on the API. The default mode only has effect when the stylesheet is invoked using apply-templates invocation. In the absence of information whether the user wants apply-templates or call-template invocation, it is up to the processor to decide on a default. If there is no initial context item, the only sensible default is call-template invocation, which will select the specified initial template, or `xsl:initial-template` if no such initial template is specified. If there is a context item, the processor has the option to choose between call-template and apply-templates invocation, and is likely to choose apply-templates invocation as a default. In that case, the default mode will be in effect.

To test whether a processor differentiates between the global context item and the initial match selection, you can write a stylesheet like the following:

```
<xsl:mode name="none" on-no-match="deep-skip"/>

<xsl:variable name="globctx">
  <xsl:try>
    <xsl:apply-templates select="." mode="none"/>
    <xsl:text>
      There is a global context item
    </xsl:text>
    <xsl:catch>
      <xsl:text>
        There is no global context item
      </xsl:text>
    </xsl:catch>
  </xsl:try>
</xsl:variable>

<xsl:template match="." default-mode="#default">
  <xsl:text>
    Invoked with apply-templates mode
    There is an initial match selection
  </xsl:text>
  <xsl:value-of select="$globctx"/>
</xsl:template>

<xsl:template name="xsl:initial-template">
  <xsl:text>
    Invoked with call-template mode
  </xsl:text>
  <xsl:value-of select="$globctx"/>
  <xsl:try>
    <xsl:apply-templates select="." mode="none"/>
    <xsl:text>
      There is an initial match selection
    </xsl:text>
    <xsl:catch>
      <xsl:text>
        There is no initial match selection
      </xsl:text>
    </xsl:catch>
  </xsl:try>
</xsl:template>
```

This code works by catching an error in the event of the absence of a context item when applying templates to a context item (there is no requirement to use `.`, but it makes it explicit that we are testing the current context item).

There is another way of coding this by using the new `xsl:context-item` instruction, but that will raise an uncatchable error (you cannot wrap this instruction inside a try/catch block). Using the pattern above allows you to actually differentiate between the four different cases: no global context item and no initial match selection, no global context item and an initial match selection, a global context item and no initial match selection and a global context item and an initial match selection.

I have uses an empty catch-clause here because the specification does not give a specific error code for this scenario, though XTTE3090 seems a good candidate, but it is specific to cases where `xsl:context-item` is used.

6. Better performance with memoization

Availability in XSLT 2.0: no related mechanism exists, though Saxon has an extension attribute that works somewhat similarly, see [15].

For functions that require processor-intensive calculations or operations, it can be beneficial to instruct the processor to only do that calculation once for the same input. The new `cache` attribute on `xsl:function` was introduced for exactly that purpose.

6.1. Syntax and use

The `cache` attribute works as a hint to the processor. The processor is not required to follow that hint. The available values for this attribute are:

- `cache="no"` this is the default, with no hints to the processor on whether to cache this function or not.
- `cache="partial"` hints the processor to cache the result of the function, but not necessarily to all extend possible. For instance, if the cached value is large, it can decide to drop it to save memory, or if the invocations are localized, it can drop the cache when it goes out of scope. It can also decide to only cache a part of the function if it finds a way to do so.
- `cache="full"` hints the processor to cache as much as it can. Still, a processor is not required to actually cache anything, but it is a strong hint to do so. The difference between `full` and `partial` is that with `full` the processor is not supposed to worry too much about memory constraints, it should simply cache as aggressively as possible.

The applied caching is dependent on the supplied arguments. If the argument is a node, for instance, and the identity of that node is different but its contents is the same, the processor will not be able to do any sensible caching because different node identities means different arguments and different arguments means potentially different results. A processor is not allowed to cache in such a way that the outcome of the function would be different if the cache attribute would not be applied.

6.2. Improving your code

Scenarios where caching can be beneficial depend highly on the processor, your input, and the different arguments of the function. Here are some rules of thumb to follow when trying to improve performance by using memoization:

- Functions that take no arguments should always be cached, or turned into a variable, unless the returned value is newly constructed node and the identity of that node is important. In most cases the identity of the returned node will not matter for your operations, or you will return an atomic result, in which case adding `cache="full"` is the right thing to do. Even without adding this hint, processors are known to turn functions into global variables internally, which has the same effect. However, by adding this hint, you make this behavior processor-independent.
- Functions that take atomic arguments that change little and that do a significant amount of processing or calculations. To know whether the overhead of caching outperforms non-caching requires careful measurement. However, some scenarios are trivial to find, for instance if your function uses an `xsl:stream` or `fn:collection()` internally, which can be expensive and is typically non-stable, meaning that on each invocation the processor would be required to load the referenced documents again. If you don't expect these documents to change, you should add memoization.
- Functions that are recursive and the recursion is dependent on the input value, where the performance is `O(n)` or worse. For instance, suppose that if the input is `xs:integer(10)` and to calculate the result the function will call itself with the values `n - 1` (i.e., 9, 8, 7 etc), it is almost a requirement to add caching to prevent all these recursive calculations to happen again. Adding memoization in such cases will change an initial call with an argument `xs:integer(10)` from 10 iterations into 1 iteration, and a subsequent call of `xs:integer(12)` into only 2 additional iterations, instead of 12 without caching.
- Functions that depend on nodes but the supplied nodes are often identity-equal. For instance, suppose you have a function that returns the settings from the first child node of the root as a map. The settings node will be the same each time, so it makes sense to memoize this.

A trivial example can be seen in the calculation of factorial. There are several ways to implement the factorial function and most will be suitable for memoization, but let's have a look at a recursive implementation:

```
<!-- definition -->
<xsl:function name="f:factorial" cache="full">
  <xsl:param name="i" as="xs:integer"/>
  <xsl:sequence select="
    if($i = 0 or $i = 1) then 1
    else $i * f:factorial($i - 1)"/>
</xsl:function>

<!-- usage -->
<xsl:template name="xsl:intial-template">
  <xsl:value-of select="
    for $i in 1 to 12
    return f:factorial($i)"/>
</xsl:template>

<!--
expected output:
1 2 6 24 120 720 5040 40320 362880 3628800
                                 39916800 479001600
-->
```

Without the `cache="full"`, the factorial function will have to go over each recursive call again and again, which is computationally intensive. With the cache attribute in place, calculating each further factorial will require only one recursive call, which returns the cached result of the previous calculation (the function `cached-value` is meant to denote the implementor's internal function used for retrieving a cached value of a function):

- `f:factorial(1)` has no recursion and returns 1.
- `f:factorial(2)` calls recursively with `2 * f:factorial(1)`, which is now cached and returns `2 * cached-value(f:factorial#1, 1)`, which returns 2.
- `f:factorial(3)` calls recursively with `3 * f:factorial(2)`, which is now cached and returns `3 * cached-value(f:factorial#1, 2)`, which returns 6.
- `f:factorial(4)` calls recursively with `4 * f:factorial(3)`, which is now cached and returns `4 * cached-value(f:factorial#1, 3)`, which returns 24.
- .....
- `f:factorial(12)` calls recursively with `12 * f:factorial(11)`, which is now cached and returns `12 * cached-value(f:factorial#1, 11)`, which returns 479001600. Without the cache this would require 11 recursive function calls.

The total number of operations is now 12, whereas without the caching it would be 1 + 2 + 3 + ... + 11 + 12 = 78 operations. A potential performance gain of 600%.

Another example where caching can be beneficial is where a single call is already expensive. For instance, suppose the structure of your input XML contains a `head` section with settings that you want to use in your pattern matching. A trivial implementation can look something like the following:

```
<xsl:template
  match="project[f:setting(., 'debug')">
  ... implementation ...
</xsl:template>

<xsl:function name="f:setting">
  <xsl:param name="node"/>
  <xsl:param name="setting"/>
  <xsl:sequence select="
    $node/(/)/head/settings[name() = $node/@name]
    [setting = $setting]/@value"/>
</xsl:function>
```

This is a potentially expensive function to be executed for each matching pattern where it is used. It is, however, dependent on the current node, which is changing each time, which makes it senseless to apply caching as it is written. However, if we modify the function slightly, we do not have to change much in our code, but we have a potential large benefit if it is called many times with the same argments:

```
<xsl:template
  match="project[f:setting(@name, 'debug')]">
  ... implementation ...
</xsl:template>

<xsl:function name="f:setting" cache="full">
  <xsl:param name="setting-name" as="xs:string"/>
  <xsl:param name="setting" as="xs:string"/>
  <xsl:sequence select="
    $node/(/)/head/settings[name() = $setting-name]
    [setting = $setting]/@value"/>
</xsl:function>
```

This new function has changed its signature to accept strings instead of nodes, which are much easier to cache. By specifying this explicitly on the `xsl:param`, the processor will first atomize any node that is passed into it, which makes it independent on the node identity and a good candidate for caching. While this function is a very straightforward example, scenarios similar to these where a lookup table is used inside a function occur very often in XSLT programming scenarios. Instead of processing the tree each time, by caching the function subsequent invocations can become instantaneous, resulting in a better overall performance.

6.3. Caveats

The main caveat with using caching is the concern for memory constraints. If you would go about caching every function you write, it may be detrimental to performance because caching itself adds a little overhead and unless a function is called multiple times with the same arguments, this will decrease performance. In the event of functions that have arguments that take nodes, the chances that a function can be actively cached are small and it is not unlikely that with many function calls, your processor will run out of memory with `cache="full"`. If your scenario can benefit from caching and your function is called with many different arguments, consider using `cache="partial"`, to hint the processor to limit the maximum amount of cached results.

If your function generates new nodes, a processor would typically have a hard time deciding whether or not it can optimize this by returning the same node each time or by returning a new node with its own identity. To hint the caching implementation, an extra attribute was introduced, `identity-sensitive`, which can be `yes`, the default, or `no`. If it is `no`, the returned node is not identity sensitive and the processor can return the same node each time, allowing for better optimization. This attribute does *not* apply to the arguments passed to the function.

6.4. Limitations

Memoization is limited to available memory and to your processor supporting it to an extend useful for your scenario. But even if a processor does not support caching fully, using the feature will not change the result of your transformation. It may, however, run much longer, especially in regards to recursive function calls.

There is currently no information on how processors support `cache="partial"`. It can be expected that processors will allow user-control over the amount of data cached, for instance with a setting that sets the maximum amount of cacheable calls for a given function.

7. Simpler templates with text value templates

Availability in XSLT 2.0: not available, you would use `xsl:value-of` in combination with `xsl:text` instead.

We have all come to love the attribute value templates with the familiar curly braces that are abundant in XSLT transformations. This syntax has now become available inside any sequence constructor as well and makes writing certain stylesheets quite a bit more readable, and as a side-effect it reduces the number of instructions to type.

7.1. Syntax and use

To use text value templates[1], you first have to enable them by adding the attribute `expand-text="yes"` to an ancestor element. This attribute is available on every instruction, declaration and literal result element. Once enabled, curly braces get the same special meaning as they have in attribute value templates: any text outside curly braces is considered normal text, any text inside curly braces is considered and XPath expression and will be evaluated as if `xsl:value-of` was called with that expression in that position[2]. A typical example of using text value templates is the following:

```
<xsl:text expand-text="yes">
  Dear {$title} {$firstname} {$lastname},
  ....
</xsl:text>
```

7.2. Improving your code

Almost every XSLT programmer has encountered the situation where he ends up writing endless combinations of `xsl:text` and `xsl:value-of` to get his output nicely formatted the way he wants it. For instance, suppose you want to take some vanilla XML[3] and turn it into a comma-separated-value (CSV) format. In XSLT 2.0, you could do this as follows:

```
<xsl:template match="row">
  <xsl:apply-templates/>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template match="field">
  <xsl:text>"</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>",</xsl:text>
</xsl:template>

<xsl:template match="field[last()]">
  <xsl:text>"</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>"</xsl:text>
</xsl:template>
```

Not really rocket science, but using text value templates, it becomes quite a bit more readable:

```
<xsl:template match="row">
  <xsl:apply-templates/>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template match="field" expand-text="yes">
  <xsl:text>"{.}",</xsl:text>
</xsl:template>

<xsl:template match="field[last()]"
              expand-text="yes">
  <xsl:text>"{.}"</xsl:text>
</xsl:template>
```

Of course, you could set the `expand-text="yes"` on the outermost `xsl:stylesheet` or `xsl:package` element, so that you do not need to repeat again and again. To futher simply the code above, you can write this now without using `xsl:text`, and assuming you have placed the `expand-text="yes"` at the stylesheet level, the final code looks like:

```
<xsl:template match="row">
  <xsl:apply-templates/>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template match="field">"{.}",</xsl:template>

<xsl:template
  match="field[last()]">"{.}"</xsl:template>
```

This particular feature is surprisingly trivial to use and once you get used to it, same as with attribute value templates, you find yourself using it everywhere[4].

7.3. Caveats

Using curlies within constructs that allow text value templates requires you to escape them as respectively {{ and }}.

Whitespace handling is not quite what you might expect. Consider the following example:

```
<xsl:text expand-text="yes">
  {@first-name}
  {@last-name}:
  {@age} years
</xsl:text>
```

[1] Attribute value templates occur in attributes inside curly braces. Text value templates occur inside any sequence constructor where literal text is allowed. Collectively both types are called *value templates* in the specification,

[2] While this is indeed typically how it works, any potential `@separator` on the `xsl:value-of` has no effect on text value templates.

[3] This is not a defined term anywhere, but when used, it typically means a flat XML file with a single root node, an element for rows and an element for each field in the row

[4] And exception can be cases where for clarity's sake it is simply better to write it out in XSLT instructions instead. As with any feature, use it when it improves your code, but stick to other techniques if it means that code becomes less readable.

This is very readable and you may have put the items on separate lines precisely to make it more readable. But other than with attribute value templates, where whitespace is normalized because it is inside attributes, that does not apply here. Whitespace is significant in sequence constructors as soon as there are text nodes inside it (other than the ones introduced with `xsl:text`), and when not inside a sequence constructor, as in this example, whitespace is significant because whitespace is always significant inside `xsl:text`. As a result, this example, when processed, will look as follows[1]:

```
John
Doe:
23 years
```

To prevent this from happening, simply remove the white-space by coding it differently:

```
<xsl:text expand-text="yes">{@first-name} {@last
-name}: {@age} years</xsl:text>
```

But this can get ugly. Another way of writing this is:

```
<xsl:text expand-text="yes">{
  @first-name,
  @last-name}:{
  @age} years</xsl:text>
```

Still not ideal, because we, as XSLT and XML addicts usually like to close an element on the same column it was opened[2]. Let's try yet another potentially (less) ugly way of coding this (using the string concatenation operator `||` from XPath 3.0):

```
<xsl:text expand-text="yes">{
  @first-name,
  @last-name
  || ': '
  || @age
  || ' years'
}</xsl:text>
```

It would have been nicer if the Working Group had decided to remove the whitespace, i.e. to let it work as if the curlies where replaced in-place by an `xsl:value-of` and if, after such expansion, no significant whitespace remains, the insignificant whitespace is removed or collapsed, similar to attribute value templates. However, processors typically remove insignificant whitespace prior to instruction expansion and the whitespace processor cannot distinguish between curlies that contain expressions and curlies that are normal text, which has been the main reason not to let it behave that way.

7.4. Limitations

Using text value templates is limited to places where `expand-text="yes"` is in scope. You cannot apply sequence normalization with a specified separator as you would with `xsl:value-of` (sequences inside a text value template are normalized with a space between the items and consecutive text value templates are concatenated without any separator).

Text value templates, similarly to attribute value templates, do not apply to themselves, that is, if the result of evaluating the expression contains curlies, this does not result in re-evalution of the result of the expression.

8. Improve production stability by introducing assertions

Availability in XSLT 2.0: limited, you could partially mimic it with `xsl:if` and `xsl:message` with `terminate="yes"`.

Assertions help in stabilizing your code by defining pre- and post-conditions for functions, input documents, templates and other constructs. If the assertion is not met, the transformation fails with an error, unless it is caught inside a try/catch construction. The advantage over using regular `xsl:if` or `xsl:message` is that assertions can be globally switched on or off.

8.1. Syntax and use

```
<xsl:assert>
  <!-- content -->
</xsl:assert>
```

Assertions can be inserted in your code with the new instruction `xsl:assert`. It behaves much the same like `xsl:message` combined with an `xsl:if`. If the `test` attribute evaluates to false, the assertion instruction will raise an error, which is by default `XTMM9001`, but can be set to anything else on the `error-code` attribute.

For instance, suppose that you want to ensure that the version attribute of the input is of at least a certain

[1] There is an opening whiteline and a closing whiteline, which may not be visible depending on the chosen rendering of the original DocBook code of this paper.

[2] This is certainly my preference, and from most examples floating around on the Net, this seems to be the preference of most programmers, though I do not have any authorative resource to proof that.

value because you do not want to support older versions of the input, you could write:

```
<xsl:template match="header" expand-text="yes">
  <xsl:assert test="number(@version) ge 2.0">
    <xsl:text>
      Incorrect version: {@version}
    </xsl:text>
  </xsl:assert>
  <xsl:apply-templates/>
</xsl:template>
```

Similarly to `xsl:message`, the result of evaluating the `xsl:assert` instruction is the empty sequence.

By default, assertions are switched on. Processors should support switching it off by any processor-dependent way through either command-line options or API settings. In the case of Exselt, you can switch assertions off by using the command-line option `-da=yes` or `-disable-assertions=yes`. For Saxon, I could not find such a global instruction, but it may be introduced in a future version[1].

8.2. Improving your code

You can use assertions to add invariants to your code, which makes testing your code easier and can give you better information in cases when something goes wrong. Once your XSLT program has been through rigorous testing and you know that your pre- and post-conditions and invariants are always true, you can switch off the assertions in subsequent processing.

A very common scenario for XSLT stylesheets is to depend on one or more external resources, for instance, auxiliary files, catalog files, lookup tables etc. In XSLT 2.0 you would write your code something like the following if you wanted to inform the user neatly if some required document was absent:

```
<xsl:template name="xsl:initial-template">
  <xsl:choose>
    <xsl:when
      test="not(
              document-available('settings.xml'))">
      <xsl:message terminate="yes">
        Settings.xml not found
      </xsl:message>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates
        select="document('settings.xml')"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

The extra required negation, plus the long-winded `xsl:choose` results in many programmers not going through all the trouble of fool-proofing their code. With `xsl:assert` it becomes a bit easier to do, plus the messages will have a bit more meaning: programmers seeing such an instruction in your code will immediately understand the purpose, while with a block of code like the one above, it requires a little bit more thought to find out that you are actually asserting something and not just having a normal program flow instruction.

With assertions, the above code becomes:

```
<xsl:template name="xsl:initial-template">
  <xsl:assert
    test="document-available('settings.xml')">
    <xsl:text>Settings.xml not found</xsl:text>
  </xsl:assert>
  <xsl:apply-templates
    select="document('settings.xml')"/>
</xsl:template>
```

This code, while more readable, may not do entirely what you expect: XSLT does not depend on order of execution. The earlier example above forced the order of execution by having and `xsl:choose` instruction. Several techniques can be used to force the order of execution. One such method is the following:

```
<xsl:template name="xsl:initial-template">
  <xsl:variable name="$doc" select="
    if(document-available('settings.xml'))
    then document('settings.xml') else ()">
    <xsl:assert test="$doc">
      <xsl:text>Settings.xml not found</xsl:text>
    </xsl:assert>
    <xsl:apply-templates select="$doc"/>
</xsl:template>
```

A similar problem arises with functions. Suppose you want to assert your input for non-zero values. You could do that as follows:

```
<xsl:function name="f:div">
  <xsl:param name="a"/>
  <xsl:param name="b"/>
  <xsl:assert test="$b != 0"
              select="'Cannot div by zero'"/>
  <xsl:sequence select="$a div $b"/>
</xsl:function>
```

But again, the error of divide-by-zero may kick in before the assertion is tested (if the arguments are integers, otherwise `INF` is returned and the assertion will kick in before the function ends). To avoid the error to be raised

[1] The commandline interface documentation currently does not seem to list such option. Once it becomes available it is likely that this documentation will get updated.

by the function, you would have to rewrite it somewhat. One approach could be to simply duplicate the logic:

```
<xsl:function name="f:div">
  <xsl:param name="a"/>
  <xsl:param name="b"/>
  <xsl:assert test="$b != 0"
              select="'Cannot div by zero'"/>
  <xsl:sequence
    select="$a div (if($b = 0) then 1 else $b)"/>
</xsl:function>
```

Once you have all your assertions in place, you probably want to be able to switch them on and off. The XSLT 3.0 specifcation does not mandate how this can be done, but suggests to do it using a `use-when` attribute. Personally, I think this is too much added clutter for a feature that should be as easy to use as possible. In fact, I think it is better to rely on the ability of the processor to switch this feature on and off. I think it ought to be off by default, but the specification states the opposite and turns it on by default.

8.3. Caveats

The main caveat was already discussed in the previous section: the inability to use `xsl:assert` in an as non-obtrusive way as possible, caused by the default behavior of how instructions are evaluated: in no pre-defined order.

There are some other things to look out for, though:

- Assertions should not have side-effects. The only way to cause side effects to kick in in an assertion is by using an extension function or instruction, either directly or indirectly. Such extension function could have a side-effect, like writing to a file, sending a message etc. This side-effect will be gone once the assertions are switched off which may result in unwanted and hard-to-predict behavior.
- Assertions should not be used to control program flow. I.e., you should not rely on the assertion to throw an exception and catch it, because again, once the assertions are switched off, the exception will never be caught and your program flow will behave differently than expected.
- Assertions have empty result. However, the specification states that assertions, other than other instructions, cannot be side-stepped for optimization purposes. That means that any code in the `test` attribute will always be executed.
- Assertions should not rely on unstable resources. If your code requires unstable resources like streams or collections[1], you should not write an assertion with `test` being dependent on such resource. Instead, use normal program flow instructions. The indeterministic nature of instructions such as `xsl:stream` make them a poor candidate for deterministic assertions.
- If you use assertions in global variables or parameters, be aware that the assertions only kick in if you actually use those variables, unless you processor does eager evaluation, though most, if not all, currently available processors are known to lazily evaluate variables.

8.4. Limitations

There are no inherent limitations with this instruction, other than limitations posed by your processor on instructions in general.

9. Meta programming with shadow attributes

Availability in XSLT 2.0: None. The only way to do meta-programming was to create a stylesheet with a stylesheet.

While this is a feature that could warrant a whole paper on itself, I will mention and touch on the essence of it here, as it is a seemingly small new feature, but one that is particularly powerful. Each and every attribute in your XSLT stylesheet can be turned into a shadow attribute and as such, it will take an attribute value template that is processed in the same way static expressions are evaluated for static parameters and `use-when` attributes.

9.1. Syntax and use

To create a shadow attribute, prepend it with an underscore. It then takes an attribute value template and takes precedence over any existing attribute by the same name. For instance, the following example uses an expression in the select statement of apply-templates that is equal to whatever atomized string is inside `$initial-select`:

```
<xsl:param name="initial-select" static="yes"/>

<xsl:template match="/">
  <xsl:apply-templates
    _select="{$initial-select}"/>
</xsl:template>
```

[1] Optionally, even `fn:doc` can be indeterministic. Here I use the term *unstable* to mean reliance on resources that by definition will cause your assertions to become non-deterministic

Because shadow attributes are evaluated in the static phase, they can only use static variables and parameters and whatever else, like document and other resources, that are available in the static context. The set of functions is limited to the functions defined in XPath, including the functions in the `map`, `array`[1] and `math` namespaces.

Any static variable or parameter that you want to use must be a global variable (static variables can never be local) and they must precede the element in which they are used in document order. You cannot use the current variable or parameter on itself, i.e. the following is illegal:

```
<xsl:param name="debug" select="true()"
           _use-when="$debug"/>
```

Shadow attributes can be any attribute, including the `static` attribute on `xsl:param` and `xsl:variable`, the `use-when` attribute and the `version` attribute on your outermost `xsl:stylesheet` or `xsl:package` declaration.

Attributes in a namespace, including attributes in the xsl namespace cannot be turned into a shadow attribute. As a result, default attributes such as `default-mode` on a literal result element appear as `xsl:default-mode` and they cannot be used as a shadow attribute by changing them into `_xsl:default-mode` or `xsl:_default-mode`.

Shadow attributes cannot appear on literal result elements as a result of this, they can only appear on XSLT instructions, declarations and other constructs.

9.2. Improving your code

The main purpose of this feature is to make it easier to write conditional inclusion of specific attributes. This was already possible using `use-when`, but for each variant of a particular instruction, it would require a full copy of the whole instruction. For instance, in XSLT 2.0 you would write something like the following to have a conditional include for a debug and a release import of a stylesheet:

```
<xsl:import href="release.xsl"
  use-when="if(document-available('release.xml')
            then true() else false()"/>
<xsl:import href="debug.xsl"
  use-when="if(document-available('debug.xml')
            then true() else false()"/>
```

The absence of the ability to use variables inside `use-when` and the inability to use it on attributes made it rather cumbersome in such situations. And suppose both the release and debug versions of the stylesheet are in the same path, it would become even more troublesome to code this correctly in XSLT 2.0. Enter shadow attributes:

```
<xsl:param name="version" static="yes"
           select="'release'"/>
<xsl:import _href="{$version}.xsl"/>
```

This simple example shows the power of the combination of static parameters with shadow attributes.

If we take that one big step further, you could, for instance, create a stylesheet that evaluates a user-input expression as follows:

```
<xsl:param name="expression" static="yes"
           select="()"/>
<xsl:template name="xsl:initial-template"
              expand-text="yes">
  <xsl:text>
    Evaluation expression: {$expression}
  </xsl:text>
  <xsl:value-of _select="{$expression}"/>
</xsl:template>
```

A stronger example of its power is when you have a stylesheet that must be executed in a certain order, dependent on a given input. There are multiple ways of programming such a requirement, but suppose you have an input configuration file as follows:

```
<config>
  <step name="run-invoices" />
  <step name="aggregate-invoices" />
</config>
```

And you have a stylesheet as follows:

```
<xsl:template name="run-invoices">
  <xsl:apply-templates select="//invoice"
                       mode="html"/>
</xsl:template>

<xsl:template name="aggregate-invoices">
  <xsl:apply-templates select="//invoice"
                       mode="aggregate"/>
</xsl:template>

<xsl:template name="send-invoices">
  <!-- the message listener is set to
       sending emails -->
  <xsl:message>
    <xsl:apply-templates select="//invoice"
                         mode="email"/>
  </xsl:message>
</xsl:template>
```

Depending on the flow, the moment of the week or other requirements, not all of these templates need to be

[1] The `array` namespace is only avaialable if your processor support XPath 3.1.

executed. As in the config file above, we only want to execute the `run-invoices` and the `aggregate-invoices`.

```
<xsl:variable name="step" static="yes"
  select="function($step) {
    (doc('config.xml')/config/step[$step],
     'none')[1]
    }"/>

<xsl:template name="none"/>

<xsl:template name="xsl:initial-template">
  <xsl:call-template _name="{$step(1)}"/>
  <xsl:call-template _name="{$step(2)}"/>
  <xsl:call-template _name="{$step(3)}"/>
  <xsl:call-template _name="{$step(4)}"/>
  <xsl:call-template _name="{$step(5)}"/>
</xsl:template>
```

This example uses a few advanced concepts, so let's go over them in detail:

- The static variable named `step` returns a function item that takes one argument. In XSLT 3.0 (and in XPath 3.0 using let-binding) it is possible to bind a function to a variable and to call the contained function using `$variablename($arg1, $arg2, ...)` syntax, i.e., the variable name, followed by parentheses containing the arguments. This is considered higher-order functions, which is a feature of XPath 3.0 and by extension a feature of XSLT 3.0.

  In this case, the returned function item has a body that opens a document `config.xml` and queries it for a `step` element based on the position given in the argument to the function.
- The result of this query is placed in a sequence of two. Since sequences eliminate empty sequences, a sequence like `((), 'none')` is the same as `('none')`. That means that if the step is not found, the first item will be the empty sequence and will be ignored. The filter expression at the end, `[1]` takes the first item of this sequence, which will be either the step, or the value `'none'` .
- The function is called five times, which suggests that our stylesheet can at most execute five steps. The result of a call like `$step(1)` is `'none'` or the name of whatever is in our configuration file at that location.
- After the static phase and with our example configuration file above, our stylesheet will look as follows:

  ```
  <xsl:template name="xsl:initial-template">
    <xsl:call-template name="run-invoices"/>
    <xsl:call-template name="aggregate-invoices"/>
    <xsl:call-template name="none"/>
    <xsl:call-template name="none"/>
    <xsl:call-template name="none"/>
  </xsl:template>
  ```
- Adding the no-op named template `none` helps us in keeping this code tidy and simple. If a step is not needed, it executed this no-op template and no harm is done.

While this particular use-case can be coded using traditional means as well, I used it here as an example of how powerful static expressions in conjunction with shadow attributes can be.

Another major use-case for shadow attributes and static parameters is for testing. For instance, suppose you want to test whether division works properly in XSLT, you could write a test-case as follows, where each argument, including the division operator, is parameterized:

```
<xsl:param name="numerator" static="yes"/>
<xsl:param name="denominator" static="yes"/>
<xsl:param name="operator" static="yes"/>
<xsl:param name="result" static="yes"/>

<xsl:template name="xsl:initial-template">
  <xsl:assert _select="{
    $numerator
    $operator
    $denominator}
    eq
    {$result}">
    <xsl:text>Not succeeded</xsl:text>
  </xsl:assert>
</xsl:template>
```

This code will either raise an error, or it will succeed in which case it returns nothing. The calling application can now simply define all the parameters for all cases it wants to test and, when these params are invoked with this stylesheet, the calling application only needs to check whether or not an error is raised.

While it is unlikely that you would want to test yourself for the stability of the division operators of a processor, it is an example that you can use in your own code, for instance by having it call your own functions with a static parameter for each argument. That would greatly simplify writing a testing framework for your functions.

9.3. Caveats

Any attribute that used to take either `yes` or `no` or `true` `false` can now take the corresponding synonyms. That means that any value that takes `yes` also accepts `true` and `1` and likewise, `no`, `false` and `0` are interchangeable. This was done to make it easier to use shadow attributes that operate on such boolean values. If you write an expression for a boolean shadow attribute, all your expression needs to do is evaluate to `true()` or `false()`, which will then be atomized as `true` or `false`.

If you have both a shadow-attribute and a non-shadow attribute, the shadow attribute takes precedence. For instance:

```
<xsl:text expand-text="yes"
          _expand-text="{$expand-text}"/>
```

The effective attribute for `expand-text` will be from evaluating the expression in `_expand-text`, the other attribute is ignored and may even contain an invalid value without leading to an error. For instance, if you would write the following, it is legal, as long as `$expand-text` evaluates to something sensible:

```
<xsl:text expand-text="invalid-value"
          _expand-text="{$expand-text}"/>
```

Other things to look out for is the order of evaluation of static parameters and variables: they must appear prior to their static usages in document order. Generally, the normal import precedence rules apply, however if a collision is detected, i.e. when two parameters have the same name and the same import precedence, than their effective values must be the same[1].

Be aware of the fact that shadow attributes take attribute value templates that in turn take static expressions and that `use-when` expressions take a static expression directly. This subtle difference is easily overlooked. I.e., the following will throw a compile-time error:

```
<xsl:param name="test" _static="true()"/>
```

Instead, write it as follows:

```
<xsl:param name="test" _static="{true()}"/>
```

9.4. Limitations

Shadow attributes take static expressions and static expressions, while allowing the full XPath syntax, are relatively limited. A few of the more prominent limitations are:

- Static expressions cannot reference variables or parameters other than static variables or parameters.
- Static variables or parameters must appear prior to their usage in document order.
- The set of statically known documents, collections and unparsed text resources is implementation defined. Make sure you check with your processor what set of documents it makes available. In the case of Exselt, the available documents is only limited by whatever the resource allocator is able to return, which is generally the same set as the dynamic set, assuming that the static and dynamic phase follow each other in the same environment.
- Static expressions are unlikely to be re-evaluated in a compiled stylesheet. If your processor supports compiled stylesheet, keep in mind that however you distribute your compiled stylesheet that your static expressions will be cast in stone. This is similar in behavior to constants in other programming languages, or compile directives in C.
- Shadow attributes do not apply to shadow attributes, that means, you cannot change a shadow attribute into a shadowed shadow attribute by prepending it with another underscore.
- Static variables and parameters are scoped to the current package. That means that you cannot override any static parameter or variables from a used package. XSLT 3.0 does not give any instrumentation to influence the static parameters in an `xsl:use-package` declaration, but your processor may[2].

10. Apply templates on atomic values

Availability in XSLT 2.0: None, it was not possible to apply templates or have an initial context item other than a selection of nodes.

XSLT 3.0 introduces new pattern syntax to be able to match on atomic values. Such patterns are called *predicate patterns*[3] and apply both to atomic items and nodes. They allow you to apply templates on other things than just nodes, for instance, a sequence of strings, singleton numeric values, maps, a sequence of functions etc.

10.1. Syntax and use

The syntax of a predicate pattern is surprisingly straightforward: a dot followed by one or more predicates. A dot without any predicates matches any item.

Examples of valid predicate patterns are:

- `.` matches any item, whether it is a node, an atomic value, a map an array or a function item.

[1] As a result of this rule, if your static parameter returns a function item, it cannot be compared if the same parameter appears with the same name and the same import precedence, because function items are not comparable.

[2] Packages are supposed to be allowed to be precompiled. As such, it makes no sense to allow static parameters to be overridden, as that would require you package to be recompiled each time it is used. Processors may, however, allow you to specify defaults for static parameters in packages, for instance, Exselt allows you to set parameters in the Package Catalog configuration file.

[3] This name seems a bit redundant, because many patterns can have predicates. But in the case patterns matching any item, the only thing to distinguish between one item and another, is by a predicate, hence the name.

- `.[self::para]` matches any item that is an element node that matches the node test `para`.
- `.[. = 'father']` matches any item that, when atomized, matches the string `'father'`.
- `.[. instance of xs:string]` matches any item that is a string.
- `.[. instance of function(*)]` matches any function item.
- `.[. instance of function(xs:integer, xs:integer) as item()]` matches any function item that takes two integers[1].
- `.[function-arity(.) = 3]` matches any function item that has arity 3.
- `.[position()]` matches any item that has a position. Atomic items do not have a position, so in effect, this only matches node items.
- `.[empty(.)]` matches nothing[2].
- `.[. instance of xs:integer][. le 100]` matches integer less then or equal to 100.
- `.[. castable as xs:double]` matches any item that can be cast to a double, like `xs:byte`, `xs:integer` and `xs:double`
- `.[..]` matches any item that has a parent, this is the same as `node()[..]`.
- `.[1]` matches any item, the position of atomic values inside a sequence is always 1.
- `.[xs:float(.)]` matches any item, including attributes and elements, that successfully parse as an `xs:float`.
- `.[matches(., '^.{2}-.{3}')]` matches any item that, after atomization, starts with two characters, followed by a dash, followed by three characters.
- `.[. instance of xs:string][matches(., '^.{2}-.{3}')]` limits the previous pattern to match only string items.
- `.[. instance of xs:string][not(.)]` matches any empty string.
- `.[. instance of map(*)]` matches any map.
- `.[. instance of map(*)][.('date')]` matches any map that has a key by the name `'date'`.
- `.[. instance of map(*)][let $keys := map:keys(.) return .($keys[1]) instance of map(*)]` matches any map that contains a nested map for each key[3].
- `.[. instance of xs:string][tokenize(., ',')[10]]` matches any CSV-style string with at least 10 fields in it.
- `.[. instance of xs:string or .[self::node()]][contains(., 'hello')]` matches any string or node that contains the word "hello".
- `.[.[self::number] or (. instance of xs:anyAtomicType and number(.) = 'NaN')]` matches either an element `number` or any atomic type that is convertible to a number.

Just as with other patterns, if a pattern raises an error, it is considered to be a negative match. Many examples above would raise an error depending on the item that is currently being matched, but these errors are never visible, they simply mean that the match failed.

10.2. Improving your code

The extra freedom of possibilities that opens up with this is so big that it is impossible to fit it into a short chapter here, but I'll give a few examples to get a general idea.

Suppose you want to process a CSV file. The XSLT 2.0 way of doing that would be inside a nested for-each loop, which can quickly get rather entangled. Using pattern matching, you can use a much cleaner approach[4]. Assuming a simple CSV style where we don't have to deal with escaping all kinds of corner cases and error scenarios, a straight-forward approach in XSLT 2.0 would look something like as follows:

```
<xsl:template name="main">
  <xsl:for-each select="
    tokenize(
    unparsed-text('file.csv'),
    '\r?\n')">
    <row>
      <xsl:for-each select="
        tokenize(., ',')">
        <cell>
          <xsl:value-of select="."/>
        </cell>
      </xsl:for-each>
    </row>
  </xsl:for-each>
</xsl:template>
```

[1] You cannot just test a function for its arguments or just for its return type. You must specify both the arguments and the return type.
[2] It does not literally *match* nothing, it actually never matches anything, because nothingness, as in empty sequences, cannot be matched.
[3] This is an example of how cumbersome it is to match nested maps. If you need matching nested maps, you are better of writing a bunch of handy functions to do the magic. While at some stage an alternative maps syntax was proposed, specifically for use in patterns, it never made it into the specification. Also note that the `map:keys` function returns the keys in an implementation-defined order, so this specific pattern is rather hit-or-miss and may behave differently under different browsers.
[4] I know that there are proponents of either of matching patterns and using for-each, my personal preference is matching patterns as it reduces nesting and generally "looks" cleaner and I just find it easier to read.

In XSLT 3.0, we can rewrite that as follows (warning: not necessary less lines):

```
<xsl:template name="xsl:initial-template">
  <xsl:apply-templates select="
    unparsed-text-lines('file'csv'''file'csv')"/>
</xsl:template>

<xsl:template match=".">
  <row>
    <xsl:apply-templates select="
      tokenize(., ',')" mode="cell"/>
  </row>
</xsl:template>

<xsl:template match="." mode="cell"
              expand-text="yes">
  <cell>{.}</cell>
</xsl:template>
```

Whether you like the approach of matching templates and patterns to structure your code or not is a personal flavor. The advantages of this approach are similar to normal push vs pull processing [16], where you are embracing change instead of trying to fight it. A more balanced comparison that favors neither approach can be find in [17].

In this example I simply matched *anything* using the dot-matches-all predicate pattern. The thought behind this is that we don't know what is inside the CSV so it doesn't make much sense to match on something more specific. Of course, we could change this matching pattern to `.[. instance of xs:string]`, but in this particular scenario it wouldn't add anything.

Another scenario is where we take the same CSV matching as above, but now we want to only process those rows that have a third column with a color and we only want the color "red". We could then do as follows:

```
<xsl:template
  match=".[tokenize(., ',')[3] = 'red']">
  <row color="red">
    <xsl:apply-templates select="
      tokenize(., ',')" mode="cell"/>
  </row>
</xsl:template>
```

Of course, we are now tokenizing twice and that is perhaps not very handy, but a good processor will cache this action, it may even recognize that the body of the template containing this match contains the same function call, and it can re-use the result without extensive caching. If not, and you want to explicitly cache this, you can write your own function and use the `cache="full"` to make it more performant. Though unless these rows are very large, I doubt it will bring much performance gain in this particular scenario, especially considering that building the result tree is likely much more expensive than breaking up a small string into smaller chunks.

Atomized values do not only appear inside unparsed text files. They can appear in generated sequences, or for instance in attributes that are of type `xs:token`. Here's an example, suppose your input is like the following, where the `ids` attribute contains the ISBN-10, ISBN-13 and the internally used ID value:

```
<books>
  <book ids="0618640150 978-0618640157 95867425"
        title="Lord of the Rings" />
  <book ... />
</books>
```

If we take this example and want to make the XML a bit more readable by expanding the `ids` attribute, a common task in XSLT processing[1], we could use the following transformation:

```
<xsl:mode on-no-match="shallow-copy"/>

<xsl:template match="book">
  <xsl:copy-of select="@* except @ids"/>
  <xsl:apply-templates select="
    tokenize(@ids, ' ')"/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match=".[string-length(.) = 10]"
              expand-text="yes">
  <isbn-10>{.}</isbn-10>
</xsl:template>

<xsl:template match=".[string-length(.) = 14]"
              expand-text="yes">
  <isbn-13>{.}</isbn-13>
</xsl:template>

<xsl:template match=".[string-length(.) = 8]"
              expand-text="yes">
  <internal-id>{.}</internal-id>
</xsl:template>
```

Essentially, this transformation works as an identity transform by virtue of the presence of `on-no-match="shallow-copy"`. By applying templates on the tokenized value of the `ids` attribute we create an easily manageable approach for any kind of items that is inside this tokenized attribute. Since we do not need the `ids` attribute anymore, we use the `except` expression to remove it from any attributes we might be copying that we do not yet know about (as opposed to explicitly copying only the `title` attribute).

As with the other examples in this paper, the `expand-text="yes"` is better suited to be placed on the outermost element of your stylesheet.

[1] Not necessarily this task, but enriching an input file, or turning attributes into elements and vice versa is very common.

10.3. Caveats

There are a number of things to be aware of when processing non-node items using pattern matching. A summary:

- The default priority of a predicate pattern without predicates is -1, which is lower than any other default priority.
- The default priority of any other predicatge pattern is 1 and is therefore higher than any other default priority. That means, for instance, that `.[self::para]` will match before `para` or even `para[@bold]` is matched.
- Errors in predicate patterns are ignored and result in a failing match.
- The default template rule matching an atomic item is to output that item unchanged, after applying the `fn:string()` function on it, except in the case of maps and functions, which are skipped.
- Because of the default priority rules, it doesn't matter how many predicates you use. The new default behavior in XSLT 3.0 for equal priority conflict resolution is to take the last in declaration order. This may lead to surprising results, to more often than not, when using this kind of pattern matching, you should resort to either using explicit priorities, or switch to different modes.
- Applying templates on string items that are significantly large can be detrimental to performance. In such case it is better to use pull processing and to split the string into smaller chunks if the problem domain allows that.

10.4. Limitations

You cannot mix predicate patterns with normal patterns. In fact, you cannot even mix a predicate pattern with a predicate pattern using the `union` operator, because that operator expects nodes on either side. If you want to match both nodes and other items in one pattern, you have to do that with a single predicate pattern, as some examples above show how to. If you want to mix multiple predicate patterns into a single pattern you should use the `or` operator, or if that doesn't work, you should split your pattern in multiple matching templates or place the matching logic inside a helper function. If the function is small enough, you can even do that inside the pattern itself:

```
<xsl:template match=".[let $f := function($m) {
  if($m instance of xs:string
      and string-length($m) gt 4)
      then true()
  else if($m instance of xs:integer
      and $m gt 9999) then true()
  else if($m instance of xs:float
      and $m lt 9.9999) then true()
  else
      false() }
  return $f(.)]" expand-text="yes">

  <xsl:text>{
  'Found an item "' || .
  ||' " that is '
  || 'a string of size above 4 '
  || 'or an integer larger than 9999 '
  || 'or a float smaller than 9.9999'
  }</xsl:text>

</xsl:template>
```

And you maybe surprised, but the above pattern, however absurd it may look, actually works. For instance, try to apply it to an integer:

```
<xsl:apply-templates select="12000"/>
```

Result:

```
Found an item "12000 " that is a string of size
above 4 or an integer larger than 9999 or a
float smaller than 9.9999
```

Other limitations include that it is far from trivial to match maps or items in maps. It gets even harder if you need to match on a map containing a map. A typical pattern to use when matching on the content of maps is build up of several steps.

Step 1, generically match a map of a certain type:

- `.[. instance of map(*)]` matches any map.
- `.[. instance of map(xs:integer, item())]` matches a map with *all* keys of type `xs:integer`, or an empty map.
- `.[. instance of map(xs:anyAtomicType, person)]` matches a map with *all* values being of element `person`, the key being any key, or an empty map.
- `.[. instance of map(xs:string, person)]` matches a map with *all* keys being of type `xs:string` and *all* values being of element `person`, or an empty map.

Step 2, match a specific item in the map (replace the three dots with anything from the previous step):

- `...[.('name') = 'John']` matches a map that has a key "name" bound to the value "John".
- `...[map:keys(.)[. = 'name']]` matches a map that has a key "name", regardless of its contents.
- `...[map:contains(., 'name')]` same as previous.
- `...[let $m := . return map:keys($m)!(if($m(.) = 123) then true() else ())]` matches a map that contains any value that equals 123[1].
- `...[.('invoice') instance of map(*)]` matches if a map item with the key "invoice" is itself a map item.
- `...[.('invoice')('billing')('total')]` matches if a map item contains a key "invoice" with a value of a map that contains a key "billing" with again a value of a map that contains a key "total" that is non-empty, not numerically zero and not `false()`.

In cases where you simply want to interrogate whether a certain key has a certain value, you can simplify this logic by writing expressions like the following:

- `.[.('invoice')('billing')('total')]` same as the last above, without the predicate determining whether it is a map. This works, because this pattern only ever matches if the currently matching item is indeed a map.
- `.[.('invoice')('total') lt 48.50]` matches a map that has key "invoice" mapping to a map that has a key "total" that itself contains value that is numerically less than `48.50`.

In general, not just for maps, to match certain items you will typically do an `instance of` to limit your matching to a given atomic type, which makes this kind of matching rather verbose to begin with. If you have an often repeated (part of a) pattern, you can wrap it inside a static variable and turn the match attribute in a shadow attribute:

```
<xsl:variable name="matchme" static="yes" select="
  . instance of xs:float or
  . instance of xs:integer or
  . instance of xs:double"/>

<xsl:template _match=".[{$matchme}][. = 0]">
  <xsl:text>Found a zero!</xsl:text>
</xsl:template>

<xsl:template
  _match=".[{$matchme}][. lt 10][. != 0]">
  <xsl:text>Found less then 10!</xsl:text>
</xsl:template>

<xsl:template _match=".[{$matchme}][. ge 10]">
  <xsl:text>Found greater/equal than 10!</xsl:text>
</xsl:template>
```

11. Improve performance helping the processor decide where to apply forking

Availability in XSLT 2.0: not available and no known extension mechanism that does the same exists, however, Saxon has a somewhat similar attribute that can be applied to `xsl:for-each` that specifies how many threads should be used, see [18].

The instruction `xsl:fork` is primarily intended for use with streaming, in that it enables a programming model that would otherwise require multiple passes over the input document, which is not always possible, let alone feasible, in streaming scenarios. At first sight, it looks like this instruction is a no-op in non-streaming scenarios, but because it introduces a logical forking of data, it can be used as a hint to the processor to open up multiple threads for each fork, and so to improve performance.

11.1. Syntax and use

```
<xsl:fork>
  <!--
    content, either:
    - zero or more xsl:sequence
    - one xsl:for-each-group
  -->
</xsl:fork>
```

[1] This particular pattern is not foolproof: if the map contains maps or function items, the item cannot be atomized and it will raise an error, resulting in a failing match. To accommodate for such scenarios, you will need to also test each map item whether it is a function item or not. Generally, once matching becomes this complex, it is better to wrap it in a function.

If you can split your sequence constructor into a bunch of `xsl:sequence` instructions, you can also modify it to use `xsl:fork`, because that instruction only takes `xsl:sequence` instructions as its children[1]. The `xsl:fork` instruction itself changes nothing to the generated sequences, the result would be exactly the same if the `xsl:fork` instruction were not there. A typical fork instruction could look something like the following:

```
<xsl:fork>
  <xsl:sequence select="beer"/>
  <xsl:sequence select="lemonade"/>
  <xsl:sequence select="other-drink"/>
</xsl:fork>
```

Here, the result would be the sequence of all `beer` elements, followed by the sequence of all `lemonade` elements, followed by the sequence of all `other-drink` elements. The only difference is that the processer gets a hint that it should use multi-threading. The user here says to the processor that it knows enough of the input data that the overhead of starting up new threads is insignificant compared to collecting these elements concurrently from the input tree.

Since XSLT is largely side-effect free, a strong hint like the one above can be easily picked up in a processor-independent way. At this moment, Exselt allows for this kind of hints to be followed up, I am not sure if Saxon also fires up multiple threads[2].

Note that, even without `xsl:fork`, processors are known to apply multi-threading where possible, often depending on the edition of the processor you have.

11.2. Improving your code

Suppose your XSLT 2.0 code looked like this:

```
<xsl:template match="publication">
  <xsl:apply-templates select="book"/>
  <xsl:apply-templates select="magazine"/>
  <xsl:apply-templates select="paper"/>
  <xsl:sequence select="f:sum-totals(.)"/>
</xsl:template>
```

Then you can turn that into a fork as follows:

```
<xsl:template match="publication">
  <xsl:fork>
    <xsl:sequence>
      <xsl:apply-templates select="book"/>
    </xsl:sequence>
    <xsl:sequence>
      <xsl:apply-templates select="magazine"/>
    </xsl:sequence>
    <xsl:sequence>
      <xsl:apply-templates select="paper"/>
    </xsl:sequence>
    <xsl:sequence select="f:sum-totals(.)"/>
  </xsl:fork>
</xsl:template>
```

This is obviously a lot more code, but if your input has a significant size and the individual sequences take some time to be processed, it makes sense to let the processor do these actions in parallel.

Whether or not it really improves performance will depend on processing vs. startup overhead, input size, computational intensity of the stylesheet and many other factors. To really find out whether it improves performance there's only one thing you can do: profile and measure it.

11.3. Caveats

Processors not supporting multi-threading or not supporting streaming, may remove the `xsl:fork` completely, as it doesn't change its outcome.

Using `xsl:fork` abundantly may result in many threads being opened, which in itself is not necessarily good for performance. It typically depends on the way threading is implemented (through enveloping, actors, tasks, divide and conquer algorithms, map/reduce algorithms, lightweight approaches or heavyweight approaches etc). As a result, on one processor it may be beneficial to have as many threads as there are processors, but not more, on other processors it can be beneficial to have as many "threads" (which in itself are not necessarily physical threads) as possible because the overhead is very light and the algorithm will automatically choose the best way to spread the tasks across waiting threads.

Again, the only way to find out what is best for a given scenario or processor is by profiling the stylesheet.

It is possible to have an empty `xsl:fork`, but this has no effect. This was only introduced to make it easier to auto-generate instructions like this and to make it orthogonal with the rest of the specification.

[1] Not entirely true. A late change, that just made it into the current Working Draft, allows `xsl:for-each-group` to appear as a child as well, which allows for a form of complex streamed grouping. For this scenario, we are only interested in `xsl:sequence` children.

[2] In fact, Saxon currently doesn't fire up new threads for streaming, but there is no mention of non-streaming scenarios in the Saxon documentation.

11.4. Limitations

There are no inherent limitations imposed by the XSLT language itself, though processors can either support this or not and it may be hard to actually find out to what level they support it if they do. In addition, for processors that will spawn actual threads, the physical processing environment may impose limitations on the amount of threads. Finally, different editions of processors may support different amounts of CPUs, threads or otherwise.

12. Conclusion

XSLT 3.0 comes with a host of new functionality that can improve your current XSLT 2.0 programming experience significantly. This paper introduced some of the possibly lesser known improvements that have been made to the language in the past couple of years. Many of the "larger" improvements have been covered in web blogs, papers and conferences, but at least some of the smaller improvements have not. By showing how these seemingly small changes can improve your programming experience, clarity of programming or speed of execution, these changes, or at least some of them, have now received the attention to detail as they deserve.

This paper can impossibly be complete, the features discussed here are, in my opinion, significant, but many other features are as well. I hope that in a subsequent paper I can discuss some of these other features that certainly warrant to be known just as much as these.

Bibliography

[1] *XSL Transformations (XSLT) Version 3.0, Latest Version*. Michael Kay.
http://www.w3.org/TR/xslt-30/

[2] *XSL Transformations (XSLT) Version 3.0, W3C Working Draft 1 February 2013*. Michael Kay. World Wide Web Consortium (W3C).
http://www.w3.org/TR/2013/WD-xslt-30-20130201/

[3] *Bugzilla - Public W3C Bug / Issue tracking system*. 2014. Miscellaneous authors.
https://www.w3.org/Bugs/Public/

[4] *XML Path Language (XPath) 3.0, Latest Version*. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xpath-30/

[5] *XQuery and XPath Data Model 3.0, W3C Candidate Recommendation 08 January 2013*. Norman Walsh, Anders Berglund, and John Snelson. World Wide Web Consortium (W3C).
http://www.w3.org/TR/2013/CR-xpath-datamodel-30-20130108/

[6] *Requirements and Use Cases for XSLT 2.1*. Petr Cimprich. 2010. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xslt-21-requirements/

[7] *The EXPath Packaging System*. Florent Georges. Proceedings of Balisage 2010.
doi:10.4242/BalisageVol5.Georges01

[8] *Collaboratively Defining Open Standards for Portable XPath Extensions*. Collaborative.
http://expath.org/

[9] *Analysing XSLT Streamability*. John Lumley. Proceedings of Balisage 2014.
doi:10.4242/BalisageVol13.Lumley01

[10] *A Streaming XSLT Processor*. Michael Kay. Proceedings of Balisage 2010.
doi:10.4242/BalisageVol5.Kay01

[11] *Streaming in XSLT 2.1*. Michael Kay. Proceedings of XML Prague 2010.
http://archive.xmlprague.cz/2010/presentations/Michael_Kay_Streaming_in_XSLT_2.1.pdf

[12] *Efficient XML processing with XSLT 3.0 and higher order functions*. Abel Braaksma. Proceedings of XML Prague 2013.
doi:10.4242/BalisageVol13.Braaksma01

[13] *In pursuit of streamable stylesheet functions in XSLT 3.0*. Abel Braaksma. Proceedings of Balisage 2014.
doi:10.4242/BalisageVol13.Braaksma01

[14] *Try/Catch in XSLT 2.0*. Florent Georges. 2007.
http://fgeorges.blogspot.nl/2007/01/trycatch-in-xslt-20.html

[15] *saxon:memo-function*. Michael Kay. 2015. Saxonica.
http://www.saxonica.com/documentation/index.html#!extensions/attributes/memo-function

[16] *Advantages of push-style XSLT over pull-style*. E. Welker. 2008.
http://www.eddiewelker.com/2008/11/25/push-style-xslt-vs-pull-style/

[17] *XML for Data: XSL style sheets: push or pull?*. Kevin Williams. 2008.
http://www.ibm.com/developerworks/library/x-xdpshpul.html

[18] *saxon:threads*. Michael Kay. 2015. Saxonica.
http://www.saxonica.com/documentation/index.html#!extensions/attributes/thread

Continuous Integration for XML and RDF Data

Sandro Cirulli

Oxford University Press

<sandro.cirulli@oup.com>

Abstract

At Oxford University Press we build large amounts of XML and RDF data as it were software. However, established software development techniques like continuous integration, unit testing, and automated deployment are not always applied when converting XML and RDF since these formats are treated as data rather than software.

In this paper we describe how we set up a framework based on continuous integration and automated deployment in order to perform conversions between XML formats and from XML to RDF. We discuss the benefits of this approach as well as how this framework contributes to improve both data quality and development.

Keywords: Jenkins, Unit Testing, Docker

1. Introduction

Oxford University Press (OUP) is widely known for publishing academic dictionaries, including the Oxford English Dictionary (OED), the Oxford Dictionary of English (ODE), and a series of bilingual dictionaries. In the past years OUP acquired a large number of monolingual and bilingual dictionaries from other publishers and converted them into the OUP XML data format for licensing purposes. This data format was originally developed for print dictionaries and had to be loosened up in order to take into account both digital products and languages other than English. Conversion work from the external publishers' original format was mainly performed out-of-house, thus producing a large number of ad hoc scripts written in various programming languages. These scripts needed to be re-run each time on in-house machines in order to reproduce the final XML. Code reuse and testing were not implemented in the scripts and external developers' environments had to be replicated each time in order to rerun the scripts.

As part of its Oxford Global Languages (OGL) programme [1], OUP plans to convert its dictionary data from a print-oriented XML data format into RDF. The aim is to link together linguistic data currently residing in silos and to leverage Semantic Web technologies for discovering new information embedded in the data. The initial steps of this transition have been described in [2] where OUP moved from monolithic, print-oriented XML to a leaner, machine-interpretable XML data format in order to facilitate transformations into RDF. [2] provides examples of conversion code as well as snippets of XML and RDF dictionary data and we recommend to refer to it for understanding the type of data modelling challenges faced in this transition.

Since the OGL programme aims at producing lean XML and RDF for 10 different languages in its initial phase and for tens of languages in its final phase, the approach of converting data with different ad hoc scripts would not be scalable, maintainable, or cost-effective. In the following chapters we describe how we set up a framework based on continuous integration and automated deployment in order to perform conversions between XML formats and from XML to RDF. We discuss the benefits of this approach as well as how this framework contributes to improve both data quality and development.

2. Continuous Integration

Continuous Integration (CI) refers to a software development practice where a development team commits their work frequently and each commit is integrated by an automated build tool detecting integration errors [3]. In its simplest form it involves a build server that monitors changes in the code repository, runs tests, performs the build, and notifies the developer who broke the build [4] (p. 1).

2.1. Build Workflow

We adopted Jenkins [5] as our CI server. Although we have not officially evaluated other CI servers, we decided to prototype our continuous integration environment with Jenkins for the following reasons:

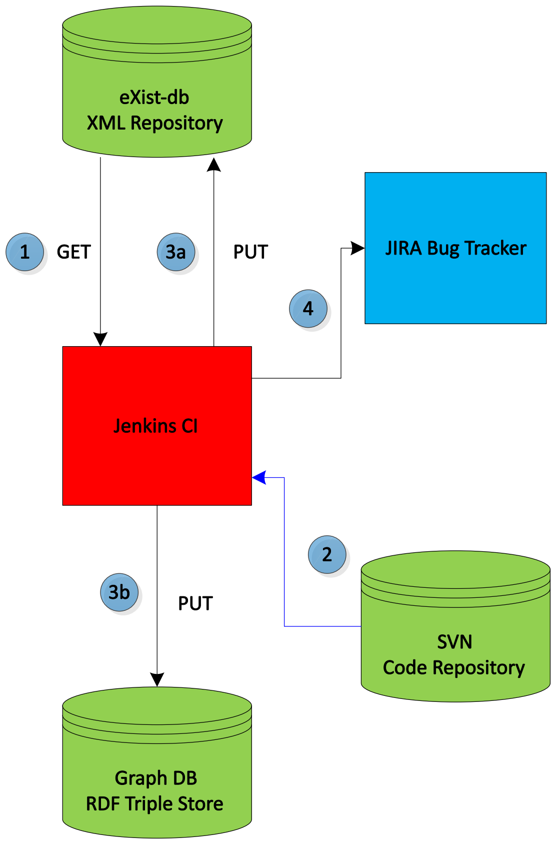
doi:10.14337/XMLLondon15.Cirulli01

- it is the most popular CI server with 70% market share [6]
- it is open source and allows to prototype without major costs
- it is supported by a large number of plugins that extend its core functionalities
- it integrates with other tools used in-house such as SVN, JIRA, and Mantis

Nevertheless, we reckon that other CI servers may have equally fulfilled our basic use cases. On the other hand, specific use cases may require different CI servers: for example, Travis CI may be a better choice for open source projects hosted on GitHub repositories due to its distributed nature whereas Bamboo may be a safer option for businesses looking for enterprise support in continuous delivery.

Figure 1, “Workflow and components for converting XML and RDF” illustrates the workflow and the components involved in converting and storing XML and RDF data via Jenkins.

Figure 1. Workflow and components for converting XML and RDF

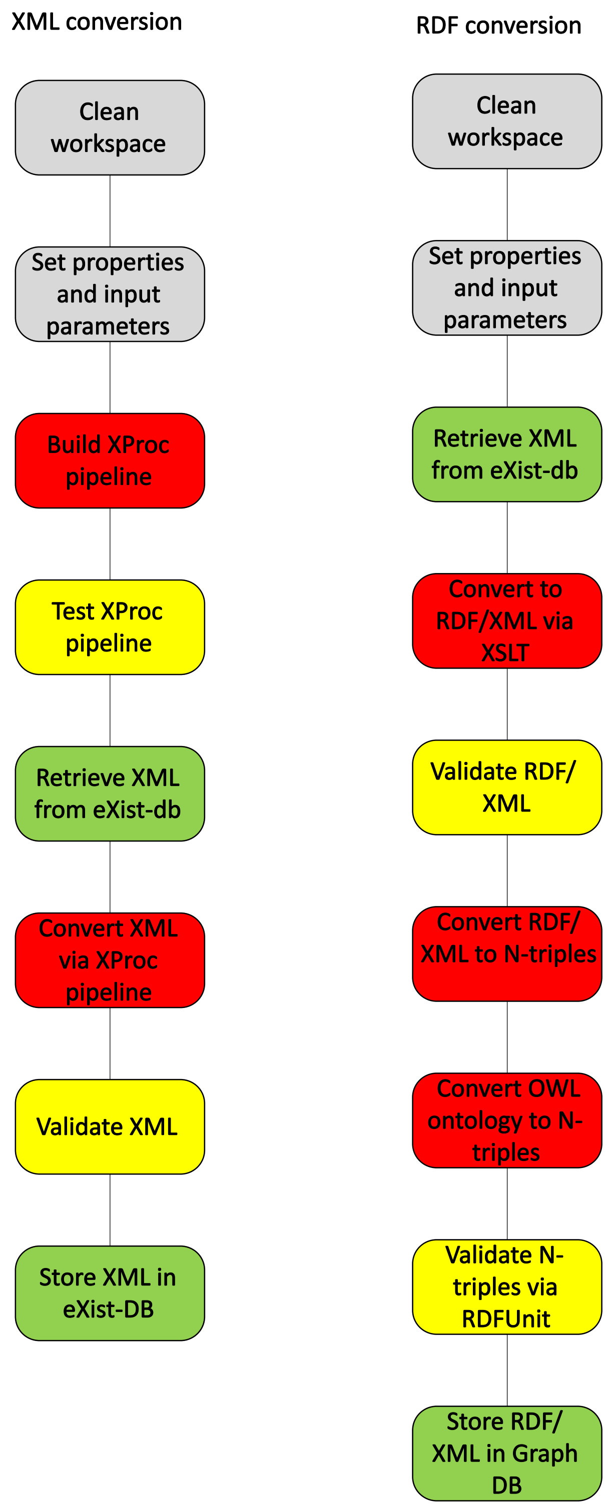


XML data in print-oriented format is stored on the XML repository eXist-db. The data is retrieved by Jenkins via a HTTP GET request (1). Code for converting print-oriented XML and building artifacts is checked out from the Subversion code repository and stored in Jenkins's workspace (2). The build process is run via an ant script inside Jenkins and the converted XML is stored in eXist-db (3a). Should the build process fail, Jenkins automatically raises a ticket in the Jira bug tracking system (4).

The same workflow occurs in the RDF conversion. XML data converted in the previous process is retrieved from eXist-db (1), converted by means of code checked out from Subversion (2), and stored in the RDF Triple Store Graph DB (3b).

The core of the build process is performed by an ant script triggered by Jenkins. Figure 2, “Build process steps for XML and RDF conversions” shows the steps involved in the build process for XML and RDF conversions.

Figure 2. Build process steps for XML and RDF conversions


XML conversion
Clean workspace
Set properties and input parameters
Build XProc pipeline
Test XProc pipeline
Retrieve XML from eXist-db
Convert XML via XProc pipeline
Validate XML
Store XML in eXist-DB
RDF conversion
Clean workspace
Set properties and input parameters
Retrieve XML from eXist-db
Convert to RDF/XML via XSLT
Validate RDF/XML
Convert RDF/XML to N-triples
Convert OWL ontology to N-triples
Validate N-triples via RDFUnit
Store RDF/XML in Graph DB

2.2. Nightly Builds

Nightly builds are automated builds scheduled on a nightly basis. We currently build data in both XML and RDF for 7 datasets and the whole process takes about 5 hours on a Linux machine with 132GB of RAM and 24 cores (although only 8 cores are currently used in parallel). The build process is performed in Jenkins via the Build Flow Plugin [7] which allows to perform complex build workflows and jobs orchestration. For our project the XML ought to be built before the RDF and each build is parametrized according to the language to be converted. The Build Flow Plugin uses Jenkins Domain Specific Language (DSL), a Groovy-based scripting language. In this case we used this scripting language as it ships with the Build Flow Plugin and the official plugin documentation provides several examples of complex parallel builds. Example 1, "XML and RDF builds for English-Spanish data" shows the DSL script for building XML and RDF for the English-Spanish dictionary data.

Example 1. XML and RDF builds for English-Spanish data

```
out.println 'English-Spanish Data Conversion'
// build lexical XML full data
build( "lexical_conversion",
      source_lang: "en-gb",
      target: "build-and-store",
      build_label: "nightly_build",
      input_type: "oxbiling",
      input: "full",
      target_lang: "es")
// build RDF full data
build( "lexical_rdf_conversion",
      input_source: "database",
      source_type: "dict",
      source_language: "en-gb",
      target_language: "es",
      target: "update-rdf")
```

Builds are run in parallel and make use of the multi-core architecture of the Linux machine. For our current needs Jenkins is set to use up to 8 executors on a master node in order to build 7 datasets in parallel. Compared to a sequential build run on a single executor, the parallel build reduced by several hours the total execution time of nightly builds. In the future we foresee to increase the number of executors as we convert more datasets and to run nightly builds and other intensive process on slave nodes in order to scale horizontally.

2.3. Unit testing

Unit testing was originally included in the build process. However, since the builds took several hours before producing results, we decided to separate the building and testing processes in order to provide immediate feedback to developers. We created validation jobs in

Jenkins that poll code repositories on the SVN server every 15 minutes and run tests within minutes from the latest commit. Should tests fail, a JIRA ticket is assigned to the latest developer who committed code and the system administrator is notified via email.

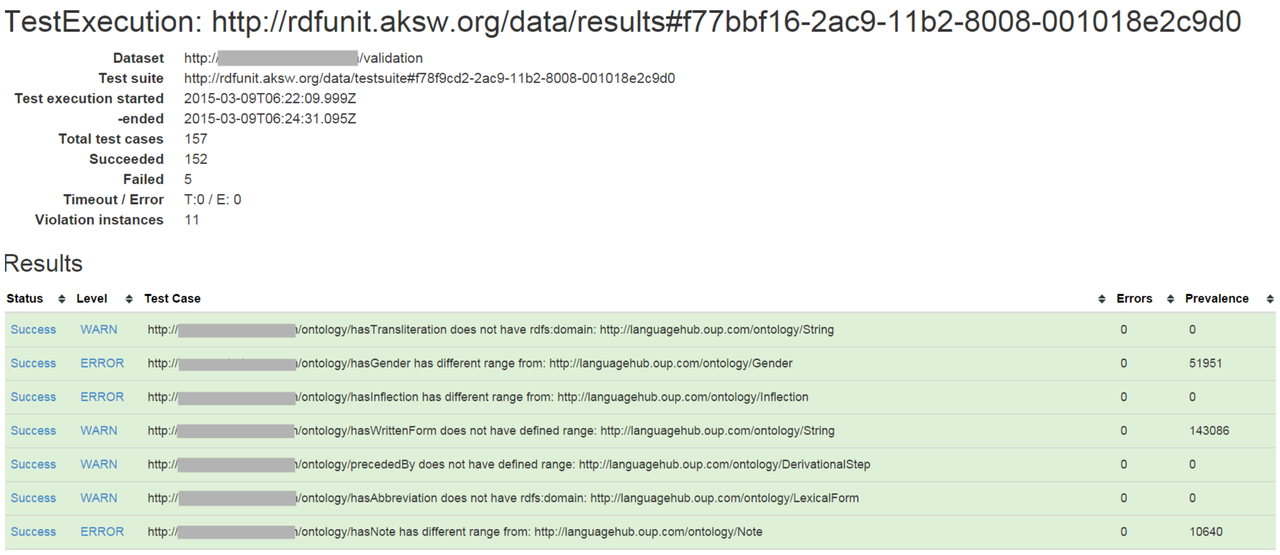
Unit testing for XSLT code is implemented using XSpec [8]. [9] suggested the use of Jxsl [10], a Java wrapper object for executing XSpec tests from Java code. We took a simpler approach which does not require the use of Java code. XSpec unit tests are run within the ant task as outlined in [11] and the resulting XSpec HTML report is converted into JUnit via a simple XSLT step. Since JUnit is understood natively by Jenkins, it is sufficient to store the JUnit reports into the directory where Jenkins would expect them to be in order to take advantage of Jenkins's reporting and statistical tools. Example 2, "XSpec unit test" shows how all the XSpec HTML reports are converted into JUnit within an ant script.

Example 2. XSpec unit test

```
<for param="file">
  <path>
    <fileset dir="${test.dir}" includes="**/*.xspec"/>
  </path>
  <sequential>
    <echo>convert XSpec test results into JUnit XML</echo>
    <propertyregex override="yes" property="basename" input="@{file}"
                   regexp=".+[\\/]([^\\/]+?)\.xspec" replace="\1"/>
    <xslt in="${test.dir}/results/${basename}-result.html"
          out="${test.dir}/results/${basename}-result.junit"
          style="${shared.dir}/xsl/xspec_to_junit.xsl" force="true">
      <classpath location="${saxon.jar}"/>
    </xslt>
  </sequential>
</for>
```

RDF data is tested using the RDFUnit testing suite [12] which runs automatically generated test cases based on a given schema. The output is generated in both HTML and JUnit. Figure 3, "Report for RDFUnit tests" shows a screenshot of the HTML report (the top level domain has been hidden for security reasons).

Figure 3. Report for RDFUnit tests



TestExecution: http://rdfunit.aksw.org/data/results#f77bbf16-2ac9-11b2-8008-001018e2c9d0

Dataset	http://[redacted]/validation
Test suite	http://rdfunit.aksw.org/data/testsuite#f78f9cd2-2ac9-11b2-8008-001018e2c9d0
Test execution started	2015-03-09T06:22:09.999Z
-ended	2015-03-09T06:24:31.095Z
Total test cases	157
Succeeded	152
Failed	5
Timeout / Error	T:0 / E: 0
Violation instances	11

Results

Status	Level	Test Case	Errors	Prevalence
Success	WARN	http://[redacted]/ontology/hasTransliteration does not have rdfs:domain: http://languagehub.oup.com/ontology/String	0	0
Success	ERROR	http://[redacted]/ontology/hasGender has different range from: http://languagehub.oup.com/ontology/Gender	0	51951
Success	ERROR	http://[redacted]/ontology/hasInflection has different range from: http://languagehub.oup.com/ontology/Inflection	0	0
Success	WARN	http://[redacted]/ontology/hasWrittenForm does not have defined range: http://languagehub.oup.com/ontology/String	0	143086
Success	WARN	http://[redacted]/ontology/precededBy does not have defined range: http://languagehub.oup.com/ontology/DerivationalStep	0	0
Success	WARN	http://[redacted]/ontology/hasAbbreviation does not have rdfs:domain: http://languagehub.oup.com/ontology/LexicalForm	0	0
Success	ERROR	http://[redacted]/ontology/hasNote has different range from: http://languagehub.oup.com/ontology/Note	0	10640

As shown in Figure 2, "Build process steps for XML and RDF conversions", the XProc pipeline for the XML conversion is built on-the-fly during the build process from a list of XSLT steps stored in a an XML configuration file. This approach simplifies and automates the creation of XProc pipelines for new datasets: for example, developers converting new datasets have to create and maintain a simple XML file with a list of steps rather than a complex XProc pipeline with several input and output ports. On the other hand, the generated XProc file needed to be tested and we therefore implemented unit tests using the xprocspec testing tool [13]. Example 3, “xprocspec unit test” shows a test that, given a valid piece of XML, expects the XProc pipeline not to generate failed assertions on the ports for Schematron reports.

Example 3. xprocspec unit test

```
<x:scenario label="test_fragment">
<x:call step="oup:main">
  <x:option name="source_lang" select="'@LANG@'"/>
  <x:input port="source">
    <x:document type="file"
                href="valid_fragment.xml"/>
  </x:input>
</x:call>
<x:context label="Schematron Validation">
  <x:document type="port"
              port="schematron_intermediate"/>
  <x:document type="port"
              port="schematron_final"/>
</x:context>
<x:expect type="xpath"
          test="count(//svrl:failed-assert)"
          equals="0"
          label="There should be no failed
                 Schematron assertions"/>
</x:scenario>
```

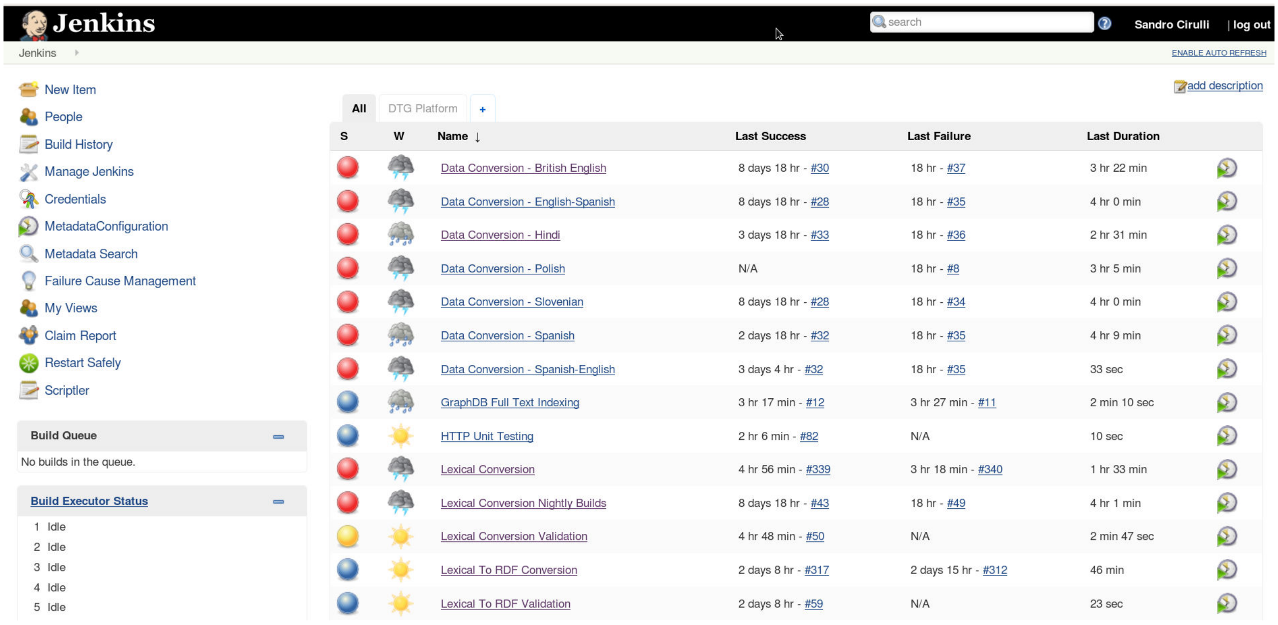
2.4. Benefits of Continuous Integration

Introducing Continuous Integration in our development workflow has been a big shift from how code used to be written and how data used to be generated in our department. In particular, we have seen major improvements in the following areas:

- **Code reuse**: on average, 70-80% of the code written for existing datasets could be reused for converting new datasets into leaner XML and RDF.
- **Code quality**: tests ensured that code is behaving as intended and minimized the impact of regression bugs as new code is developed.
- **Bug fixes**: bugs are spotted as soon as they appear, developers are notified instantly, and bugs are fixed more rapidly.
- **Automation**: removing manual steps made the building process faster and less error-prone.
- **Integration**: a fully automated building process reduced risks, time, and costs related to integration with existing and new systems and tools.

Figure 4, “Jenkins projects” and Figure 5, “Parametrized build” show respectively the list of Jenkins projects and a parametrized build inside the Lexical Conversion project. In Figure 4, “Jenkins projects” we illustrate on purpose a critical situation showing projects with failed builds in red, projects with unstable builds (i.e. failing unit tests) in amber, and project with successful builds in blue; the weather icon illustrates the general trend.

Figure 4. Jenkins projects



Build Queue

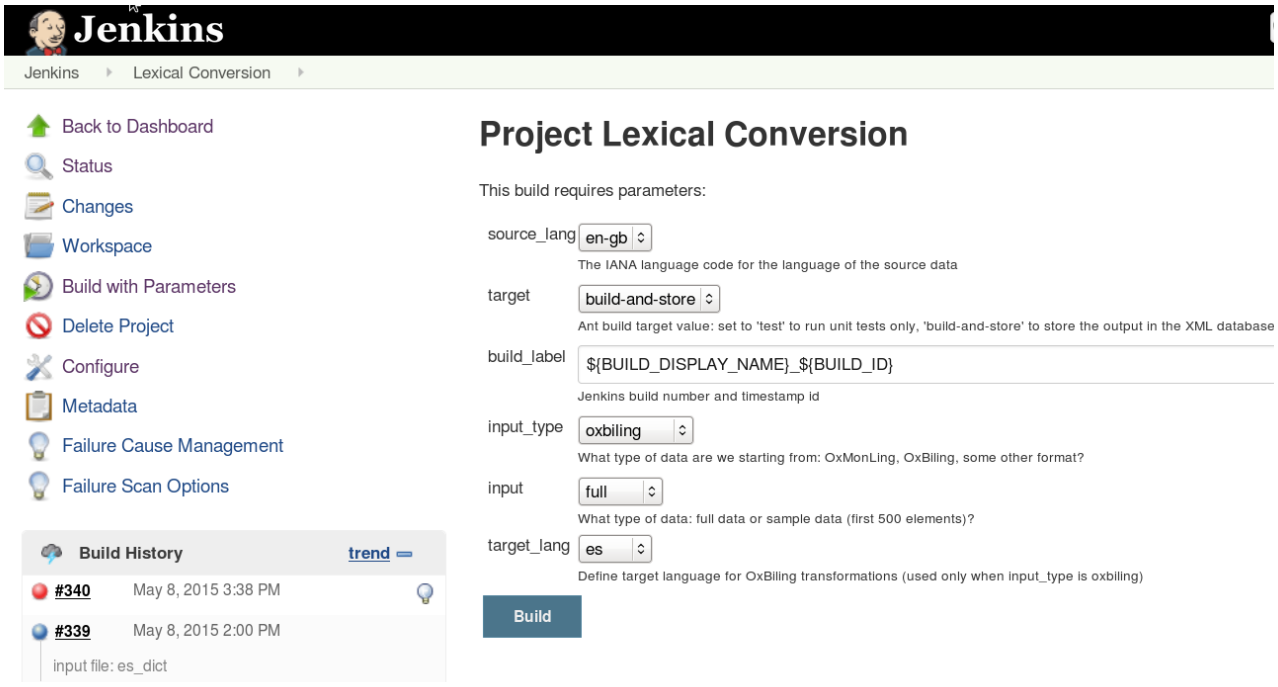
No builds in the queue.

Build Executor Status

1	Idle
2	Idle
3	Idle
4	Idle
5	Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
🔴		Data Conversion - British English	8 days 18 hr - #30	18 hr - #37	3 hr 22 min
🔴		Data Conversion - English-Spanish	8 days 18 hr - #28	18 hr - #35	4 hr 0 min
🔴		Data Conversion - Hindi	3 days 18 hr - #33	18 hr - #36	2 hr 31 min
🔴		Data Conversion - Polish	N/A	18 hr - #8	3 hr 5 min
🔴		Data Conversion - Slovenian	8 days 18 hr - #28	18 hr - #34	4 hr 0 min
🔴		Data Conversion - Spanish	2 days 18 hr - #32	18 hr - #35	4 hr 9 min
🔴		Data Conversion - Spanish-English	3 days 4 hr - #32	18 hr - #35	33 sec
🔵		GraphDB Full Text Indexing	3 hr 17 min - #12	3 hr 27 min - #11	2 min 10 sec
🔵	☀	HTTP Unit Testing	2 hr 6 min - #82	N/A	10 sec
🔴		Lexical Conversion	4 hr 56 min - #339	3 hr 18 min - #340	1 hr 33 min
🔴		Lexical Conversion Nightly Builds	8 days 18 hr - #43	18 hr - #49	4 hr 1 min
🟡	☀	Lexical Conversion Validation	4 hr 48 min - #50	N/A	2 min 47 sec
🔵	☀	Lexical To RDF Conversion	2 days 8 hr - #317	2 days 15 hr - #312	46 min
🔵	☀	Lexical To RDF Validation	2 days 8 hr - #59	N/A	23 sec

Figure 5. Parametrized build



Project Lexical Conversion

This build requires parameters:

source_lang: en-gb
The IANA language code for the language of the source data

target: build-and-store
Ant build target value: set to 'test' to run unit tests only, 'build-and-store' to store the output in the XML database

build_label: ${BUILD_DISPLAY_NAME}_${BUILD_ID}
Jenkins build number and timestamp id

input_type: oxbiling
What type of data are we starting from: OxMonLing, OxBiling, some other format?

input: full
What type of data: full data or sample data (first 500 elements)?

target_lang: es
Define target language for OxBiling transformations (used only when input_type is oxbiling)

#340	May 8, 2015 3:38 PM
#339	May 8, 2015 2:00 PM

input file: es_dict

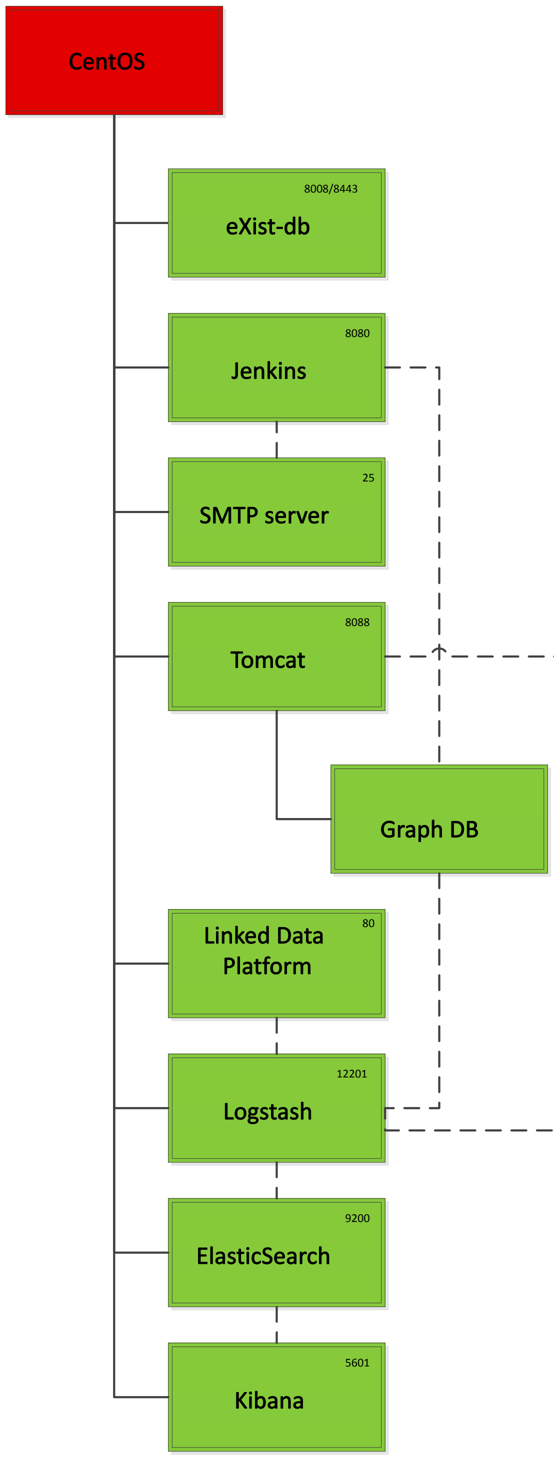
3. Deployment

One of the issues we faced when working with out-of-house freelancers is that their working environments needed to be replicated in-house in order to rerun scripts. Indeed, even within an in-house development team it is not uncommon to use different software tools and operating systems. In addition, the need of development, staging, and production environments for large projects usually causes integration problems when deploying from one environment to another.

In order to minimize integration issues and avoid the classic 'but it worked on my machine' problem, we picked up Docker as our deployment tool. Docker is an open source software for deploying distributed applications running inside containers [14]. It allows applications to be moved portably between development and production environments and provides development and operational teams with a shared, consistent platform for development, testing, and release.

As shown in Figure 6, “Docker Containers”, we based our environment on a CentOS image base. This container also deploys all the software tools employed by subsequent containers (e.g. Linux package utilities, Java, ant, maven, etc.). Separate ports are allocated to each component and re-deploying the components to a different port is simply a matter of re-mapping the Docker container to the new port. Graph DB is deployed as an application inside a Tomcat container. The Jenkins container is linked to the SMTP server container in order to send email notifications to the system administrator and to Graph DB for reindexing purposes. Most of the components send their logs to logstash which acts as a centralized logging system. Logs are then searched via ElasticSearch and visualized with Kibana. Software components like SVN and Jira are deployed on separate servers managed by other IT departments, therefore there was no need to deploy them via Docker containers.

Figure 6. Docker Containers



Example 4, "Dockerfile for deploying eXist-db" illustrates an example of Dockerfile for deploying eXist-db inside a Docker container. The example is largely based on an image pulled out from the Docker hub registry. The script exist-setup.cmd is used to set up a basic configuration (e.g. admin username and password).

Example 4. Dockerfile for deploying eXist-db

```
FROM centos7:latest
MAINTAINER Sandro Cirulli  <sandro.cirulli@oup.com>

# eXist-db version
ENV EXISTDB_VERSION 2.2

# install exist
WORKDIR /tmp
RUN curl -LO http://downloads.sourceforge.net/exist
/Stable/${EXISTDB_VERSION}/eXist-db-setup-${EXISTDB
_VERSION}RC2.jar
ADD exist-setup.cmd /tmp/exist-setup.cmd

# run command line configuration
RUN expect -f exist-setup.cmd
RUN rm eXist-db-setup-${EXISTDB_VERSION}RC2.jar exi
st-setup.cmd

# set persistent volume
VOLUME /data/existdb

# set working directory
WORKDIR /opt/exist

# change default port to 8008
RUN sed -i 's/default="8080"/default="8008"/g'
tools/jetty/etc/jetty.xml

EXPOSE 8008 8443

ENV EXISTDB_HOME /opt/exist

# run startup script
CMD bin/startup.sh
```

4. Future Work

The aim of the OGL programme is to convert into lean XML and RDF tens of language datasets and our project is a work-in-progress that changes rapidly. Although we are in the initial phase of the project, we believe we have started building the initial foundations of a scalable and reliable system based on continuous integration and automatic deployment. We have identified the following areas of further development in order to increase the robustness of the system:

- **Availability**: components in the system architecture may be down or inaccessible thus producing cascading effects on the conversion workflow. In order to minimize this issue, we introduced HTTP unit tests using the HttpUnit testing framework [15]. These tests are triggered by a Jenkins project and regularly poll the system components to ensure that they are up and running. A more robust approach would involve the implementation of the Circuit Breaker Design Pattern [16] which early detects system components failures, prevents the reoccurrence of the same failure, and reduces cascading effects on distributed systems.

- **Scalability**: we foresee to build large amounts of XML and RDF data as we progress with the conversion of other language datasets. As our system architecture matures, we also feel an urgent need to deploy development, staging, and production environments. Consequently, we plan to move part of our system architecture to the cloud in order to run compute-intensive processes such as nightly builds and to deploy different environments. Cloud computing is particularly appealing for our project thanks to auto-scaling features that allow to start and stop automatically instances of powerful machines. Another optimization in terms of scalability would be to increase the number of executors for parallel processing and to distribute builds across several slave machines.
- **Monitoring**: we introduced a build monitor view in Jenkins [17] that tracks the status of builds in real time. The monitor view also allows to display automatically the name of the developer who may have broken the last build, to identify common failure causes by catching the error message in the logs, and to assign or claim broken builds so that developers can fix them as soon as possible. We hope that this tool will act as a deterrent for unfixed broken builds and will increase the awareness of continuous integration in both our team and our department.
- **Code coverage and further testing**: we introduced code coverage metrics (i.e. the amount of source code that is tested by unit tests) for Python code related to the development of the Linked Data Platform and we would like to add code coverage for XSLT code. Unfortunately, there is a lack of code coverage frameworks in the XML community since we could only identify two code coverage tools (namely XSpec and Cakupan), one of which requires patching at the time of writing [18]. In addition, we plan to increment and diversify the types of testing (e.g. more unit tests, security tests, acceptance tests, etc.). Finally, in order to avoid unnecessary stress on Jenkins and SVN servers, we would like to replace the polling of SVN via Jenkins with SVN hooks so that an SVN commit will automatically trigger the tests execution.
- **Deployment orchestration**: the number of containers increased steadily since we started to deploy via Docker. Moreover, some containers are linked and need to be started following a specific sequence. We plan to orchestrate the deployment of Docker container and there are several tools for this task (e.g. Machine, Swarm, Compose/Fig).

5. Conclusion

In this paper we described how we set up a framework based on continuous integration and automated deployment for converting large amounts of XML and RDF data. We discussed the build workflows and the testing process and highlighted the benefits of continuous integration in terms of code quality and reuse, integration, and automation. We illustrated how the deployment of system components was automated using Docker containers. Finally, we discussed our most recent work to improve the framework and identified areas for further development related to availability, scalability, monitoring, and testing.

In conclusion, we believe that continuous integration and automatic deployment contributed to improve the quality of our XML and RDF data as well as our code and we plan to keep improving our workflows using these software engineering practices.

6. Acknowledgements

The work described in this paper was carried out by a team of developers at OUP. This team included Khalil Ahmed, Nick Cross, Matt Kohl, and myself. I gratefully acknowledge my colleagues for their precious and professional work on this project.

Bibliography

[1] *Oxford's Global Languages Initiative*. OUP. Accessed: 8 May 2015.
http://www.oxforddictionaries.com/words/oxfordlanguage

[2] Matt Kohl, Sandro Cirulli, and Phil Gooch. *From monolithic XML for print/web to lean XML for data: realising linked data for dictionaries*. In Conference Proceedings of XML London 2014. June 7-8, 2014.
doi:10.14337/XMLLondon14.Kohl01

[3] Martin Fowler. 2006. *Continuous Integration*. Accessed: 8 May 2015.
http://martinfowler.com/articles/continuousIntegration.html

[4] John Ferguson Smart. 2011. *Jenkins - The Definitive Guide*. O'Reilly Media, Inc.. Sebastopol, CA. ISBN 978-1-449-30535-2.

[5] Jenkins CI. *Jenkins*. Accessed: 8 May 2015. http://jenkins-ci.org

[6] ZeroTurnaround. *10 Kick-Ass Technologies Modern Developers Love*. Accessed: 8 May 2015. http://zeroturnaround.com/rebellabs/10-kick-ass-technologies-modern-developers-love/6

[7] Jenkins CI. *Build Flow Plugin*. Accessed: 8 May 2015. https://wiki.jenkins-ci.org/display/JENKINS/Build+Flow+Plugin

[8] Jeni Tennison. *XSpec - BDD Framework for XSLT*. Accessed: 8 May 2015. http://code.google.com/p/xspec

[9] Benoit Mercier. *Including XSLT stylesheets testing in continuous integration process*. In Proceedings of Balisage: The Markup Conference 2011. Balisage Series on Markup Technologies. vol. 7. August 2-5, 2011. doi:10.4242/BalisageVol7.Mercier01

[10] *Jxsl - Java XSL code library*. Accessed: 8 May 2015. https://code.google.com/p/jxsl/

[11] Jeni Tennison. *XSpec - Running with ant*. Accessed: 8 May 2015. https://code.google.com/p/xspec/wiki/RunningWithAnt

[12] Agile Knowledge Engineering and Semantic Web (AKSW). *RDFUnit*. Accessed: 8 May 2015. http://aksw.org/Projects/RDFUnit.html

[13] Jostein Austvik Jacobsen. *xprocspec - XProc testing tool*. Accessed: 8 May 2015. http://josteinaj.github.io/xprocspec/

[14] Docker. *Docker*. Accessed: 8 May 2015. https://www.docker.com/whatisdocker/

[15] Russell Gold. 2008. *HttpUnit*. Accessed: 8 May 2015. http://httpunit.sourceforge.net

[16] Michael T. Nygard. 2007. *Release it! Design and Deploy Production-Ready Software*. The Pragmatic Programmers, LLC. Dallas, Texas - Raleigh, North Carolina. ISBN 978-0978739218.

[17] Jenkins CI. *Build Monitor Plugin*. Accessed: 8 May 2015. https://wiki.jenkins-ci.org/display/JENKINS/Build+Monitor+Plugin

[18] Google Groups. *XSpec Coverage*. Accessed: 8 May 2015. https://groups.google.com/forum/#!topic/xspec-users/VRlCTR5KvIU

Vivliostyle - Web browser based CSS typesetting engine

How browser based typesetting systems can be made to work also for printed media

Shinyu Murakami (村上真雄)

Vivliostyle Inc.

`<murakami@vivliostyle.com>`

Johannes Wilm

Vivliostyle Inc.

`<johanneswilm@vivliostyle.com>`

Abstract

All currently available typesetting systems and formats are rather limited, and the integration between workflows related to print are quite different than those related to web publishing and ebooks.

In this article we argue that the best way forward to unite the workflows is to focus on an HTML-centric workflow, using CSS for styling, and leveraging the power of browsers through the usage of Javascript for print-based layouts.

The Vivliostyle project is working on a new typesetting engine for the next phase of the digital publishing era in which web, ebook and print publishing are unified. We seek to demonstrate here that such a project is needed to bring the three publishing workflows together.

Keywords: HTML, CSS, Page based media

1. Introduction

Publishing of long format text in 2015 usually takes three different forms: print as a book, a version to be used on the internet and possibly an ebook.

Ebooks are in most cases EPUB files. The textual content of EPUBs is provided by files containing a restricted version of Hyper Text Markup Language (HTML), the same format used for web pages. The styling of both web pages and EPUBs is defined through Cascading Style Sheets (CSS). Converting content between EPUBs and web pages is therefore not that difficult.

In contrast, most print typesetting systems are using quite different formats and standards than those for ebooks and the web. The workflows from document creation, through editing to final publication differ considerably with different tools and different file formats used. Publishing the same document for print, web and ebooks is therefore difficult, especially for documents that require updating after initial publication as a change in oen fo the files needs to be propagated to all other versions.

The simplest way to unify the publication processes is to introduce HTML and CSS to the print publishing process. Other projects that provide print processing functionality using HTML/CSS already exist. Among these are PrinceXML[1], the Antenna House Formatter[2], PDFreactor[3] or Pagination.js[4] and SimplePagination.js[5].

However, none of these solutions have been able to establish themselves as the industry standard. In the following, we will argue that all the existing solutions have fundamental shortcomings and that the Vivliostyle project is needed to effectuate a change to web technologies in the print publishing industry.

[1] http://www.princexml.com
[2] http://www.antennahouse.com
[3] http://www.pdfreactor.com
[4] https://github.com/fiduswriter/pagination.js (requires CSS Regions, previously known as BookJS)
[5] https://github.com/fiduswriter/simplePagination.js (does not require CSS Regions, but has less features than Pagination.js)

2. CSS Paged Media and the limitations of current implementations.

To style elements of pages (electronic or physical) there is a CSS module called "CSS Paged Media" [1]. It is one of several CSS modules defining styling elements needed to make exact specifications in CSS for printed and paged output. There are already several typesetting engines supporting CSS Paged Media: The Antenna House Formatter supports CSS as an alternative to XSL-FO, and also PrinceXML supports it.

However, these proprietary and paid-for solutions were never able to establish CSS Paged Media as a standard neither for the web nor for their industry. Web browsers have not implemented much of it, even though they have provided features to print web pages and convert to the Portable Document Format (PDF), the file format most commonly used to ensure consistency in print outputs. Even ebook systems, which show individual pages on the screen, have not been very concerned with implementing CSS Paged Media.

What is more, the formatters that do support CSS Paged Media each have their own proprietary vendor extensions that are not compatible with web browsers or even each other.

Even though CSS Paged Media formatters are getting acknowledgment in the XML publishing world, they are therefore still far away from becoming mainstream tools that are widely used.

3. Enhancing web browser's page layout with JavaScript

An approach to try to bring page layout to web browsers are Pagination.js and simplePagination.js, which use Javascript in combination with CSS to draw pages. They provide some of the features used for book printing such as table of contents, running headers, page floats, footnotes, word indexes, and margin notes.

However, they are limited to features of books, they do not interpret CSS but take configuration options only through Javascript function arguments, and they use tricks to achieve their results in current browsers. The usage of tricks means that they only currently work but that this may not work for all future. For this reason they may be usable for certain cases of print, but will not be able to replace broader printing solutions.

4. Standardizing and implementing next generation CSS standards

The Vivliostyle projects seeks to combine and enhance both approaches: Use CSS standards and Javascript for browser based implementations.

Vivliostyle seeks to work with the World Wide Web Consortium (W3C) to enhance and promote specifications such as CSS Paged Media and other related specifications such as "CSS Page Floats" [2], working with web browsers to work towards implementation of these specifications in browsers.

Until such support is fully implemented in browsers, Vivliostyle develops Vivliostyle.js, a polyfill which will use Javascript to layout pages inside browsers, similar to simplePagination.js and Pagination.js, but it will do so by reading and interpreting the CSS that accompanies the source files and it will provide for a broader usage field, so that styling options can be defined through CSS and will work for a broader usage field than just books.

Additionally, the Vivliostyle Formatter, a Command-Line Interface (CLI) application, and the Vivliostyle Browser, a Graphic User Interface (GUI) application, will embed Vivliostyle.js to allow for PDF output of HTML/XHTML and CSS source files to fit professional publishing needs.

Bibliography

[1] Melinda Grant, Elika Etemad, Håkon Wium Lie, and Simon Sapin. *CSS Paged Media Module Level 3*. W3C Working Draft. 14 March 2013. World Wide Web Consortium (W3C). http://www.w3.org/TR/css3-page/

[2] Johannes Wilm. *CSS Page Floats*. Editor's Draft. 7 April 2015. World Wide Web Consortium (W3C). http://dev.w3.org/csswg/css-page-floats/

Magic URLs in an XML Universe

George Bina

Syncro Soft / oXygen XML Editor

`<george@oxygenxml.com>`

Abstract

XML is an ideal format for structured content but we need to accept that are also other formats that can encode information in a structured way. We can try to get everyone to use XML but that may not be always possible for various reasons. Now it is possible to dynamically unify different formats by converting all content to XML simply by pointing to that content though a "magic" URL that performs this conversion on the fly.

We experimented with creating URLs to convert from various formats to XML, including Java classes and JavaDoc files, Excel and Google Sheet spreadsheets, Markdown and HTML, CSV files, etc. A major advantage of this approach is that it works immediately with any URL-aware application and allows to extend single source publishing across formats in a transparent way.

Keywords: XML, dynamic conversion, URL, DITA, Markdown, Excel, CSV, SVG, HTML, Java, Javadoc

1. The problem

If you need to write a tutorial for an SDK in DITA, then there is already some information available inside the source code - let's consider that to be Java in this example - so we can have comments inside the Java source code describing different methods or fields from each Java class, return values, possible exceptions and so on. It is not easy to reuse this information outside the basic copy and paste type of reuse. One possibility will be to generate JavaDoc documentation and link to that from DITA, but then again, any reference inside the DITA code, like the syntax for a method needs to be recoded and duplicated in DITA.

This problem appears also if you receive a spreadsheet from accounting and you need to use that information in a solution proposal and in many other situations.

It is not possible to have developers write in DITA, the Javadoc documentation is needed to work within IDEs, when a user tries to insert a method they will be assisted by the IDE showing the description of that method. So, it is not possible in the real word to have a single common language for everything, we need to accept that there are multiple formats for information available and we need to look into possibilities to make them work easily together.

2. The idea

If we look more closely, it is not the actual format, but the degree of structure that matters, if we want to be able to process information. And Java source code with its comments is a very structured language so it can be reliably converted automatically to a different format.

In general, any reference to a resource is done though a URL. For example, an HTTP URL has the following format:

```
http://user:password@www.example.com/path/to/
file.ext?param1=val1&param2=val2
```

where we can identify

http
    The URL scheme / access protocol
user:password
    Credentials
www.example.com
    Server location
path/to/file.ext
    Resource path
param1=val1&param2=val2
    Parameters with some specified values

Even in the case of an HTTP URL, the resource part may not represent what it is actually returned when the URL is read, the resource may be a script that does some processing, depending on the suplied parameters. Other example URLs:

- `file:/path/to/file.dita`
- `http://server/cgi?file=file.dita`
- `https://server/path/to/file.dita`
- `ftp://server/path/to/file.dita`
- `zip:URL!/path/to/file.dita`

In general, the URL has a URL scheme or protocol and the rest of the URL is protocol dependent. As we can see in the `zip` URL example, the rest of the URL may contain also another URL and in this case it is clear that the access to a resource does not imply only reading the content of that resource and returning it, in this case the resource is extracted from the ZIP archive, so a conversion process takes place from the ZIP encoded form.

Because all references to resources are done though URLs and URLs can encode information and decode that when reading, we can think for example that a Java class encodes a DITA topic and when we read that though a `java2dita` URL, we get back the DITA topic.

3. URLs in Java

Java has a pluggable system for URL support, one can register a URL handler for a URL scheme, that is a Java class that implements a specific interface, and that Java class handles the parsing of the URL syntax for that URL scheme/protocol as well as how content should be read from or written to that resource.

Thus, if we register a URL handler class that can convert from Java to DITA for the `java2dita` URL scheme, then we can use URLs starting with `java2dita:/` to point to a Java class and when we read from that URL, the control will be given to our registered URL handler that will provide a stream from where DITA content will be read, representing the dynamic conversion of the Java source code to a DITA topic.

The same can be imagined for any type of conversion from one format to another - as long as there is enough structural information to have this conversion possible in an automatic way.

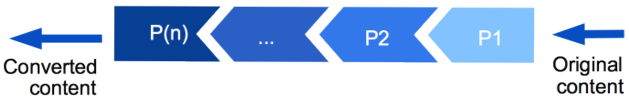
But, instead of registering a protocol scheme for each conversion we can also register a single URL scheme, `convert`, and use the syntax of the convert URLs to control the actual conversion process.

4. The `convert` URLs

We define the convert URLs to have the following syntax:

```
convert:/pipelineStepN/.../pipelineStep1!/
targetContentURL
```

Figure 1. Simple conversion pipeline



where we can identify

convert
: The URL scheme / protocol

pipelineStepN
: Information for the conversion step N, applied on the output of conversion step N-1

…
: and so on …

pipelineStep1
: Information for the first conversion step, applied on the target URL

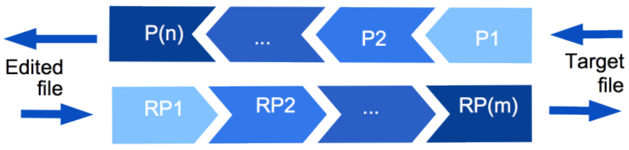
targetContentURL
: A URL pointing to a resource whose content will be passed though the conversion pipeline formed by the conversion steps

This allows us to apply different conversion steps on a resource, also identified by a URL, the target URL, and the result when the convert URL is read will be the dynamic conversion of the resource content though the specified conversion steps.

We can generalize this to support also writing, that happens when we change the content of the URL and we want to save. For that we can specify also a reverse pipeline, with similar conversion steps as the pipeline that converts from the target URL but this time the conversions are performed in the other direction, from the URL content to the target URL. The content to be saved will be the input of the first reverse step, and so on, and the output of the last reverse step will be the new content of the target URL.

```
convert:/reverseStep1/.../reverseStepM/pipeline
StepN/.../pipelineStep1!/targetContentURL
```

Figure 2. Complete conversion pipeline



Using such a URL we can transparently round-trip from one format to another, from the target URL format to our URL format and back.

Each pipeline step starts with `processor=processingStep` if it is a step representing part of the direct conversion pipeline and with `rprocessor=processingStep` if it is a conversion step part of the reverse conversion pipeline. This is followed by parameters specific to that processing step, specified as `paramName=paramValue` and separated by `;` as separator character.

To proof this concept, we implemented a Java URL handler for the convert protocol and as basic processing steps we implemented:

- XSLT
- XQuery
- Java
- JavaScript
- Excel to XML
- JSON to XML
- HTML to XHTML
- Wrap text into an XML element

4.1. XSLT conversion step

The XSLT conversion step allows to apply an XSLT stylesheet on the input and send the result of the XSLT processing to the output. The processor name is `xslt` and as parameters we have:

ss
: This identifies the XSLT stylesheet to be applied on the input

[any other parameter name]
: Optional parameters can be specified other than `ss`, they will be transmitted as XSLT parameters to the XSLT stylesheet

Example 1. Sample XSLT step

```
processor=xslt;ss=urn:processors:convert.xsl;p1=v1
```

This specifies that the XSLT stylesheet identified by `urn:processors:convert.xsl` should be applied on the input received by this processing step and the result of the transformation should go to the output, either as the result of the convert URL, in case this is the last step of a direct conversion pipeline or as the result of the target URL, if this is the last step of a reverse conversion pipeline, or to the input of the next step in the pipeline. A parameter with the name `p1` will be set for the transformation and it will have the value `v1`.

4.2. XQuery conversion step

The XQuery conversion step is similar to the XSLT one, but it allows to apply an XQuery script on the input instead of an XLST script. The processing name is `xquery` and as parameters we have:

ss
: This identifies the XQuery script to be applied on the input

[any other parameter name]
: Optional parameters can be specified other than ss, they will be transmitted as parameters to the XQuery script

Example 2. Sample XQuery step

```
processor=xquery;ss=urn:processors:convert.
xquery;p1=v1
```

This specifies that the XQuery script identified by `urn:processors:convert.xquery` should be applied on the input received by this processing step and the result of the XQuery execution should go to the output. A parameter with the name `p1` will be set for the transformation to the value `v1`.

4.3. Java conversion step

The Java conversion step allows to apply processing specified in a Java class on the input and the result of the Java processing will be sent to the output. The processing name is `java` and as parameters we have

jars
: A comma separated list of libraries that will be added to the class path

ccn
: Identifies the conversion class name, as a fully qualified Java class name

[any other parameter name]
: Optional parameters can be specified other than `jars` and `ccn`, they will be transmitted as properties for the Java conversion

The conversion class needs to have a constructor without parameters and a method with the following signature:

```java
public void convert(
  String systemID,
  String originalSourceSystemID,
  InputStream is,
  OutputStream os,
  LinkedHashMap<String, String> properties)
throws IOException
```

Example 3. Sample Java step

```
processor=java;jars=urn:processors:jars;
ccn=j.to.xml.JavaToXML
```

This specifies that the Java class `j.to.xml.JavaToXML` will be applied for the conversion process, and that the folder identified by `urn:processors:jars` will be added to the class path.

4.4. JavaScript conversion step

The JavaScript conversion step allows to apply processing specified in a JavaScript method on the input and the result of the processing will be sent to the output. The processing name is `js` and as parameters we have

js
: Points to the JavaScript file

fn
: Identifies the method name that should be invoked for conversion, method that must take a string as an argument and return a string

Example 4. Sample JavaScript step

```
processor=js;js=urn:processors:md.js;
fn=convertExternal
```

This specifies that the JavaScript method `convertExternal` from the `urn:processors:md.js` script file will be applied for the conversion process.

4.5. Excel to XML conversion step

The Excel conversion step converts an excel sheet to an XML form that can then be easily processed further with XSLT towards your desired final format. The processing name is `excel` and as parameter we have

sn
: This identifies the sheet name

Example 5. Sample XSLT step

```
processor=excel;sn=test
```

This specifies that the sheet with the name test should be converted to XML.

4.6. JSON to XML conversion step

The JSON conversion step converts JSON to XML. The processing name is `json` and it does not have any parameters.

Example 6. Sample JSON step

```
processor=json
```

4.7. HTML to XHTML conversion step

The HTML conversion step converts HTML to XHTML. The processing name is `xhtml` and it does not have any parameters.This is a very useful conversion step as it allows us to process HTML content further with XSLT and XQuery.

Example 7. Sample HTML conversion to XHTML step

```
processor=xhtml
```

4.8. Wrap text conversion step

The wrap conversion steps helps putting an XML tag around text content so it can be further processed with XML-aware processing steps, like XSLT and XQuery. The processing name is `wrap` and as parameter we have

rn
: Optional, specifies the root name, by default that will be `wrapper`.

This is useful if we want to take advantage of the RexExp support in XSLT 2.0 for example in a following conversion step.

Example 8. Sample wrap step

`processor=wrap`This specifies that the input should be placed within an XML element called `wrapper` (the default element name).

5. URL aliases

Using an XML catalog we can easily implement simple aliases for a more complex convert protocol syntax. For example with a mapping like in the following code fragment we can simplify the URL syntax to the point that authors can just type it in:

```
<rewriteURI uriStartString="excel2dita:/"
rewritePrefix="convert:/processor=xslt;ss=urn:processors:e2d.xsl/processor=excel;sn=sample!/"/>
```

This allows us to write a convert URL that bundles together multiple processing steps, and maybe also reverse processing steps, in case of a URL supporting round-tripping, as a single new protocol followed by a pointer to the target resource:

Example 9. Short URL for Excel to DITA conversion

```
excel2dita:/urn:files:sample.xls
```

The XML Catalog can be also used to define URN names pointing to different locations where we have files used in these URLs either as target files or files used in the conversion processes:

```
<rewriteURI uriStartString="urn:files:"
            rewritePrefix="resources/"/>
<rewriteURI uriStartString="urn:processors:"
            rewritePrefix="processors/"/>
```

6. Sample conversion pipelines

Here we have a few sample conversion pipelines that can be assembled as convert URLs and used though an alias. These cover some common use cases.

6.1. Excel to DITA

We apply two processing steps, Excel to XML using the `excel` processing step and XML to DITA topic using the `xslt` processing step:

```
convert:/proc=xslt;ss=excel2d.xsl/proc=excel;
sn=sample!/urn:files/sample.xls
```

By defining `excel2dita` as alias for

```
convert:/proc=xslt;ss=excel2d.xsl/proc=excel;
sn=sample!
```

we will be able to write the same URL as:

`excel2dita:/urn:files/sample.xls`

This allows us to reuse as DITA tables the spreadsheet tables and we can take advantage of the computations automatically done in Excel. Also, we have true single sourcing, if the spreadsheet file changes we automatically see the changes in our DITA content.

6.2. Google Sheets to DITA

Google sheets allows to access a spreadsheet content as HTML, so we can apply an HTML to XHTML conversion using the `xhtml` processing step and then use two XSLT stylesheets though `xslt` steps to filter the information that we want to convert to DITA and to transform that to a DITA topic.

By defining as alias for

```
convert:/processor=xslt;ss=urn:processors:h2d.x
sl/processor=xslt;ss=urn:processors:googleSheet
s2dita.xsl/processor=xhtml!/
```

the `gs2dita:/` prefix, we can easily express URLs that bring Google sheets as DITA just by specifying the target URL after this prefix

```
gs2dita:/https://docs.google.com/spreadsheets/
d/[...docid...]/edit?usp=sharing
```

This provides a very simple way to enable people to collaborate on creating the table information, using the Google Drive shared editing functionality and being able to include the up-to-date information automatically in a DITA processing workflow.

6.3. HTML to DITA

This is a more generic conversion, as we do not make any assumptions on the HTML input. We accomplish this with two processing steps, the first will convert HTML to XHTML using the `xhtml` processor, and then we use the `xslt` processor to apply an XHTML to DITA conversion.

An example URL looks like this

```
convert:/proc=xslt;ss=h2d.xsl/proc=xhtml!/urn:f
iles/care.html
```

But it can be reduced to

`html2dita:/urn:files/care.html`

by defining an alias in the XML Catalog.

6.4. Markdown to DITA

Markdown source files can be already available from developers and it may be useful to be able to include them in your DITA workflow. We can setup a pipeline that converts Markdown to HTML using JavaScript, HTML to XHTML and then XHTML to DITA using XSLT. A full convert URL with these pipeline steps like this

```
convert:/processor=xslt;ss=urn:processors:h2d.x
sl/processor=xhtml/processor=js;js=urn:processo
rs:pagedown%2FMarkdown.Converter.js;fn=convertE
xternal!/urn:files/sample.md
```

can be shortened to `md2dita:/urn:files/sample.md` using an XML Catalog to define an alias for the more complex convert URL.

6.5. XML Schema to DITA

XML Schema to DITA topic can be easily converted though XSLT, so we can package that as a convert URL like

```
convert:/processor=xslt;ss=urn:proc:xsdToTopic.
xsl!/urn:files/personal.xsd
```

or in the short form it can be `xsd2dita:/urn:files/personal.xsd`. This allows to easily integrate basic XML Schema documentation into a DITA-based project.

6.6. Java to DITA

Back to our initial problem, API documentation, we can get Java source code to DITA using a processing pipeline that converts the Java source to an XML format using some processing implemented in a Java class, then we convert to DITA using XSLT.

```
convert:/processor=xslt;ss=urn:processors:javaT
oTopic.xsl/processor=java;jars=urn:processors:j
ars;ccn=j.to.xml.JavaToXML!/urn:files:WSAuthorE
ditorPage.java
```

with the equivalent short form

`java2dita:/urn:files:WSAuthorEditorPage.java`

6.7. Javadoc to DITA

JavaDoc does more processing and records additional information about Java components, for example it can contain information about where a component is used so it may be interesting to have also this information available from DITA. Because JavaDoc is HTML, we can setup a conversion pipeline that normalizes the HTML to XHTML, then converts XHTML to DITA:

```
convert:/proc=xslt;ss=urn:proc:jdToTopic.xsl/
proc=xhtml/!/urn:files:ButtonEditor.html
```

or the short form
`javadoc2dita:/urn:files:ButtonEditor.html`
Publish DITA conversion of Javadoc to PDF

6.8. Custom XML to SVG

If we have a set of values in an XML file, we can easily get them as a graphic, by generating an SVG as a convert URL

```
convert:/processor=xslt;ss=urn:processors:sales
.xsl!/urn:files:sales.xml
```

Using this method we can publish graphs which dynamically change depending on the values found in the original source.

6.9. DITA Map to Schematron

Another interesting example is related to an intelligent style guide, that encodes business rules within the style guide prose. By default a transformation converts that to Schematron, but instead of that we can just point to the style guide through a magic URL that will bring a Schematron schema.

6.10. Round-tripping CSV to DITA and back

For CSV we implement round-tripping by providing both a direct conversion pipeline, from CSV to DITA and a reverse conversion pipeline, from a DITA topic to a CSV file. We then add a mapping in the catalog to be able to refer to these convert URLs though a simple `csv2dita` protocol:

```
<rewriteURI uriStartString="csv2dita:/"
  rewritePrefix="convert:/rprocessor=xslt;ss=urn
:processors:dita2csv.xsl/processor=xslt;ss=urn:p
rocessors:csvtext2dita.xsl/processor=wrap!/"/>
```

When a CSV file is read through a `csv2dita` URL, the processing applied is a wrapping of the CSV in and XML element done with the `wrap` processor and then that is passed though the `csvtext2dita.xsl` stylesheet which will return a DITA topic containing a table with all the CSV data. When the resource is saved, the reverse conversion pipeline is executed, the content of the DITA document being sent to the `dita2csv.xsl` stylesheet and the output of this transformation will be stored in the CSV file.

The round-tripping support can be very useful also to edit a document as a slightly different version in terms of markup, allowing basically to perform a transformation before editing and a transformation before saving. For example, if a document makes havy use of attributes, these can be transformed to elements when the document is read and then, after it is changed, when the document is about to be saved, the elements representing atribute information will be transformed back to attributes.

7. Conclusions

The idea of using URLs to dynamically convert content back and forth between different formats is very simple and in the same time very powerful. It seamlessly brings together different formats in a single publishing framework, and allows any processing flow based on URLs to automatically handle the references to resources that otherwise were incompatible to the supported format.

This is a generic approach that allows to make any type of automatic conversion dynamic, allowing us to implement true single sourcing across multiple formats.

The DITA specific examples mentioned here are made available also in a project published on GitHub at
`https://github.com/oxygenxml/dita-glass`
so you can experiment with these type of conversions or use them as a starting point if you want to try your own ideas.

A rendering language for RDF

Fabio Labella

The University of Edinburgh

NCR Corporation

Henry S. Thompson

The University of Edinburgh

Abstract

Since RDF is primarily intended to be processed by machines, there is a need for a technology to render it into human readable (HTML) content, similarly to what XSLT does for XML. This is made hard, however, by the high syntactic variability of RDF serialisations, that allows the same graph to be expressed in many different ways.

In this paper we propose an approach, called just-in-time reflection, that allows the application of normal XSLT stylesheets directly to RDF graphs. It follows that the transformation does not depend on any particular serialisation format: stylesheets can be written in terms of RDF's abstract syntax.

1. Introduction

With the increasing adoption of RDF comes the need for transforming RDF graphs into a human readable format like HTML. While XML has a core technology, XSLT, devoted to it, a similar tool is sorely missing for RDF. Let's take for example the simple graph represented in Figure 1. We might like to be able to transform it into an HTML output like the one in Figure 2, with the constraints that it must be possible to express the transformation declaratively, and that the result should not depend on the particular serialisation syntax which was used to create the RDF graph at hand.

Figure 1. A simple RDF graph representation. The predicate names have been left blank for simplicity

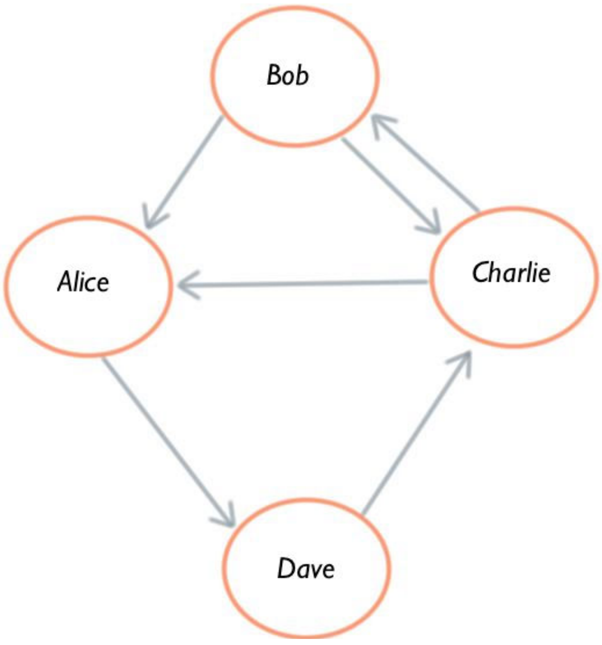
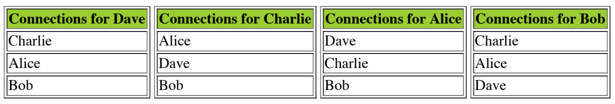


Figure 2. A possible HTML output.



Connections for Dave	Connections for Charlie	Connections for Alice	Connections for Bob
Charlie	Alice	Dave	Charlie
Alice	Dave	Charlie	Alice
Bob	Bob	Bob	Dave

We call this transformation process *rendering*, and offer the following contributions:

- We propose an approach, called *just-in-time reflection*, that provides access to RDF data, originating from **any** format, via XPath expressions (See Section 3, "Just-in-time reflection").
- We design a normal form that allows the stylesheet to be written against RDF's abstract syntax, without having to know the details of any particular concrete syntax (See Section 4, "Normal form design").

- We present an enhanced XSLT processor that uses *just-in-time reflection* to give standard XSLT stylesheets access to RDF graphs via the XPath data model. This processor implements all the XPath axes and can be used to transform any RDF graph (See Section 5, "An XSLT engine for RDF Graphs").

2. Rendering RDF

Figure 3. Different representations of the same RDF graph [AKKP08]

```
@prefix alice: <alice/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

alice:me a foaf:Person.
alice:me foaf:knows _:c.
_:c a foaf:Person.
_:c foaf:name "Charles".
```

(a)

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <foaf:Person rdf:about="alice/me">
      <foaf:knows>
         <foaf:Person foaf:name="Charles"/>
      </foaf:knows>
   </foaf:Person>
</rdf:RDF>
```

(b)

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:nodeID="x">
      <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
      <foaf:name>Charles</foaf:name>
   </rdf:Description>
   <rdf:Description rdf:about="alice/me">
      <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
      <foaf:knows rdf:nodeID="x"/>
   </rdf:Description>
</rdf:RDF>
```

(c)

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:about="alice/me">
      <foaf:knows rdf:nodeID="x"/>
   </rdf:Description>
   <rdf:Description rdf:about="alice/me">
      <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   </rdf:Description>
   <rdf:Description rdf:nodeID="x">
      <foaf:name>Charles</foaf:name>
   </rdf:Description>
   <rdf:Description rdf:nodeID="x">
      <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   </rdf:Description>
</rdf:RDF>
```

(d)

The RDF specification [CWL14] defines an abstract syntax where RDF datasets, composed of triples of the form *subject-predicate-object*, have a natural representation as a directed graph.

However, the actual form RDF documents take is entirely dependant on the serialisation format used: as an example, Figure 3, "Different representations of the same RDF graph [AKKP08]" shows the same graph in Turtle (a), and different flavours of RDF/XML. Path-based query languages can be easily extended to fit RDF's graph-based data model, making XSLT an attractive candidate to solve the RDF rendering problem.

In contrast, attempting to render generic RDF via XSLT stylesheets targetting RDF/XML directly has serious drawbacks:

- it would not apply to non-XML formats directly, whereas the general trend seems to move away from RDF/XML in favour of simpler formats, especially Turtle.
- RDF/XML itself is highly flexible (Figure 3, "Different representations of the same RDF graph [AKKP08]"), allowing the same graph to be serialised in several different ways.
- That very flexibility means there is no standard approach to re-serialising graphs independently of their original format (if any).

An apparently more serious problem lies in the mismatch between XPath's tree-based data model and RDF's graph-based data model, which means that is not possible, in general, to serialise an RDF graph trivially into XML in a way that is compatible with path-based traversal.

Alternative XML formats such as TriX [CS04] have been designed specifically to be compatible with XSLT, but, in order to ensure predictability, they move away from a graph-based encoding of RDF, opting instead to represent it as a collection of statements, which seems more suitable for a pattern matching query language.

For these reasons, currently the only available solution to the rendering problem consists in processing RDF using native engines, since they are oblivious of the input format and work directly in terms of the abstract syntax. This obviously means that for every transformation custom code must be written to select the appropriate RDF data and generate the desired HTML output. Moreover, the semantics of these engines are usually modelled after SPARQL, and are based on pattern matching instead of path traversal.

3. Just-in-time reflection

The starting point for this work was a paper by Thompson et al. [TKC03], which proposed an approach, called *just-in-time reflection*, to access XML Infosets and their extensions from XPath.

The main contribution of our paper is that the same approach can be applied successfully to RDF: this section will hence explain what is meant by *just-in-time reflection*, and how it can turn XSLT into a syntax-unaware RDF processor.

3.1. Static reflection

Thompson et al. [TKC03] defined reflection as *" a process whereby a syntactic form in some language, in our case XML, is analysed and represented in some underlying data model, in our case the XML Infoset, and then the constituents of that representation are themselves expressed using the same syntactic form, in our case XML once again."*. Adapting this definition to RDF, reflecting means that the underlying graph is derived from an RDF/XML serialisation, and then serialised once again in XML.

Listings Figure 4, “Three different serialisations of the same RDF statement” and Figure 6 and Figure 5 show the process of reflecting the statement "Alice is a person", bearing in mind however, that in this paper the concept of reflection is used more loosely, for the resulting XML is not necessarily the reflection of an RDF/XML input, since the graph it processes can be derived from any RDF format.

Figure 4. Three different serialisations of the same RDF statement

TURTLE:

```
alice:me a foaf:Person .
```

RDF/XML - ABBREV:

```
<foaf:Person rdf:about="alice/me"/>
```

RDF/XML:

```
<rdf:Description rdf:about="alice/me">
 <rdf:type
  rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
</rdf:Description>
```

Figure 5. The underlying graph

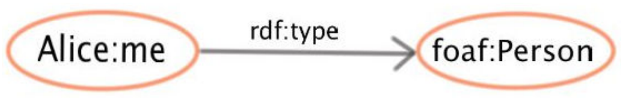


Figure 6. A possible reflection

```
<alice:me>
  <rdf:type>
    <foaf:Person>
      .....
```

In our approach the reflection is generated from an RDF data model exploiting a generic RDF engine, and then fed to XSLT using *extension functions*, a native mechanism to call external functions, written in another programming language, from any stylesheet. As a result, path expressions are evaluated against the reflection instead of the original file, meaning that:

- XSLT can work with different input formats, not necessarily XML-based.
- Syntactic variability, the single greatest obstacle to RDF processing with XSLT, is eliminated, provided that the reflection is expressed in a *normal form*.

3.2. Generating reflections dynamically

As its name would suggest, a *just-in-time reflection* differs from a mere normalisation in that it is not the result of preprocessing the input data, but is instead generated dynamically when evaluating a path expression.

More specifically, the system evaluates a path expression one step at the time, interprets it according to the normal form used, and returns a nodeset populated by querying the RDF engine to retrieve the appropriate resources. Therefore, even though the XSLT processor behaves as if it were operating on a physical XML file encoding the reflection of an RDF graph, with this approach no actual serialisation is needed, which is a significant advantage over a simple normalisation.

In fact, the complexity of RDF/XML stems from the mismatch between the **graph** structure of RDF and the **tree** structure of XML, for a graph in the general case is not constrained to be acyclic, rooted or connected. Since the edges that violate the aforementioned constraints cannot be encoded directly into the structure of XML [Wal03], different approaches to represent them lead to high syntactic flexibility, which makes RDF/XML very unfit for XSLT processing[1].

On the other hand, when serialisation is not required, it becomes much easier to devise a normal form to model closely RDF's abstract syntax, rendering path expressions straightforward to write. Let's consider for instance the cyclic graph described by the statement "Alice knows Bob and Bob knows Alice", which can be encoded as in Listing Figure 7, “Simple representation of a cyclic graph”, that is:

[1] Technically, a graph could be consistently encoded in standard XML by using only IDREF attributes to represent edges (see Layman Normal Form [Tho01]), but this would make path expressions hard to write.

Figure 7. Simple representation of a cyclic graph

```
<alice:me>
  <foaf:knows>
    <bob:me>
      <foaf:knows>
        <alice:me>
          <foaf:knows>
            <bob:me>
              <foaf:knows>
                ...
              </foaf:knows>
            </bob:me>
          </foaf:knows>
        </alice:me>
      </foaf:knows>
    </bob:me>
  </foaf:knows>
</alice:me>
```

This normal form is very easy to query, but results in an infinitely deep tree and is hence impossible to serialise: using *just-in-time reflection*, however, one can write a path expression for this normal form, leaving to the system the synthesis of the appropriate result nodeset (whose elements are retrieved using any generic RDF engine's basic navigation capabilities), **as if** such a serialisation could exist and be manipulated by XSLT.

Problems only arise when querying along the descendant axis, since a recursive depth-first search of the tree would be performed, leading to non-termination (because the system tries to reflect the same structure infinitely often). To solve this problem, the reflection generator must then be equipped with a cycle detection algorithm, so that when the same node is visited twice with an unchanged path expression, the search stops exploring that branch, avoiding infinite recursion.

4. Normal form design

The *just-in-time reflection* approach effectively overcomes the limitations of XSLT with respect to the variability in the concrete syntax of RDF, but it is not sufficient to ensure that is possible to work directly with the graph-based data model. In fact, the normal form used for the reflection is crucial for that purpose, for it must strive to model closely RDF's abstract syntax, so that one can easily write a path expression given only knowledge of the structure of the graph.

Before discussing the design of the normal form we used, it's necessary to clarify what we exactly mean by "normal form". Thompson [Tho01] distinguished two uses of the term: one concrete, to refer to a representation of a dataset in XML, and one abstract, to refer to "*a set of principles for constructing and/or interpreting concrete normal forms*".

We use "normal form" here in the latter sense, since it represents a set of rules by which the system interprets path expressions in order to construct a result nodeset, rather than a concrete XML encoding, as discussed above.

The original work on reflection [TKC03] proposed the so-called Edinburgh Normal Form as a suitable representation for reflected Post Schema-Validation Infosets (PSVIs). Due to its blend of simplicity and conciseness, it was chosen as the starting point for the one used here, although some changes were necessary to adapt it to RDF.

This section lays out the rules that define what we are calling, showing a remarkable lack in imagination, Edinburgh Normal Form for RDF (ENFR).

4.1. Resources

RDF resources are encoded using XML elements, named after the resource's QName, or its full URI if no QName is available. It was decided not to use `rdf:resource`, `rdf:Description` etc., since they are not directly concerned with the RDF model, but appear to be details of RDF/XML that make the resulting syntax less human-friendly.

4.2. Predicates

To reduce verbosity, the representation for predicates is different depending on whether the object of a statement is a resource or a literal.

4.2.1. Resource-valued properties

Borrowing from Alternating Normal Form [Tho01], triples whose object is a resource are represented by nesting alternately elements representing nodes and elements representing edges, as shown in Listing Figure 7, "Simple representation of a cyclic graph". Path expressions would then take the form:

```
subject/predicate/object/predicate/object/...
```

so that, given the Turtle statements:

```
Alice:me foaf:knows Bob:me .
Bob:me foaf:knows Charlie:me .
```

Charlie can be accessed, assuming that Alice is the root of the reflection, using the expression:

```
Alice:me/foaf:knows/Bob:me/foaf:knows/Charlie:me.
```

Several strategies are possible for representing two properties with the same name and different objects: the one chosen here is to have the two properties as two different children of the subject element, so as to guarantee that each property element has exactly one child. This restriction is more faithful to RDF's data model, where properties are required to be binary.

4.2.2. Literal-valued properties

Expressions of the form outlined above are desirable as they model paths through a graph in a very intuitive manner, but they are unnecessarily verbose when dealing with literals. Since literals in RDF can only appear in the object position of a triple, they ought to never have children in ENFR, and can thus be represented using attributes, in order to shorten path expression involving them. This choice, however, has some drawbacks:

- The result is undefined when two properties have the same name and they both have literal values, however this appears to be pretty rare.
- Datatypes for literals are not easily supported, as XML attributes cannot have attributes themselves. This could be addressed by the use of XML Schema datatypes, but we will not explore this option here.

It's important to stress, however, that these are shortcomings of the normal form, and they are not intrinsic in using a *just-in-time reflection*. In fact, they could both be overcome, at the price of verbosity, by using Alternating Normal Form.

Furthermore, since the implementation is based on extension functions, one may choose the most appropriate normal form on demand by calling the corresponding extension function. Since time constraints limited the implementation to one normal form, ENFR seemed more useful for the majority of cases.

4.3. Multiple roots

In general, a graph is not rooted, hence one may start navigating through it starting from any node. For this reason a reflection expressed in Edinburgh Normal Form has multiple roots, one for each subject in the graph, but this is a strength rather than a weakness, since we are generating a reflection dynamically instead of serialising.

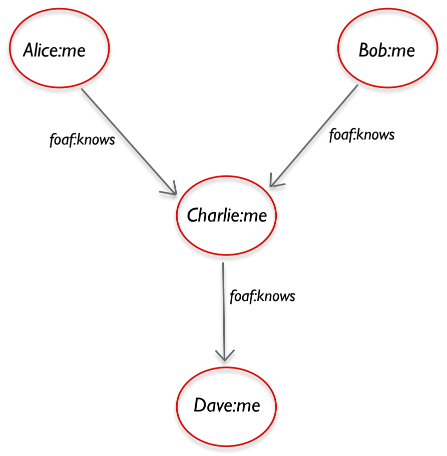
However, since a root node, by definition, has no parent [1], one cannot access properties having the root as their object using the parent axis: if one wants to do so, one has to choose a different starting node to act as the root.

4.4. The parent and ancestor axis:backtracking

Since an XML document is modelled as a rooted tree, every element has at most one parent node. This constraint is of course violated by an RDF graph, since a node can be the object of multiple statements, as in Figure 8, “The parent axis carries problems with directed non rooted graphs.”.

To understand the exact behaviour of our implementation of the parent axis given this, one must consider how nodes are generated by the reflection process: they can either be returned by an extension function to act as the root of a reflection, or reflected during the evaluation of a path expression. We 'solve' the parent axis problem by implementing it to backtrack one step along the path that led to the creation of the current node, returning the empty nodeset in case of a root node.

Figure 8. The parent axis carries problems with directed non rooted graphs.



Taking the graph in Figure 8, “The parent axis carries problems with directed non rooted graphs.” as an example, let's consider the following expressions [2], where the node following `reflect()` is the root of the reflection:

```
reflect()/Alice:me/..
```

returns the empty nodeset.

```
reflect()/Alice:me/foaf:knows/Charlie:me/../..
```

returns `Alice:me`.

```
reflect()/Bob:me/foaf:knows/Charlie:me/../..
```

returns `Bob:me`.

[1] Technically, it has a parent: the Document node, but this makes no difference in this context since the document is generated on demand via *reflection*, starting from the root node.

[2] `\..` is a short form for `\parent::*`.

Therefore, unlike in XML, the parent axis is path-dependant, meaning that the ancestors of a node are all the nodes visited in the evaluation of the current path that are reachable via the parent access. It follows that one can trace back from a node to the root used to evaluate the expression that generated it by simply querying the ancestor axis.

4.5. The descendant axis: reentrancy and circularity

Since RDF graphs, in the general case, are not acyclic, they can result in infinitely deep trees when using a serialisation that represents edges using nesting, such as ENFR. However, reflecting a graph dynamically ensures that this is not an issue in most cases, since the system only goes as deep as required by the query at hand, even if the graph contains a cycle.

A notable exception is the descendant axis: since it performs a depth first search, it exhibits non-terminating behaviour if the tree is infinitely deep. Using *just-in-time reflection* alone does not solve the problem in this case, because the search forces the reflection of a cyclic structure recursively, which is effectively the same as trying to serialise it.

For this reason, having an algorithm capable of detecting and breaking cycles is essential. This is however not sufficient to ensure a consistent behaviour of the descendant axis, due to the fact that, unlike in trees, the same node could be a descendant of the context node along multiple paths. Since graphs can exhibit this property, called *reentrancy*, independently from circularity, one might have duplicate nodes in the result nodeset, even when the search does terminate.

We have chosen to change the behaviour of the descendant axis by implementing a graph-based depth-first search algorithm, which deals explicitly with reentrancy and circularity.

The key idea of the search algorithm is to avoid visiting the same node twice, thus preventing endless loops or duplicate results when dealing with circular or reentrant graphs. For this purpose, an adequate definition of equality is crucial: in RDF two things are equal if they have the same URI, but, in this case, this definition cannot handle properties correctly.

In fact, if two properties with the same URI but different objects were to be considered equal, some branches would be left unexplored by the search algorithm, since it will erroneously think such branches were visited before.

To overcome this problem, the equality test for properties was changed so that two properties are equal if they have the same URI, **and** their children also have the same URI. Note that this definition is not ambiguous since in Edinburgh Normal Form properties are constrained to have exactly one child (the object of the statement that they encode).

4.6. The sibling axis: sibling properties

As a final remark, it's appropriate to discuss briefly the behaviour of the sibling axis. In the Edinburgh Normal Form for RDF, an element representing a node in the graph can either be:

- The **only child** of a property element, thus having no siblings.
- The **root node** of the reflection, which has no siblings by definition.

It should therefore be evident how the concept of sibling is only meaningful for property elements. In particular, sibling properties can be thought of as properties with the same subject.

5. An XSLT engine for RDF Graphs

Having discussed both *just-in-time reflection* and Edinburgh Normal Form, it is now possible to present the resulting engine. We will not go into much detail on its architecture, whereas greater attention will be given to the form of the stylesheets it can process.

5.1. Overall architecture

This section will describe the overall architecture of the system. While the discussion will be kept to a fairly high level, it should be enough to understand how a typical XSLT transformation is carried out.

The system consists mainly of a pipeline, with the RDF engine on one end and the XSLT processor on the other: these two components are connected by a set of extension functions, that basically allow access to the graph API from XSLT stylesheets. More specifically, the system takes two files as input: an RDF file, which is parsed by the RDF engine, and an XSLT stylesheet, which is compiled and executed by the XSLT processor, and can contain one or more calls to interface extension functions.

The different extension functions implemented have slightly different behaviours, but they can all be thought of as returning a special **wrapper object** to act as the root of the reflection. Since both expose the same interface to the XSLT processor, such a wrapper looks no different from the in-memory representation of a node generated during the parsing of a physical XML document, but it differs greatly in the implementation, since:

- It wraps an RDF resource, named after its QName, if available, or URI otherwise.
- It is capable of reflecting other nodes, so that, when used as the context node in a path expression step, it will use the rules laid out in the normal form to interpret it, and then query the RDF engine to retrieve the desired resources. The result is a nodeset accordingly populated with other wrappers, which are in turn used to reflect the next step in the path, and so on until the expression has been completely evaluated.
- It uses the depth-first graph search algorithm described above to ensure termination and absence of duplicate results when reflecting along the descendant axis.

Specifically, Jena and Saxon were used as the RDF and XSLT engine respectively: since Saxon represents XML nodes via the `NodeInfo` interface, the wrapper object, in which lies all the *just-in-time reflection* logic, is simply a custom implementation of this interface.

As briefly mentioned before, this architecture is not dependent on the normal form used: a different kind of wrapper object could implement an entirely different normal form. Furthermore, since the root wrapper is accessed through an extension function, different functions can return different wrappers, allowing to mix and match the normal form to the input data on demand.

5.2. Stylesheet structure

RDF stylesheets look almost identical to standard XML ones, the only difference being the use of extension functions in the initial template. Once their semantics are understood, an XSLT user should be able to write stylesheets for RDF easily, because, apart from the initial template, stylesheets look exactly the same as if they were processing a physical XML file expressed in Edinburgh Normal Form.

5.2.1. The Reflect function

Reflect is the most important function of the engine, in that it gives access to the Document node of the reflection. To better understand its usage, let's notice that when using XSLT with XML, if the user does not write an initial template, the processor implicitly executes the instruction:

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

where `/` is the document node and `*` is the outermost element, i.e. the root node of the tree. Subsequent templates then match the various children of the root node.

For RDF, the initial template is written explicitly, like so:

```
<xsl:template name="main">
  <xsl:apply-templates select="enf:reflect()/*"/>
</xsl:template>
```

There is an important difference though: in XML the outermost element is unique, for a Document node is required to have exactly one Element node among its children, whereas an ENFR reflection has multiple roots, and therefore one can replace the wildcard match with the name of any subject in the graph: the argument to `select` takes then the form `enf:reflect/root-qname`.

This characteristic is still compatible with the XPath 2.0 data model [BFM+10] and, while it can be useful in general to shorten some path expression, it becomes essential in the case of disconnected graphs.

However, note how one cannot write directly something like:

```
<xsl:template
  match="reflect()/Alice:me/foaf:knows/*">
```

Due to the fact that a template `match` expression cannot start with a function call or variable. This limitation should disappear in XSLT 3.0 [Kay13], but until then, the correct form is:

```
<xsl:template name="main">
 <xsl:apply-templates select="reflect()/Alice:me"/>
</xsl:template>

<xsl:template match="foaf:knows/*">
...
```

Finally, a name for the initial template is strictly required when writing a stylesheet for RDF: since the Saxon processor is not running on any physical XML source file, the initial template acts as the starting point for the transformation.

5.2.2. The Select function

Let's consider the following template, which operates on an RDF graph containing several statements, some of which are about Persons:

```
<xsl:variable name="doc" select="enf:reflect()"/>

<xsl:template name="main">
  <xsl:for-each select="$doc/*[rdf:type/
                               foaf:Person]">
    <div>
      <xsl:value-of select="@rdfs:comment"/>
    </div>
  </xsl:for-each>
</xsl:template>
```

The `foreach` loop outputs the textual description of the resources in the graph, but only if they are of type Person. Filtering by type is indeed a very common need, so it seemed appropriate to devote a function to it, called `select()`, which returns a RdfDocWrapper whose children are filtered according to the specified type. The stylesheet then becomes:

```
<xsl:variable name="doc"
              select="enf:select('foaf:Person')"/>

<xsl:template name="main">
  <xsl:for-each select="$doc/*">
    <div>
      <xsl:value-of select="@rdfs:comment"/>
    </div>
  </xsl:for-each
</xsl:template>
```

5.3. Transformation example

As an example of the capabilities of our engine, we will use it to perform the transformation outlined at the very beginning of this paper: the graph in Figure 1, “A simple RDF graph representation. The predicate names have been left blank for simplicity” is encoded by the following Turtle, which describes the connections between Alice, Bob, Charlie and Dave, using the `foaf:knows` predicate:

```
@prefix crc: <http://www.inf.ed.ac.uk/~fl/crc#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf:
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:
  <http://www.w3.org/2000/01/rdf-schema#> .

crc:A a foaf:Person ;
        foaf:name "Alice" ;
        foaf:knows crc:D .

crc:B a foaf:Person ;
        foaf:name "Bob" ;
        foaf:knows crc:C , crc:A .

crc:C a foaf:Person ;
        foaf:name "Charlie" ;
        foaf:knows crc:A , crc:B .

crc:D a foaf:Person ;
        foaf:name "Dave" ;
        foaf:knows crc:C .
```

This data is an ideal candidate to test the behaviour of the descendant axis, since it has several reentrant and cyclic paths. The idea is to use the descendant axis with the predicate `foaf:knows` to find out all the connections that each Person has, where a connection is defined as a Person that is at any degree of separation in the chain of acquaintances. The output in Figure 2 is produced by the following stylesheet:

```
<xsl:variable name="Node" select="enf:reflect()"/>

<xsl:template name="main">
  <html>
    <body>
      <table>
        <tr>
          <xsl:for-each select="$Node/*">
            <td>
              <table border="1">
                <tr bgcolor="#9acd32">
                  <th>
                    Connections for
                    <xsl:value-of
                      select="@foaf:name"/>
                  </th>
                </tr>
                <xsl:for-each select=".//*
                  [parent::foaf:knows]">
                  <tr>
                    <td>
                      <xsl:value-of
                        select="@foaf:name"/>
                    </td>
                  </tr>
                </xsl:for-each>
              </table>
            </td>
          </xsl:for-each>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>
```

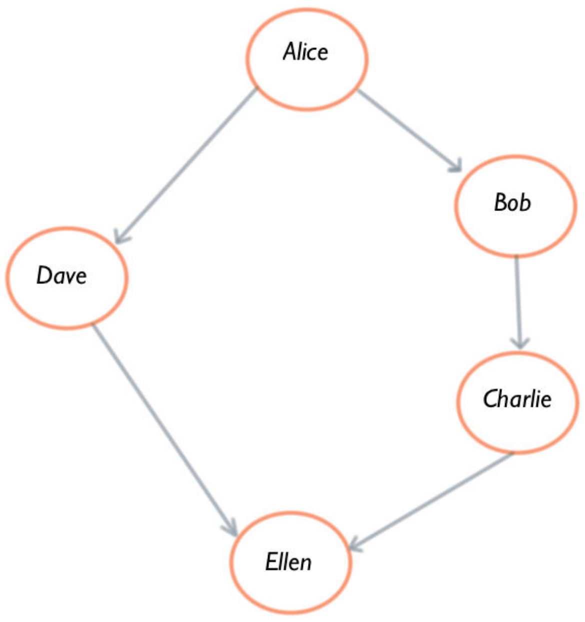
6. Limitations

While this project goes a good deal towards implementing a rendering language for RDF, it is not without limitations. Some of them are due to the relatively limited time available for the implementation, whereas others are intrinsic. Here is a brief overview:

- If two properties with the same name both have a literal object, the result of querying them is undefined, as ENFR encodes literal-valued properties as attributes, which are required to be unique. However is possible to implement another, more verbose normal form that does not have this limitation. More importantly, it's possible to use both at the same time by calling the appropriate extension function.
- Variables or function calls are not allowed in match expressions: this is inconvenient, as one has to resort to inelegant workarounds. This limitation is enforced by XSLT and therefore very little can be done about it, however it should disappear in XSLT 3.0.
- RDF Containers are unsupported. In theory they could be queried using standard XPath expressions, but no tests have been done to ensure they will work correctly.
- We would like to thank an anonymous reviewer for pointing out that there might be other ways to break the XSLT processor with a graph data model, including fetching the string value of an element, `fn:deep-equal()`, `<<` , `is`, and `fn:root()` and also for highlighting that document order might be an important factor. The current implementation guarantees to preserve document order as long as the underlying RDF engine does so during the parsing, and appears to deal well with `fn:string()` (in ENFR there are no text nodes). However, we have not explored these aspects in detail and they deserve further investigation.
- Using the ancestor and descendant axes in conjunction may lead to unexpected results. Let's consider for example the graph in Figure Figure 9, "DFS can cause non intuitive behaviour of the ancestor axis."; if one is interested in retrieving the descendants of Alice that are also descendants of Bob, one could write:

```
select="enf:reflect()/Alice//*[ancestor::Bob]"
```

Figure 9. DFS can cause non intuitive behaviour of the ancestor axis.



While it's clear that this query should select both Charlie and Ellen, only Charlie is guaranteed to always appear in the result nodeset. This happens because there are two paths from Alice to Ellen, and since the depth first search will avoid reflecting her twice, Ellen will be reached **either** through Bob and Charlie or through Dave, depending on the document order. If Ellen is reached via Dave, than her ancestors will be Dave itself and Alice, and therefore she will not be selected by the above query.

7. Related work

Besides the approach followed by TriX, various attempts have been made that do not require a change in the existing formats: most of them are now abandonware, yet some of the ideas used may still be worth exploring.

7.1. XSPARQL

XSPARQL does not deal directly with rendering RDF, but with the related and somewhat more general problem of RDF *lifting* and *lowering* [KVBF07],[FL07].

In this context lifting is not relevant, but it's not hard to envision how lowering, the task of translating RDF into arbitrary XML, originally for use with Web Services, could be adapted to produce XHTML for formatting purposes.

XSPARQL [AKKP08] stems from merging SPARQL into XQuery, so that RDF data is accessed using SPARQL, and then processed as in normal XQuery.

While undoubtedly an interesting approach, it has a few drawbacks: it is an entirely new technology, with only one implementation available, and, perhaps more importantly, does not use a path semantics, which is very familiar to users that deal with templating, since this is usually done in XSLT.

7.2. TriAl

TriAl [LRV13] is a query language for RDF capable of working directly on triples, in order to account for the few but significant differences between directed graphs and RDF's data model. TriAL semantics model RDF more closely than XPath does, but they also prevent its usage with XSLT, therefore making it a worse fit in the context of RDF rendering.

7.3. RDFXSLT

RDFXSLT [Kop07] is basically an XSLT stylesheet that turns RDF/XML into a more predictable form, which can then be queried using standard XPath expressions, along with a set of XSLT extension functions.

The main attractive of this approach is that is written in pure XSLT, and hence independent from any particular XSLT processor or platform. However, RDFXSLT is clearly limited to RDF/XML, and the RDF/XML subset it produces, although predictable and hence usable with XSLT, is admittedly ugly.

7.4. RDF Twig

RDF Twig [Wal03] is implemented through a set of XPath extension functions, which provide access to different XML serialisation of a graph, mainly breadth-first and depth-first trees. Twig is quite limited, in that it does not attempt a uniform mapping between RDF's abstract syntax like the one presented in this paper. However, it contains two important ideas:

1. It uses an RDF engine to access the graph, and then provides the result in a form amenable to XSLT processing.
2. The graph is not preprocessed, but rather the serialisations needed (e.g. a breadth-first tree) are generated on demand.

7.5. TreeHugger

TreeHugger [McD12] is an extension function that builds on Twig's two ideas as outlined above, and improves on them in that the serialisation it gives access to is a more accurate mapping from the RDF model to XML's than a mere search tree, like the ones used by Twig.

TreeHugger's approach is very similar to the one used in this project, however its implementation is limited to the parent, attribute and child axes[1].

7.6. RxSLT

The RxPath language [Sou06] is a more complete attempt at RDF rendering that supports all the XPath axes. However, it does not use XPath extension functions, but is a new language with a native processor.

Even though RxPath's syntax is designed to be identical to XPath 1.0, the choice of using a native processor carries a few consequences:

- It cannot benefit from improvements in newer versions of XPath.
- For implementation reasons, it constraints the behaviour of the ancestor and descendant axes [Sou06].

[1] This means three out of the thirteen XPath axes. Most notably, the descendant axis is missing.

- A new processor is likely to be not nearly as popular as the alternatives available for XSLT (like Saxon or Xalan).

However, it is the only attempt trying a complete, deterministic mapping from RDF to the XPath data model, and has been influential on this work.

RxSLT emulates XSLT by replacing XPath expressions with RxPath expressions to select nodes.

8. Conclusions and future work

After more than a decade, RDF is on its way to become a widespread technology. However, the lack of a suitable mechanism for rendering it to humans makes it costly to integrate into web-facing applications, and hardly appealing to the casual user. It is worth mentioning the case of SGML, whose diffusion was hindered by the lack of a template language, a mistake that its successor, XML, avoided thanks to the introduction of XSLT.

While XSLT cannot work directly on arbitrary RDF, and can only deal with RDF/XML at the price of great complexity, we showed how *just-in-time reflection* can be used to abstract away from the concrete syntax, and work directly on the data model. Furthermore, changing the descendant algorithm to a graph-based search accounting for both reentrancy and circularity completes the mapping to the tree based model used by XPath.

Now that a working implementation is available, several extensions are possible. First and foremost, new normal forms can be implemented, in order to avoid trading off generality for concision, as ENFR does. Also, it will be useful to add support for inference, which is currently lacking. This does not seem excessively hard, as it will probably only require adding the inferred triples to the Jena model.

Finally, one could improve on the Document node, so that it can act as the Saxon Source for the transformation. This is a very interesting possibility, as extension functions would then become unnecessary, making RDF stylesheets exactly identical to the ones used on XML. Achieving this goal, however, would require substantially more implementation effort than the approach reported here.

References

[AKKP08] Waseem Akhtar, Jacek Kopecky, Thomas Krennwallner, and Axel Polleres. *XSPARQL: Traveling between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage. The Semantic Web: Research and Applications*. Springer. 2008.
doi:10.1007/978-3-540-68234-9_33

[BBC+10] Anders Berglund, Scott Boag, Don Chamberlin, Mary Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. *XML path language (XPath) 2.0 (second edition)*. W3C recommendation. W3C. December 2010.
http://www.w3.org/TR/xpath20/

[BFM+10] Anders Berglund, Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 data model (XDM) (second edition)*. W3C recommendation. W3C. December 2010.
http://www.w3.org/TR/xpath-datamodel/

[BM12] Dan Brickley and Libby Miller. *Foaf vocabulary specification 0.98. Namespace Document*. 9. 2012.
http://xmlns.com/foaf/spec/

[BYM+08] Tim Bray, Francois Yergeau, Eve Maler, Jean Paoli, and Michael Sperberg-McQueen. *Extensible markup language (XML) 1.0 (fifth edition)*. W3C recommendation. W3C. November 2008.
http://www.w3.org/TR/REC-xml/

[CP14] Gavin Carothers and Eric Prud'hommeaux. *RDF 1.1 turtle*. W3C recommendation. W3C. February 2014.
http://www.w3.org/TR/turtle/

[CS04] Jeremy J Carroll and Patrick Stickler. *RDF triples in XML*. 412-413. Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. ACM. . 2004.
doi:10.1145/1010432.1010566

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 concepts and abstract syntax*. W3C recommendation. W3C. February 2014.
http://www.w3.org/TR/rdf11-concepts/

[FL07] Joel Farrell and Holger Lausen. *Semantic annotations for WSDL and XML schema*. W3C recommendation. W3C. August 2007.
http://www.w3.org/TR/sawsdl/

[GS14] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML syntax*. W3C recommendation. W3C. February 2014.
http://www.w3.org/TR/rdf-syntax-grammar/

[HB11] Tom Heath and Christian Bizer. *Linked data: Evolving the web into a global data space*. 1--136. *Synthesis lectures on the semantic web: theory and technology*. 1. 1. 2011.
doi:10.2200/s00334ed1v01y201102wbe001

[HS13] Steven Harris and Andy Seaborne. *SPARQL 1.1 query language*. W3C recommendation. W3C. March 2013.
http://www.w3.org/TR/sparql11-query/

[Kay09] Michael Kay. *XSL transformations (XSLT) version 2.0 (second edition)*. W3C recommendation. W3C. April 2009.
http://www.w3.org/TR/xslt20/

[Kay13] Michael Kay. *XSL transformations (XSLT) version 3.0*. Last call WD. W3C. December 2013.
http://www.w3.org/TR/2013/WD-xslt-30-20131212/

[Kop07] Jacek Kopecky. *Dx. yvzz rdfxslt: Xslt-based data grounding for rdf wsmo working draft 12 april 2007*. *WSMO Working Draft, WSMO*. 2007.
http://www.wsmo.org/TR/d24/d24.2/v0.1/20070412/rdfxslt.html

[KVBF07] Jacek Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. *Sawsdl: Semantic annotations for wsdl and xml schema*. 60--67. *Internet Computing, IEEE*. 11. 6. 2007.
doi:10.1109/mic.2007.134

[LRV13] Leonid Libkin, Juan Reutter, and Domagoj Vrgoc. *Trial for rdf: adapting graph query languages for rdf data*. 201--212. Proceedings of the 32nd symposium on Principles of database systems. ACM. . 2013.
doi:10.1145/2463664.2465226

[McD12] Mat McDermott. *Treehugger*. *TreeHugger*. 2012.

[MM04] Frank Manola and Eric Miller. *RDF primer*. W3C recommendation. W3C. February 2004.
http://www.w3.org/TR/rdf-primer/

[RGN+01] Jonathan Robie, Lars Marius Garshol, Steve Newcomb, M Fuchs, L Miller, D Brickley, V Christophides, and G Karvounarakis. *The syntactic web: Syntax and semantics on the web*. 411--440. *Markup Languages: Theory and Practice*. 3. 4. 2001.
doi:10.1162/109966202760152176

[Sou06] Adam Souzis. *Rxpath: a mapping of rdf to the xpath data model*. Extreme Markup Languages. . 2006.
http://conferences.idealliance.org/extreme/html/2006/Souzis01/EML2006Souzis01.html

[SR14] Guus Schreiber and Yves Raimond. *RDF 1.1 primer*. W3C note. W3C. June 2014.
http://www.w3.org/TR/rdf11-primer

[Tho01] H Thompson. *Normal form conventions for xml representations of structured data*. Proceedings of XML 2001. GCA. . 2001.
http://www.ltg.ed.ac.uk/~ht/normalForms.html

[TKC03] Henry S Thompson, K Ari Krupnikov, and Jo Calder. *Uniform access to infosets via reflection.*. Proceedings of Extreme Markup Languages 2003. Extreme Markup Languages. . 2003.
http://conferences.idealliance.org/extreme/html/2003/Thompson01/EML2003Thompson01.html

[Wal03] Norman Walsh. *Rdf twig: accessing rdf graphs in xslt.*. Extreme Markup Languages. Citeseer. . 2003.
http://nwalsh.com/docs/articles/extreme2003/

Publishing with XProc

Transforming documents through progressive refinement

Nic Gibson

Corbas Consulting and LexisNexis

`<nicg@corbas.co.uk>`

Abstract

Over the last few years, we, as a community, have spent a great deal of time writing code to convert Microsoft Word documents into XML. This is a common task with fairly predictable stages to it. We need to read the .Docx or WordML file and and transform the flat, formatting-rich XML in a well structured XML document.

One approach to this problem is to create a pipeline that uses a progressive refinement technique to achieve a simple sequence of transformations from one format to another. Given that this approach requires the ability to chain multiple transformations together, we decided to build a framework to enable that.

This paper explores the implementation of this kind of pipelining through XProc and examine the pipeline processing used. We discuss the use of progressive enhancement to convert Microsoft Word files to an intermediate format, considering the challenges involved in converting Word in context. We look at the features of XProc which enable this sort of processing.

Keywords: XProc, XSLT, Word

1. Introduction

Authors like Microsoft Word, transformers of content don't. Traditionally, a publishing house (or publishing organisation) would receive manuscript in Word, copy edit it, typeset it, proofread it and publish it. Structured authoring is not something the majority of authors are able do. Organisations and individuals involved in digital publishing conversion are interested in converting Word files to XML as it enables multi-output publishing, querying of documents and simplifies reuse. Conversion from Word to structured formats has become an important part of the publishing process. Currently, many publishers are migrating existing content from Word and/or RTF sources to XML. There is a need for a robust framework to enable this kind of conversion. The framework we present here was originally created in order to fulfil that requirement although it can be used for any multi-stage XML conversion task.

XSLT is the obvious choice for a processing tool but the environment in which that tool operates is a less clear choice. Apache Ant has been used but this usage is not within the realm which Ant was designed for and bulk conversion in Ant suffers from performance problems. We chose to implement an XProc based processing environment because the language supports in-memory pipelining of multiple documents through multiple steps. This dramatically decreases the overheads while proving a more elegant solution to the problem.

2. Progressive enhancement

It is, in general, simpler to create a sequence of transformations, each one focussed on a particular aspect of a task than it is to define a single complex transformation.

It is worthwhile considering that XML content is generally found in two basic forms: linear and structured. Microsoft Word (and sometimes HTML) are the archetypal linear formats. Linear formats are well suited to authoring as authoring tends to be a linear activity. XML languages such as DocBook, JATS and the internal formats used by many publishers are the structured forms to which we need to transform. Structured formats are well suited for transformation to multiple outputs, storage and analysis.

Conversion, is therefore, primarily as task of adding structure to content. Any reasonably complex, mature, publishing activity such as legal publishing will have complex formatting rules used by authors and editors when preparing documents. Conversion to structured XML can be considered primarily as the addition of structure. However, the conversion of an element in one language to semantically appropriate equivalent in the other is still a required portion of the process.

Complex conversions can be challenging for a developer to maintain. A sequence of simpler transformations has both a better change of being maintained and a better change of reuse.

When converting from WordML to XML we choose to convert to XHTML 5 before we convert to the final format. The vast majority of narrative documents can be decomposed into a sequence of nested sections and blocks of text.

3. Microsoft Word

The approach we have taken starts with conversion of Microsoft Word OOXML elements to XHTML 5 elements. Following that we, add structure to the intermediate XML document in several stages.

Legal documents tend to contain structured at several levels:

- the sectional structure of the document
- clauses
- numbered paragraphs

This structure is also found in legislation (although legislation tends to have additional structures such as chapters).

The approach we are describing here is intended for use in bulk conversion. This allows us to ignore problems of performance in some ways. There may be performance issues caused by this approach and we intend to investigate this a later date.

4. WordML conversion

In current versions Word emits an XML based format (either packaged into a zip file or as a single XML file). The difference between the various versions in the wild is not hugely significant and support for the varying formats can be easily achieved.

Fundamentally, the vast majority of constructs in a Word document are paragraphs. Lists are represented as paragraphs so grouping is required to identify and mark up list content. Tables are marked up using a format very different to either that of CALS or HTML.

Figure 1. Formatting in Word

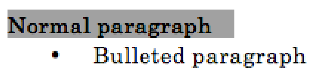


Image and picture markup can be very complex. In the context of the projects in which the toolkit described here is used, image markup has not been complex and no attempt has been made to find a general solution to the problem. Consider the XML created by Word for those two paragraphs:

```
<w:p>
  <w:r>
    <w:t>Normal paragraph</w:t>
  </w:r>
</w:p>
<w:p>
  <w:pPr>
    <w:pStyle
     w:val="ListParagraph"/>
    <w:numPr>
      <w:ilvl
       w:val="0"/>
      <w:numId
       w:val="1"/>
    </w:numPr>
  </w:pPr>
  <w:r>
    <w:t
    >Bulleted paragraph</w:t>
  </w:r>
</w:p>
```

An initial conversion step would convert these paragraphs to XHTML preserving some of the important information:

```
<p>Normal paragraph</p>
<li cword:list-level="0"
 cword:list-mark="1"
 >Bulleted paragraph</li>
```

Obviously, this content is not yet close to a structured output but it has been simplified and important information retained.

A second step would be to add the list markup around the `li` element:

```
<p>Normal paragraph</p>
<ul>
        <li>Bulleted paragraph</li>
</ul>
```

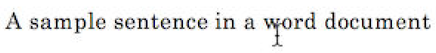
At this point the namespaced attributes are no longer required because we have determined that the list is a simple bulleted list (by examining other structures in the WordML).

4.1. Challenges in conversion of WordML to XHTML

There are several challenges in conversion. Change tracking markup and annotations have no obvious analogs in XHTML. However, these are simply challenges in that a decision has to be made and implemented (we have generally converted change tracking markup to sequences of spans and annotations to XHTML spans).

Inline text can be complex when converting from Word markup to XHTML. A WordML paragraph consists of one or more “runs” of text. These can have complex properties and, sometimes, it can be unclear why a span exists at all.

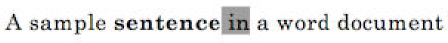
Example 1. Inline formatting in Word


A sample sentence in a word document

```
<w:p>
  <w:r>
    <w:t xml:space="preserve"
    >A sample sentence in a word document
    </w:t>
  </w:r>
</w:p>
```

If a user formats any of the text in the paragraph (either using a character style or direct formatting), Word will add additional run elements:

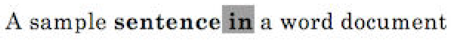
Example 2. User marks text as bold


A sample **sentence** **in** a word document

```
<w:p>
  <w:r>
    <w:t xml:space="preserve"
    >A sample </w:t>
  </w:r>
  <w:r w:rsidRPr="0077343A">
    <w:rPr>
      <w:b/>
    </w:rPr>
    <w:t>sentence</w:t>
  </w:r>
  <w:r>
    <w:t xml:space="preserve"
    > in a word document</w:t>
  </w:r>
</w:p>
```

Here, a word has been marked as bold. This processing is relatively simple. However, almost every user change of this type will be represented in this way even if those changes lead to identical formatting:

Example 3. User changes an adjacent run


A sample **sentence** **in** a word document

```
<w:p>
  <w:r>
    <w:t xml:space="preserve"
    >A sample </w:t>
  </w:r>
  <w:r >
    <w:rPr>
      <w:b/>
    </w:rPr>
    <w:t>sentence</w:t>
  </w:r>
  <w:r>
    <w:rPr>
      <w:b/>
    </w:rPr>
    <w:t xml:space="preserve"
    > in</w:t>
  </w:r>
  <w:r>
    <w:t xml:space="preserve"
    > a word document</w:t>
  </w:r>
</w:p>
```

The initial conversion to XHTML for a paragraph like this would be:

```
<p>A sample <strong>sentence</strong>
  <strong> in</strong> a word document</p>
```

Word does not merge runs in identical formatting so the initial, simplistic, conversion won’t either. This is something that needs to be resolved in structured XML. The obvious solution to this problem is to create a step in our pipeline to handle this situation. It should be possible to create a general transformation.

5. Progressive enhancement and XSLT

Progressive enhancement is XSLT is simple. The identity transformation does much of the work for us. We will demonstrate the process using a transformation to XHTML 5. This transformation is useful as the first stage of a transformation into another XML format.

The first step of our conversion process is generally to convert the WordML elements to XHTML 5 elements without regard to validity. During this step we convert paragraphs and tables and we maintain the original Word styles using a namespaced attribute.

There is no major complexity in the initial conversion.

```
<xsl:stylesheetl
version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:cword="http://www.corbas.co.uk/ns/word"
  xmlns:w="http://schemas.microsoft.com/office/wo
rd/2003/wordml"
  xmlns="http://www.w3.org/1999/xhtml"
 xpath-default-namespace="http://schemas.microso
ft.com/office/word/2003/wordml">

  <xsl:import href="identity.xsl"/>

  <xsl:template match="w:wordDocument">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head/>
      <xsl:apply-templates select="w:body"/>
    </html>
  </xsl:template>

  <xsl:template match="w:body">
    <body>
      <xsl:apply-templates/>
    </body>
  </xsl:template>

  <xsl:template
  match="w:p[w:pPr/w:numPr[w:numId and w:ilvl]]"
  priority="1">
    <li
cword:list-level="{w:pPr/w:numPr/w:ilvl/@w:val}"
cword:list-mark="{w:pPr/w:numPr/w:numId/@w:val}">
     <xsl:next-match/>
    </li>
  </xsl:template>

  <xsl:template match="w:p">
    <p><xsl:apply-templates/></p>
  <xsl:template/>

  <xsl:template match="w:pPr/w:pStyle">
    <xsl:attribute name="cword:style"
      select="@w:val"/>
  </xsl:template>

</xsl:stylesheet>
```

Each step is used to transform the content to a form nearer to that desired in the final output. This process allows us to maintain relatively simple XSLT whilst creating a complex transformation. Additional steps can be written to improve the transformation.

5.1. Using meta-programming to structure content taken from Word

Adding structure to content taken from Microsoft Word can be challenging. There is nothing inherently different about a heading or title in Microsoft Word — they are simply paragraphs with a style. It is relatively simple to convert the Word built-in paragraph styles to HTML heading elements:

Style	Element
Heading 1	h1
Heading 2	h2
Heading 3	h3

However, publishers conventionally use document and task specific styles in Word:

- Title
- Sub Title
- Clause Title
- A-Head
- B-Head

We could write a new stylesheet to convert these elements to the right headings and insert this into our sequence of stylesheets:

```
<xsl:template match="p[@cword:style='Title']">
  <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="p:[@cword:style='Sub Title']">
  <h2><xsl:apply-templates/></h2>
</xsl:template>
```

It is clear that any stylesheet as repetitive as this could be replaced with a configuration file and another stylesheet to generate it. We took this approach because it allows a stylesheet to be generated from a standard configuration:

```
<map xmlns="http://www.corbas.co.uk/ns/transforms/map"
    xmlns:cword="http://www.corbas.co.uk/ns/word"
    source-attribute="cword:style"
    ns="http://www.w3.org/1999/xhtml"
    source-element="p">
  <mapping source-value="Title"
    target-element="h1" heading-level="1"/>
  <mapping source-value="Sub Title"
    target-element="h2" heading-level="2"/>
  <mapping source-value="Clause Title"
    target-element="h3" heading-level="2"/>
</map>
```

We can then write a stylesheet that will generate the appropriate stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
  xmlns:axsl=
    "http://www.w3.org/1999/XSL/TransformAlias"
  exclude-result-prefixes="xs xd axsl"
  version="2.0">

  <xsl:strip-space elements="*"/>
  <xsl:namespace-alias stylesheet-prefix="axsl"
                       result-prefix="xsl"/>

  <xsl:template match="mapping" as="element()">

    <axsl:template>
      <xsl:apply-templates select="@source-value"/>
      <xsl:apply-templates select="."
        mode="generate-elements"/>
    </axsl:template>

  </xsl:template>

  <xsl:template match="@source-value">
    <xsl:attribute name="match"
      select="concat(../@source-element, '[@',
      /map/@source-attribute, ' = ''', ., ''']')"/>
  </xsl:template>

  <xsl:template match="mapping"
    mode="generate-elements">

    <xsl:param name="element-list" as="xs:string*"
      select="tokenize(@target-element, '\s+')"/>
    <xsl:param name="top-level" as="xs:boolean"
                                select="true()"/>

    <xsl:choose>

      <!-- If there are no input elements in the
           sequence, create an apply-templates only
           - stop the recursion -->
      <xsl:when test="count($element-list) = 0">
        <axsl:apply-templates select="node()"/>
      </xsl:when>

      <xsl:otherwise>

        <!-- Generate a literal element -->
        <xsl:element name="{$element-list[1]}"
          namespace="{/map/@ns}">

         <!-- If top level, process mapping
              attributes and generate an apply
              templates for the input ID attributes
              (if any) -->
          <xsl:if test="$top-level = true()">
            <xsl:apply-templates
              select="@hint|@heading-level"/>
            <axsl:apply-templates
              select="@*[local-name() = 'id']"/>
          </xsl:if>

          <xsl:apply-templates select="."
                               mode="copy-attributes"/>

          <!-- Recursing passing the tail of the
               sequence  and setting top-level to
               false -->
          <xsl:apply-templates select="."
                              mode="generate-elements">
            <xsl:with-param name="element-list"
             select="subsequence($element-list, 2)"/>
            <xsl:with-param name="top-level"
                                select="false()"/>
          </xsl:apply-templates>

        </xsl:element>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet generates a new stylesheet which we can apply to the content. We consider that the ability to generate stylesheets for predictable parts of the process is very important. It allows us to reduce the amount of new code written for each transformation. The purpose of the toolkit we have created is to maximise reuse. Configuration driven approaches to transformations allow us to reduce the custom code requirements dramatically.

5.2. Implementation

The XSLT approach techniques discussed here have allowed us to create a set of stylesheets that maximise reuse and minimise coding. However, we require an environment in which this code can be run. We considered several approaches to constructing the framework in which these stylesheets could be executed.

The first approach constructed a shell script using XSLT. This proved inefficient since the overhead caused by IO requirements and the overhead of restarting the JVM proved excessive. A second approach was taken where we wrote custom software using Perl and the XML::LibXML libraries. This was more efficient but required us to maintain and support a second codebase. The first XProc implementations appeared at the time at which we were creating the Perl implementation. We chose to investigate using XProc to process our documents.

Given that the stylesheets required for any given transformation process may differ from that required for any other process, we decided to find an approach which allowed us use configuration files to indicate which stylesheets should be run and in which sequence.

6. XProc

We chose to implement this framework using XProc because we believed that it had all of the features we required built into the implementations. We chose to use Norm Walsh's Calabash XProc processor.

XProc is a language for specifying pipelined transformations on XML documents. A pipeline takes one or more documents as input and produces one or more as output. There is a certain amount of symmetry to an XProc pipeline since steps in the pipeline and the pipeline itself share semantics allowing an existing pipeline to be used as a step in another pipeline.

6.1. Manifest files

We have defined a schema and processing mechanism for manifest files which allows an XProc step to load a sequence of XSLT files via a manifest file and prepare them for evaluation against a document.

At its simplest a manifest file may be a simple set of stylesheets:

```
<manifest
  xmlns="http://www.corbas.co.uk/ns/transforms/data"
  version="1.0">
  <item href="word-to-xhtml5-elements.xsl"/>
  <item href="wrap-blocks.xsl"/>
  <item href="merge_sups.xsl"/>
  <item href="merge_spans.xsl"/>
</manifest>
```

This manifest simply indicates that four documents should be loaded and a sequence of documents returned by the XProc step Additional features are available to allow metadata to be stored with the document after loading and to allow documents to be processed with an XSLT stylesheet before being returned:

```
<item href="merge_spans.xsl"/>

<processed-item
  stylesheet="build-mapping-stylesheet.xsl">
  <item xml:base="../../mapping/" href="efp.xml"/>
</processed-item>

<item href="rewrite-para-numbers.xsl"/>

<item href="merge_spans.xsl">
  <meta name="merge-sup" value="false"/>
</item>
```

Finally, inclusion and grouping syntax is available to allow metadata items to be included and to enable and disable items and groups.

6.2. Applying XProc to the problem

If we break down the problem into multiple stages we find that it can be viewed as:

1. Load a sequence of XSLT stylesheets to be applied to a source document
2. Load a source document to be processed
3. Thread the source document through the XSLT files
4. Return the final output

6.2.1. Loading the manifest

The XProc specification provides for 31 required steps and 10 optional steps. Additionally, custom steps can be written. An XProc step is an operation on (generally) one or more XML documents resulting in one more XML documents. The two major stages identified above map to custom XProc steps.

XProc operates on documents and sequences of documents in much the same way as XSLT operates on nodes and sequences of nodes. This feature allows us to create an XProc step which reads a configuration file, loads the documents referenced in that file and return a sequence of files as the result

```
<p:declare-step ❶
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:ccproc=
  "http://www.corbas.co.uk/ns/xproc/steps"
  type="ccproc:load-sequence-from-file"
  name="load-sequence-from-file">

  <p:input port="source" primary="true"/> ❷
  <p:output port="result" primary="true"
    sequence="true">
    <p:pipe port="result" step="load-iterator"/>
  </p:output> ❸

  <p:for-each name="load-iterator"> ❹

    <p:output port="result" primary="true">
      <p:pipe port="result" step="load-doc">

    <p:iteration-source select="/data:manifest/*">
      <p:pipe port="source"
      step="load-sequence-from-file"/>
    </p:iteration-source>

    <p:variable name="href"
      select="p:resolve-uri(/data:item/@href,
                  p:base-uri(/data:item))"/>

    <p:load name="load-doc">
      <p:with-option name="href" select="$href"/>
    </p:load>

  </p:for-each>

</p:declare-step>
```

❶ A custom step in XProc is created using `p:declare-step`

❷ In this context the `p:input` statement declares an input. The `primary` attribute is used to indicate that this is the primary input to the step (it will connect automatically and acts as the default XPath context).

❸ The `p:output` statement declares an output. The output of the step is the output of the substep called **load-manifest**. We also indicate that it returns a sequence of documents.

❹ This substep loops over items in the input document, resolves the URI against the document itself and then loads the file.

This step (simplified from real-world code) shows the basic features of the XProc environment. In order to load the files listed in the manifest file, we use the manifest as an input document, use XPath to locate and resolve the URIs and then load them using the built-in `p:load` step. The `p:load` step returns an XML document, the loop returns a sequence of them. This is used as the custom step output. Once this code has been saved, we can import it and use it as it were a built-in step. This is one of the great advantages of working with XProc — the ability to extend functionality and reuse high level operations. This step alone provides us with the ability to list the transformations applicable to a document (or set of documents) and laod them.

6.2.2. Processing the manifest

In order to process a source document and convert it to XHTML, it is necessary to apply the sequence of stylesheets loaded by the step we previous described. This initially presented some challenges when we considered implementing it in XProc. There is nothing in the XProc specification that would allow a document to be threaded through a sequence of stylesheets. Therefore, it was necessary to define a custom step.

Jostein Jacobsen suggested an approach using recursion and Romain Deltour provided a simple implementation which forms the basis of the step we wrote. We can approach the problem indirectly. Given a document and a sequence of stylesheets the task can be broken down into repeated application of a simpler task: apply the first stylesheet in a sequence to a document and then apply the same approach to the output and the next document.

This is relatively simple in XProc and the custom step model allows us to hide the complexity of the process in a simple appearing interface.

```
<p:declare-step
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:ccproc=
    "http://www.corbas.co.uk/ns/xproc/steps"
  name="threaded-xslt"
  type="ccproc:threaded-xslt">

  <p:input port="source" sequence="false"
    primary="true"/>

  <p:input port="stylesheets"
    sequence="true"/>

  <p:output port="result" primary="true"
    sequence="true">
    <p:pipe port="result"
      step="determine-recursion"/>
  </p:output>

  <!-- Split of the first transformation from
       the sequence -->
  <p:split-sequence name="split-stylesheets"
    initial-only="true" test="position()=1">
    <p:input port="source">
      <p:pipe port="stylesheets"
      step="threaded-xslt"/>
  </p:input>
</p:split-sequence>

<!-- How many of these are left? We actually
     only care to know  if there are *any*
     hence the limit. -->
<p:count name="count-remaining-transformations"
  limit="1">
  <p:input port="source">
    <p:pipe port="not-matched"
      step="split-stylesheets"/>
  </p:input>
</p:count>


<!-- run the stylesheet/ -->
<p:xslt name="run-single-xslt">
  <p:input port="stylesheet">
    <p:pipe port="matched"
      step="split-stylesheets"/>
  </p:input>
  <p:input port="source">
    <p:pipe port="source"
      step="threaded-xslt"/>
  </p:input>
  <p:input port="parameters">
    <p:empty/>
  </p:input>
</p:xslt>


<!-- If there are any remaining stylesheets
     recurse. The primary input is the result of
     our XSLT and the remaining sequence from
     split-transformations above will be the
     stylesheet sequence  -->
<p:choose name="determine-recursion">

  <p:xpath-context>
    <p:pipe port="result"
      step="count-remaining-transformations"/>
  </p:xpath-context>


  <!-- If we have any transformations
       remaining recurse -->
  <p:when test="number(c:result)>0">

    <p:output port="result" sequence="true">
      <p:pipe port="result"
        step="run-single-xslt"/>
      <p:pipe port="result"
        step="continue-recursion"/>
    </p:output>

    <ccproc:threaded-xslt-impl
      name="continue-recursion">

      <p:input port="stylesheets">
        <p:pipe port="not-matched"
          step="split-stylesheets"/>
      </p:input>

      <p:input port="source">
        <p:pipe port="result"
          step="run-single-xslt"/>
      </p:input>

    </ccproc:threaded-xslt-impl>

  </p:when>

  <!-- Otherwise, pass the output of our
       transformation back as the result -->
  <p:otherwise>

    <p:output port="result" sequence="true">
      <p:pipe port="result"
        step="terminate-recursion"/>
    </p:output>

    <p:identity name="terminate-recursion">
      <p:input port="source">
        <p:pipe port="result"
          step="run-single-xslt"/>
      </p:input>
    </p:identity>

  </p:otherwise>

</p:choose>

</p:declare-step>
```

The listing above is a simplified version of the step in the framework. In addition to processing content with XSLT, the framework version of the code supports XSLT parameters and debug output.

This step allows us to emulate the process of iterating and threading using recursion. Each call to the step processes the output of the preceding steps as input using the first stylesheet in the sequence. If the sequence is not empty after the first has been consumed, then the step calls itself using the output as the input document for the next call and the tail of the stylesheet sequence.

The power of the XProc `p:declare-step` statement is again demonstrated here. The relatively complex processing of the sequence of stylesheets can be hidden behind a simple interface:

```
<ccproc:threaded-xslt
  name="thread-content">
  <p:input port="source">
    <p:document href="my-doc.xml"/>
  </p:input>
  <p:input port="stylesheets">
    <p:pipe port="result" step="load-stylesheets"/>
  </p:input>
</ccproc:threaded-xslt>
```

6.2.3. A complete pipeline

In order to appreciate the power of the XProc approach to building pipelines, we must examine the final pipeline code.

```
<p:declare-step
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:ccproc=
   "http://www.corbas.co.uk/ns/xproc/steps"
  version="1.0" name="process-doc">

  <p:input port="manifest"/>
  <p:input port="source"/>
  <p:output port="result">
    <p:pipe port="result" step="run-stylesheets"/>
  </p:output>

  <p:import href="load-sequence-from-file.xpl"/>
  <p:import href="threaded-xslt.xpl"/>

  <!-- load the stylesheets -->
  <ccproc:load-sequence-from-file
    name="load-transformations">
    <p:input port="source">
      <p:pipe port="manifest" step="process-doc"/>
    </p:input>
  </ccproc:load-sequence-from-file>

  <!-- run it through the stylesheets
    in the manifest -->
  <ccproc:threaded-xslt name="run-stylesheets">
    <p:input port="source">
      <p:pipe port="result" step="process-doc"/>
    </p:input>
    <p:input port="stylesheets">
      <p:pipe port="result"
        step="load-transformations"/>
    </p:input>
  </ccproc:threaded-xslt>

</p:declare-step>
```

The final pipeline chains together the two custom pipelines and allows a user process a document using the progressive transformation framework. The model of chaining and layering steps used by XProc allows for high levels of modularity and reuse. The combination of XSLT and XProc provides and elegant solution to the problem of reliable conversion from Microsoft Word to useful structured XML.

7. Conclusions

Processing Word (or other complex XML sources) can be challenging using conventional single stylesheet or Ant driven approaches. The majority of the XSLT code required is quite generic. However, the requirements imposed by varying style information on the Word document make it impossible to design a stylesheet that can both cope with all situations and be maintained. Therefore, an combination approach of using progressive enhancement and careful metaprogramming techniques provides a poweverful mechanism for building generic processing tools. XProc provides a valuable and elegant environment in which to build the systems required to chain transformations agains one or more files together.

All of the code discussed in this paper is available from our Github repositories at https://github.com/Corbas/xproc-tools and https://github.com/Corbas/mapping-tools. The code presented here is a simplified version of that held in the repositories.

We have implemented this code as the basis of a conversion pipeline at LexisNexis which has been used to successfully convert 49.000 legal precedents from RTF to a proprietory XML format. The only made was to replace the single input document with the results of a directory traversal. We believe that this demonstrats the validity of both the approach and implementation.

7.1. Issues with XProc

Chaining steps can be opaque. The default input and outputs of a step and their relationship to other steps are not clear. Explicit input and output definitions are simpler to use but are complex. In the examples in this paper we have avoided the use of implicit inputs and outputs because they do not add to code clarity and they are generally of little use as soon as steps with multiple inputs or outputs are used. We believe that the implicit default mechanism is valuable and would prefer to see a mechanism for simplifying the definition of all inputs and outputs.

Iteration. The ability to iterate over a sequence of steps would be a great advantage to processing XProc pipelines. The `xsl:iterate` statement added to XSLT 3 could provide a useful template. A new compound step allowing iteration over content with the ability to wire inputs to outputs would reduce the need for recursion and mitigate the risk that a complex pipeline could cause memory issues.

Memory. The pipeline can be very memory intensive. We have seen XProc pipelines using over 10GB of memory because each document is held for the duration of the script. There does not appear to be any mechanism by which an XProc processor can determine when a document is no longer required by the pipeline. The addition of some mechanism which would allow the pipeline implementer to indicate that a document is no longer required which go some way to resolving the problem.

Data-Driven Programming in XQuery

Eric van der Vlist

Dyomedea

Abstract

Data-driven development is a popular programming paradigm often implemented using reflection in object-oriented programming languages.

Even if this is less common, data-driven development can also be implemented with functional programming languages, and this paper explores the possibilities opened by higher-order functions in XQuery 3.0 to develop data-driven applications.

1. Problem statement

1.1. Definitions

In computer programming, **data-driven programming** is a programming paradigm in which the program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken.[1] Standard examples of data-driven languages are the text-processing languages sed and AWK,[1] where the data is a sequence of lines in an input stream – these are thus also known as line-oriented languages – and pattern matching is primarily done via regular expressions or line numbers.

--Wikipedia

Data Driven Programs are programs which process data files whose contents cause the program to do something different. The extreme case is an interpreter and the interpretable program files.

--c2.com

If the ultimate result of an application is data, and all input can be represented by data, and it is recognised that all data transforms are not performed in a vacuum, then a software development methodology can be founded on these principles, the principles of understanding the data, and how to transform it given some knowledge of how a machine will do what it needs to do with data of this quantity, frequency, and it's statistical qualities. Given this basis, we can build up a set of founding statements about what makes a methodology data-oriented.

--Data-oriented Design

1.2. Real world examples

These definitions make it clear that any XSLT transformation making good use of templates is a data-driven program and that XSLT is the best example of a programming language rooted in the data-driven programming paradigm.

Even so, I had to wait until 2003 to discover the notion of data-driven programming in "Data-Driven Classes in Ruby", a most inspiring presentation by Michael Granger and David McCorkhill at OSCON 2003. So inspiring that I followed up with my own "XML Driven Classes in Python" presented at OSCON 2004 and then TreeBind in 2005.

The common point between these three methodologies or libraries and many others is to bridge the gap between data-driven and object-oriented programming by driving object-oriented classes and methods through data.

These approaches are very useful in a number of cases. For instance, the Python utility which is backing up all my servers and managing their archives hes, since 2006, been driven by XML configuration files processed by the library presented at OSCON 2004. Similarly, the mailing list manager handling emails sent to the XML Guild is powered by TreeBind driven by the following XML document (sensitive information have been removed for obvious reasons):

doi:10.14337/XMLLondon15.Vlist01

```
<?xml version="1.0"?>

<listManager>
  <server>localhost</server>
  <storeType>imap</storeType>
  <user>...</user>
  <password>...</password>
  <port>143</port>
  <folderManager>
    <folder>INBOX</folder>
    <messageHandler>
      <ifEither>
        <ifIsRecipient>
          info@xmlguild.org
        </ifIsRecipient>
        <ifIsRecipient>
          info@thexmlguild.org
        </ifIsRecipient>
        <ifIsRecipient>
          info@xmlguild.info
        </ifIsRecipient>
        <ifIsRecipient>
          info@xml-guild.org
        </ifIsRecipient>
        <ifIsRecipient>
          info@xml-guild.com
        </ifIsRecipient>
      </ifEither>
      <sendToList>
        <subjectPrefix>
          [the XML Guild]
        </subjectPrefix>
        <footer><![CDATA[
--
The XML Guild
    where you find established XML experts . . .
          http://xmlguild.org/
          info@xmlguild.org
]]></footer>
        <recipient>vdv@dyomedea.com</recipient>
        <recipient>...</recipient>
        <recipient>
          <!-- other recipients removed -->
        </recipient>
        <envelopeFrom>
          info-bounce@xmlguild.org
        </envelopeFrom>
        <header name="Precedence">
          List
        </header>
        <header name="List-Id">
          &lt;info.xmlguild.org&gt;
        </header>
        <header name="List-Post">
          &lt;mailto:info@xmlguild.org&gt;
        </header>
        <server>localhost</server>
        <user>...</user>
        <archive>archive</archive>
      </sendToList>
      <moveTo>done</moveTo>
    </messageHandler>
    <messageHandler>
      <moveTo>unparsed</moveTo>
    </messageHandler>
  </folderManager>
</listManager>
```

1.3. More precisely

Data-driven programming is a paradigm.

Like for any paradigm, programming languages and libraries influence the easiness with which data-driven applications can be written:

- In Apple][basic you'd have to use a bunch of if/then/else statements to implement data-driven applications.
- In object-oriented languages it is convenient to use a data binding library to directly bound data to object and methods to write data-driven applications.
- In XSLT, templates are natively bound to data through patterns.
- For functional programming languages, the frontier between code and data is fuzzy since functions are considered as data and we can expect a very special relation to data "driveness" as we will see in the next sections.

2. Data Driving XQuery

If XSLT is natively data-driven and object-oriented languages can be used with a twist to develop data-driven programs, what can we say about functional programming languages in general and XQuery in particular?

2.1. A simple example

As an example to illustrate our discussion, we will implement the example Hangman game given by

Wikipedia. The state of the current game will be defined by the following XML document:

```
<hangman status="in-progress" misses="Z">
  <word>
    <letter guessed="true">H</letter>
    <letter guessed="true">A</letter>
    <letter guessed="true">N</letter>
    <letter guessed="false">G</letter>
    <letter guessed="true">M</letter>
    <letter guessed="true">A</letter>
    <letter guessed="true">N</letter>
  </word>
  <display>
    <head shown="true"/>
    <body shown="true"/>
    <right_arm shown="true"/>
    <left_arm shown="true"/>
    <right_leg shown="false"/>
    <left_leg shown="false"/>
  </display>
</hangman>
```

And the implementation will consist in updating the document based on a letter given as a parameter.

2.2. Python

Before jumping to XML technologies, we can have a look at a possible Python implementation using the XML data-driven classes that I presented at OSCON 2004:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
__author__ = 'vdv'

import XmlObject


# Class Hangman: overall logic
class Hangman(XmlObject.XmlObjectElement):

  def isInProgress(self):
    return self.status._value() == 'in-progress'

  def addGuess(self, guess):
    isAGuess = self.word.addGuess(guess)
    if isAGuess:
      if self.word.areAllGuessed():
        self.status._set('won')
      else:
        self.misses._set(
          self.misses._value() + guess )
        self.display.addFailure()
        if self.display.areAllShown():
          self.status._set('lost')

XmlObject.XmlObjectElement_hangman = Hangman

# Class Word: handle guesses
class Word(XmlObject.XmlObjectElement):

  def addGuess(self, guess):
    result = False
    for letter in self.letter:
      if letter.addGuess(guess):
        result = True
    return result

  def areAllGuessed(self):
    for letter in self.letter:
      if not letter.isGuessed():
        return False
    return True

XmlObject.XmlObjectElement_word = Word

# Class Letter: individual letters to be guessed
class Letter(XmlObject.XmlObjectElement):

  def addGuess(self, guess):
    if self._value() == guess:
      self.guessed._set('true')
      return True
    else:
      return False

  def isGuessed(self):
    return self.guessed._value() == 'true'

XmlObject.XmlObjectElement_letter = Letter

# Class Display: displays the hangman
class Display(XmlObject.XmlObjectElement):

  def addFailure(self):
    result = False
    for member in self._childElements:
      if not member.isShown():
        member.show()
        return

  def areAllShown(self):
    return self._childElements[-1].isShown()

XmlObject.XmlObjectElement_display = Display

# Class members (generic to different
# element types): hangman members to be displayed
class Member(XmlObject.XmlObjectElement):

  def isShown(self):
    return self.shown._value() == 'true'

  def show(self):
    self.shown._set('true')


XmlObject.XmlObjectElement_head = Member
XmlObject.XmlObjectElement_body = Member
XmlObject.XmlObjectElement_right_arm = Member
XmlObject.XmlObjectElement_left_arm = Member
XmlObject.XmlObjectElement_right_leg = Member
XmlObject.XmlObjectElement_left_leg = Member

# Main

x = XmlObject.XmlObjectDocument()
x._Parse('hangman.xml')
hangman = x.hangman
print hangman._xml.toxml()

while hangman.isInProgress():
  guess = raw_input('Enter a letter: ')
  hangman.addGuess(guess[0].upper())
  print hangman._xml.toxml()
```

In this first example we have followed the principles of object orientation by defining a class for each element and never short-circuiting intermediary classes by accessing descendant properties directly.

2.3. XSLT

In XSLT on the contrary, we tend to use XPath to access the information wherever it can be found, and can implement the hangman as:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  extension-element-prefixes="xs" version="2.0">

  <xsl:strip-space elements="*"/>
  <xsl:output indent="yes"/>

  <xsl:param name="guess"/>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="hangman">
    <xsl:copy>
      <xsl:variable name="updated-content"
                    as="element()+">
        <xsl:apply-templates select="*"/>
      </xsl:variable>
      <xsl:attribute name="status">
        <xsl:choose>
          <xsl:when test="not($updated-content/
            self::word/letter[@guessed='false'])">
            won
          </xsl:when>
          <xsl:when test="not($updated-content/
            self::display/*[@shown='false'])">
            lost
          </xsl:when>
          <xsl:otherwise>
            in-progress
          </xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
      <xsl:attribute name="misses">
        <xsl:value-of select="@misses"/>
        <xsl:if test="not(word/letter = $guess)">
          <xsl:value-of select="$guess"/>
        </xsl:if>
      </xsl:attribute>
      <xsl:copy-of select="$updated-content"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="letter[.=$guess]/@guessed">
    <xsl:attribute name="guessed">
      true
    </xsl:attribute>
  </xsl:template>

  <xsl:template match="hangman[
      not(word/letter = $guess)]/display/*[
        @shown = 'false'][1]/@shown">
    <xsl:attribute name="shown">
      true
    </xsl:attribute>
  </xsl:template>

</xsl:stylesheet>
```

2.4. XQuery, using transform.xq

After this XSLT transformation, it is tempting to use the amazing transform.xq library presented by John Snelson at XML Prague 2012.

There are a number of ways to write a transformation with this library, and the most straightforward is to define it as a sequence of rules:

```
xquery version "3.0";

import module namespace tfm =
  "http://snelson.org.uk/functions/transform" at
  "transform.xq/transform.xq";

declare namespace f =
  "http://ns.dyomedea.com/functions/";

declare variable $guess := 'G';

declare function f:isAGuess(
  $node as node(),
  $guess as xs:string) as xs:boolean {
    $node/ancestor-or-self::hangman/word/letter =
      $guess
};

let $hangman :=
  <hangman status="in-progress" misses="Z">
    <word>
      <letter guessed="true">H</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
      <letter guessed="false">G</letter>
      <letter guessed="true">M</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
    </word>
    <display>
      <head shown="true"/>
      <body shown="true"/>
      <right_arm shown="true"/>
      <left_arm shown="true"/>
      <right_leg shown="false"/>
      <left_leg shown="false"/>
    </display>
  </hangman>

let $mode := tfm:mode((

(:
  Default rule: identity template for elements
:)
  tfm:rule('*', function($mode, $node) {
    element {xs:QName(name($node))} {
      $mode($node/(@*|node()))
    }
  }),

(:
  Default rule: identity template for attributes
:)
  tfm:rule('@*', function($mode, $node) {
    $node
  }),

(: Hangman root element :)

  tfm:rule('hangman', function($mode, $node) {
    let $updated-content := $mode($node/node())
    return
    <hangman
      status="{
       if (not($updated-content/self::word/
               letter[@guessed='false']))
        then 'won'
        else if (not($updated-content/
                 self::display/*[@shown='false']))
        then 'lost'
        else 'in-progress'}"
      misses = "{concat($node/@misses,
        if (f:isAGuess($node, $guess)) then ''
        else $guess)}"
    > {
      $updated-content
    } </hangman>
  }),

  (: Guesses :)
  tfm:rule('letter/@guessed',
    function($mode, $node) {
      attribute {xs:QName(name($node))} {
        if ($node/.. = $guess)
        then 'true'
        else $node
      }
  }),


  (: Display :)
  tfm:rule('display/*/@shown',
    function($mode, $node) {
      attribute {xs:QName(name($node))} {
        if (not(f:isAGuess($node, $guess)) and
            not($node/../preceding-sibling::*/
                @shown = 'false') )
        then 'true'
        else $node
      }
  }),

  ()
))

return $mode($hangman, ())
```

I won't go into the details, let's just note that the result of `tfm:mode()` is a function that is then used to perform a transformation and that `tfm:rule()` is another function which is used to associate patterns (defined as a subset of XSLT 2.0 match patterns) and actions defined as XQuery functions.

Note

transform.xq includes its own XSLT match patterns parsers to convert each pattern expressed as strings into a function performing pattern evaluation!

The principle of all these binding tools is to bind functions, classes or templates to specific nodes in a document. In the Python library, classes were associated to elements based on element names. In XSLT, templates are bound to nodes through their `@match` attributes. In XQuery 3.0, we need to find a trick to bind a match pattern, a function or any other property to actions defined as functions.The transform.xq library proposes two different ways to do this association. In this first example, this is done through this `tfm:rule` function, and we'll see how that works in one of our next examples. The second way is using standard MarkLogic extension functions to perform reflection, and this binding is done using annotations:

```
xquery version "3.0";

import module namespace tfm =
  "http://snelson.org.uk/functions/transform" at
  "transform.xq/transform.xq";

declare namespace f =
  "http://ns.dyomedea.com/functions/";
```

```
declare variable $guess := 'G';

declare function f:isAGuess(
  $node as node(),
  $guess as xs:string) as xs:boolean {
  $node/ancestor-or-self::hangman/word/letter =
    $guess
};

declare function f:copy(
  $node as node(),
  $content as item()*) as node()? {
  if($node instance of element()) then
    element {xs:QName(name($node))} { $content }
  else if ($node instance of attribute()) then
    attribute {xs:QName(name($node))} { $content }
  else $node
};

declare %tfm:rule("default","*",1)
function f:identity-elt($mode, $node){
  f:copy($node, $mode($node/(@*|node())))
};

declare %tfm:rule("default","@*",1)
function f:identity-att($mode, $node) {
  $node
};

declare %tfm:rule("default","hangman",2)
function f:hangman-elt($mode, $node) {
  let $updated-content := $mode($node/node())
  return
  <hangman
    status="{
     if(not($updated-content/self::word/letter
           [@guessed='false']))
      then 'won'
     else if(not($updated-content/self::display/*
           [@shown='false']))
      then 'lost'
     else 'in-progress'}"
     misses =
       "{concat($node/@misses, if(f:isAGuess(
          $node, $guess)) then '' else $guess)}"
  > {
    $updated-content
  } </hangman>
};

declare %tfm:rule("default","letter/@guessed",2)
function f:guessed-att($mode, $node) {
  f:copy($node,
    if ($node/.. = $guess)
      then 'true'
      else $node
  )
};

declare %tfm:rule("default","display/*/@shown",2)
function f:shown-att($mode, $node) {
  f:copy($node,
    if (not(f:isAGuess($node, $guess)) and
        not($node/../preceding-sibling::*/
            @shown = 'false') )
      then 'true'
      else $node
  )
};


let $hangman :=
  <hangman status="in-progress" misses="Z">
    <word>
      <letter guessed="true">H</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
      <letter guessed="false">G</letter>
      <letter guessed="true">M</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
    </word>
    <display>
      <head shown="true"/>
      <body shown="true"/>
      <right_arm shown="true"/>
      <left_arm shown="true"/>
      <right_leg shown="false"/>
      <left_leg shown="false"/>
    </display>
  </hangman>

let $mode := tfm:named-mode('default')

return $mode($hangman, ())
```

We have also taken advantage of this second version to define a `f:copy()` function which is more or less equivalent to `<xsl:copy>` but otherwise it is equivalent to the previous version.

2.5. XQuery, simple recursion

Transform.xq is really well thought out, and it includes an incredible number of bells and whistles, but it's worth seeing what we can do straight away in XQuery.

Before we jump into higher-order functions we can see what we can do with good old recursion, and in fact that's not that bad:

```
xquery version "3.0";

declare namespace f =
  "http://ns.dyomedea.com/functions/";

declare variable $guess := 'G';

declare function f:isAGuess(
  $node as node(),
  $guess as xs:string) as xs:boolean {
  $node/ancestor-or-self::hangman/word/letter =
    $guess
};

declare function f:copy(
  $node as node(),
  $content as item()*) as node()? {
  if ($node instance of element()) then
    element {xs:QName(name($node))} { $content }
  else if ($node instance of attribute()) then
    attribute {xs:QName(name($node))} { $content }
  else $node
};


declare function f:transform(
  $node as node()) as node()? {

  if ($node/self::hangman) then
    let $updated-content := $node/node()
                        ! f:transform(.)
    return
    <hangman
      status="{
       if (not($updated-content/self::word/
               letter[@guessed='false']))
         then 'won'
       else if (not($updated-content/self::display/
                *[@shown='false']))
         then 'lost'
       else 'in-progress'}"
      misses = "{concat($node/@misses,
                 if(f:isAGuess($node, $guess)) then
                   '' else $guess)}"
    > {
      $updated-content
    } </hangman>
  else if ($node instance of attribute(guessed)) then
    f:copy($node,
     if ($node/.. = $guess)
       then 'true'
       else $node
    )
  else if ($node instance of attribute(shown)) then
    f:copy($node,
     if (not(f:isAGuess($node, $guess)) and
         not($node/../preceding-sibling::*/
             @shown = 'false') )
      then 'true'
      else $node
    )
  else if ($node/self::*) then
    f:copy($node, $node/(@*|node()) !
           f:transform(.))
  else $node
};

let $hangman :=
  <hangman status="in-progress" misses="Z">
    <word>
      <letter guessed="false">H</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
      <letter guessed="false">G</letter>
      <letter guessed="true">M</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
    </word>
    <display>
      <head shown="true"/>
      <body shown="true"/>
      <right_arm shown="true"/>
      <left_arm shown="true"/>
      <right_leg shown="false"/>
      <left_leg shown="false"/>
    </display>
  </hangman>

return f:transform($hangman)
```

This new example is functionally equivalent to what we've done with transform.xq, and I find it surprisingly readable.

It could have been made more modular by using function calls for each alternative and we would them have been left with individual functions similar those defined in the second transform.xq implementation called from a central multi level `if/then/else` block.

2.6. XQuery using higher-order functions

The last exercise will be to implement our own simple transformation mechanism using higher-order functions.

Here again we need to associate several items in a simple structure. Maps would have been ideal but they've not made their way into XQuery 3.0. However we can use the same trick as transform.xq does and mimic a map with two fixed keys (k1 and k2):

```
xquery version "3.0";

declare namespace f =
  "http://ns.dyomedea.com/functions/";

declare function f:map-hack(
  $v1,
  $v2) as function(*) {
  function($k as xs:string) {
    switch($k)
      case 'k1' return $v1
      case 'k2' return $v2
      default return ()
  }
};

let $map := f:map-hack('foo', 'bar')

return ($map('k1'), $map('k2'))
```

This method will be used to associate match patterns defined as a functions and actions defined as functions as well as rules.

Where transform.xq has an elaborated mechanism to derive priority from match pattern selectivity similarly to XSLT, we can adopt a simpler priority system similarly to Schematron where the rules are evaluated in the order in which they appear in a sequence and the evaluation stops after the first matching rule.

With these principles, the implementation becomes:

```
xquery version "3.0";

declare namespace f =
  "http://ns.dyomedea.com/functions/";

declare variable $guess := 'W';

declare function f:isAGuess(
  $node as node(),
  $guess as xs:string) as xs:boolean {
  $node/ancestor-or-self::hangman/word/letter =
    $guess
};

declare function f:copy(
  $node as node(),
  $content as item()*) as node()? {
  if($node instance of element()) then
    element {xs:QName(name($node))} { $content }
  else if($node instance of attribute()) then
    attribute {xs:QName(name($node))} { $content }
  else $node
};

(: borrowed from transform.xq :)
declare function f:rule(
  $predicate as (function(node()) as xs:boolean),
  $action as
    (function(node(),
    function(*)) as node()?)) as function(*)
{
  function($k as xs:string) {
    switch($k)
      case 'predicate' return $predicate
      case 'action' return $action
      default return ()
  }
};

(: Some higher-order functions magic :)
declare function f:transform(
  $rules as function(*)*) as function(*) {
  function($node as node(),
           $transform as function(*)) as node()? {
    if (head($rules)('predicate')($node))
      then head($rules)('action')($node,
                                  $transform)
    else if (exists(tail($rules)))
      then f:transform(tail($rules))($node,
                                     $transform)
      else ()
  }
};

(: The transformation itself :)
let $t := f:transform((
  (: hangman element :)
  f:rule(
  function($node as node()) as xs:boolean {
    boolean($node/self::hangman)
  }, function(
        $node as node(),
        $transform as function(*)) as node() {
      let $updated-content := $node/node() !
                   $transform(., $transform)
      return
      <hangman
        status="{
         if (not($updated-content/self::word/
                 letter[@guessed='false']))
           then 'won'
         else if (not($updated-content/
                  self::display/*[@shown='false']))
           then 'lost'
         else 'in-progress'}"
        misses = "{concat($node/@misses,
                   if (f:isAGuess($node, $guess))
                     then '' else $guess)}"
      > {
        $updated-content
      } </hangman>
  }),

  (: @guessed attributes :)
  f:rule(
  function($node as node()) as xs:boolean {
    $node instance of attribute(guessed)
  }, function($node as node(),
            $transform as function(*)) as node()
  {
    f:copy($node, if ($node/.. = $guess)
          then 'true'
          else $node)
  }),

  (: @shown attributes :)
  f:rule(
  function($node as node()) as xs:boolean {
    $node instance of attribute(shown)
  }, function($node as node(),
            $transform as function(*)) as node()
  {
    f:copy($node,
      if (not(f:isAGuess($node, $guess)) and
          not($node/../preceding-sibling::*/@shown
            = 'false')) then
        'true'
      else
        $node)
  }),

  (: Any other attribute :)
  f:rule(
    function($node as node()) as xs:boolean {
      $node instance of attribute()
    },
    function($node as node(),
             $transform as function(*)) as node() {
      f:copy($node, $node)
    }),

  (: Anything else :)
  f:rule(
  function($node as node()) as xs:boolean {
    true()
  },
  function($node as node(),
           $transform as function(*)) as node() {
    f:copy($node, $node/(@*|node()) !
           $transform(., $transform))
  })
))

let $hangman :=
  <hangman status="in-progress" misses="Z">
    <word>
      <letter guessed="true">H</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
      <letter guessed="false">G</letter>
      <letter guessed="true">M</letter>
      <letter guessed="true">A</letter>
      <letter guessed="true">N</letter>
    </word>
    <display>
      <head shown="true"/>
      <body shown="true"/>
      <right_arm shown="true"/>
      <left_arm shown="true"/>
      <right_leg shown="true"/>
      <left_leg shown="false"/>
    </display>
</hangman>


return $t($hangman, $t)
```

3. Conclusion

Even if, unlike XSLT, XQuery doesn't have any feature to make it natively data-driven you can easily write your own mechanisms to bind data into functions.

These mechanisms can be generic such as transform.xq but it is also easy to write you own specialized one to perform specific tasks.

4. Acknowledgments

Many thanks to Tony Graham, Patrick Durusau and my three anonymous reviewers for their reviews, comments, questions and suggestions and to John Snelson for his very inspiring transform.xq.

XML Processing with Scala and yaidom

Chris de Vreeze

EBPI

<chris.de.vreeze@ebpi.nl>

Abstract

Yaidom is a uniform XML query API, written in the Scala programming language and leveraging its Collections API. Moreover, yaidom provides several specific-purpose DOM-like tree implementations offering this XML query API.

In this paper the yaidom library is introduced, using examples from XBRL (eXtensible Business Reporting Language).

Keywords: Scala, XML, yaidom

1. Introduction

This article introduces the open source *yaidom* XML query library[1], using examples in the domain of XBRL (eXtensible Business Reporting Language)[2].

It is assumed that the reader has some experience with XML processing in Java (e.g. JAXP) or another OO programming language (such as Scala or C#).

XSLT, XQuery and XPath are standard XML transformation/query languages, yet in this article yaidom (with Scala) is introduced as an *alternative* approach to in-memory XML querying/transformation, leveraging the Scala programming language. Still, yaidom can also be used together with standard languages such as XQuery, for example when using an XML database.

As shown in the table of contents, after introducing Scala, Scala Collections and yaidom, a brief introduction to XBRL follows. XBRL is an XML-based business reporting standard. Business reports in XBRL format are called *XBRL instances*. XBRL instances must obey many requirements, in order for them to be considered valid. After the brief XBRL introduction, the remainder of this paper shows how many of these rules can be expressed using yaidom and Scala. It will be shown that using Scala and yaidom instead of standard XML query and transformation languages actually makes expressing these rules relatively easy.

There are several other papers about XML processing in Scala, mostly about Scala's own standard XML libary. For example, [1] contains many well-chosen examples that show how to process XML in Scala. Moreover, it first introduces Scala, assuming some familiarity with XQuery on the part of the reader. To get an appreciation of XML processing using Scala in general, and of XML processing using Scala and yaidom in particular, it makes sense to read both papers, starting with [1].

2. Brief introduction to Scala and Scala Collections

The Scala programming language is the most popular alternative to the Java language on the Java virtual machine. It is *object-oriented* (more so than Java) and also *functional*, in that functions are first-class objects. It is *statically typed*, but it feels like a dynamically typed language, because of features such as type inference.

Scala is a *safe* and *expressive* language, typically leading to good productivity and low bug counts in skilled disciplined teams. Its rich *Collections API*, its strong support for *immutable* data structures, and its focus on *expressions* rather than statements enables programmers to work at a higher level of abstraction in Scala than in Java.

The Collections API of a programming language (which in the case of Scala and Java is a part of the standard library of the language, not of the core language) often says a lot about the expressive power of that language. Below follows some Scala code that manipulates collections, to illustrate Scala's expressiveness.

Consider a book store and some queries about books (using sample data from the Stanford University online course *Introduction to Databases*). The Scala code is as follows:

```
case class Author(
  firstName: String, lastName: String)

case class Book(
  isbn: String,
  title: String,
  authors: List[Author],
  price: Int)
```

[1] https://github.com/dvreeze/yaidom
[2] https://www.xbrl.org

```scala
val someBooks = List(
  Book(
    "ISBN-0-13-713526-2",
    "A First Course in Database Systems",
    List(
      Author("Jeffrey", "Ullman"),
      Author("Jennifer", "Widom")),
    85),
  Book(
    "ISBN-0-13-815504-6",
    "Database Systems: The Complete Book",
    List(
      Author("Hector", "Garcia-Molina"),
      Author("Jeffrey", "Ullman"),
      Author("Jennifer", "Widom")),
    100),
  Book(
    "ISBN-0-11-222222-3",
    "Hector and Jeff's Database Hints",
    List(
      Author("Jeffrey", "Ullman"),
      Author("Hector", "Garcia-Molina")),
    50),
  Book(
    "ISBN-9-88-777777-6",
    "Jennifer's Economical Database Hints",
    List(Author("Jennifer", "Widom")),
    25)
)

// Return all books that cost no more than 50
// dollars (i.e., the last 2 books)

val cheapBooks =
  someBooks.filter(book => book.price <= 50)

// Return all books having Jeffrey Ullman as an
// author (i.e., the first 3 books)

def hasAuthor(book: Book, authorLastName: String):
  Boolean = {

  book.authors.exists(
    author => author.lastName == authorLastName)
}

val booksByUllman =
  someBooks.filter(book =>
    hasAuthor(book, "Ullman"))

// Return all book authors, without duplicates

val allAuthors =
  someBooks.flatMap(book => book.authors).distinct

// Return all titles of books having Jeffrey
// Ullman as an author

val bookTitlesByUllman =
  someBooks.filter(bk => hasAuthor(bk, "Ullman")).
    map(bk => bk.title)
```

Note how the queries in prose naturally map to their counterparts in Scala code, using a small vocabulary of *higher-order functions* such as `map`, `flatMap` and `filter`. The code shows the *"what"* more than the *"how"*. In that respect, the Scala code is more like XQuery than Java (especially than Java before version 8). In a sense, *the Scala core language along with its Collections API* form a *universal query (and transformation) language*. Of course, Scala is a lot more than that, but for the purposes of this article this is a fitting description.

3. Brief introduction to yaidom

The *yaidom* library can be used for querying and transforming XML in Scala. It interoperates very well with the Scala Collections API.

It was mentioned above that Scala and its Collections API can be used as a universal query and transformation language. The yaidom library offers an *XML element query API* that turns elements into Scala collections of elements. So yaidom can be said to turn a universal query and transformation language into an *XML querying and transformation language*. In other words, *Scala + its Collections API + yaidom* can be used as an "XML querying/transformation stack". Below it will become clear that yaidom can plug in different "XML backends", thus making the "XML stack" very powerful.

Using the bookstore example above, some simple yaidom XML queries are shown below. The XML is as follows:

```xml
// The book store XML

<Bookstore>
  <Book ISBN="ISBN-0-13-713526-2"
        Price="85" Edition="3rd">
    <Title>A First Course in Database Systems
    </Title>
    <Authors>
      <Author>
        <First_Name>Jeffrey</First_Name>
        <Last_Name>Ullman</Last_Name>
      </Author>
      <Author>
        <First_Name>Jennifer</First_Name>
        <Last_Name>Widom</Last_Name>
      </Author>
    </Authors>
  </Book>
  <Book ISBN="ISBN-0-13-815504-6" Price="100">
    <Title>Database Systems: The Complete Book
    </Title>
    <Authors>
      <Author>
        <First_Name>Hector</First_Name>
        <Last_Name>Garcia-Molina
        </Last_Name>
      </Author>
      <Author>
        <First_Name>Jeffrey</First_Name>
        <Last_Name>Ullman</Last_Name>
      </Author>
      <Author>
        <First_Name>Jennifer</First_Name>
        <Last_Name>Widom</Last_Name>
      </Author>
    </Authors>
    <Remark>
Buy this book bundled with "A First Course"
    </Remark>
  </Book>
  <Book ISBN="ISBN-0-11-222222-3" Price="50">
    <Title>Hector and Jeff's Database Hints
    </Title>
    <Authors>
      <Author>
        <First_Name>Jeffrey</First_Name>
        <Last_Name>Ullman</Last_Name>
      </Author>
      <Author>
        <First_Name>Hector</First_Name>
        <Last_Name>Garcia-Molina
        </Last_Name>
      </Author>
    </Authors>
    <Remark>
An indispensable companion to your textbook
    </Remark>
  </Book>
  <Book ISBN="ISBN-9-88-777777-6" Price="25">
    <Title>
    Jennifer's Economical Database Hints
    </Title>
    <Authors>
      <Author>
        <First_Name>Jennifer</First_Name>
        <Last_Name>Widom</Last_Name>
      </Author>
    </Authors>
  </Book>
</Bookstore>
```

Below follow the yaidom XML queries corresponding to the (non-XML) queries above. Written rather verbosely, they are as follows:

```
// Assume a root element called bookstore.

val someBooks =
  bookstore.filterChildElems(bk =>
    bk.localName == "Book")

// Return all books that cost no more than 50
// dollars (i.e., the last 2 books)

val cheapBooks =
  someBooks.filter(book =>
    book.attribute(EName("Price")).toInt <= 50)

// Return all books having Jeffrey Ullman as an
// author (i.e., the first 3 books)

def hasAuthor(
  book: simple.Elem, authorLastName: String):
    Boolean = {

  require(book.localName == "Book")

  book.findElem(e =>
    e.localName == "Author" &&
      e.getChildElem(che =>
        che.localName == "Last_Name").text ==
          authorLastName).isDefined
}

val booksByUllman =
  someBooks.filter(bk => hasAuthor(bk, "Ullman"))

// Return all book author elements (with
// duplicates, this time)

val allAuthors =
  someBooks.flatMap(
    book => book.filterElems(e =>
      e.localName == "Author"))

// Return all titles of books having Jeffrey Ullman
// as an author

val bookTitlesByUllman =
  someBooks.filter(book =>
    hasAuthor(book, "Ullman")).
      map(book => book.getChildElem(e =>
        e.localName == "Title").text)
```

Above, the `EName` type stands for "expanded name". It corresponds to Java's `javax.lang.namespace.QName`, except that it does not retain the prefix, if any.

For clarity, these XML queries were written more verbosely than needed. Even when writing these queries in a less verbose way than has been done above, there would still be some verbosity related to XML handling. This is intentional: yaidom is a *precise* XML query API.

For example, yaidom does not abstract away the distinction between elements and attributes, or between names with a namespace and those without any namespace. Despite the syntax dedicated by yaidom to XML node manipulation, the yaidom query examples are not *that* much more verbose than the non-XML query examples presented earlier. Compared to ad-hoc XML querying in Java (using JAXP), however, ad-hoc XML querying in Scala using yaidom is *much* less verbose. Even when invoking XPath queries (returning node sets) from Java code, the processing of the resulting node sets would add "syntactic clutter" that the use of Scala with yaidom could have prevented.

Why use yaidom and not Scala's own XML library? As will become apparent in this article, yaidom has very precise support for *XML namespaces*, more so than Scala XML. Using the yaidom API it is always clear if queries are namespace-aware. If it is intentional to query for elements that have specific local names, regardless of the namespace, then yaidom forces the user to be explicit about that. Precise namespace support in yaidom even goes as far as the ability to express a simple *theory* of XML namespaces (relating namespace declarations, in-scope namespaces, qualified names and expanded names) in yaidom code itself, outside of any particular XML tree!

There are more reasons why yaidom may be preferable to Scala's own XML library. For example, yaidom has a precise *uniform XML query API* that is offered by *multiple "XML backend"* implementations. Not only does yaidom offer own native DOM-like tree (XML backend) implementations with different strengths and weaknesses, but it is also possible to wrap existing XML library tree implementations (DOM, JDOM, XOM, Saxon etc.) in yaidom, offering the same yaidom query API. For example, yaidom wrappers around Saxon-EE `NodeInfo` trees offer the best of Saxon and the *Scala-yaidom* combination: on the one hand the completeness and schema-type-awareness of Saxon-EE, and on the other hand a "Scala Collections API querying experience", using yaidom as the natural bridge between Saxon "nodes" and Scala collections processing. Unlike yaidom, the Scala XML library does not offer multiple tree implementations backing the same query API.

This extensibility of yaidom goes even further than specific XML backends. It is also possible to extend yaidom for custom "XML dialects" (or "vocabularies"). This will be explained and shown later in this paper. Arguably this could be the best reason to prefer yaidom to Scala's own XML library.

Why not just use *standards* such as XSLT or XQuery? XBRL processing is a good example where yaidom shines, as will become clear below. After all, XBRL is a lot more than "just" XML, so XBRL processing is a lot more than just XML processing. Performing most or all of this processing in Scala using yaidom offers the following advantages:

- No Scala syntax is spent on *processing sequences* (or node sets) resulting from XPath/XQuery evaluation
- In a programming language (such as Scala) it is quite natural and easy to store *intermediate results* in variables (unlike XPath)
- As a rich (functional) *OO* programming language, Scala has a lot of *expressive power*, which makes it easy to build layered models on top of DOM-like element trees (as will be shown below)
- Yaidom leverages the *Scala Collections API*, which enables the user to achieve a lot using only a small vocabulary
- There is a large *ecosystem* around Scala (and Java), offering many high quality libraries.
- Yaidom offers (and enables) element implementations optimized for fast querying (although no benchmarks are provided in this paper)
- For programmers on the JVM, Scala and yaidom have more familiar *semantics* than XPath and the XQuery and XPath Data Model (XDM):
  - In XDM there is no difference between an item (node or atomic value) and a singleton sequence containing that item
  - Sequences in XDM cannot be nested, so are always flattened
  - In Scala (as in Java), "equality" is expected to be an equivalence relation (unlike the general comparison equality operator in XPath, which is not transitive)

If desired, yaidom can be used with XQuery when using an XML(-enabled) database, where XQuery joins and filters database XML data into "raw result sets", which are further processed using yaidom queries. Still, it makes sense to keep the number of boundaries between XQuery and yaidom/Scala relatively low, for each such boundary has some (syntactic and semantic) costs. In summary, the more some XML processing task can benefit from the use of Scala, the more attractive the use of yaidom becomes.

4. Brief introduction to the XBRL examples

So far, this article has introduced Scala and yaidom, using only trivial examples. In the remainder of this article, yaidom examples in the domain of *XBRL* are used. First, this section gives a very brief introduction to XBRL.

XBRL (eXtensible Business Reporting Language) is a standard for business reporting. Many (but not all)

XBRL reports are financial statements. XBRL reports ("XBRL instances") are XML documents, following a specified structure.

Suppose we want to report that for a given organization ("CIK") the average number of employees in 2003 was 220, and that the corresponding numbers for 2004 and 2005 were 240 and 250, respectively. More precisely, *concept* `gaap:AverageNumberEmployees` (described by the so-called US-GAAP XBRL *taxonomy*) has the *value* `220` in the given *context* (organization "CIK", year 2003). Then we can report the 3 *facts* above in XBRL format as follows:

```
<xbrl xmlns="http://www.xbrl.org/2003/instance"
      xmlns:gaap="http://xasb.org/gaap">

   <context id="D-2003">
      <entity>
         <identifier
           scheme="http://www.sec.gov/CIK">
         1234567890
         </identifier>
      </entity>
      <period>
         <startDate>2003-01-01</startDate>
         <endDate>2003-12-31</endDate>
      </period>
   </context>

   <context id="D-2004">
      <entity>
         <identifier
           scheme="http://www.sec.gov/CIK">
           1234567890
         </identifier>
      </entity>
      <period>
         <startDate>2004-01-01</startDate>
         <endDate>2004-12-31</endDate>
      </period>
   </context>

   <context id="D-2005">
      <entity>
         <identifier
           scheme="http://www.sec.gov/CIK">
           1234567890
         </identifier>
      </entity>
      <period>
         <startDate>2005-01-01</startDate>
         <endDate>2005-12-31</endDate>
      </period>
   </context>

   <unit id="U-Pure">
     <measure>pure</measure>
   </unit>

   <gaap:AverageNumberEmployees
     contextRef="D-2003"
     unitRef="U-Pure"
     decimals="INF">220
   </gaap:AverageNumberEmployees>
   <gaap:AverageNumberEmployees
     contextRef="D-2004"
     unitRef="U-Pure"
     decimals="INF">240
   </gaap:AverageNumberEmployees>
   <gaap:AverageNumberEmployees
     contextRef="D-2005"
     unitRef="U-Pure"
     decimals="INF">250
   </gaap:AverageNumberEmployees>

</xbrl>
```

This example comes from a non-trivial sample XBRL instance written by Charles Hoffman, also known as "the father of XBRL".

There are many requirements that have to be met in order for an XBRL instance to be valid. The XBRL Core specification (as well as other XBRL specifications) describes many of these requirements. There are also many common best practices that have been formalized as complementary rules. For example, the International FRIS Standard places additional constraints on XBRL instances. [2]

Most of the remainder of this article will show how many of those FRIS rules can be written naturally as Scala expressions using yaidom. Yaidom is in no way married to XBRL, but XBRL validations are good XML processing examples where Scala and yaidom really shine.

5. Simple yaidom query examples

The XBRL snippet above is part of this sample instance[1]. In this section, some simple yaidom XML queries are performed on the XBRL instance.

Before showing these queries on this XBRL instance, it should be noted that knowing only 3 yaidom query API methods to some extent means knowing them all. These 3 methods are `filterChildElems`, `filterElems` and `filterElemsOrSelf`. They all filter elements, based on the passed element predicate function. The difference is that they filter *child* elements, *descendant* elements, and *descendant-or-self* elements, respectively. The word "descendant" is left out from the method names.

It should also be noted that methods `filterChildElems` and `filterElemsOrSelf` have shorthands `\` and `\\`, respectively. Method `attributeOption` has shorthand `\@`. Moreover, some element predicate functions have names, such as `withLocalName` and `withEName`.

[1] http://www.xbrlsite.com/examples/comprehensiveexample/2008-04-18/sample-Instance-Proof.xml

Some yaidom queries on the sample XBRL instance are as follows:

```
// Let's first parse the XBRL instance document

val docParser = DocumentParserUsingSax.newInstance

val doc = docParser.parse(sampleXbrlInstanceFile)

// Check all gaap:AverageNumberEmployees facts
// have unit U-Pure.

val xmlNs = "http://www.w3.org/XML/1998/namespace"
val xbrliNs = "http://www.xbrl.org/2003/instance"
val gaapNs = "http://xasb.org/gaap"

val avgNumEmployeesFacts =
  doc.documentElement.filterChildElems(
    withEName(gaapNs, "AverageNumberEmployees"))

println(avgNumEmployeesFacts.size) // prints 7

val onlyUPure =
  avgNumEmployeesFacts.forall(fact =>
    fact.attributeOption(EName("unitRef")) ==
      Some("U-Pure"))
println(onlyUPure) // prints true

// Check the unit itself, minding the default
// namespace

val uPureUnit =
  doc.documentElement.getChildElem(e =>
    e.resolvedName == EName(xbrliNs, "unit") &&
    (e \@ EName("id")) == Some("U-Pure"))

println(
  uPureUnit.getChildElem(
    withEName(xbrliNs, "measure")).text)
// prints "pure"

// Now we get the measure element text, as QName,
// resolving it to an EName (expanded name)
println(
  uPureUnit.getChildElem(
    withEName(xbrliNs, "measure")).
      textAsResolvedQName)
// prints EName(xbrliNs, "pure")

// Knowing the units are the same, the
// gaap:AverageNumberEmployees facts are
// uniquely identified by contexts.

val avgNumEmployeesFactsByContext:
  Map[String, simple.Elem] =
  avgNumEmployeesFacts.groupBy(_.attribute(
    EName("contextRef"))).
      mapValues(_.head)

println(avgNumEmployeesFactsByContext.keySet)
// prints the set:
// "D-2003", "D-2004", "D-2005", "D-2007-BS1",
// "D-2007-BS2", "D-2006", "D-2007"

println(
  avgNumEmployeesFactsByContext("D-2003").text)
// prints 220
```

The uniform query API of yaidom consists of several query API traits. They are like LEGO blocks, that can easily be combined. Yaidom (native and wrapper) element tree implementations all mix in some or most of these query API traits. The example queries above are not bound to any particular element implementation, but use a common query API trait, namely `ScopedElemApi`, which is itself a combination of query API traits. This trait offers methods like `filterElemsOrSelf`, `filterChildElems` (from trait `ElemApi`), as well as methods to get text content, qualified names, expanded names, attributes etc. In other words, it offers a query API abstraction that is valid for almost all element implementations.

The query API traits themselves are not visible in normal yaidom client code. They are relevant for creators of custom yaidom element implementations, for example wrappers around elements offered by existing XML libraries. Yaidom users that do not extend yaidom may still want to know which query API traits are offered by some XML tree implementation, of course.

Sometimes we want to use methods that are only offered by specific element implementations, and not by any query API traits. The default native yaidom element implementation is `simple.Elem`. It knows about elements and text content (as per the mixed-in `ScopedElemApi` trait), but it also knows about comments, processing instructions and CDATA sections (if passed by the XML parser). For example:

```
println(doc.comments.map(_.text.trim).mkString)
// prints
// "Created by Charles Hoffman, CPA, 2008-03-27"

val contexts =
  doc.documentElement.filterChildElems(
    withEName(xbrliNs, "context"))

println(contexts forall (e =>
  !e.commentChildren.isEmpty))
// prints true: all contexts have comments

// Being lazy, and ignoring the namespace here
val facts =
  doc.documentElement.filterChildElems(
    withLocalName(
      "ManagementDiscussionAndAnalysisTextBlock"))

println(
  facts.flatMap(e => e.textChildren.filter(
    _.isCData)).size >= 1)
// prints true
```

6. Namespace examples

Yaidom has very precise namespace support. Like the article [3], yaidom distinguishes qualified names from expanded names, and namespace declarations from in-scope namespaces. Their yaidom counterparts are immutable classes `QName`, `EName`, `Declarations` and `Scope`. Having these 4 distinct concepts, their relationships can be expressed very precisely, even in yaidom code, and even outside of the context of any particular XML tree.

In the example XBRL instance above, all namespace declarations are in the root element, and therefore all descendant-or-self elements have the same in-scope namespaces. In code:

```
val rootScope = doc.documentElement.scope

val sameScopeEverywhere =
  doc.documentElement.findAllElemsOrSelf.forall(
    e => e.scope == rootScope)

println(sameScopeEverywhere) // prints true
```

Let's consider the first FRIS rule taken from [2], expressed in yaidom. Rule 2.1.5 states that some commonly used namespaces should use their "preferred" namespace prefixes in XBRL instances. The rule can be expressed in yaidom as follows:

```
val standardScope = Scope.from(
  "xbrli" -> "http://www.xbrl.org/2003/instance",
  "xlink" -> "http://www.w3.org/1999/xlink",
  "link" -> "http://www.xbrl.org/2003/linkbase",
  "xsi" ->
    "http://www.w3.org/2001/XMLSchema-instance",
  "iso4217" -> "http://www.xbrl.org/2003/iso4217")

val standardPrefixes = standardScope.keySet
val standardNamespaceUris =
  standardScope.inverse.keySet

// Naive implementation: expects only namespace
// declarations in root element

def usesExpectedNamespacePrefixes(
  xbrlInstance: simple.Elem): Boolean = {

  val rootScope = xbrlInstance.scope
  require(
    xbrlInstance.findAllElemsOrSelf.forall(
      e => e.scope == rootScope))

  val subscope =
    xbrlInstance.scope.withoutDefaultNamespace
      filter {
        case (pref, ns) =>
          standardPrefixes.contains(pref) ||
            standardNamespaceUris.contains(ns)
      }
  subscope.subScopeOf(standardScope)
}
```

Above, there is no useful error reporting, but that is easy to add, because the implementation is entirely in the rich Scala programming language. In prose, method `usesExpectedNamespacePrefixes` checks that if some of the 5 namespace prefixes above are used, that they all map to the expected namespace URIs. The method also checks the other side: if some of the namespace URIs are in-scope, then the corresponding namespace prefixes are the expected ones, with the exception that they may be the default namespace.

The example above illustrates yaidom's precise support for namespaces in the uniform query API, and therefore offered by diverse element tree implementations. Yet the namespace support goes further than that. As article [3] makes clear, namespaces are not only used in element and attribute names, but can also be used in text content and attribute values.

FRIS rule 2.1.7 must take namespaces in text content and attribute values into account, because it states that XBRL instances should not have any unused namespace declarations. Yet how do we detect the use of namespaces in text content or attribute values? We know this from the XML schema(s) describing XBRL instances. For example, the `xbrli:measure` element has type `xs:QName`. So the text content of an `xbrli:measure` should be interpreted as an expanded name. The namespace of that expanded name is therefore one of the namespaces used in the XBRL instance.

Yaidom makes it possible to code a `DocumentENameExtractor` strategy, holding information about ENames and therefore namespaces occurring in text content or attribute values. So, looking at the XML schema(s), we can easily code such a strategy ourselves (yaidom itself has no XML Schema awareness). Then, using method `NamespaceUtils.findAllNamespaces`, all namespaces used in the XBRL instance can be found.

Method `NamespaceUtils.findAllNamespaces` does not work on the default "simple" elements, however, because simple elements do not know their ancestry. For this purpose, yaidom offers so-called "indexed" elements, that do know their ancestry. Like simple elements, indexed elements are immutable, because they are just wrappers around a root as simple element along with an "index" into that element tree. The indexed and simple elements also share most of the query API, in particular the `ScopedElemApi` query API trait.

Let's now implement FRIS rule 2.1.7, but only for the sample XBRL instance:

```scala
val xbrliDocumentENameExtractor:
  DocumentENameExtractor = {
  // Not complete, but suffices for this example!

  new DocumentENameExtractor {

    def findElemTextENameExtractor(
      elem: indexed.Elem):
        Option[TextENameExtractor] =

      elem.resolvedName match {
        case EName(Some(xbrliNs), "measure")
          if elem.path.containsName(
            EName(xbrliNs, "unit")) =>

          Some(SimpleTextENameExtractor)
        case EName(
          Some(xbrldiNs), "explicitMember") =>
          Some(SimpleTextENameExtractor)
        case _ => None
    }

    def findAttributeValueENameExtractor(
      elem: indexed.Elem, attrEName: EName):
        Option[TextENameExtractor] =

      elem.resolvedName match {
        case EName(
          Some(xbrldiNs), "explicitMember")
          if attrEName == EName("dimension") =>
            Some(SimpleTextENameExtractor)
          case _ => None
      }
    }
}

val indexedDoc = indexed.Document(doc)

val namespaceUrisDeclared =
  indexedDoc.documentElement.scope.inverse.keySet

import NamespaceUtils._

// Check that the used namespaces are almost
// exactly those declared in the root element
// (approximately rule 2.1.7)

val companyNs = "http://www.example.com/company"

val usedNamespaces =
  findAllNamespaces(
    indexedDoc.documentElement,
    xbrliDocumentENameExtractor).diff(Set(xmlNs))

// The "company namespace" is an unused namespace
// in our sample XBRL instance
require(usedNamespaces == namespaceUrisDeclared.
  diff(Set(companyNs)))
```

Although yaidom itself has no XML Schema awareness, yaidom can still be useful in a context where schema-awareness is needed. For example, Saxon-EE NodeInfo objects can be wrapped as yaidom trees, thus getting the best of Scala Collections processing and Saxon-EE XML and XML Schema support.

Let's now remove the unused namespaces (the "company" namespace in this example), and compare the result with the original XBRL instance. Yet how do we compare two XML trees (as simple elements) for equality? In order to do so, note that namespace prefixes are irrelevant to equality comparisons, but namespace URIs do count. (Be careful with prefixes in text content and attribute values!) Yaidom offers an XML element implementation in which namespace prefixes do not occur. These elements are called "resolved" elements. They share much of the same query API with simple and indexed elements, but not all of it. After all, resolved elements do not know about namespace prefixes, so they do not know about qualified names. Therefore they do not mix in the `ScopedElemApi` trait, but they do mix in traits like `ElemApi` and `HasTextApi`, that is, all traits extended by `ScopedElemApi` that do not know about qualified names. Hence, resolved elements still have much of the yaidom query API in common with simple and indexed elements.

The following code strips unused namespaces, and shows that the result is the same, when comparing the trees as resolved elements.

```
val editedRootElem =
  stripUnusedNamespaces(
    indexedDoc.documentElement,
    xbrliDocumentENameExtractor)

val areEqual =
  resolved.Elem(
    indexedDoc.document.documentElement) ==
    resolved.Elem(editedRootElem)

println(areEqual) // prints true
```

7. Extending yaidom for custom XML dialects

Above, all XBRL instance processing was coded as normal XML processing, mostly using yaidom simple and indexed elements. That's not very convenient. It would be nice if we could talk about contexts, units, facts etc., instead of just XML elements that happen to be contexts, units, facts, etc. In general, it would be nice if yaidom would make it easy to support custom XML dialects. That is indeed the case. We already knew that yaidom is *extensible*, in that new element implementations offering the same yaidom query API can easily be added. Yet, what's more, yaidom also facilitates a "yaidom querying experience" for *custom XML dialects*, such as XBRL instances (or DocBook files, or Maven POM files, or any other XML dialect described by schemas).

To that end, yaidom offers the `SubtypeAwareElemApi` query API trait. Whereas the `ElemApi` trait offers querying for child/descendant/descendant-or-self elements, trait `SubtypeAwareElemApi` extends this to class hierarchies (for XML dialects), offering querying for child/descendant/descendant-or-self elements of specific sub-types of the root class of the class hierarchy.

In this XBRL instance class hierarchy[1] we can see this action. Each part of an XBRL instance is of type `XbrliElem` or a sub-type. Common sub-types are those for contexts, units, item facts, tuple facts, and, of course, XBRL instances themselves. Super-type `XbrliElem` mixes in traits `ScopedElemApi` and `SubtypeAwareElemApi`. Trait `ScopedElemApi` offers the most common yaidom element query API, as we know, and trait `SubtypeAwareElemApi` makes it easy to query for elements of specific types, with little boilerplate. The latter is used internally in the code of the `XbrliElem` class hierarchy, but can also be used in client code, if need be.

[1] http://dvreeze.github.io/code-snippets/xbrl-instances.html

For the remaining FRIS validations in this article, we will use the `XbrliElem` class hierarchy.

Consider FRIS rule 2.1.10. It states that there is a specific expected order of the child elements of the root element. One way to code that is as follows:

```
// Assume xbrlInstance variable of type XbrlInstance

val remainingChildElems =
  xbrlInstance.findAllChildElems dropWhile {
    case e: SchemaRef => true
    case e => false
  } dropWhile {
    case e: LinkbaseRef => true
    case e => false
  } dropWhile {
    case e: RoleRef => true
    case e => false
  } dropWhile {
    case e: ArcroleRef => true
    case e => false
  } dropWhile {
    case e: XbrliContext => true
    case e => false
  } dropWhile {
    case e: XbrliUnit => true
    case e => false
  } dropWhile {
    case e: Fact => true
    case e => false
  } dropWhile {
    case e: FootnoteLink => true
    case e => false
  }

require(remainingChildElems.isEmpty)
```

Now consider FRIS rule 2.4.2 stating that all contexts must be used. It is also checked that all context references indeed refer to existing contexts. Note in this case how friendly the XBRL instance model is compared to raw XML elements:

```
val contextIds =
  xbrlInstance.allContextsById.keySet

val usedContextIds = xbrlInstance.findAllItems.
  map(_.contextRef).toSet

require(usedContextIds.subsetOf(contextIds))

// Oops, some contexts are not used, namely
//I-2004, D-2007-LI-ALL and I-2003
println(contextIds.diff(usedContextIds))
```

The next rule is more complex. FRIS rule 2.4.1 states that S-equal contexts should not occur. S-equality ("structural equality") is defined in the Core XBRL specification. A good implementation of S-equality requires type information. Therefore Saxon-EE backed yaidom wrappers would be a good choice. A very naive approximation is given below:

```
def transformContextForSEqualityComparison(
  context: XbrliContext): resolved.Elem = {

  // Ignoring "normalization" of dates and
  // QNames, as well as dimension order etc.
  val elem = context.indexedElem.elem.copy(
    attributes = Vector())
  resolved.Elem(elem).
    coalesceAndNormalizeAllText.
      removeAllInterElementWhitespace
}
```

Then rule 2.4.1 applied to our XBRL instance is as follows:

```
val contextsBySEqualityGroup =
  xbrlInstance.allContexts.groupBy(e =>
    transformContextForSEqualityComparison(e))

require(contextsBySEqualityGroup.size ==
  xbrlInstance.allContexts.size)
```

As we can see, the more complex the rules, the more we profit from the fact that all code is Scala code, and that there is no needed effort in bridging between Scala and XSLT, for example. The Scala language, its Collections API, and yaidom form a powerful combination.

Finally, consider FRIS rule 2.8.3, stating that concepts are either top-level or nested in tuples, but not both. Using the XBRL instance model, the code is simple:

```
val topLevelConceptNames =
  xbrlInstance.allTopLevelFactsByEName.keySet

val nestedConceptNames =
  xbrlInstance.allTopLevelTuples.
    flatMap(_.findAllFacts).
      map(_.resolvedName).toSet

require(topLevelConceptNames.intersect(
  nestedConceptNames).isEmpty)
```

8. Conclusion

In this article, the *yaidom Scala XML query library* was introduced. We used examples from XBRL. It turned out that Scala, its Collections API, and the yaidom library form a powerful precise XML processing stack. This stack is even more powerful when using custom mature yaidom backends as Saxon-EE. It also turned out that yaidom makes it easy to support custom XML dialects (such as XBRL instances), offering more type-safety and leading to less boilerplate. The *extensibility* of yaidom (in more than one way) is one of its strengths, along with its precise *namespace support* and *uniform precise element query API* (offered by multiple *XML backends*).

The FRIS rule examples show that a programming language like Scala is a natural fit for implementing those rules. Had we used XSLT or XQuery instead, how would we easily have found unused namespaces, for example? Moreover, how would we have supported custom XML dialects in the same way that yaidom facilitates such support?

The examples only used XBRL instances. These instances are described by XBRL taxonomies. Such taxonomies have to obey many rules as well. Taxonomies typically span many files, and their validation is usually much more complex than instance validation. The advantages of using a Scala yaidom XML stack would even be greater than for XBRL instances.

As a concluding remark, yaidom is used in production code developed at EBPI[1]. Its usage in several projects has certainly helped it mature. I want to thank my colleagues Jan-Paul van der Velden, Andrea Desole, Johan Walters and Nicholas Evans for their valuable feedback on earlier versions of yaidom.

Bibliography

[1] *XML Processing in Scala*. Dino Fancellu and William Narmontas. XML London 2014. doi:10.14337/XMLLondon14.Narmontas01

[2] *Financial Reporting Instance Standards 1.0*. XBRL Consortium. http://www.xbrl.org/technical/guidance/FRIS-PWD-2004-11-14.htm

[3] *Understanding Namespaces*. Evan Lenz. http://www.lenzconsulting.com/namespaces/

[1] EBPI http://www.ebpi.nl

Lizard

A Linked Data Publishing Platform

Andy Seaborne

Epimorphics

Abstract

Publishing data means delivering it to a wide and changing variety of data consumers. Instead of defined, agreed use by fixed applications, data is used in ways that the publisher will find hard to predict as users find ingenious ways to use and combine data. Data services don't do "9 to 5" and publishing of the data must aim for high levels of available service.

Yet the operation of data services will need to be resilient to operational needs as well as updates. By looking at some real data publishing services, we will see that while hardware failures happen, the main causes of service disruption are operational.

This paper describes a new, open source, RDF database that addresses operation needs using fault tolerance systems techniques to provide a scalable, consistent, and resilient data publishing platform for RDF.

Keywords: SPARQL, RDF, Linked Data, Semantic Web, High Availability

1. Introduction

When publishing data, there is no limit to the ways in which the data is used. As the data is discovered and used by new applications, the usage of the published material changes over time. Data publishers who want to attract these applications want to provide a high quality service including 24x7 availability in order to encourage usage as a trusted, reliable publishing service.

Experience running two public-facing data publishing service shows that the most frequent causes of machine interruption do not come from hardware faults or software bugs. System administration is a much more frequent cause but the availability requirements exclude the idea of scheduled downtime. Sometimes, security alerts need to be handled at very short notice.

A single server can not continue providing the application-facing service. It requires more than one machine, and that in turn complicates the service operation. Yet that in turn introduces complexity in the form of update to multiple copies of the data.

We describe a linked data publishing platform that can provide 24x7 operation while at the same time allowing machines to be brought in and out of service. It provides consistent, robust update across the multiple copies of the data.

2. Prior Experience with Data Publishing Platforms

This section covers two services operating on Amazon Web Services (AWS). They were built to provide continuous operation using single application. Experience from these informs the design of the Lizard publish platform.

Each publishing service operates as a number of identical replica application stacks, normally 2, behind a load balancer. Each replica stack is itself a pair – a application server providing a data explorer via HTML, and a SPARQL database server. The stack also provides query access to the database via the application layer.

One service is small, of the order of 10 million triples, and updated several times a day in small quantities, and the other is a larger database of 370 million triples, updated once a month. Both are public facing services.

The main cause of system updates has been due to administration tasks, either planned when conducting routine updates including security updates, or unplanned in the case of short-notice security issues, as in the case of SSH Heartbleed. Customer contracts stipulate maintaining the systems are up-to-date and penetration testing covers issues encountered by the customer beyond the minimum for the service.

Managing the system is achieved by exploiting the "read mainly" nature of the services. Updates come via different route, from the data owner, and not part of the public facing service. Machines can be stopped by holding up updates, taking a replica out of the load balancer, performing system administration, and putting back in the load balancer. The other replicas are updated by repeating the process.

Another source of system change relates to changes in usage patterns. A general data publishing service is

doi:10.14337/XMLLondon15.Seaborne01

different to a database-backed web site because the nature of queries to the underlying database are much more varied. There isn't a controlled set of local application queries that can be audited and tuned. Some queries may involve powerful views of the data including sorting and aggregation which are costly operations, whether performed in the database or in the presentation service. Some access patterns defeat caching techniques: a web crawler does not visit the same page more then once so caching of previously calculated pages has little benefit. The major search engines operate multiple machines to crawl a site and, even when each on it's own is a tolerable load, having 4 or more search engines all active at once has generated excessive load. In addition, not all software respects web controls such as robots.txt and "nofollow" link mark-up.

Such load increases can also arise by external events such as publicity around new or greatly enhanced data offerings. To meet this, new replicas may need to be added to a running system, and removed later when load subsides.

The administration tasks are scripted but they are relying on the ability to hold back updates for a period of time.

3. Advantages and Disadvantages

This approach has the advantage of not requiring the majority of software to be modified for replicated use. Updating the data is not coordinated to provide a consistent view but because any request of generated page, or SPARQL query, is only driven from one database each request is seeing a consistent view. During update, it is possible one replica is serving new data but another is serving old data. The short update time makes this acceptable for the class of data publishing involved. Should a failure happen during updates, the task of bring the system back to a consistent state is mainly manual and the time during which the data is inconsistent is longer.

So in addition to continuous operation, we would like to have some transaction features for consistency and robustness to make platform operation easier.

The last desirable feature is being able to flex the system in terms of scale. For replicated separate systems, scaling horizontally adapts to changes in the number of requests being made of the overall platform. Given the nature of new events such as publicity around new data sources, being able to add new capacity, and remove it later to reduce costs, is desirable.

4. Apache Jena

Apache Jena[1] is an open source RDF system. It provides an RDF database (TDB) with a complete implementation of SPARQL 1.1 query [1] and update [2] languages. It provides a singe-machine database server accessed with the standard SPARQL protocols (query [3], update and graph store [4] protocols). This is Apache Jena Fuseki.

5. Storing RDF

In this section, we outline the single-machine design for RDF storage and query processing in Apache Jena TDB. This designed is extended to a multi-machine setup in the next section.

An RDF Dataset is a default graph, and zero or more named graphs. TDB stored these as a triples table, for the default graph, and a quads table for the named graphs. It can also use the dynamically computed union of all the named graphs as the triple table instead of concrete storage. We will discuss RDF triples and RDF graphs – the same principles apply to RDF Datasets and quads used to represent named graphs, a quad being an RDF triple with an extra field to record the graph name.

TDB stores RDF terms (URIs, Literals, blank nodes) using a dictionary. Each RDF term is allocated a fixed length binary node id, and the representation of the RDF term is stored in the dictionary using that fixed length binary node id as a key.

A triple is a tuple of three RDF terms (subject, predicate object) The same node id is used for every occurrence of the same RDF term (URI, literal, blank node) and matching SPARQL basic patterns is done by node id. Incoming queries are translated to node id form and the results mapped back to RDF terms in the results.

For certain value types, the node id is used to encode the value within the identifier. This avoids the need to look into the dictionary to retrieve the RDF term and then convert it into a value for use in SPARQL FILTERs. Encoding the value directly avoids this process and can make a significant difference to performance. Values are recorded, not lexical forms,; for integers 01 and +1, two ways to write the lexical form of integer value 1, so they become the same node id. If a value does not fit into a node id, the RDF term is stored in the dictionary as normal. Datatypes supported include the XSD datatypes [5] integers, and the derived types, decimals, dateTime and dates.

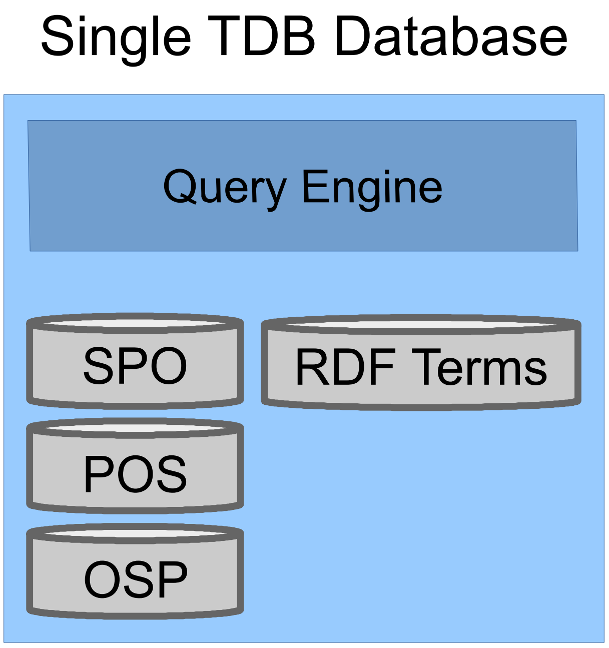
Indexes are used to used to store the triples, with the default choice being SPO, POS, OSP, where the letters

[1] Apache Jena - http://jena.apache.org/

indicate the ordering. The SPO index can return all tuples, all tuples starting with a specific S or all tuples with specific SP; it can not efficiently return all tuples with specific O and P but any S values.

The OSP index is little used in real world queries. Because of caching effects, its presence affects loading speed but does not influence query performance.

The indexes record all 3 parts of the triple tuple so there is no need to additionally store the triple tuple itself.



For example, if the triple is

```
?s rdf:type :Class
```

the POS index can be used, with a lookup of

```
                P=id(rdf:type) O=id(:Class) S=any
```

where `id(..)` is the node id of the given URI mapped by the RDF Term dictionary.

TDB uses B+trees. A B+tree is a datastructure using fixed sized blocks to hold a number of key- value pairs. When storing triple tuples, the tupel forms the key and there is no additional value part. The B+Tree algorithms balance the tree so that the blocks are at least half full.

A B+tree keeps its key entries in a sorted ordered so to answer all triple tuples matching "SP?", a lookup of "SP-" (where "-" is the value 0, the first possible O node id), then a short range scan until the first turtle that starts with a different prefix SP is seen, where (P+1) is node id treated as a number, +1.

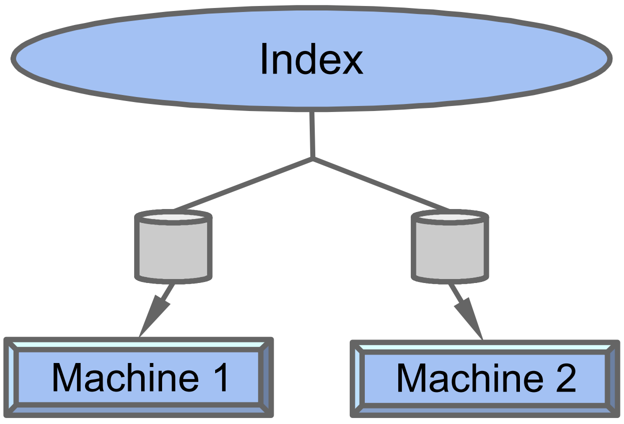
6. Extending to a Cluster

The data for the RDF needs to be replicated for 24x7 availability. The SPARQL query engine itself does not have any persistent state; the persistent state is all held in the indexes and the RDF term dictionary. Replications strategies for both are similar.

Considering the indexes, and starting from the design of single-machine TDB, there are two main points at which replication can be added. One is the interface to the indexes (the B+Trees) and the other is at the storage layer, replicating the blocks used to store the B+Tree.

The attraction of replicating blocks is that either a replicated file storage solution or a general key- value store could be used, using the block number as the key and the block contents as the value. However, replicating the blocks leads to any inefficient design that would limit scalability because in any lookup, several blocks must be traversed in a sequential manner, leading to several cross-cluster operations, especially during the start-up period when the server is started and caches have yet to fill up properly. In addition, general key-value stores return the whole of the block, yet only a part is needed for searching so not only more network operations are performed but also more data is transferred across the network.

It is more efficient to replicated the indexes themselves, so that a single cluster operation is involved for a lookup and a stream of matching triple tuple returned.



With N replicas, consistency is achieved if R replicas are read for a rad operation and W replicas updated for a write operation where R+W > N. When a read occurs over R copies, at least one up to date write copy is included. Because this is a read-dominated publishing platform, the usual choice is R=1 and W=N. An effect of this choice is that when an update is completed, there is no further replication to be done asynchronously, making recovery simpler because there is no work done as a a background backlog.

The main change in the query engine is to change the join algorithms. On a single machine, the SPARQL query engine when used with TDB, can use index joins. This join algorithm has the advantage that it uses a fixed amount of working space, regardless of the size of the data being joined. Because TDB is often run in the same JVM as the application, the fixed size workspace, and the fact the query engine and the storage are in the same JVM, means that the database engine does not excessively compete for Java heap resources with application code (file system caching is used extensively, ; this is not part of the java heap space).

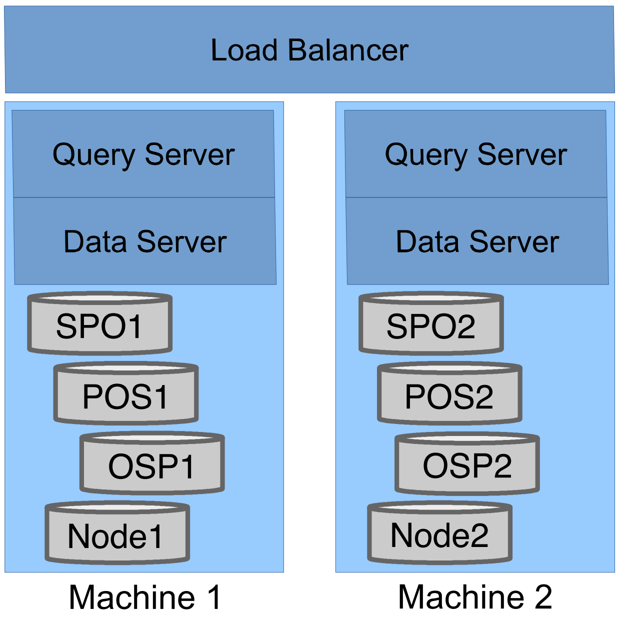
These assumptions are not valid on a cluster. It is a single system and while possibly multiple queries at the same time, all the system resources can be devoted to SPARQL execution. The index join algorithm requires multiple probes into one side of the data streams to be joined, which in turn would result in multiple cross-machine operations. Instead, Lizard uses pipeline hash joins for matching SPARQL basic graph patterns.

7. Deployment

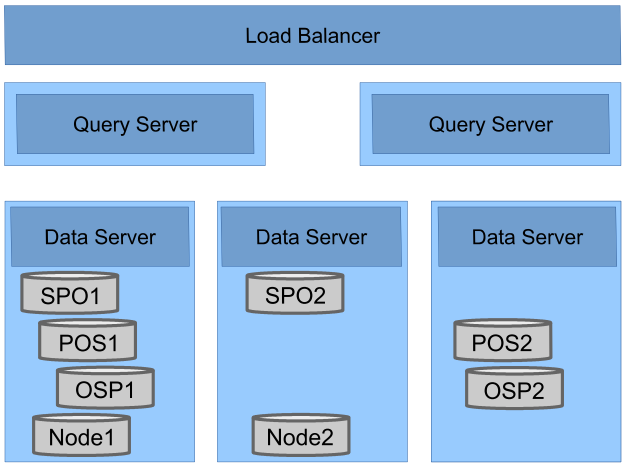
This design decomposes Lizard into a number services, separated by a network connection: query engines, a number of index replicas and a number of node dictionaries. Each of these service instances can be placed across a number of machines such that each machine has only one copy of each replica type so that the loss of one machine, whether a planned or unplanned, does not make part of the database inaccessible.

A local balancer is used to provide a logical single point of connection (round-robin-DNS can be used as well). Apache Zookeeper[1] is used to coordinate which machines are in the cluster and to provide the cluster wide locking to coordinate update transactions.

Small deployment might consistent of two machines, with each machine having one copy of each service instance. Updates are performed across both machines, read requests can be performed by one machine.



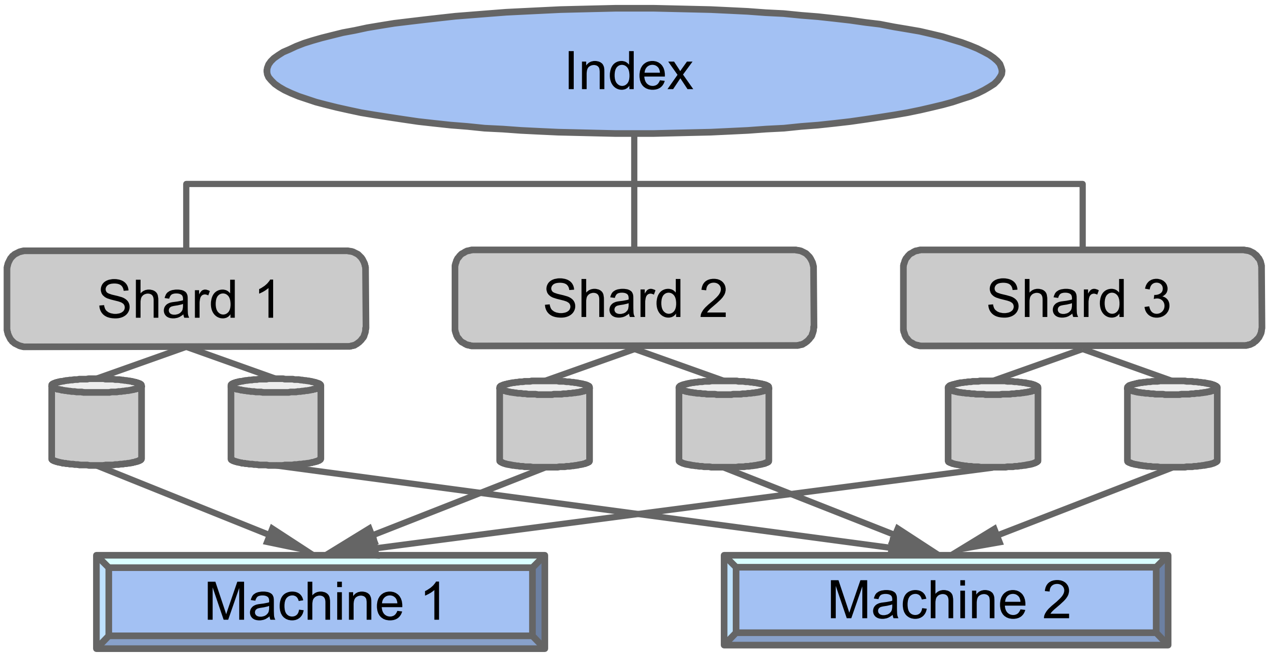
A larger system might consist of different machines for different components. Some query work loads require significant amounts of CPU resources so separating query servers from the data-storing index and dictionary services ma be useful.



[1] Apache Zookeeper - https://zookeeper.apache.org

8. Scale

Scale, to service increasing workloads, can be achieved by adding more machines, as shown in the deployment using 4 server machines. The Lizard design can form the basis of a scalable store, reaching to larger datasets by partitioning the data storage elements. For example: a replicated index,with 3 shards, 2 copies of each shared, mapped to 2 machines, might be:



Bibliography

[1] Steve Harris and Andy Seaborne. 21 March 2013. *SPARQL 1.1 Query Language*. World Wide Web Consortium (W3C).
http://www.w3.org/TR/sparql11-query/

[2] Paul Gearon, Alexandre Passant, and Axel Polleres. 21 March 2013. *SPARQL 1.1 Update*. World Wide Web Consortium (W3C).
http://www.w3.org/TR/sparql11-update/

[3] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 21 March 2013. *SPARQL 1.1 Protocol*. World Wide Web Consortium (W3C).
http://www.w3.org/TR/sparql11-protocol/

[4] Chimezie Ogbuji. 21 March 2013. *SPARQL 1.1 Graph Store HTTP Protocol*. World Wide Web Consortium (W3C).
http://www.w3.org/TR/sparql11-http-rdf-update/

[5] Paul V Biron and Ashok Malhotra. 28 October 2004. *XML Schema Part 2: Datatypes Second Edition*. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xmlschema-2/

Streamlining XML Authoring Workflows

Phil Fearon

DeltaXML

Abstract

When preparing XML content for publication, even small-scale projects can involve many people at different stages in the process; the process itself will often repeat several times. It follows that an XML review and approval workflow should allow everyone to contribute to the process at various stages; this is often critical to the quality and timeliness of the end-product.

This paper explores ideas on how XML document merge features can allow contributors and reviewers to, when necessary, work concurrently on content within an XML authoring workflow. A 'proof of concept' application called XMLFlow is used as a vehicle to demonstrate some of these ideas; some detail on the design and implementation of this proof of concept is also covered here.

1. Sequential and Concurrent Editing Workflows

In its simplest form, there are two ways that updates can be made to document content: sequentially or concurrently. Sequential updates (where each contributor updates content in turn) sit easily within a conventional editing chain. With concurrent updates, several people edit a copy of the content at the same time, a 'merge' process is then required to incorporate all changes back into a single copy.

Figure 1. Sequential updates: each contributor edits the document in turn.

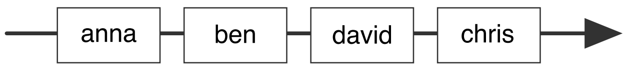
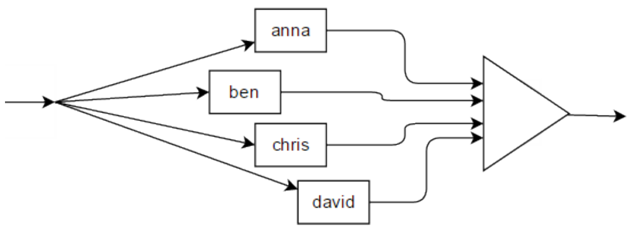


Figure 2. Concurrent updates: each contributor edits a copy of the document in the same time-frame.



So which is the preferred way to update documents, sequentially or concurrently? Sequential updates seem at first to be the more obvious solution for many cases. This approach does however impose significant restrictions including:

- Requires a locking mechanism to prevent conflicting changes being made to the same document.
- Contributors need network access to a central copy of the document.
- If contributors are unavailable they hold up the entire editing chain.
- The correct positioning of people within the editing chain is often critical.

Given the above restrictions for sequential editing, even when this is the preferred workflow, support for occasional concurrent editing provides additional flexibility when it is needed.

2. Current Systems

2.1. XML Authoring

In general, XML authoring systems provide good support for sequential editing workflows, for example by providing tracked changes for multiple authors and document locking features. Few of these system however provide specific support for concurrent working; one possible reason for this is the perception that document merge is time consuming and error prone. The counter to this perception is that XML documents with well-defined semantics are particularly suited to merging as their formal structure minimizes potential conflict, this also allows complex changes to be broken down into parts that are much easier to understand and manage.

2.2. Software Development

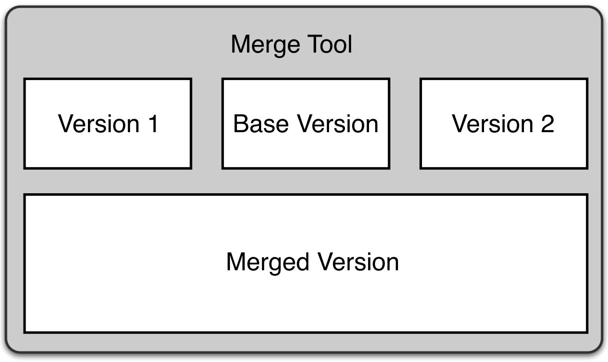
With few examples of support for concurrent working in XML authoring systems, it is useful to look in other areas outside of this field. One example is in the 'branch and merge' system used frequently as part of a software development cycle. Here, version control systems such as Git and Mercurial provide built in tools to support development of code on different branches, these

doi:10.14337/XMLLondon15.Fearon01

branches can then be merged with each other or merged back on to the main branch.

When developing the proof of concept described in this paper, inspiration was sought from the merge tools found in software version control systems. A decision was made however (rightly or wrongly) to tackle the issue from a new viewpoint; this was due both to differences in the type of content being merged and perceived differences in the typical preferences of end-users. Another significant factor in this decision is that the goal for the proof of concept is to allow up to ten documents to be merged at a time, software merge tools are typically restricted to just three versions of the code (including the common ancestor), where side-by-side views are more effective. A layout diagram for a typical code merge tool is shown below:

Figure 3. The layout of a typical code merge tool using side-by-side views



3. Proof of Concept Design

The proof of concept application, XMLFlow, was developed to be an experimental front-end for rendering the result of a document merge using DeltaXML's DITA Merge product. It has however evolved to support a more complete XML authoring workflow with the merge capability built in.

With integrated document merge capability, this application now also allows us to explore and demonstrate options to support concurrent document authoring workflows in a more general way. The aim was to provide a solution that is fully functional so that all parts of the workflow, except the actual editing, can be tested. It was decided not to include any editing capability as this allows us to focus on the merge, review and approval part of the authoring workflow, it also reduces development effort, and keeps the workflow flexible. Files uploaded to XMLFlow are not encrypted and are sent and retrieved over a standard HTTP connection.

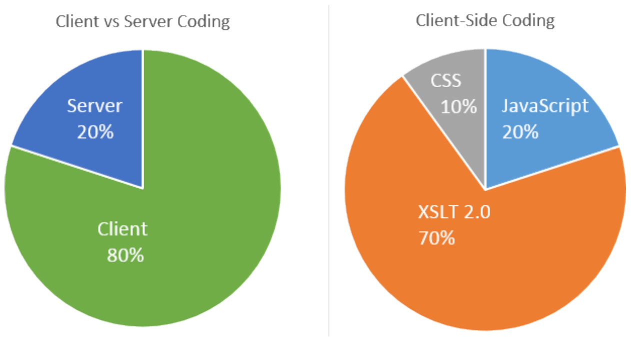
Note

XMLFlow's design is strongly influenced by experiences learnt in the formal document review process for enterprise-scale projects. Here, even minor edits would need to be recorded and approved. In this context, the absence of an editing capability in a review and approval tool such as XMLFlow could actually be seen as a benefit.

3.1. High-level Architecture

The application uses a client-server architecture, all server-side functionality was implemented in Java as a Java servlet; the 'client' is an HTML5/JavaScript single page web application. The Saxon-CE XSLT 2.0 processor is used on the client for all significant HTML and XML transforms. This architecture meets the need for the application to run on a wide variety of operating systems.

Figure 4. Breakdown of effort for development of the proof of concept.



The server performs two key roles: 1) it provides a remote file system for storing and retrieving files, and 2) it provides the 'merge' service which invokes DeltaXML's DITA Merge product. Apache Tomcat 8 is used as the web application server that hosts the Java servlets. This, in combination with other Apache libraries provides the required multi-part HTTP POST and WebSocket connectivity for the merge service

3.2. User Interface Style

For this type of application, effort spent on cosmetic changes to the user interfaces should be minimised, yet the look and feel must still be good enough to avoid being a distraction. To help with this, the BootStrap CSS framework was used, but with JavaScript event-handling functionality removed - so XSLT-based event-handling (using Saxon-CE's interactive extensions) could be used instead. The user interface for this web app was designed from the outset to suit an Apple iPad tablet, desktop-specific features are also included for cases when the app is run on laptops or desktops with a hardware keyboard.

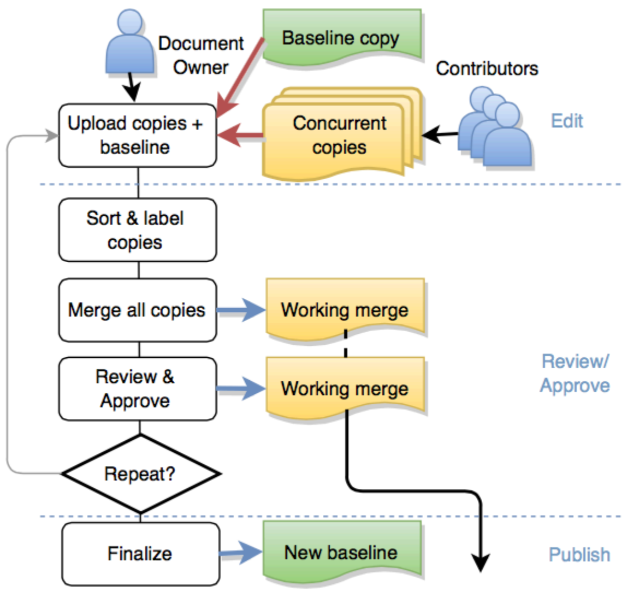
3.3. DITA XML Document Format

The principles demonstrated by XMLFlow apply to all structured XML document formats. The DITA document format is used here simply because the DITA Merge product is integral to this app (running server-side), and was DeltaXML's first product to support an n-way merge (more are in the pipeline). The document view in the proof of concept provides basic styling for most but not all DITA elements in the DITA topic, concept and reference document types.

3.4. Proof of Concept Workflow

The proof of concept is built around an XML authoring workflow with three high-level phases: edit, review/approve and publish. As already mentioned, the 'edit' phase is outside scope here, the workflow therefore starts with a step for retrieving updated copies of a baselined document from each of the contributors. The basic steps for the workflow are shown below:

Figure 5. Flow diagram for a typical XMLFlow workflow



In this scenario, the 'document owner' initiates the process by launching the XMLFlow web app in their browser and selecting the files to load, either by providing URLs or simple drag and drop.

Once all files are uploaded into the client, they are shown in the Files list; here they can be reordered and labelled with meaningful short names. Critically, the baselined document must be located first in the file list, this is because of the way n-way merge works: each updated document is first compared with its 'common ancestor', which is the 'baselined copy' in this case.

The document merge is started when the user presses the 'Merge' button in XMLFlow. Once the merge operation is complete (which may take a few seconds), a 'working merge' document is rendered in the document-view, with changes from all contributors highlighted.

Now there is a working merge, the review/approval phase can begin. The XMLFlow user can select 'Approve' and 'Reject' modes, the corresponding action is then performed when the user selects any change, this can be in the document-view, the attribute changes list-view or the content-changes list-view. Affected changes are restyled to reflect whether they have been accepted or rejected.

When all changes have been accepted or rejected the working merge is ready for finalizing. The 'finalize' process works on a copy of the last version of the working merge, it uses approval data embedded in the XML file to modify the content and attributes to reflect the approval decisions. The embedded approval data is then removed so that the finalized XML file is valid. This file is suitable to be used as the baseline for a further workflow. The working merge can also be kept so that there is a record of the approval process, a critical requirement in some situations.

Note

The term 'baseline' is used a number of times in this document. This term is frequently used in configuration management or project management processes, but can also apply to more formal publishing processes. In this context, a new 'baseline' is created each time a set of significant changes for a document are agreed at a certain stage in the life-cycle of the document. A document may of course be part of a larger project which will have its own baselines.

3.5. Identifying features in the workflow

To support a concurrent XML authoring workflow, the XMLFlow GUI design reflects the following areas of functionality that were identified:

- File management
  - Upload - DITA input files or a working merge
  - Label - add meaningful short labels for each file
  - Reorder - sort files, with the baselined file first
  - Store/Download - any file including working and finalized merge files

  - Merge - create a working merge file for review/approval
  - Finalize - create a finalized version from the working merge
  - Reset - clear all files ready for another merge
- Document change review
  - Show changes inline in a document view
  - Show changes in vertical lists for:
    - content
    - attributes
- Support approval states
  - Accept
  - Reject
  - Defer

3.6. Document-view design

The document view was by far the most challenging part of the XMLFlow GUI design; a number of different approaches were tried before settling on the current solution. The document view is effectively an HTML div element with content generated by a client-side XSLT transform on the result of the merge operation. In the relatively simple transform, XML document elements are renamed to prevent conflict with HTML elements, new attributes are also added to assist with CSS styling.

The problem here is to determine how to show the different types of changes made by multiple users in a single view - without overloading the end-user with too much information. In an n-way merge, there are three basic change types: 'add', 'delete' and 'modify'. A 'modify' type occurs when a word or phrase has been changed by one or more contributors, if a modified word or phrase has been deleted by another contributor this is still regarded as a 'modify' with an empty-string as the new value. Changes marked as 'modify' are presented as a choice of two or more mutually exclusive options, with selection of the baselined text version being the first option.

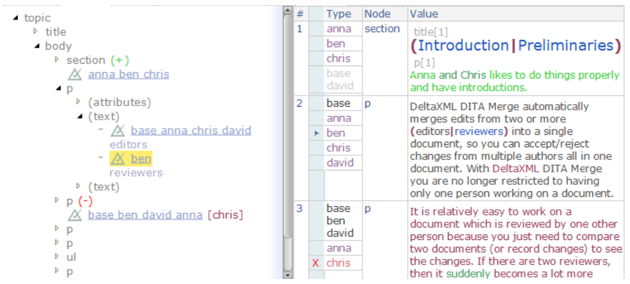


Figure 6. A first attempt at the document view: element tree and grid controls, with no WYSYWIG styling

```
topic
  title
  body
    section (+)
    anna ben chris
      p
        (attributes)
        (text)
          - base anna chris david
            editors
          - ben
            reviewers
        (text)
    p (-)
    base ben david anna [chris]
    p
    p
    ul
    p
```

#		Type	Node	Value
1		anna	section	title[1] (Introduction|Preliminaries)
		ben		p[1]
		chris		Anna and Chris likes to do things properly
		base david		and have introductions.
2		base	p	DeltaXML DITA Merge automatically
		anna		merges edits from two or more
		ben		(editors|reviewers) into a single
		chris		document, so you can accept/reject
		david		changes from multiple authors all in one document. With DeltaXML DITA Merge you are no longer restricted to having only one person working on a document.
3		base ben david	p	It is relatively easy to work on a document which is reviewed by one other person because you just need to compare
		anna		two documents (or record changes) to see
	X	chris		the changes. If there are two reviewers, then it suddenly becomes a lot more

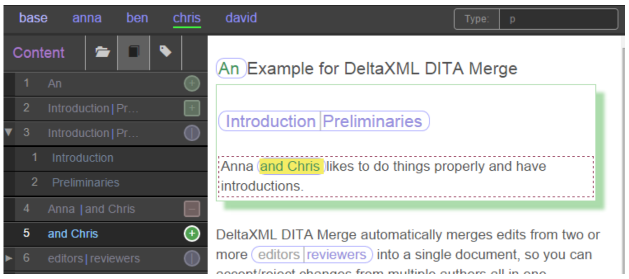


Figure 7. The current evolution of the document view: element grid replaced by a simplified WYSYWIG view

1 An
2 Introduction | Pr...
3 Introduction | Pr...
1 Introduction
2 Preliminaries
4 Anna | and Chris
5 and Chris
6 editors | reviewers

An Example for DeltaXML DITA Merge

Introduction Preliminaries

Anna and Chris likes to do things properly and have introductions.

DeltaXML DITA Merge automatically merges edits from two or more editors reviewers into a single document, so you can

Red, green and blue foreground text colors are used in the document-view to indicate delete, add and modify respectively. To distinguish between an element change and a text change, square borders are added for elements and round borders for text. Similarly, in the change list-view, square and round icons indicate element changes and text changes respectively. To help differentiate between adds and deletes, the square borders for element changes are dashed for deletes and supplemented by a shadow for adds.

3.6.1. Nested changes

Nested changes occur when two or more contributors add an element that is broadly similar in each case but has minor changes within it. For example, we have a nested change if 'anna' and 'ben' and 'chris' add a section that aligns, but 'ben' deletes a paragraph within the section, a further level of nesting occurs if 'chris' also added a phrase within the paragraph.

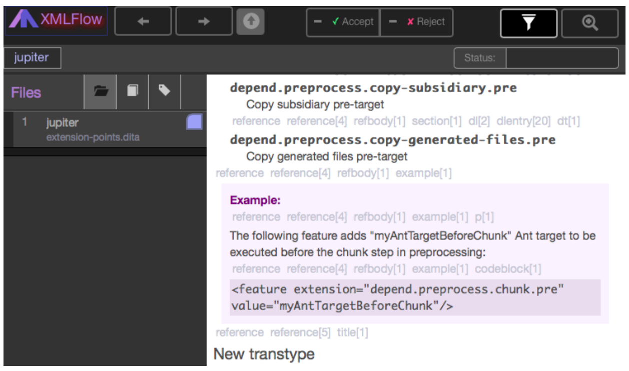
In nested changes the baseline document (the common ancestor) is no longer within the context. Without this context, it cannot be determined whether a nested change is an add or a delete, it could be either depending on your viewpoint. In these cases, XMLFlow takes the approach whereby a change type is never nested within a similar change type, thus a change occurring within an add is always marked as a delete and vice-versa. This arbitrary typing of nested changes is undesirable, but necessary without any further information about why the same parent change was made by more than one contributor.

3.6.2. Styling of content

The original DITA XML is transformed into HTML span elements representing each original DITA XML element, attributes are added to the span elements to indicate their original type and allow them to be styled via CSS rules. In this way, CSS rules for the attributes style tables, lists and headings and other block-level types, inline formatting is also applied.

For this proof of concept advanced styling is not included, for example, multi-column spans on table cells are not rendered properly and `conref` element references are indicated by a simple placeholder. A style filter is normally applied to the view, when this is switched off, content that is not normally rendered such as metadata and DITA comments can be viewed.

Figure 8. Simple WYSYWIG styling for DITA is provided through CSS rules



depend.preprocess.copy-subsidiary.pre
Copy subsidiary pre-target
reference reference[4] refbody[1] section[1] dl[2] dlentry[20] dt[1]
depend.preprocess.copy-generated-files.pre
Copy generated files pre-target
reference reference[4] refbody[1] example[1]

Example:
reference reference[4] refbody[1] example[1] p[1]
The following feature adds "myAntTargetBeforeChunk" Ant target to be executed before the chunk step in preprocessing:
reference reference[4] refbody[1] example[1] codeblock[1]

```
<feature extension="depend.preprocess.chunk.pre"
value="myAntTargetBeforeChunk"/>
```

reference reference[5] title[1]
New transtype

XMLFlow's CSS styling of inline text changes works by exploiting attributes on wrapper elements that are part of the DeltaV2 format output by the DITA Merge component, this format is described later in this paper.

3.6.3. WYSYWIG Vs Code View

End-users often prefer a 'What You See Is What You Get' (WYSYWIG) view to a raw XML view as this avoids the clutter and distraction of lots an angle-brackets and attributes. Sometimes however, it is necessary to see the raw XML to fully understand the document structure; ideally XMLFlow would include a code view, at this stage it does not, but it does however reveal the XPaths for key element types when the view 'filter' is switched off. An element tree-view can also help with understanding XML structure, this was present in an early prototype but removed after feedback suggested it added too much clutter for an end-user.

3.7. Content and attribute change lists

To supplement the document view, XMLFlow presents two lists, one for attribute changes and one for content changes; these share the same tabbed view as the files list and are reached via tab-buttons immediately above them. List items representing 'modify' changes (as opposed to simple adds and deletes), are marked with an arrow to indicate they can be expanded to show the list options.

3.8. Accept and Reject Modes

When XMLFlow is launched it is effectively in a 'Review' mode. That is, when a document change is selected in the document-view, the content changes-list or the attribute changes-list, the change is simply highlighted with information shown about the selected change in other views. To switch to the 'Accept' or 'Reject' mode, the corresponding button is pressed in the toolbar. A single button press switches mode for a one-off operation - two presses are required to make the mode 'stick' for all further operations. An accept or reject on a change is quickly backed-out from by pressing the change again.

A change is highlighted, both in the list-view and the document-view to show whether it has been accepted or rejected. In list-views, accepted changes are shown with a red bar and a cross, rejected changes with a green bar and a check-mark; for multi-choice changes (which can't be rejected) the selected option is highlighted in blue and a check-mark. In document views, borders are replaced by bars or symbols for block-level and inline changes respectively; text is hidden for accepted deletes or rejected adds, text is shown for accepted adds or an accepted multi-choice option.

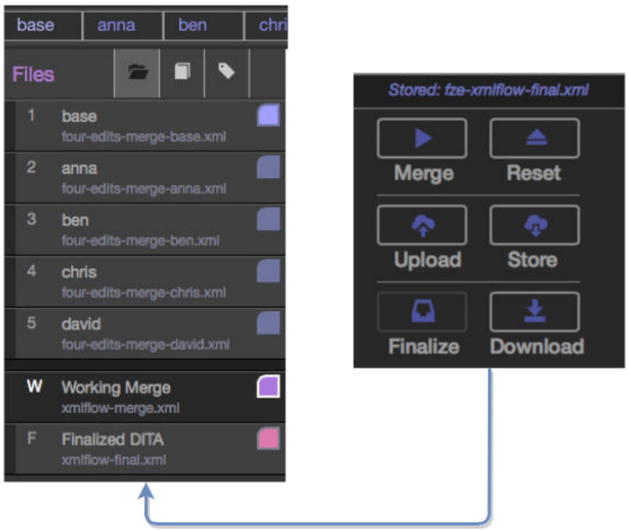
3.8.1. User Experience

With a large number of document changes, it can be tedious for the reviewer to go through and press on each change, the use of modes means that only one press is required for each change. This could be improved further if a block of several changes in a list could be accepted or rejected with a single click.

3.9. The Files Panel

All input and output files in XMLFlow are managed through the files panel; this comprises a list of files at the top and a control panel below. Input files are either drag and dropped from the desktop into the files panel or uploaded via a supplied URL.

Figure 9. In the files panel, the control panel appears below the files list


base
anna
ben
chri
Files
1 base four-edits-merge-base.xml
2 anna four-edits-merge-anna.xml
3 ben four-edits-merge-ben.xml
4 chris four-edits-merge-chris.xml
5 david four-edits-merge-david.xml
W Working Merge xmlflow-merge.xml
F Finalized DITA xmlflow-final.xml
Stored: fze-xmlflow-final.xml
Merge
Reset
Upload
Store
Finalize
Download

The labels bar is immediately above the tabbed panel hosting the files panel tab. When the files panel is selected, the labels bar becomes editable. Each label is also shown in the corresponding file item in the files list, along with the long filename.

Before starting a merge, at least three files are required, the first must be the common ancester. All input files within the list can be sorted and labelled. Labels allow short meaningful names to be ascribed to each file, these are also shown in the labels bar so that labels are highlighted when a change associated with a label is selected. Any label in the labels bar can also be selected to highlight all changes associated with the selected label.

All operations on files are managed through buttons on the control panel. The 'Merge' button initiates a merge operation on the remote server, because this operation may take a few seconds, merge progress is reported by highlighting a bar adjacent to each file item in the list. Once the merge is complete a 'Working Merge' file item is added to the files list and the merge result shown in the document view, ready for review and approval.

Pressing the 'Finalize' button produces a new 'Finalized DITA' item in the files list, when this file item is selected, the finalized document can be seen in the document-view. Once the finalize button is pressed it is disabled until all files are reset. This is because if the user carried on with the working merge, further finalize operations are performed automatically each time the 'Finalized DITA' file item is selected.

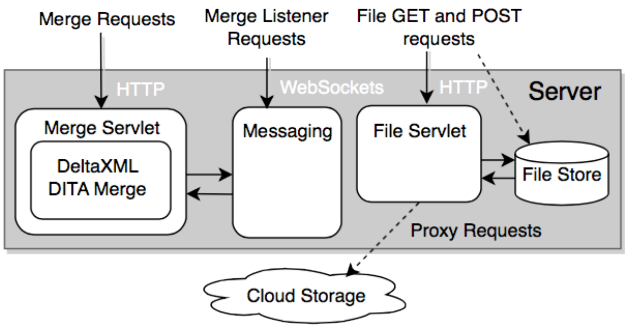
The 'Upload', 'Store' and 'Download' buttons are for managing files on the remote server. All stored files are given a URL that includes a unique 'hard to guess' number within the filename (e.g. 4392800660168831878), the URL appears in the header bar so it can be easily copied to the clipboard; files can be retreived at a later time by anyone who knows the URL.

4. Proof of Concept Implementation

XMLFlow is a single-page web application, with functionality split between the browser client and the remote server. The server and client side implementation is described in the following sections, for this paper the focus is on the client.

4.1. Server

The Server's role is to provide a set of high-level services for the client, no application-specific logic is embedded in the server, allowing this to be used with an entirely different front-end. The provided services are shown as reqeusts in the server component diagram below:



The Apache Tomcat 8.0 web application server is used for hosting the proof of concept server. All server side functionality is coded in Java to allow easy integration with the DeltaXML DITA Merge component which has a Java API. The included Apache Commons FileUpload component provides the required support for multi-file requests from the client conforming to RFC 1867; this is compatible with the HTML5 FormData API used in the browser client.

The support for the 'Merge Request' is the most significant feature of the server. This allows the client application to make a single merge request that includes all the files to be merged, along with a label for each file.

Because the Merge Request is handled by a standard HTTP Servlet request, only one response can be made, when the merge operation completes (or fails). A merge operation on all the files can take several seconds, thus another connection is therefore required to keep the client informed of progress of the merge on each file. For this purpose, a WebSocket connection is made by the client, the server side response is to invoke a special handshake that binds the WebSocket connection with the HTTP Session. Java running in the HTTP session can now send notifications to the client while it is still processing the merge and thus holding the HTTP connection.

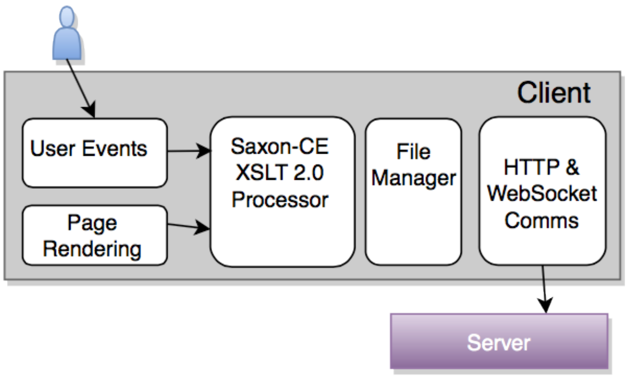
The simplified Java code below shows how, for each uploaded file, methods are called on the DitaConcurrentMerge class, the `setAncestor` method is called for the first file, all further files are added to the merge through the `addVersion` method call. A WebSocket text message is sent to the client by a `sendMessage` method call before and after call to the DITA Merge component.

```
private void processFile(
  FileItem item,
  DitaConcurrentMerge ditaMerge) throws Exception {
  // the short file label used as
  // a version identifier
  String fieldName = item.getFieldName();
  String fileName = item.getName();

  String fileId = "file" + (uploadFileCount + 1);
  sendMessage("upload," + fileId + "," + fileName);
  File uploadedFile = File.createTempFile("merge-",
                    "-" + fileName, uploadDir);
  item.write(uploadedFile);
  if(uploadFileCount == 0) {
    ditaMerge.setAncestor(uploadedFile, fieldName);
  } else {
    ditaMerge.addVersion(uploadedFile, fieldName);
  }
  sendMessage("add," + fileId + "," + fileName);
  uploadFileCount++;
}
```

4.2. Client

The XMLFlow client effectively comprises the browser, JavaScript and XSLT 2.0 code, CSS styling and a single HTML page. File handling, user-event handling, web-layer communications and page-rendering are the key areas of functionality, this is summarised in the diagram below:

Figure 10. A component view of client-side features



4.2.1. File Management

Files management is used in XMLFlow for managing XML documents that are to be merged, the 'working merge' document, and the 'finalized' document. The HTML5 File and FormData APIs are exploited to allow files to be retrieved, stored locally and then sent as part of a Merge request. File management functionality is coded in JavaScript, simplified code for sending a Merge request is shown below:

```
var mergeDocument = function () {

  // synchronously create a remote HTTP session and
  // bind this to a new WebSocket connection for
  // progress monitoring
  setupComms();

  // append each uploaded file along with its label
  // to a new FormData object
  var form = new FormData();
  var labels =
    document.getElementById("labels-group")
            .getElementsByTagName("input");
  for (var i = 0; i < rawFilename.length; i++) {
    var rawIndex = parseInt(
      listButtons[i].getAttribute("data-fileindex")
    );
    form.append(
      labels[i].value, rawfileData[rawIndex],
      rawFilename[rawIndex]
    );
  }
  // POST the updated FormData instance and set the
  // 'statechange' event handler function
  var oReq = new XMLHttpRequest();
  oReq.open("POST", reqMergeHttp, true);
  oReq.send(form);
  oReq.onreadystatechange =
    handleMergeStateChange(oReq);
};
```

The JavaScript above shows how, for each file to be merged, the file label, the file object (rawFileData), and the filename are added to a FormData object. This FormData object is then sent asynchronously via a POST XMLHttpRequest to the remote server, a function is assigned to handle the response. The first method call in this code is `setupComms`, this creates a WebSocket connection with the server and assigns a function to handle WebSocket messages sent from the server.

4.2.2. Page Rendering

This applicaton is rendered within a single HTML web page. The static page is effectively a skeleton to which dynamically updated parts of the application are added using XSLT. While standard XSLT normally transforms an entire document, the Saxon-CE processor extends the `xsl:result-document` to allow specified parts of the HTML DOM to be updated instead. There are three parts of the application that are updated in this way, the content change list, the attribute change list and the document view. The two change lists are updated using the same XSLT sylesheet that contains two result-document instructions, the document view is updated by a separate XSLT stylesheet with just one result-document instruction. Both XSLT transforms are invoked using Saxon-CE's JavaScript API as soon as a response from a Merge request is received.

4.2.3. Transforming a Merge Result

The result of a DITA Merge operation is a DITA document that effectively combines all the input documents into one. Where there are differences between versions, document elements are annotated with 'deltaxml:deltaV2' attributes. Extra wrapper elements are also added to allow differences in attributes and text nodes to be represented in a lossless way; these are namely 'deltaxml:textGroup', 'deltaxml:text' and 'deltaxml:attributes'.

Coding the XSLT was relatively straightforward once the DeltaV2 format was understood. The deltaV2 attribute of an element combined with the deltaV2 attribute of its parent has all the information needed to determine not only the type of change, but who made the change. This attribute holds the labels for all the documents that contain a match for that element. The attributes labels are arranged to describe the equality characteristics of the sub-tree of the element, they are separated by '=' if they are part of the same 'equality group', '!=' character-pairs are used to separate equality groups. An example deltaV2 attribute value would be: `X=anna=ben!=chris=david`.

An extract from a DeltaV2 formatted document is shown below, the label 'X' is used to denote the 'common ancestor' version, the other labels are 'anna','ben','chris' and 'david'. This shows a `p` element with an attribute that has been deleted by 'chris', we know this is a deletion because the deltaV2 attribute for the `p` element contains the label 'X' representing the common ancestor. Within the text node, the word 'DeltaXML' is deleted by 'anna' and 'chris' in two places. For each text deletion there is are two wrapper elements `deltaxml:textGroup` and `deltaxml:text` The `textGroup` element allows one or more `text` elements so that modifications of a word or phrase can be shown as well as additions and deletions.

```
<p deltaxml:deltaV2="X=ben=david!=anna=chris">
  <deltaxml:attributes
    deltaxml:deltaV2="X=anna=ben=david">
    <dxx:id xmlns:dxx=
"http://www.deltaxml.com/ns/xml-namespaced-attribute"
      deltaxml:deltaV2="X=anna=ben=david">
      <deltaxml:attributeValue
        deltaxml:deltaV2="X=anna=ben=david">
        legacy
      </deltaxml:attributeValue>
    </dxx:id>
  </deltaxml:attributes>Now, with
  <deltaxml:textGroup
    deltaxml:deltaV2="X=ben=david">
    <deltaxml:text deltaxml:deltaV2="X=ben=david">
      DeltaXML </deltaxml:text>
  </deltaxml:textGroup>DITA Merge, this job becomes
    much, much easier, because
  <deltaxml:textGroup
    deltaxml:deltaV2="X=ben=david">
    <deltaxml:text deltaxml:deltaV2="X=ben=david">
        DeltaXML
      </deltaxml:text>
  </deltaxml:textGroup>DITA Merge merges all the
  changes into a single document.
</p>
```

Part of the top-level XSLT template used to update the attribute and content change lists is shown below, with each xsl:result-document instruction updating a different list. The referenced `xsl:member-count` variable is an integer returned by a JavaScript extension function (another Saxon-CE extension that treats function names in a special namespace as native JavaScript functions). In this specific case, using a user-defined JavaScript function is not strictly necessary, but it serves well as an illustration of how JavaScript can be exploited, note the string() cast is used on the $deltv2 attribute argument because the JavaScript will not auto-cast attribute nodes to strings.

```
<xsl:variable name="member-count" as="xs:integer"
select="count(js:getV2Members(string($deltav2)))"/>

<xsl:result-document href="#changes"
                     method="replace-content">
  <xsl:apply-templates select="
      //deltaxml:textGroup |
      //*[not(self::deltaxml:*)]
      [exists(@deltaxml:deltaV2)
      and not(contains(@deltaxml:deltaV2, '!='))
      and count(tokenize(@deltaxml:deltaV2, '='))
        ne $member-count
      and not(parent::deltaxml:attributes)
      or (exists(@deltaxml:deltaV2)
        and count(tokenize(@deltaxml:deltaV2, '='))
      ne
        count(tokenize(
                parent::*/@deltaxml:deltaV2, '='))
      and not(parent::deltaxml:attributes))
      and exists(parent::*)]" mode="setchange"/>
</xsl:result-document>

<xsl:result-document href="#att-changes"
                     method="replace-content">
  <xsl:apply-templates
    select="*" mode="set-element-att-change">
    <xsl:with-param name="location" select="''"/>
  </xsl:apply-templates>
</xsl:result-document>
```

4.2.4. Creating a Working Merge

The 'Working Merge' is the result of the merge document with extra XML elements added at the start of the document to hold information about each change, and the approval data for each change (whether it has been accepted or rejected). This document format is designed so that the document merge and the current merge state can be stored remotely and reloaded at any time. When it is reloaded the embedded approval data for each change is used to update the changes lists and the document views in the XMLFlow page.

An example of the extra embedded elements in the working merge is shown below. Attribute and element changes are kept separate in their own respective deltaxml:attribute-updates and deltaxml:updates wrapper elements; this keeps the XSLT transforms simple as this is the way changes are viewed in XMLFlow.

```
<deltaxml:updates>
  <deltaxml:update index="1" change="add"
                   data-accept="yes"
                   data-descriptor="add"
                   data-dgroup="add"/>
  <deltaxml:update index="2" change="elementAdd"
                   data-accept="no"
                   data-descriptor=""
                   data-dgroup="elementAdd"/>
  <deltaxml:update index="3" change="replace"
                   data-descriptor="replace"
                   data-dgroup="modify"
                   option="1"/>
  <deltaxml:update index="4" change="elementDelete"
                   data-accept="no"/>
  <deltaxml:update index="5" change="add"/>
</deltaxml:updates>
<deltaxml:attribute-updates>
  <deltaxml:update index="1" change="elementAdd"
                   data-accept="yes"/>
  <deltaxml:update index="2" change="modify"
                   data-accept="yes"
                   data-dgroup="modify"
                   option="2"/>
  <deltaxml:update index="3" change="elementAdd"/>
</deltaxml:attribute-updates>
```

Note

The 'deltaxml' prefix is bound to the standard DeltaXML namespace, ideally a different namespace would be used, but an early decision was made to minimize the number of namespaces due to XML serialization issues encountered in the iOS implementation of Safari running on the iPad. A custom XML serializer was eventually implemented in JavaScript to deal with namespace issues.

4.2.5. Creating a Finalized Merge

The 'Finalized Merge' takes the working merge and uses the approval data to remove or add elements, attributes or parts of text nodes to produce a complete and valid DITA document. To ensure validity, the approval data is removed; in more formal review cases it is therefore prudent to keep a copy of the last working merge also, this is a record of whose changes were accepted or rejected.

4.2.6. Handling User Events

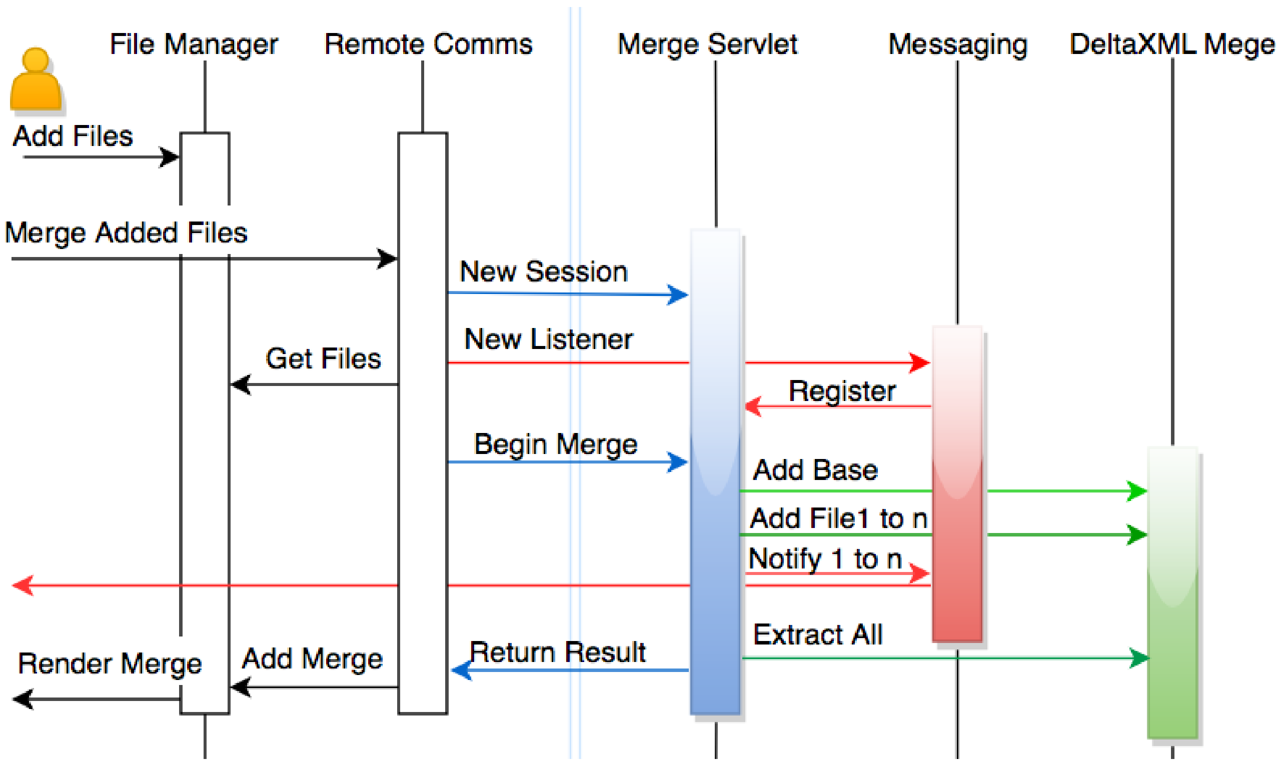
Using Saxon-CE's interactive extensions, XSLT templates with special `mode` attributes that match the event type handle most user events in the browser. This works reasonably well even for the iPad because `touchstart` and `touchend` events are supported by Saxon-CE's interactive extensions. There were however problems associated with scrolling panels for the iPad, an element might only be touched for the purpose of grabbing the entire panel to scroll. Here it was necessary to add some low-level JavaScript to intercept certain touch start events and only rethrow them if a corresponding touch end event happened within a certain interval. Other techniques tried meant there was a noticeable delay between the user touching an element and the user interface being updated to reflect the change. Note that event handling varies between different mobile platforms but event handling has only been specialised for the iPad.

4.3. A Document Merge Scenario

Having described how the main features of the client and server are used to perform a document merge, it is time to look at the flow of information between client and server for a single merge operation. The sequence diagram below shows the information flow in diagrammatic form.

The scenario begins with the user adding files for the merge through the user interface, these files are stored as objects in the File Manager. The process ends after the DeltaXML merge component has completed the merge, with the result sent back as the response to an HTTP request (via Remote Comms), the File Manager adds this result to its set of files, and finally the file is rendered in the user interface (via XSLT).

Figure 11. A simplified sequence diagram for a typical merge operation



One thing the diagram above illustrates is that the 'Messaging' component that wraps a WebSocket connection does add a degree of complexity because of the 'Register' method required as part of the initial connection handshake with the HTTP session.

5. Conclusions

The proof of concept, though restricted in certain areas of functionality still demonstrates the potential for XML document merge (and thus concurrent working) within an authoring workflow. Changes made concurrently to different copies of an XML document by a number of contributors can be explained to the user in an understandable way. Moreover, by using 'approval modes' the process of accepting, rejecting and deferring changes, can be performed relatively effortlessly.

In practice, the publication process and authoring workflow is affected and determined by such a wide variety of factors that it would be very difficult to claim that all processes could be streamlined by adding support for concurrent working. However, we should at least consider the software development process, where branch and merge has been an intrinsic part of version control for many years now.

This paper describes how DeltaXML's DITA Merge component was used by the proof of concept to gather all information required by a merge into a single DITA document augmented with 'DeltaV2' attributes and elements. Currently, XMLFlow uses this format directly to create the document-view and change-lists; with the benefit of hindsight, it would have simplified the XSLT to first convert the output to a form that characterised the XML so each change was described in terms of the change type and the change 'owner(s)'. This first 'analysis stage' XSLT could have been run server side also which would help reduce the load for resource-limited mobile devices like iPads. The drawback to a first 'analysis stage' is that, for nested changes, special care would be needed to qualify the type according to its context.

The XMLFlow design makes extensive use of client-side XSLT, reducing significantly the JavaScript skills needed to develop XMLFlow. The declarative nature of XSLT also means that when the design changed considerably, changes to XSLT still had to be made in a considered way; there was thus less potential for the code structure to degenerate significantly with each design change. This makes a good case for using client-side XSLT 2.0 (with interactive extensions) for rapid development of a proof of concept, especially when XSLT developer resources are already available.

Implementation of Portable EXPath Extension Functions

Adam Retter

Evolved Binary

<adam@evolvedbinary.com>

Abstract

Various XPDLs (XPath Derived Languages) offer many high-level abstractions which should enable us to write portable code for standards compliant processors. Unfortunately the reality is that even moderately complex applications often need to call additional functions which are non-standard and typically implementation provided. These implementation provided extension functions reduce both the portability and applicability of code written using standard XPDLs. This paper examines the relevant existing body of work and proposes a novel approach to the implementation of portable extension functions for XPDLs.

Keywords: XQuery, Portability, EXPath, Haxe

1. Introduction

High-level XML processing/programming languages such as XQuery, XSLT, XProc and XForms have long held the promise of being able to write portable code that can execute on any W3C compliant implementation. Unfortunately the specification of these languages leave several issues to be "implementation defined"; Typically a pragmatic necessity, most often occurring where the language must interact with a lower-level interface, e.g. performing I/O or integrating with the environment of the host system.

If we put to one-side the potential "implementation defined" incompatibilities, which in reality are often few and can likely be worked around, there is another issue which hinders the creation of portable code, and that is the issue of implementation provided extension functions. XQuery, XSLT, XProc and XForms are all built atop XPath, which defines a Standard Library in the form of the F+O specification (XPath and XQuery Functions and Operators). XQuery 3.0 and XSLT 3.0 provide F+O 3.0 [1], and whilst XProc 1.0 and XForms 2.0 provide the older F+O 2.0 [2] it is most likely that new versions of those specifications will also adopt F+O 3.0.

Whilst F+O 3.0 offers some 164 distinct functions and 71 operators, it is predominantly focused on manipulating XML, JSON and text, unfortunately for creating complex processes or applications with XPDLs (XPath Derived Languages) e.g XQuery, XSLT, XProc and XForms, these functions by themselves are not enough. To fill this gap, many implementations have provided their own modules of extension functions to their users; for instance eXist 2.2 provides some 53 modules [1] of extension functions for XQuery, and similarly MarkLogic 8 provides some 55 modules [2] of extension functions for XQuery.

The aim of this paper is that through examining the existing approaches to portable extension functions for XPDLs, an new approach is developed for their implementation which should enable them to be reused by any XPDL processor with the minimum of effort.

1.1. Extension Function Costs

Initially these extension functions are most welcome as they enable the user to quickly and easily perform additional operations which would be impossible (or costly) to implement in an XPDL. Unfortunately over time these extension functions add a burden with regards to portability [3] [4], which typically manifests itself in two distinct ways: directly and indirectly.

1.1.1. Directly

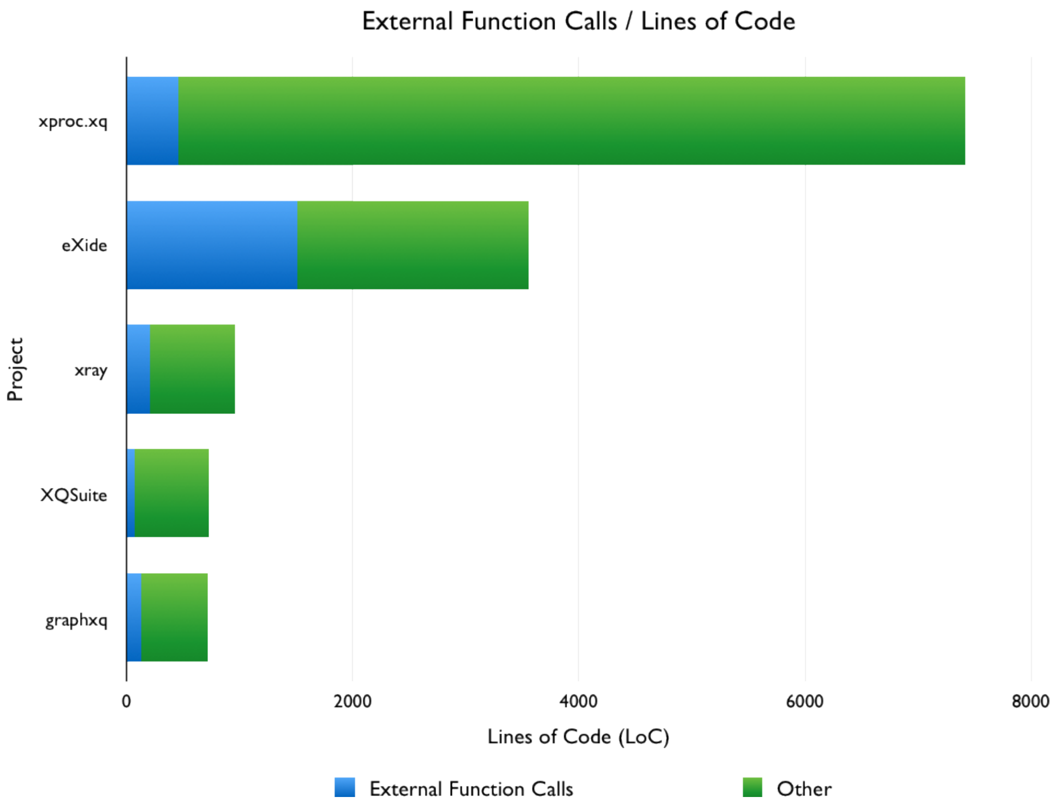
--- Restricting User Freedom

The use of proprietary implementation extension functions can adversely restrict the ability of a user to freely move between implementations or reuse their

[1] eXist XQuery extension modules were counted by examining the eXist source code at https://github.com/eXist-db/exist/tree/eXist-2.2
[2] MarkLogic XQuery extension modules were counted by examining the MarkLogic documentation at https://docs.marklogic.com/all

existing code across implementations. An examination of several Open Source projects (eXide[1], graphxq[2], xproc.xq[3], xray[4] and XQSuite[5]) which are implemented in XQuery reveals that the impact of this is typically a function of the size of the code base as illustrated in Figure 1, and the variety of extension functions that have been used as illustrated in Figure 2 and Figure 3.

Figure 1. External Function Calls / Lines of Code



[1] https://github.com/wolfgangmm/exide revision 07207a2 (12 April 2015)
[2] https://github.com/apb2006/graphxq revision 0b19756 (8 March 2015)
[3] https://github.com/xquery/xproc.xq revision f0f0697 (13 December 2014)
[4] https://github.com/robwhitby/xray revision dc03243 (25 April 2015)
[5] https://github.com/eXist-db/exist/tree/develop/src/org/exist/xquery/lib/xqsuite revision c32784a (5 May 2015)

Figure 2. Total Function Calls

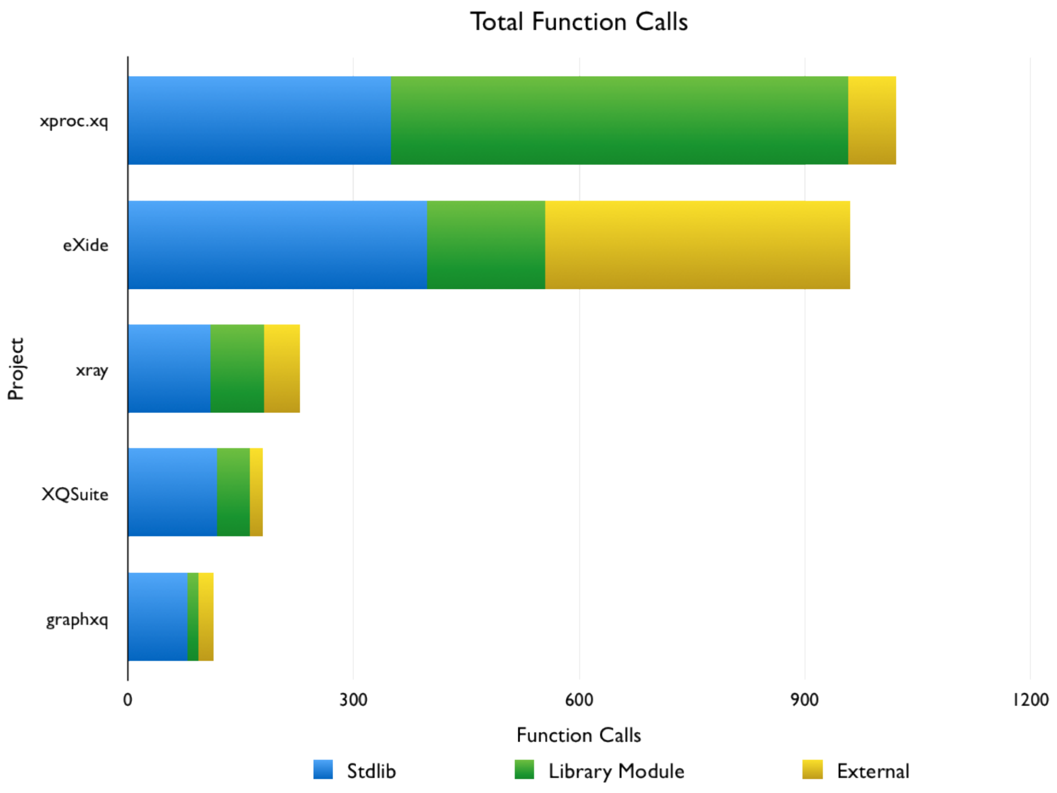
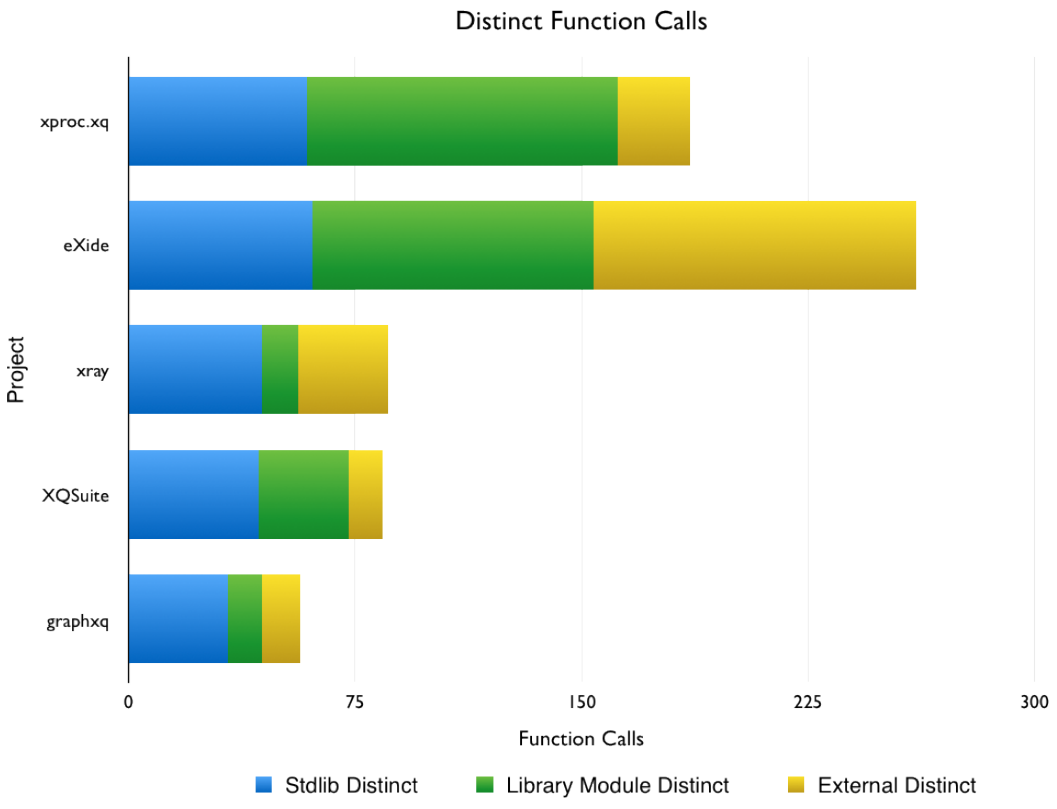


Figure 3. Distinct Function Calls



The most extreme example of this impact is often felt by XQuery framework providers (e.g. XRay and xproc.xq, xqmvc[1], etc.) who often have to attempt to abstract out various implementation extension functions to be able to provide frameworks which will work on more than one implementation. We therefore conclude that implementation specific extension functions restrict freedom by impairing code reuse.

1.1.2. Indirectly

--- Fragmenting the Community

In comparison to the C++ or Java communities, the XPDL communities are considerably smaller. The TIOBE Programming Community Index [5] for May 2015 shows that Java is the most popular programming language and that C++ is third, no XPDL languages appear in the top 100. Likewise, the PYPL Index [6] for May 2015, shows Java and C++ to hold first and fifth positions respectively, with no XPDL languages appearing in the top 16. The Redmonk Programming Language Ratings [7] for January 2015 place Java in second and C++ in joint fifth position in terms of popularity rank across StackOverflow and GitHub. From the plot produced by Redmonk we can infer that in comparison XSLT has ~57% and ~77% of the popularity rank on GitHub and Stack Overflow respectively, whilst XQuery has just ~14% and ~50%.

The last 10 years has produced an exponential growth in Open Source projects; Deshpande and Riehle reported in 2008 [8] from analysing statistics for Open Source projects over the previous 10 years that Open Source growth was doubling about every 14 months. With recent social coding services such as GitHub and BitBucket and physical events facilitated by meetup.com and others, there is likely a much greater tendency to publish even small snippets of code or utilities as open source for others to reuse.

However, when publishing XPDL code projects, if those projects depend on implementation specific extension functions, then it is often non-trivial for a user of a differing implementation to adapt the code. Even if a user can adapt the code to their implementation, if they then wish to improve it, the ability to contribute these changes back upstream is also impaired as the code bases have most likely diverged; As such further forking is implied. We conclude from this that implementation specific extension functions further fragment the XPDL communities into smaller implementation specific sub-communities by restricting portability and code sharing.

2. Prior Art

This paper is not the first work to look at improving the portability of XPDLs. In this section, previous efforts in the area of improving the portability of extension functions within one or more XPDLs are examined.

2.1. EXSLT

The EXSLT project [9] which first appeared in March 2001, at the time focused on extension functions and elements for XSLT 1.0. Arguably, XSLT 1.0 had a very limited standard library provided by the core function library of XPath 1.0 [10], with just 27 functions, augmented with 7 additional functions. EXSLT recognised that much of the XSLT community required additional extension functions and elements and that it would be desirable if such functions and elements were the same across all XSLT implementations to ensure the portability of XSLT code. EXSLT specified a set of 8 modules which include extension functions allowing XSLT developers to write portable code for tasks that were not covered by the XSLT 1.0 specification. EXSLT itself did not provide an implementation of the functions, rather it tightly defined the XSLT signatures and operational expectations and constraints of its extension functions. Any vendor may choose to implement the EXSLT modules within their XSLT implementation, however the standards set out by the EXSLT project ensure that their invocation and outcome must be the same across all implementations.

The last update to EXSLT was in October 2003, and whilst still used by many XSLT developers its relevance has decreased since the release of XSLT 2.0 [11] which expanded its standard library by adopting F+O 2.0 which provides 114 functions and 71 operators, many of which were likely inspired by EXSLT. The utility of EXSLT will likely be further reduced by the upcoming release of XSLT 3.0 which adopts F+O 3.0.

2.2. XSLT 1.1

XSLT 1.1 [12] of which the last public working draft was published in August 2001 (although the first working draft appeared in December 2000), had the stated primary goal to “improve stylesheet portability”, and included a new and comprehensive mechanism for working with extension functions in XSLT.

XSLT 1.1 like XSLT 1.0 permits the use of extension functions which are implementation defined and whose presence is testable through the use of the fn: function-

[1] The XQMVC projects' attempt at supporting both MarkLogic and eXist XQuery processors - https://code.google.com/p/xqmvc/source/browse/#svn%2Fbranches%2Fdiversify%2Fsystem%2Fprocessor%2Fimpl

available function. However, XSLT 1.1 went much further than its predecessor by introducing the `xsl:script` element which made possible the implementation of an extension function within the XSL document itself either directly in program code or by URI reference. When two distinct programming languages interact, there is always the issue of type mapping, to solve this XSLT 1.1 specified explicit DOM2 core model and argument type mappings for ECMAScript, JavaScript and Java. Extension function implementation was not limited to just ECMAScript, JavaScript or Java, however bindings and mappings for other languages were considered outside of the scope of XSLT 1.1 and were left to be implementation defined.

Providing a user of XSLT 1.1 had used extension functions implemented in either ECMAScript, JavaScript or Java, and those functions were either implemented inside an `xsl:script` element or available from a resolvable URI on the Web, then it was entirely possible to consume and/or create portable extension functions for XSLT.

The addition of `xsl:script` in XSLT 1.1 was highly controversial [13] with opponents on both sides of the debate [14][15][16]. Unfortunately, before XSLT 1.1 was finished, it was considered unworkable for several reasons by the W3C XSLT Working Group [17], and was permanently suspended to be superseded by XSLT 2.0 [11]. XSLT 2.0 adds little more than XSLT 1.0 in the area of extension functions and altogether abandons the type mapping from XSLT 1.1, clearly stating that: “The details of such type conversions are outside the scope of this specification”.

2.3. FunctX

FunctX [18] released by Priscilla Walmsley in July 2006 provides a library of over 150 useful common functions for users of XQuery and XSLT. The purpose of this library is to remove the need for users to each implement their own approaches to common tasks and to provide a code set that beginners could learn from.

FunctX provides two implementations, one in XQuery 1.0 and the other in XSLT 2.0; neither require any implementation specific extensions and as such are entirely portable and useable with any W3C compliant XQuery or XSLT processor.

The availability of FunctX has almost certainly reduced the amount of duplicated effort that otherwise would have been spent by developers working with XPDLs and also removes the temptation for vendors to provide proprietary alternatives to assist their users.

2.4. EXQuery

The EXQuery project [19] which started in October 2008 as a collaborative community effort set out with the initial goal of raising awareness of the portability problems that could result from the use of non-standard vendor extensions in XQuery. Focused solely on XQuery, the non-standard extensions which could causes issues were set out as including extension functions, indexing definitions, collections, full-text search and the URI schemes used for the XPath `fn:doc` and `fn:collection` functions.

The EXQuery project firstly approached the problem of non-standard implementation specific extension functions for XQuery, with the desire to define standard function signatures and behaviour for similar XQuery functions which appeared across several implementations.

The EXQuery project shortly abandoned its work on defining standard function signatures for XQuery extension modules in favour of the EXPath project (see Section 2.5, “EXPath”) which appeared in 2009, instead focusing on XQuery specific portability issues like server-side scripting resulting in RESTXQ [4].

The EXQuery project goes further than just defining standards documents that define intention and behaviour of a specific system, it also provides source code for a common implementation that may be adopted as the base for any implementation [20]; Although currently limited to Java the project has also expressed interest in producing C++ implementations.

2.5. EXPath

The EXPath project [21] started in January 2009 whilst independent had many similar goals to the EXQuery project. Critically, with regards to extension functions, it is recognised that defining standards for these at the lower XPath level as opposed to the XQuery or XSLT level would make them more widely applicable to any XPDL.

The EXPath project provides two types of specification for XPDLs, the first looks at the broader ecosystem of delivering XPDL applications (e.g. Application Packaging and Web), whilst the second and more widely adopted, focuses on defining extension function modules. It is this second specification type of extension function modules that are of interest to this paper.

The EXPath project to date has released three specifications for standard extension modules for XPDLs: Binary Data Handling, File System API and HTTP Client. In addition, at the time of writing there are another five extension module specifications under development which focus on: File Compression, Cryptography, Geospatial and NoSQL database access. The EXPath project like the EXSLT project focuses on defining function signatures and behaviour, albeit at the XPath as opposed to the XSLT level; again the goal being that any vendor may implement an extension module standard and that users will benefit from code portability across all implementations that support the EXPath specifications.

Whilst the EXPath project has predominantly focused on defining standards documents that specify the intention and behaviour of a number of modules of related XPath extension functions, there have been some related efforts [22][23][24] to produce common implementation code for JVM (Java Virtual Machine) based implementations.

3. Analysis

The review of prior art in Section 2, “Prior Art”, uncovers three distinct approaches to reduce the impact of non-portable extension functions in XPDLs:

1. Function Standardisation

   Specifying function libraries and the exact behaviour of those functions so that vendors may each implement the same functions. EXSLT, EXQuery and EXPath all take this approach, although EXQuery and EXPath also have some support for reducing the overhead of implementing (for the JVM) by providing common code.
2. Function Distributions

   Providing libraries of ready-to-use common functions that are implemented in a language known to every implementation. This is the approach taken by FunctX, whose implementations are provided in pure XQuery or XSLT.
3. Implementation Type Mapping

   Tightly defining the function interface and type mapping between the host language and the extension function language. This is the approach taken by XSLT 1.1, which when restricting implementation to ECMAScript, JavaScript or Java, would have enabled the creation and use of libraries of portable extension functions for XSLT. Arguably XSLT 1.1 also overlaps with the Function Distributions approach as it allows the implementation of the extension function to be embedded within the XSLT itself.

Function Standardisation is a great start, but without a majority of significant implementations [25], adoption is likely to remain a problem. Implementation can be assisted by reducing the overhead for vendors to achieve this, one such mechanism is providing common code; however, this must be inclusive to languages other than those atop the JVM (See Section 3.2, “XPDL Implementation Survey”).

Ignoring source-level interoperability for the moment, one issue with providing common code is that each implementation almost certainly has a different type system and approach to representing the XDM [26] types (amongst others). The Implementation Type Mapping approach taken by XSLT 1.1 demonstrates an interesting mechanism for solving this by explicitly laying out a type model and mappings from XSLT to the implementation language. Both the EXQuery and EXPath projects have also made embryonic attempts at defining mappings for XDM types, however both are restricted to the JVM through their use of Scala [27] and Java [28] respectively.

Function Distributions of Standardised Functions is the ultimate goal; The ability to distribute extension functions for XPDLs that will interoperate with any implementation. However, without Implementation Type Mapping and standard interfaces it is certainly impossible that an implementation of an XPDL extension function would work with an unknown vendors XPDL implementation.

Implementation Type Mapping should be considered as the foundation layer for any form of interoperability between an XPDL extension function and varying XPDL implementations. Without this every implementation of an XPDL extension function for a specific XPDL platform would require re-implementation.

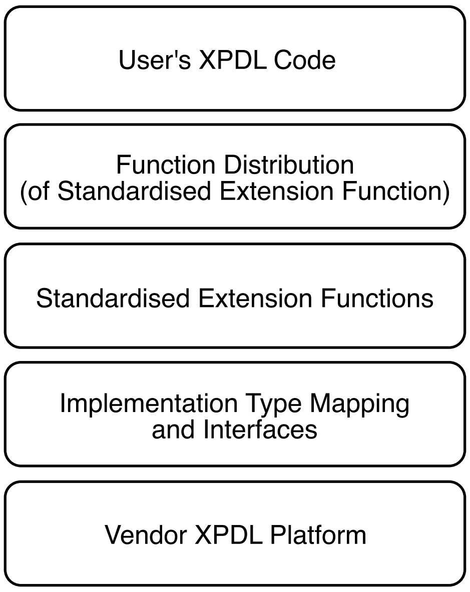
If we want to solve the problem of portable extension functions for XPDLs then it would seem that we must adopt a layered approach where we combine aspects of all three existing approaches:

1. An Implementation Type mapping needs to be created which is either at a level of abstraction that is not specific to any particular implementation language or can be losslessly implemented in a specific language, yet is still specific enough to constrain implementations to extension function standard specifications.

   Function Standardisation for extension functions needs to take place at the XPath level so as to ensure that the functions are applicable to the widest range of XPDLs.

   Standardised Functions need to be implemented according to an Implementation Type Mapping to form a Function Distribution, but in a language that allows them to be distributed in either source or binary form for any vendor implementation regardless of platform.

Figure 4. Layered Approach to Portable XPDL Extension Functions


User's XPDL Code
Function Distribution (of Standardised Extension Function)
Standardised Extension Functions
Implementation Type Mapping and Interfaces
Vendor XPDL Platform

3.1. Commonality of EXPath Standardised Extension Functions and Implementation Type Mapping

Whilst the EXPath project has provided definitions for several modules of Standardised Functions for XPDL extension functions, there has been little work by EXPath or others [29] in reducing the duplication of effort across vendors who wish to implement these functions, i.e. by exploring Implementation Type Mapping.

Consider the signature of the `file:exists` function (as shown in Example 1, "file:exists function signature") which is just one of the Standardised Functions from the EXPath File Module [30].

Example 1. file:exists function signature

```
file:exists($path as xs:string) as xs:boolean
```

When we examine the three known implementations of this for BaseX 8.1.1 [31], eXist 2.2 [32] and Saxon 9.6.0.5 [33] we find that each implementation is very similar; Each implements a host interface which represents an XPDL function, and within that implements a host function which has access to the arguments and context of the XPDL function call. A *simplified* representation of the interfaces of these processors is extracted:

Example 2. BaseX Extension Function Interface

```
interface StandardFunc {
  Item item(QueryContext qc, InputInfo ii)
      throws QueryException;
}
```

Example 3. eXist Extension Function Interface

```
interface BasicFunction {
  Sequence eval(Sequence[] args,
      Sequence contextSequence)
      throws XPathException;
}
```

Example 4. Saxon Extension Function Interface

```
interface ExtensionFunctionCall {
  SequenceIterator call(
    SequenceIterator[] arguments,
    XPathContext context) throws XPathException;
}
```

Whilst there is currently no non-Java implementation of the EXPath File Module, if we examine a similarly simple function such as XPath's `fn:year-from-date` in XQilla [34] (a C++ implementation) then we can again extract a *simplified* function interface:

Example 5. XQilla Function Interface

```
class XQFunction {
  public:
    Sequence createSequence(DynamicContext*
        context, int flags=0) const;
};
```

The similarity of these interfaces leads us to conclude that there is further room for common abstraction and that specifying a standard Implementation Type Mapping and interfaces could lead to a reduction in duplicated effort for implementers of these EXPath extension functions and therefore any XPDL extension functions.

3.2. XPDL Implementation Survey

To achieve the broadest appeal between implementers of XPDL extension functions, it cannot be assumed that primary support for Java, ECMAScript or JavaScript in itself will be acceptable to the larger community; As partially demonstrated by the failure of XSLT 1.1 (see Section 2.2, "XSLT 1.1"). Therefore, any Implementation Type Mapping or Function Distribution should be applicable to any platform and most likely not just limited to the JVM [27][28]. To inform how such a Mapping or Distribution may be implemented, we should first understand the variety of source languages of existing XPDL processors. The results of a survey of XPDL processors is presented in Table 1, “Survey of XPDL Implementations”.

Table 1. Survey of XPDL Implementations

C	C++	Haskell	Java	JavaScript	.NET	Objective-C	Pascal
libxml2[a]	Berkley DBXML (libxquery-devel)[a][b]	Haskell XML Toolbox[a]	Altova Raptor XML[a][b][c]	Frameless[a][c]	Exselt[a][c] (F#)	GDataXML[a]	Xidel[a][b]
libxslt[c]	Intel SOA Expressway XSLT[c]	HXQ[a][b]	Apache VXQuery[b]	Saxon/CE[c][d]	.NET Standard Library `XmlNode`[a]	NSXML[a][b][c]	
Saxon/C[c][d]	MarkLogic[a][b][c]		BaseX[a][b]	xpath NPM[a]	XMLPrime[a][b][c] (C#)	Panthro[a][b]	
	pugixml[a]		DataDirect XQuery[b]	XQIB[b]	xsltc[c] (C#)		
	QtXmlPatterns[a][b][c]		EMC Documentum[a][b][c]				
	Sedna[a][b]		eXist-db[a][b][c]				
	Sablotron[a][c]		GNU Qexo[a][b][c]				
	TinyXPath[a]		IBM WebSphere Application Server Feature Pack for XML[a][b][c]				
	Xalan-C++[a][c]		Qizx[a][b][c]				
	XQilla[a][b]		Saxon[a][b][c]				
	Zorba[a][b][c]		Xalan-J[a][c]				

[a] Implements XPath
[b] Implements XQuery
[c] Implements XSLT
[d] Source-level port of Saxon from Java

The survey was produced from aggregating the W3C XML Query list of implementations [35], the EXPath CG list of XPath engines [36] and relevant Google searches. The aggregate list was then reduced to those implementations for which information was still available and up-to-date. The list of programming languages for the survey was chosen based on the available implementations, and the native language of that implementation; For example with the Go programming language the approach appears to be to call xsltproc [37] (a wrapper around libxslt which itself is implemented in C), and the common approach from Python seems to be to use the lxml python wrapper [38] for libxml2 and libxslt (both themselves implemented in C).

From the survey it is clear that there is no lack of native programming language implementations of XPDL processors. The majority of implementations are written in Java and C++, which could likely be justified by the size of the C++ and Java communities (as briefly discussed in Section 1.1.2, "Indirectly").

4. Portable XPDL Extension Function Implementation

From an analysis of the current state of the art in regard to extension functions for XPDLs it can be determined that if we want to reduce the effort to implement a standardised extension function then we need to provide an Implementation Type Mapping; This allows the implementer of a standardised extension function to code to a standard interface without worrying about vendor specifics. However, such an Implementation Type Mapping needs to take into account the implementation language of the vendors XPDL processor, and this could potentially lead to an issue of fanout with many similar Implementation Type Mappings, one for each implementation language, which is far from ideal.

In addition, we have seen that providing common code can help to reduce the effort which is duplicated by each vendor implementing the same XPDL extension functions. Unfortunately this further compounds the fanout issue, as it would be very time consuming to provide common implementation code for each XPDL extension function in every known XPDL platform implementation language.

4.1. Implementation Portability

Ideally we would like to be able to specify a single Implementation Type Mapping and implement any XPDL extension function just once according to that mapping and have it execute with any vendors XPDL implementation.

Sun Microsystems coined the phrase "Write Once, Run Anywhere" (WORA) around 1996 in relation to Java [39]. Java is a high-level language which avoids platform specific implementation details by compiling to byte-code which is then executed by a virtual machine. The ability to distribute an XPDL extension function as byte-code has several attractions, such as the user not having to compile any code. However, the promise is somewhat shallow as executing Java requires a JVM to be installed on the target platform, without that the byte-code cannot be executed. For those vendors whose implementations are themselves not written in Java, they could still execute an XPDL extension function written in Java via JNI (Java Native Interface), however it may not be desirable to also force their users to install a JVM on their systems. A WORA experience for XPDL extension functions could eliminate the fanout cost of implementation, however Java is not suitable for all implementations.

If we can't achieve WORA we could instead consider falling back to a WOCA (Write Once, Compile Anywhere) approach where we distribute the Implementation Type Mapping and any common implementation source code in a single language that can be compiled on any platform. At first, C or C++ would seem a suitable choice for WOCA due to the fact that many XPDL processors are implemented in C or C++ and any XPDL processor implemented in Java could call a C or C++ implementation of an XPDL extension function via JNI. Through SWIG [40] we could also make any C or C++ XPDL extension function applicable to XPDL processors implemented in many other languages. Whilst C and C++ have many desirable properties, such as instruction set portability, compiler availability, and interoperability, the code is often highly hardware (e.g. big-endian vs little-endian), Operating System specific (e.g. Win32 API vs Posix API) and library specific (e.g Std vs Boost vs Qt etc), thus imposing a great deal of constraints to actually achieve WOCA; Therefore we would most likely still require several C or C++ variants for different systems.

Having identified issues with both, WORA where we would distribute a compiled intermediate byte-code for a VM (Virtual Machine), and WOCA where we would distribute source code which could be compiled to machine code, we are naturally led to investigate Source-to-source compilation. Source-to-source compilation allows us to take source code expressed in one language and translate it into a different target language. Regardless of the language of our initial source code, based on the results of our survey (see Table 1, "Survey of XPDL Implementations") we know that we would need to generate code for at least C++ and Java targets.

An examination of the available source-to-source compilers leads us to the Haxe Cross-platform Toolkit which fits our requirements well as it has targets for C++, C#, Java and JavaScript amongst others [41], with targets in development for C and LLVM [42]. Haxe uses a single source language also called Haxe which is similar to ECMAScript but with influences from ActionScript and C#. The Haxe toolkit also provides a cross-target standard library for the Haxe language. With Haxe it seems entirely possible that we can entirely eliminate the fanout issue of implementation by: 1) specifying an Implementation Type Mapping between XDM and the Haxe Language and 2) going further than providing common code for the implementation of an XPDL extension function, instead implement the entire function according to the Implementation Type Mapping in the Haxe language itself. The vendor of an XPDL processor could then take the Haxe code and compile it to the implementation language of their processor to produce a distribution of standardised extension functions; This role could also perhaps also be taken by an intermediary such as the EXPath project.

4.2. Implementation Type Mapping for Haxe

We have developed a partial Implementation Type Mapping between XDM and Haxe (the source code is available from the EXQuery GitHub repository [43]) that provides enough functionality to allow implementation of a single EXPath extension function: the `file:exists` function (as discussed in Section 3.1, "Commonality of EXPath Standardised Extension Functions and Implementation Type Mapping"). In addition to implementing Type Mappings for the XDM, we also need to implement interfaces to map the XPath concept of calling a function and passing arguments.

To produce interfaces for mapping the concept of an XPDL extension function, which is effectively an externally declared function in terms of the XPath specification, we need to understand both how a function is declared and subsequently called. A function call in XPath 3.0 [44]is made up of the several constructs expressed in EBNF (Extended Backus-Naur Form) as reproduced in Example 6, "XPath 3.0 Function Call EBNF".

Example 6. XPath 3.0 Function Call EBNF

```
FunctionCall ::= EQName ArgumentList
ArgumentList ::= "(" (Argument ("," Argument)*)? ")"
```

XPath only specifies how to call a function, it does not specify how to define a function, so here we have opted to follow the XQuery 3.0 specification which does specify how to define a function [45]. A function definition in XQuery 3.0 is made up of the EBNF constructs as reproduced in Example 7, "XQuery 3.0 Function Declaration EBNF".

Example 7. XQuery 3.0 Function Declaration EBNF

```
FunctionDecl        ::=    "function" EQName "(" ParamList? ")"
                           ("as" SequenceType)? (FunctionBody | "external")
ParamList           ::=    Param ("," Param)*
Param               ::=    "$" EQName TypeDeclaration?
FunctionBody        ::=    EnclosedExpr

TypeDeclaration     ::=    "as" SequenceType
SequenceType        ::=    ("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)
OccurrenceIndicator ::=    "?" | "*" | "+"

EQName              ::=    QName | URIQualifiedName
```

As our XPDL extension functions will always be external in nature, we can transform the `FunctionDecl` construct, to extract a `FunctionSignature`. As our functions are always external we can also ignore the `FunctionBody` construct as this will instead be implemented in Haxe code. As our extension functions are always the target of a function call, we can reduce `EQName` to `QName`. All of the other constructs can be translated into interfaces for our Implementation Type Mapping to the Haxe language.

Example 8. Function Type Mapping for Haxe

```
package xpdl.extension.xpath;

interface Function {
  public function signature() : FunctionSignature;
  public function eval(
    arguments: Array<Argument>,
    context: Context) : Sequence;
}

class FunctionSignature {
  var name: QName;
  var returnType: SequenceType;
  var paramLists: Array<Array<Param>>;

  public function new(name, returnType, paramLists)
  {
    this.name = name;
    this.returnType = returnType;
    this.paramLists = paramLists;
  }
}
```

Example 8, “Function Type Mapping for Haxe” shows part of our Implementation Type Mapping for functions (full code in Appendix A, *Function Type Mapping in Haxe*). The mapping is direct enough that anyone with a knowledge of the relevant EBNF constructs of XPath and XQuery can understand the simplicity of the function type mapping between an XPDL extension function and Haxe.

Yet, being able to specify the interface for a function is not enough; we also need to create Implementation Type Mappings for the XDM types. Whilst ultimately we need to map all XDM types, within this paper we focus exclusively on the types required for our partial implementation, i.e. those types needed by the function signature of the `file:exists` function (see Example 1, “file:exists function signature”).

The signature of `file:exists` shows how we only need to create type mappings for `xs:string` and `xs:boolean` to appropriate Haxe types. We take the approach to encapsulate the Haxe types inside representations of the XDM types, as we believe that this will provide greater flexibility for future changes.

Example 9. Haxe Implementation Type Mapping for `xs:string` and `xs:boolean`

```
package xpdl.xdm;

import xpdl.HaxeTypes.HString;

interface Item {
  public function stringValue() : xpdl.xdm.String;
}

interface AnyType {
}

interface AnyAtomicType extends Item extends AnyType
{
}

class Boolean implements AnyAtomicType {
  var value: Bool;

  public function new(value) {
    this.value = value;
  }

  public function stringValue() {
    return new xpdl.xdm.String(Std.string(value));
  }

  public function haxe() {
    return value;
  }
}

class String implements AnyAtomicType {
  var value: HString;

  public function new(value) {
    this.value = value;
  }

  public function stringValue() {
    return this;
  }

  public function haxe() {
    return value;
  }
}
```

There is certainly an argument concerning whether we should actually implement the `xs:string` and `xs:boolean` XDM types in Haxe by providing classes, or whether we should simply provide interfaces for a vendor to implement. Further research through a survey of vendor requirements would be required to answer this definitively. For the purposes of this paper, classes have been implemented for these basic atomic types.

4.3. Implementation of a portable `file:exists`

Given the function type mapping and Implementation Type Mapping that we have defined in Section 4.2, “Implementation Type Mapping for Haxe” we can now implement our first truly portable XPDL extension function by making use of Haxe.

Example 10. Implementation of the `file:exists` function in Haxe

```
class ExistsFunction implements Function {
  private static var sig = new FunctionSignature(
    new QName(
      "exists",
       "http://expath.org/ns/file",
       "file"),

    new SequenceType(
      Some(new ItemOccurrence(Boolean))),
      [
        [ new Param(new QName("path"),
          new SequenceType(Some(
            new ItemOccurrence(
              xpdl.xdm.Item.String))))
        ]
      ]
  );

  public function new() {}

  public function signature() {
    return sig;
  }

  public function eval(
    arguments : Array<Argument>,
    context: Context)
  {
    var path = arguments[0].getArgument().
               iterator().next().
               stringValue().haxe();
    var exists = FileSystem.exists(path);
    return new ArraySequence( [
                          new Boolean(exists) ] );
  }
}
```

Example 10, “Implementation of the `file:exists` function in Haxe” shows the main concern of our implementation of file:exists (full code in Appendix B, *`file:exists` implementation in Haxe*).

4.4. XPDL Processor Vendor Implementation

We have defined both an Implementation Type Mapping for XDM and associated interfaces for functions in the Haxe language, and subsequently created an implementation of an XPDL Extension Function, the EXPath File Module's `file:exists` function in Haxe written for the type mapping and interfaces. However, such an XPDL extension function implementation is still not useful without vendor support, as the Haxe code must be compiled to the XPDL processors implementation language and made available to the XPDL from the processor.

As a proof-of-concept we have compiled the Haxe code to both Java source and byte code using the Haxe compiler and modified eXist-db to support XPDL Extension Functions (the source code is available from the eXist GitHub repository [46]). Modifying eXist to recognise any XPDL Extension Function Module and make its functions available as extension functions in XQuery was achieved in approximately 300 lines of Java code; For the partial implementation, only support for the XDM types `xs:string` and `xs:boolean` was required, but we recognise that the amount of code required will increase as further XDM types are mapped.

Whilst modifying eXist to support XPDL Extension Function Modules, we recognised that there were several different approaches that could be taken to implement a mapping between eXists own XDM model and our Haxe XDM model. These approaches, whilst not exhaustive, will most likely also apply to other XPDL processors, and so we briefly enumerate them here for reference:

1. Mapping of Haxe XDM types to eXist XDM types and vice-versa. This could be achieved either statically or dynamically, or through a combination of both approaches. A static implementation would be coded in source, whereas a dynamic mapping would be generated as needed at runtime.
2. Modify eXists XDM classes to implement the Haxe XDM interfaces. This would allow us a single XDM model and we could transparently pass eXists XDM types into the Haxe compiled functions.
3. Inversion of Responsibility, using byte-code generation at runtime to have the Haxe XDM interfaces implement the eXist XDM interfaces. This would make the Haxe XDM model compatible with the eXist XDM model, so that Haxe XDM types could be used transparently by eXist.

For expediency in creating the proof-of-concept modifications in eXist, we used a static mapping of XDM types in combination with a dynamic mapping of functions. For the dynamic mapping of functions we used byte-code generation to generate classes at runtime to bridge between eXist's concept of an extension function and our Haxe XPDL extension function.

5. Summary and Conclusion

Having explicitly laid out the issues with portability of XPDLs in regard to non-standard extension functions (see Section 1, “Introduction”), we have reviewed both the past and current works on improving the status-quo (see Section 2, “Prior Art”), and performed a critical analysis of these approaches (see Section 3, “Analysis”). From our critical analysis we have identified three common approaches to improving portability: Function Standardisation, Function Distributions and Implementation Type Mappings. To resolve the issue of portability with respect to extension functions for XPDL users, we argue that there have to be solutions in place for all three approaches and that these must work together holistically.

Function Standardisation is already well supported by the EXPath project, a community oriented organisation which is vendor agnostic and has already proven itself capable of coordinating stakeholders to define modules of common XPDL extension functions and their behaviour.

Function Distributions require implementations of extension functions which they can then make available. These extension functions themselves however need to be portable, so that the resultant XPDL code that uses them is also portable. Arguably the FunctX distribution was successful because its extension functions were portable, as they were written in XSLT and XQuery, making them useable on any vendors XQuery or XSLT processor. For more complex extension functions which cannot be expressed in an XPDL, a portable Implementation Type Mapping is a required enabler to creating Function Distributions.

We have presented a solution for a portable Implementation Type Mapping through the use of source-to-source compilation (section Section 4.2, “Implementation Type Mapping for Haxe”), and implemented what we believe to be the first truly portal extension function for an XPDL whilst using a non-XPDL to implement the function (section Section 4.3, “Implementation of a portable `file:exists`”). Further, we have created a proof-of-concept by integrated support for the Implementation Type Mapping into a real-world XPDL processor (section Section 4.4, “XPDL Processor Vendor Implementation”).

The use of Haxe for source-to-source compilation is an interesting and novel approach towards solving the issue of portable extension functions for XPDLs. Whilst it does not eliminate the need of some effort by XPDL processor vendors to support it, it greatly reduces the work to a one-off exercise to support a portable Implementation Type Mapping. In this manner an XPDL extension function written once in Haxe, when compiled will work on any XPDL processor (in a target language supported by Haxe) which implements the Implementation Type Mapping. For authors of portable XPDL extension functions, rather than just creating a standardisation of a function module through the EXPath project and waiting for each vendor to implement this, they can now also write a single implementation which can be adopted quickly by the widest possible audience.

5.1. Future Work

The Implementation Type Mapping and the proof-of-concept currently only implement the basic XDM types required for this paper, a full XDM Implementation Type Mapping in Haxe is desirable and would likely provide new insights into creating a portable Implementation Type Mapping.

The target code generated by the Haxe compiler can be somewhat verbose and even confusing to the consuming developer. It is possible to tune the code generation by tightly controlling DCE (Dead Code Elimination) and native vs reflective generation. The use of various Haxe language annotations should be investigated to achieve the generation of cleaner target code.

The options for implementation approach discussed in Section 4.4, “XPDL Processor Vendor Implementation” are likely coupled to the observation at the end of Section 4.3, “Implementation of a portable `file:exists`” over how concrete the XDM Implementation Type Mapping should be. Further research is required in this area, likely informed by creating more proof-of-concept integrations with several other XPDL processors.

Whilst Haxe does not favour Java as a target above any other, a non-Java proof-of-concept would reinforce our argument that Haxe allows us to create a portable implementation. A C++ integration for the Zorba XQuery processor could perhaps serve as a suitable reinforcement.

A. Function Type Mapping in Haxe

```
package xpdl.extension.xpath;

interface Function {
  public function signature() : FunctionSignature;
  public function eval(arguments: Array<Argument>, context: Context) : Sequence;
}

interface Context {
}

class FunctionSignature {
  var name: QName;
  var returnType: SequenceType;
  var paramLists: Array<Array<Param>>;

  public function new(name, returnType, paramLists) {
    this.name = name;
    this.returnType = returnType;
    this.paramLists = paramLists;
  }
}

class QName {
  public static var NULL_NS_URI = "";
  public static var DEFAULT_NS_PREFIX = "";

  var localPart : String;
  var namespaceUri(default, null) : String;
  var prefix(default, null) : String;

  public function new(localPart, ?namespaceUri, ?prefix) {
    this.localPart = localPart;
    this.namespaceUri = (namespaceUri == null) ? NULL_NS_URI : namespaceUri;
    this.prefix = (prefix == null) ? DEFAULT_NS_PREFIX : prefix;
  }
}

class SequenceType {
  var type: Option<ItemOccurrence>; //None indicates empty-sequence()

  public function new(type) {
    this.type = type;
  }
}

class ItemOccurrence {
  var itemType: Class<Item>;
  var occurrenceIndicator: OccurrenceIndicator;

  public function new(itemType, ?occurenceIndicator) {
    this.itemType = itemType;
    this.occurrenceIndicator = (occurrenceIndicator == null) ?
                               OccurrenceIndicator.ONE : occurrenceIndicator;
  }
}

enum OccurrenceIndicator {
  ZERO_OR_ONE;    // ?
  ONE;            // implementation detail
  ONE_OR_MORE;    // +
  ZERO_OR_MORE;   // *
}

class Param {
  var name: QName;
  var type: SequenceType;

  public function new(name, type) {
    this.name = name;
    this.type = type;
  }
}

interface Argument {
  public function getArgument() : Sequence;
}

interface Module {
  public function name() : String;
  public function description() : String;
  public function functions() : List<Class<Function>>;
}
```

B. `file:exists` implementation in Haxe

```
package example.expath.file;

import xpdl.extension.Module;
import xpdl.extension.xpath.*;
import xpdl.extension.xpath.SequenceType.ItemOccurrence;
import xpdl.xdm.Sequence;
import xpdl.xdm.Item.Item;
import xpdl.xdm.Item.Boolean;
import sys.FileSystem;

class ExistsFunction implements Function {

  private static var sig = new FunctionSignature(
    new QName("exists", FileModule.NAMESPACE, FileModule.PREFIX),
    new SequenceType(Some(new ItemOccurrence(Boolean))),
    [
      [ new Param(new QName("path"), new SequenceType(Some(new ItemOccurrence(xpdl.xdm.Item.String)))) ]
    ]
  );

  public function new() {}

  public function signature() {
    return sig;
  }

  public function eval(arguments : Array<Argument>, context: Context) {
    var path = arguments[0].getArgument().iterator().next().stringValue().haxe();
    var exists = FileSystem.exists(path);
    return new ArraySequence( [ new Boolean(exists) ] );
  }
}
```

```
class ArraySequence implements Sequence {
  var items: Array<Item>;

  public function new(items: Array<Item>) {
    this.items = items;
  }

  public function iterator() {
    return new ArraySequenceIterator(items.iterator());
  }
}

class ArraySequenceIterator implements xpdl.support.Iterator<Item> {
  var it: Iterator<Item>;

  public function new(it) {
    this.it = it;
  }

  public function hasNext() {
    return it.hasNext();
  }

  public function next() {
    return it.next();
  }
}

class FileModule implements Module {
  @final public static var NAMESPACE = "http://expath.org/ns/file";
  @final public static var PREFIX = "file";

  public function name() {
    return "FileModule.hx";
  }

  public function description() {
    return "Haxe implementation of the EXPath File Module";
  }

  public function functions() : List<Class<Function>> {
    var lst = new List<Class<Function>>();
    lst.add(ExistsFunction);
    return lst;
  }
}
```

Bibliography

[1] *XPath and XQuery Functions and Operators 3.0*. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-functions-30/

[2] *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. W3C. 14 December 2010.
http://www.w3.org/TR/xpath-functions/

[3] *Unifying XSLT Extensions*. xml.com. Leigh Dodds. 29 March 2000.
http://www.xml.com/pub/a/2000/03/29/deviant/index.html

[4] *RESTful XQuery*. Standardised XQuery 3.0 Annotations for REST. XML Prague. . XML Prague. Adam Retter. 12 February 2012.
http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[5] *TIOBE Programming Community Index*. TIOBE Software. May 2015.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[6] *PYPL PopularitY of Programming Language Index*. Pierre Carbonnelle. May 2015.
http://pypl.github.io/PYPL.html

[7] *Redmonk Programming Language Ratings*. RedMonk. January 2015.
https://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

[8] *The Total Growth of Open Source*. Amit Deshpande and Dirk Riehle. SAP Research, SAP Labs LLC. The Fourth Conference on Open Source Systems (OSS 2008). . Springer Verlag. 197-209. 2008.

[9] *The EXSLT Project*.
http://www.exslt.org

[10] *XML Path Language (XPath) Version 1.0*. W3C. 16 November 1999.
http://www.w3.org/TR/xpath/

[11] *XSL Transformations (XSLT) Version 2.0*. W3C. 23 January 2007.
http://www.w3.org/TR/xslt20/

[12] *XSL Transformations (XSLT) Version 1.1*. W3C. 24 August 2001.
http://www.w3.org/TR/xslt11/

[13] *XSLT Extensions Revisited*. xml.com. Leigh Dodds. 14 February 2001.
http://www.xml.com/pub/a/2001/02/14/deviant.html

[14] *Re: [xsl] XSLT 1.1 comments*. W3C xsl-editors Mailing List. Michael Kay. 11 February 2001.
https://lists.w3.org/Archives/Public/xsl-editors/2001JanMar/0087.html

[15] *Re: [xsl] XSLT 1.1 comments*. xsl-list Mailing List. Steve Muench. 12 February 2001.
http://markmail.org/message/5fpk5gecmslzepdy

[16] *Petition to withdraw xsl:script from XSLT 1.1*. xml-dev Mailing List. Uche Ogbuji. 1 March 2001.
http://markmail.org/thread/tquj4ozsax3pjkm2

[17] *Minutes of the Face-to-face meeting of the W3C XQuery Working Group in Bangkok*. W3C XQuery Working Group. January 2001.
https://lists.w3.org/Archives/Member/w3c-xsl-wg/2001Feb/0083.html

[18] *FunctX*. Datypic. Priscilla Walmsley. July 2006.
http://www.functx.com

[19] *EXQuery*. Collaboratively Defining Open Standards for Portable XQuery Applications. EXQuery. October 2008.
http://www.exquery.org

[20] *EXQuery Common Implementation Source Code*. The EXQuery Project.
https://github.com/exquery/exquery

[21] *EXPath*. Collaboratively Defining Open Standards for Portable XPath Extensions. EXPath. January 2009.
http://www.expath.org

[22] *EXPath HTTP Client Module Common Implementation Source Code*. Florent Georges.
https://github.com/fgeorges/expath-http-client-java

[23] *EXPath File Module Common Implementation Source Code*. The EXQuery Project. Adam Retter.
https://github.com/exquery/exquery/tree/master/expath-file-module

[24] *EXPath File Module Common Implementation Source Code*. Florent Georges.
https://github.com/fgeorges/expath-file-java

[25] *Implementations of EXPath Modules*. W3C EXPath Community Group. 8 May 2015.
https://www.w3.org/community/expath/wiki/Modules#Implementation

[26] *XQuery and XPath Data Model 3.0*. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-datamodel-30/

[27] *API for XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. The EXQuery Project. Adam Retter. 12 February 2015.
https://github.com/exquery/exquery/tree/xdm-model/xdm

[28] *XML Model for Java*. The EXPath Project. Florent Georges. 6 January 2015.
https://github.com/expath/tools-java

[29] *Minutes of Face-to-face meeting of the W3C EXPath Community Group in Prague*. W3C EXPath Community Group. 12 February 2015.
https://lists.w3.org/Archives/Public/public-expath/2015Feb/0005.html

[30] *File Module 1.0*. W3C EXPath Community Group. 20 February 2015.
http://expath.org/spec/file

[31] *BaseX 8.1.1 implementation of EXPath file:exists function*. BaseX. 9 January 2015.
https://github.com/BaseXdb/basex/blob/8.1.1/basex-core/src/main/java/org/basex/query/func/file/FileExists.java

[32] *eXist implementation of EXPath file:exists function*. Adam Retter. 21 February 2015.
https://github.com/adamretter/exist-expath-file-module/blob/master/src/main/scala/org/exist/expath/module/file/FileModule.scala

[33] *Saxon implementation of EXPath file:exists function*. Florent Georges. 16 January 2015.
https://github.com/fgeorges/expath-file-java/blob/master/file-saxon/src/org/expath/file/saxon/props/Exists.java

[34] *XQilla implementation of XPath fn:date-from-year function*. XQilla. 16 November 2011.
http://xqilla.hg.sourceforge.net/hgweb/xqilla/xqilla/file/6468e5681607/include/xqilla/functions/FunctionYearFromDate.hpp

[35] *W3C XML Query*. Implementations. W3C XQuery Working Group.
http://www.w3.org/XML/Query/#implementation

[36] *W3C EXPath Community Group Wiki*. XPath Engines. W3C EXPath Community Group. 8 May 2015.
https://www.w3.org/community/expath/wiki/Engine

[37] *Using XSLT with Go*. William Kennedy. 3 November 2013.
http://www.goinggo.net/2013/11/using-xslt-with-go.html

[38] *lxml*. XML Toolkit for Python.
http://lxml.de/

[39] *Write once, run anywhere*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Write_once,_run_anywhere

[40] *SWIG*. Simplified Wrapper and Interface Generator.
http://www.swig.org/

[41] *Haxe Compiler Targets*.
http://haxe.org/documentation/introduction/compiler-targets.html

[42] *Experimental C and LLVM Targets for Haxe*.
https://github.com/waneck/haxe-genc

[43] *Source code for Implementation Type Mapping of XPDL Extension Functions in Haxe*. The EXQuery Project. Adam Retter. 12 May 2015.
https://github.com/exquery/xpdl-extension-lib

[44] *XML Path Language (XPath) 3.0*. Static Function Calls. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-30/#id-function-call

[45] *XQuery 3.0: An XML Query Language*. Function Declaration. W3C. 8 April 2014.
http://www.w3.org/TR/xquery-30/#FunctionDecln

[46] *Source code of XPDL Extension Functions integration with eXist*. Adam Retter. 12 May 2015.
https://github.com/eXist-db/exist/tree/xpdl-extensions/src/org/exist/xpdl

Validating XSL-FO with Relax NG and Schematron

Tony Graham

Antenna House, Inc.

`<tgraham@antenna.co.jp>`
`<tony@antennahouse.com>`

Abstract

XSL-FO defies conventional validation, so much so that it hasn't been done successfully before now. This paper describes a combination of hand-written and auto-generated Relax NG plus hand-written and auto-generated Schematron that can validate XSL-FO markup. The project is available on GitHub at https://github.com/AntennaHouse/focheck

Keywords: XSL-FO, Relax NG, Schematron

1. Introduction

XSL-FO documents are typically generated as the result of an XSLT transformation and are rarely edited by hand. However, validating generated XSL-FO markup is useful as a check of the correctness of the transformation. Also, people do edit XSL-FO by hand either when prototyping the XSL-FO markup that will later be generated using XSLT or when debugging generated XSL-FO. Being able to validate the XSL-FO in an XML editor helps in both scenarios.

Validating XSL-FO is not easy because:

- Constraints in the definitions of FOs are hard or impossible to express in structure-checking schema languages.
- Some FOs can appear almost anywhere in an XSL-FO document but, equally, cannot appear where they are not allowed.
- The properties of an FO are expressed in the XML as attributes of the XML element representing the FO, but inherited properties[1] are allowed to appear on any FO, not just on the FOs for which they are defined.
- While the XSL 1.1 Recommendation [1] defines the allowed values of properties, most properties can contain expressions in the expression language[2] that is defined in the spec, so determining the correctness of an attribute in the XML initially requires evaluating it.

A schema for XSL-FO was in the requirements for XSL 2.0 [2], but the design of XSL was shaped by the requirements of formatting rather than any requirement to conform to a schema language. The result has been that XSL-FO was hard to validate except by running it through an FO formatter. Systems for checking the formatted result exist [3] [4], but they require usable input.

Schemas for XSL-FO do exist, including several from RenderX [5] and the schema that is provided by the oXygen XML Editor[3], but they do not cover XSL 1.1, they each cut corners in their models for element content, and they do not properly evaluate property value expressions. Also, none of them cover the numerous extensions supported by Antenna House AH Formatter.

One of the validation methods tried by RenderX but noted as longer used [5] is a validator written in XSLT[4]. A 2004 paper [6] by Alexander Peshkov of RenderX describes the XSLT approach as powerful but requiring more resources than, for example, DTD validation and also not being suitable for "visual XSL-FO editors or document builders." That paper then describes a Relax NG schema that includes a limited ability to handle property value expressions.

The approach taken by Antenna House combines Relax NG and Schematron for detailed validation of the XSL-FO. The Relax NG handles structural validation and is, we believe, more correct than pre-existing schemas. The Schematron handles the additional constraints that cannot be expressed in Relax NG. The Schematron parses property value expressions using an XSLT-based parser generated by the REx parser generator [7] plus an XSLT library for reducing the parse tree to XSL-FO datatypes.

[1] Section 5.1.4 Inheritance, in Extensible Stylesheet Language (XSL) Version 1.1 [1]
[2] Section 5.9 Expressions, in Extensible Stylesheet Language (XSL) Version 1.1 [1]
[3] http://www.oxygenxml.com/
[4] http://xep.xattic.com/xep/resources/validators/folint/folint.xsl

The Relax NG and Schematron is available on GitHub (https://github.com/AntennaHouse/focheck) and you can download an oXygen add-on framework for XSL-FO validation directly from the GitHub page[1].

We also considered wiring the Schematron directly to the expression parser built into an FO formatter through XSLT extension functions. However, doing the interfacing would have been a non-trivial task, plus the Antenna House AH Formatter is a native application on each platform and wouldn't be as portable as purely-XSLT Schematron.

2. Why Relax NG?

Three features of Relax NG made it the best choice for the schema:

- Non-deterministic content models
- Easy extensibility by redefining or extending patterns
- Ability to interleave elements in content models

3. Why Relax NG Compact Syntax?

The schema is written in Relax NG compact syntax and then converted into the XML syntax for use with oXygen. It is not written directly in the XML syntax for multiple reasons:

- It was easy to write and check the initial patterns that would be replicated by the programmatically generated schema.
- Relax NG compact syntax closely matches the syntax of the content models in the spec, which made it easier to include them in the generated schema.
- Reading the generated schema to check it is easier with the compact syntax than with the XML syntax.
- The handwritten parts, including the schema module defining Antenna House extensions, were only ever going to be written in the compact syntax.

4. Generating the Relax NG and Schematron

The bulk of the FO portion of the Relax NG and Schematron is generated by transforming the XML source[2] for the XSL 1.1 Recommendation using XSLT. The XML is consistent enough for this to be feasible: it's not the first time that I've generated code from the XML, nor am I the only person to have done it.

5. Validating FOs

At first glance, this seems quite straightforward to do using Relax NG: the content models are in the spec, where every FO is in a separate div3 element and the FO's content model is easy to identify:

```
<div3 id="fo_block">
  <head>fo:block</head>

  <p>
    <emph>Common Usage:</emph>
  </p>

  <p>The fo:block formatting object is commonly
  used for formatting paragraphs, titles,
  headlines, figure and table captions, etc.</p>
...
  <p>
    <emph>Contents:</emph>
  </p>
<eg xml:space="preserve">
(#PCDATA|<loc href="#inline.fo.list"
xlink:type="simple"
xlink:show="replace" xlink:actuate="onRequest"
xmlns:xlink="http://www.w3.org/1999/xlink">%inline;
</loc>|<loc
href="#block.fo.list" xlink:type="simple"
xlink:show="replace"
xlink:actuate="onRequest"
xmlns:xlink="http://www.w3.org/1999/xlink">%block;
</loc>)*
</eg>

<p>In addition this formatting object may have a
sequence of zero or more fo:markers as its initial
children, optionally followed by an
fo:initial-property-set.</p>
```

The `%inline;` and `%block;` behave like parameter entities in a DTD, though there isn't a DTD, and their expansions are given in the text of the Recommendation[3]:

[1] Oxygen add-on hosted on GitHub - http://inasmuch.as/2013/10/23/oxygen-add-on-hosted-on-github/
[2] http://www.w3.org/TR/2006/REC-xsl11-20061205/xslspec.xml
[3] 6.2 Formatting Object Content, in Extensible Stylesheet Language (XSL) Version 1.1 [1]

The parameter entity, "%block;" in the content models below, contains the following formatting objects:
block
block-container
table-and-caption
table list-block
The parameter entity, "%inline;" in the content models below, contains the following formatting objects:
bidi-override
character
external-graphic
instream-foreign-object
inline
inline-container
leader
page-number
page-number-citation
page-number-citation-last
scaling-value-citation
basic-link
multi-toggle
index-page-citation-list

So far, so good; the corresponding Relax NG pattern generated for `fo:block` looks like:

```
fo_block.model =
    (text|inline.fo.list|block.fo.list)*
```

where `inline.fo.list` and `block.fo.list` are defined in literal text that is included in the generated schema.

However, the XSL 1.1 Recommendation defines `neutral` and `out-of-line` classes of FOs that can appear anywhere where `#PCDATA`, `%inline;` or `%block;` is allowed in FO content models (although additional constraints apply). Handling those simply required matching on `#PCDATA`, `%inline;`, or `%block;` in the content models in the spec. The generated pattern for `fo:block` then becomes:

```
fo_block.model =
   (text|inline.fo.list|block.fo.list |
    neutral.fo.list)* &
   (inline.out-of-line.fo.list)*
```

The `neutral` and `out-of-line` FO classes were also in XSL 1.0. XSL 1.1 added `fo:change-bar-begin` and `fo:change-bar-end` as `point` FOs that “may be used anywhere as a descendant of fo:flow or fo:static-content” [FOC]. Since that couldn’t be handled by just looking at either the FO or its content model, the XSLT contains a list of FOs to which to not add the `point` FOs:

```
<xsl:variable name="no-point-fos"
    select="'root layout-master-set declarations
bookmark-tree page-sequence page-sequence-wrapper
color-profile title folio-prefix folio-suffix
simple-page-master page-sequence-master flow-map
single-page-master-reference
repeatable-page-master-reference
repeatable-page-master-alternatives
conditional-page-master-reference region-body
region-before region-after region-start
region-end flow-assignment flow-source-list
flow-target-list flow-name-specifier
region-name-specifier'" as="xs:string"/>

<xsl:variable
     name="no-point-fo-list"
     select="tokenize($no-point-fos, '\s+')"
     as="xs:string+"/>
```

such that every FO *not* in the list will allow `fo:change-bar-begin` and `fo:change-bar-end`, so the model for `fo:block` becomes:

```
fo_block.model =
   ( ( (text|inline.fo.list|block.fo.list |
         neutral.fo.list)* &
       (inline.out-of-line.fo.list)* ) &
     (point.fo.list)* )
```

But there’s also the additional constraints about allowing `fo:marker` and `fo:initial-property-set` as initial children of an `fo:block`. This is handled adding those elements to content models only where the significant “zero or more fo:markers” or “optionally followed by an fo:initial-property-set” text occurs in the FO’s definition. The complete, and completely auto-generated, model for `fo:block` is:

```
fo_block.model =
   fo_marker*,
   fo_initial-property-set?,
   ( ( (text|inline.fo.list|block.fo.list |
         neutral.fo.list)* &
       (inline.out-of-line.fo.list)* ) &
     (point.fo.list)* )
```

`fo:block` is actually a quite straightforward FO to validate. `fo:footnote`[1], for example, would appear to be even easier, since its content model is:

```
(inline,footnote-body)
```

The `neutral` and `out-of-line` FOs don’t apply, but the `point` FOs do, so the generated model is:

```
fo_footnote.model =
    ( (fo_inline,fo_footnote-body) &
      (point.fo.list)* )
```

[1] Section 6.12.3 fo:footnote, in Extensible Stylesheet Language (XSL) Version 1.1 [1]

If only it was that simple. There are additional constraints in the text of the XSL 1.1 Recommendation:

> It is an error if the fo:footnote occurs as a descendant of a flow that is not assigned to one or more region-body regions, or of an fo:block-container that generates absolutely positioned areas...
>
> ...
>
> An fo:footnote is not permitted to have an fo:float, fo:footnote, or fo:marker as a descendant.
>
> Additionally, an fo:footnote is not permitted to have as a descendant an fo:block-container that generates an absolutely positioned area.

From its content model, `fo:retrieve-table-marker`[1] (added in XSL 1.1) would appear to be even simpler:

```
EMPTY
```

producing:

```
fo_retrieve-table-marker.model = ( empty )
```

but it has its own constraints:

> An fo:retrieve-table-marker is only permitted as the descendant of an fo:table-header or fo:table-footer or as a child of fo:table in a position where fo:table-header or fo:table-footer is permitted.

These are the sorts of constraints that can't be expressed in Relax NG (except by exploding the size of the schema through making separate versions of every FO that can appear in each constrained context) but that are well suited to Schematron. There aren't enough of these constraints that are expressed in a consistent way for it to be worthwhile autogenerating them, so they have to be written by hand. For example, this is the `fo:retrieve-table-marker` constraint as a Schematron rule:

```
<rule context="fo:retrieve-table-marker">
  <assert test="
exists(ancestor::fo:table-header) or
exists(ancestor::fo:table-footer) or
(exists(parent::fo:table) and
 empty(preceding-sibling::fo:table-body) and
 empty(following-sibling::fo:table-column))">An
 fo:retrieve-table-marker is only permitted as
 the descendant of an fo:table-header or
 fo:table-footer or as a child of fo:table in a
 position where fo:table-header or
 fo:table-footer is permitted.</assert>
</rule>
```

6. Validating properties

Generating Relax NG patterns for the properties is straightforward. The XML for each FO includes a list of its allowed properties or groups of properties. For example, for `fo:footnote`[fo-footnote]:

```
<p>
 <emph>The following properties apply to this
 formatting object:</emph>
</p>
<slist>
 <sitem>
   <specref ref="common-accessibility-properties"/>
 </sitem>
 <sitem>
   <specref ref="id"/>
 </sitem>
 <sitem>
   <specref ref="index-class"/>
 </sitem>
 <sitem>
   <specref ref="index-key"/>
 </sitem>
</slist>
```

Here, the `specref/@ref` refers either to a `div2` containing the `div3` for multiple properties or to a `div3` for a property. The `div2` each generate a named pattern, so the pattern for the properties of `fo:footnote` is:

```
fo_footnote.attlist =
    common-accessibility-properties,
    id,
    index-class,
    index-key
```

where `common-accessibility-properties` is:

```
common-accessibility-properties =
    source-document,
    role
```

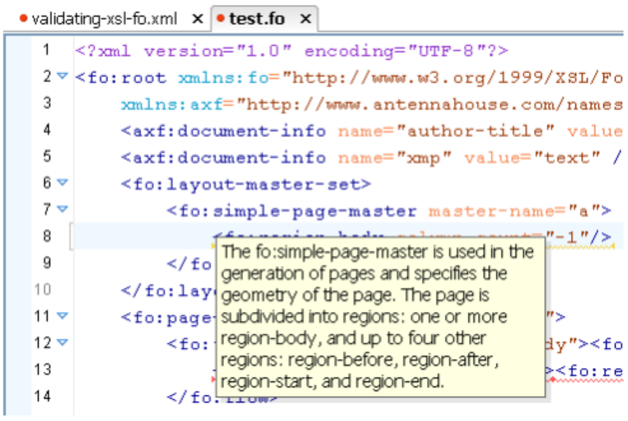
Because, as stated previously, the properties are evaluated as expressions, each property is generated in the Relax NG as containing only text. For example, for the `column-count` property:

```
column-count =
    ## <number> | inherit
    attribute column-count { text }?
```

where `##` begins an annotation that is the property's allowed value as extracted from the XML for the XSL 1.1 spec (and, similarly, annotations for FOs are also extracted from the spec). The annotations appear in oXygen as tool-tips, as shown in Figure 1, "Tool-tip in oXygen".

[1] Section 6.13.7 fo:retrieve-table-marker, in Extensible Stylesheet Language (XSL) Version 1.1 [1]

Figure 1. Tool-tip in oXygen



```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Fo
    xmlns:axf="http://www.antennahouse.com/names
    <axf:document-info name="author-title" value
    <axf:document-info name="xmp" value="text" /
    <fo:layout-master-set>
        <fo:simple-page-master master-name="a">
            <fo:region-body column-count="-1"/>
        </fo
    </fo:lay
    <fo:page                                    ">
        <fo:                                    dy"><fo
                                                ><fo:re
        </fo:flow>
```

The fo:simple-page-master is used in the generation of pages and specifies the geometry of the page. The page is subdivided into regions: one or more region-body, and up to four other regions: region-before, region-after, region-start, and region-end.

Whether or not a particular property is required for an FO is not easy to automatically determine from the XML for the XSL 1.1 spec, so that is enforced by the Schematron, not by the Relax NG.

Some properties values are described in terms of compound datatypes[1], which are expressed in the XML as multiple attributes. For example a "space-before" property may be specified as:

```
space-before.minimum="2.0pt"
space-before.optimum="3.0pt"
space-before.maximum="4.0pt"
space-before.precedence="0"
space-before.conditionality="discard"
```

so properties that may have a value that is a compound datatype each generate multiple attribute definitions. For example:

```
space-before =
    ## <space> | inherit
    attribute space-before { text }?,
    attribute space-before.minimum { text }?,
    attribute space-before.optimum { text }?,
    attribute space-before.maximum { text }?,
    attribute space-before.precedence { text }?,
    attribute space-before.conditionality { text }?
```

As stated previously, property values are evaluated using a parser generated by the REx parser generator [7]. The productions in the XSL 1.1 spec[expressions] were mostly suitable for feeding to REx, although it took a lot of making modifications based on the grammar for XPath 2.0[2] that is provided on the REx website to get a functioning parser.

Running the parser on a property value expression produces markup corresponding to the productions in the grammar. For example, for "-1 - -2", the expression parser produces:

```
<Expression>
   <Expr>
      <AdditiveExpr>
         <MultiplicativeExpr>
            <UnaryExpr>
               <TOKEN>-</TOKEN>
               <UnaryExpr>
                  <PrimaryExpr>
                     <Numeric>
                        <AbsoluteNumeric>
                           <AbsoluteLength>
                              <Number>1</Number>
                           </AbsoluteLength>
                        </AbsoluteNumeric>
                     </Numeric>
                  </PrimaryExpr>
               </UnaryExpr>
            </UnaryExpr>
         </MultiplicativeExpr>
         <TOKEN>-</TOKEN>
         <MultiplicativeExpr>
            <UnaryExpr>
               <TOKEN>-</TOKEN>
               <UnaryExpr>
                  <PrimaryExpr>
                     <Numeric>
                        <AbsoluteNumeric>
                           <AbsoluteLength>
                              <Number>2</Number>
                           </AbsoluteLength>
                        </AbsoluteNumeric>
                     </Numeric>
                  </PrimaryExpr>
               </UnaryExpr>
            </UnaryExpr>
         </MultiplicativeExpr>
      </AdditiveExpr>
   </Expr>
   <EOF/>
</Expression>
```

This, obviously, has little resemblance to an XSL-FO datatype. The Schematron uses a handwritten `parser-runner.xsl` library that runs the expression parser and (mostly) reduces the elements for the grammar productions into elements representing XSL-FO datatypes. For example, this is the current

[1] Section 5.11 Property Datatypes, in Extensible Stylesheet Language (XSL) Version 1.1 [1]
[2] http://www.bottlecaps.de/rex/xpath20.ebnf

implementation of the function for the AdditiveExpr element:

```
<xsl:function name="ahf:AdditiveExpr"
              as="element()">
  <xsl:param name="parse-tree" as="element()*"/>

  <xsl:variable name="term1" as="element()"
    select="ahf:reduce-tree(
              $parse-tree/MultiplicativeExpr[1])"/>

  <xsl:choose>
    <xsl:when test="count(
              $parse-tree/MultiplicativeExpr) = 1">
      <xsl:sequence select="$term1"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence
          select="ahf:nextAdditiveExpr($term1,
$parse-tree/MultiplicativeExpr[position() > 1])"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

The `parser-runner.xsl` is implemented solely using `xsl:function` because of problems with oXygen's Schematron support when using `xsl:apply-templates`: with an earlier version that did use `xsl:apply-templates` on the `Expression` element, oXygen reported the context for Schematron errors as the line in the parser XSLT where the Expression element was created, not as the line in the XSL-FO document where the property expression occurred. Using only `xsl:function` works well enough that it was not necessary to delve further into why this was happening.

The result returned from the `parser-runner.xsl` for "-1 - -2" is:

```
<Number value="1" is-positive="yes" is-zero="no"/>
```

Expression evaluation is used in three Schematron phases:

- Automatically generated Schematron rules that report syntax errors and incorrect datatypes.
- Handwritten Schematron rules for the additional constraints in the XSL 1.1 Recommendation.
- Handwritten Schematron rules for Antenna House extensions.

For example, the value of the `column-count` property is defined as "<number> | inherit", but the definition of `<number>` for `column-count` is:

> <number>
>
> A positive integer. If a non-positive or non-integer value is provided, the value will be rounded to the nearest integer value greater than or equal to 1.

The automatically generated rule for `column-count` is:

```
<rule context="fo:*/@column-count">
  <let name="expression"
       value="ahf:parser-runner(.)"/>

  <assert test="local-name($expression) =
('Number', 'EnumerationToken', 'ERROR', 'Object')">
'column-count' should be Number, EnumerationToken.
'<value-of select="."/>' is a <value-of
select="local-name($expression)"/>.</assert>

   <report test="$expression instance of
element(EnumerationToken) and
 not($expression/@token = ('inherit'))">
      Enumeration token is: '<value-of
              select="$expression/@token"/>'.
      Token should be 'inherit'.</report>

   <report test="local-name($expression) =
              'ERROR'">Syntax error:
   'column-count="<value-of select="."/>"'</report>
</rule>
```

and the manually generated rule is:

```
<rule context="fo:*/@column-count"
      role="column-count">
  <let name="expression"
       value="ahf:parser-runner(.)"/>

  <report test="local-name($expression) = 'Number'
          and (exists($expression/@is-positive) and
               $expression/@is-positive eq 'no' or
                   $expression/@is-zero = 'yes' or
                   exists($expression/@value) and
   not($expression/@value castable as xs:integer))"
 role="column-count">Warning: @column-count should
be a positive integer.  The FO formatter will
round a non-positive or non-integer value to the
nearest integer value greater than or equal to 1.
  </report>
</rule>
```

Note that the expression evaluation stops short of evaluating the inherited value. Also, a `<Number>` might not have a `@value`; for example, if it is the result of 'evaluating' an XSL-FO function that isn't fully implemented in `parser-runner.xsl`.

Some of the property value definitions need to be expanded[1] into multiple enumeration tokens or XSL-FO datatypes before generating the Schematron for checking

[1] Section 7.4 Additional CSS Datatypes, in Extensible Stylesheet Language (XSL) Version 1.1 [1]

a property's value. For example, the value of the border-start-width property[1] is defined as:

```
<border-width> | <length-conditional> | inherit
```

but <border-width> is considered a 'notational shorthand' in XSL 1.1[cssdat], so the value to be checked for expands to:

```
thin | medium | thick | <length> |
<length-conditional> | inherit
```

although only the presence or absence of attributes for border-start-width.length or border-start-width.conditionality would determine whether a length value for border-start-width is a <length> or a <length-conditional>.

7. Antenna House extensions

Antenna House AH Formatter [8] implements a number of extensions[2][3][4] to the XSL 1.1 Recommendation to provide improvements to the formatted output. Validation of AH Formatter extensions is also implemented using a combination of Relax NG and Schematron.

The documentation for the AH Formatter extensions is in XML, as you would expect. However, it's not in a format that's useful for automating the connections between extensions and applicable FOs and properties, so the Relax NG and Schematron both needed to be handwritten.

The modules for the AH Formatter extensions use the Relax NG include pattern to include the schema for XSL-FO and merge it with the definitions of the extensions. The shortened schema module below demonstrates this:

```
default namespace axf =
"http://www.antennahouse.com/names/XSL/Extensions"
namespace fo = "http://www.w3.org/1999/XSL/Format"

include "fo.rnc" {

# http://www.antennahouse.com/product/ahf60/docs/
#ahf-ext.html#fo.change-bar-begin
fo_change-bar-begin.model =
    ( fo_float? )

# http://www.antennahouse.com/product/ahf60/docs/
#ahf-ext.html#axf.document-info
fo_root.model =
    ( (axf_document-info*,
       fo_layout-master-set,
       axf_document-info*,
       fo_declarations?,
       axf_document-info*,
       fo_bookmark-tree?,
       axf_document-info*,
       (fo_page-sequence|
        fo_page-sequence-wrapper)+) )

}

# http://www.antennahouse.com/product/ahf60/docs/
#ahf-ext.html#axf.document-info
axf_document-info =
  element axf:document-info {
    attribute name {
('document-title' | 'subject' | 'author' |
'author-title' | 'description-writer' | 'keywords' |
'copyright-status' | 'copyright-notice' |
'copyright-info-url' | 'xmp' | 'pagemode' |
'pagelayout' | 'hidetoolbar' | 'hidemenubar' |
'hidewindowui' | 'fitwindow' | 'centerwindow' |
'displaydoctitle' | 'openaction' )},
    attribute value { text },
    empty
}


common-border-padding-and-background-properties &=
    axf_border-radius,
    axf_border-top-right-radius

axf_border-radius =
    attribute axf:border-radius { text }?
axf_border-top-right-radius =
    attribute axf:border-top-right-radius { text }?
```

This module includes fo.rnc. The definitions of fo_change-bar-begin.model and fo_root.model redefine and override the corresponding definitions in fo.rnc.

[1] Section 7.8.15 "border-start-width", in Extensible Stylesheet Language (XSL) Version 1.1 [1]
[2] http://www.antennahouse.com/CSSInfo/extension.html
[3] http://www.antennahouse.com/CSSInfo/float-extension.html
[4] http://www.antennahouse.com/CSSInfo/ruby-extension.html

Conversely, the definition of `common-border-padding-and-background-properties` that is outside the `include` interleaves the `axf_border-radius` and `axf_border-top-right-radius` patterns with the existing `common-border-padding-and-background-properties` defined in `fo.rnc` to add additional optional attributes to any FO defined by the XSL 1.1 spec to already have the common border, padding, and background properties.

The definitions of `axf_document-info`, `axf_border-radius`, and `axf_border-top-right-radius` have to be outside the `include` pattern. It would be an error to put any of them inside the `include` since there are no corresponding definitions in `fo.rnc` that they would override.

8. Putting it all together - the onion and the string

The Relax NG schema resembles an onion: the outer layer is `axf.rnc` with the definitions and redefinitions for the Antenna House extensions. The next layer, which is included by `axf.rnc` is the auto-generated definitions that interleave the inheritable Antenna House extension properties with the properties that are defined for each FO. That includes the auto-generated module with definitions for the XSL 1.1 inherited properties, which in turn includes the inner layer that is the autogenerated definitions for the FOs and their properties.

The Relax NG compact syntax schema is also converted into Relax NG XML syntax for use with oXygen (since oXygen does not support schema-directed editing using a compact syntax schema) and into W3C XML Schema for use with other editors. As noted previously, the annotations in the schema, which were extracted from the XML for the XSL 1.1 spec, are presented as tool-tips when editing an FO document with oXygen.

The Schematron is written as multiple phases strung together. With a Schematron implementation that supports progressive validation by executing each phase in order of its appearance, this will lead to progressively more refined error checking. The phases are:

- Handwritten rules for FO constraints that aren't captured by the Relax NG.
- Autogenerated rules for checking property values for syntax errors and correct datatypes.
- Handwritten rules for extra constraints on property values, such as the rule that `column-count` should be a positive integer.
- Handwritten rules for the Antenna House extensions.

There is an oXygen framework file that refers to both the Relax NG and the Schematron so that oXygen can automatically validate FO files using them. The framework is available as a downloadable add-on for oXygen.

You can also validate FO files from the command-line using the `validate` target from the `build-focheck.xml` Ant build file.

9. Testing

There are multiple levels of testing of the Relax NG and Schematron.

At the lowest level, the `parser-runner.xsl` XSLT library is tested using XSpec[1] tests, for example:

```
<x:scenario label="AdditiveExpr">
    <x:call function="axf:parser-runner"/>
    <!-- ... -->
    <x:scenario label="-1 - -2">
        <x:call>
            <x:param name="input"
                     select="'-1 - -2'"
                     as="xs:string"/>
        </x:call>
        <x:expect label="is a number">
            <Number value="1" is-positive="yes"
                    is-zero="no"/>
        </x:expect>
    </x:scenario>
</x:scenario>
```

At the next highest level, the Schematron is tested using stf[2]:

```
<?stf column-count:1 ?>
<fo:retrieve-table-marker column-count="-1"
    xmlns:fo="http://www.w3.org/1999/XSL/Format" />
```

Finally, complete documents will be validated using both Relax NG and Schematron using the `validate` Ant task.

10. Need for speed

The bottleneck for the validation is obviously going to be executing the XSLT for the Schematron validation and, in particular, the expression parser.

[1] https://code.google.com/p/xspec/
[2] https://github.com/MenteaXML/stf

Saxon has dropped the ability to compile stylesheets[1], but the non-free versions of Saxon can memorise values returned by functions[2]. oXygen uses Saxon EE when running Schematron, so any property value is only evaluated once, no matter how many times it appears in the document. Additionally, oXygen also caches the schema used to validate a document[3], and oXygen Support have confirmed that this includes caching a Schematron validator, so the memorised property expression values are available across documents.

The REx parser generator [7] is able to generate a parser as a Saxon extension function. It should, therefore, be possible to optionally include the compiled extension function in the classpath for Saxon and make the 'parser runner' library use the compiled extension function if it is available and to fallback to the XSLT parser when the function is unavailable. However, using Saxon extension functions with Schematron validation is not a common use-case, so it is not possible with oXygen 17 to just add the Jar file for an extension function to an oXygen framework and have it be used when validating Schematron. Using the extension function currently requires registering the extension function in the default Saxon configuration in the oXygen preferences. Since that can't be done just by downloading the oXygen add-on, it's not currently part of focheck.

11. Future improvements

A necessary improvement is adding to and improving the handwritten parts of the Relax NG and Schematron. The constraints in the XSL 1.1 spec are spread through much of the spec, and some of the details require careful reading. For example, the Schematron rule for `column-count` was modified after re-reading the definition while writing this paper. Getting the Relax NG and Schematron to be complete and correct is an ongoing and iterative process. Pull requests on the GitHub project will be appreciated.

The current expression parser cannot evaluate some of the shorthand properties, such as font[4], that were borrowed from CSS2. Handling those will require either writing custom XSLT or generating a different parser using REx.

12. Conclusion

Validating XSL-FO documents can be useful when debugging generated XSL-FO documents and when prototyping XSL-FO by hand in an XML editor. The XSL-FO validation from Antenna House available on GitHub uses Relax NG, Schematron, and an XSLT-based property expression parser to provide unprecedented accuracy when validating XSL-FO documents.

Thanks go to the folks at Oxygen XML Editor Support for helping with some of the issues encountered when developing focheck.

Bibliography

[1] *Extensible Stylesheet Language (XSL) Version 1.1*. 05 December 2006. Anders Berglund. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xsl11/

[2] *Extensible Stylesheet Language (XSL) Requirements Version 2.0*. 11.5 Schema for XSL-FO. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xslfo20-req/#N67198

[3] *A Visual Comparison Approach to Automated Regression Testing*. Celina Huang. In Conference Proceedings of XML London 2014. June 7-8, 2014.
doi:10.14337/XMLLondon14.Huang01

[4] xmlroff. Tony Graham.
https://github.com/xmlroff/xmlroff/tree/master/testing

[5] Validators by RenderX. RenderX Inc..
http://www.renderx.com/tools/validators.html

[1] http://www.saxonica.com/documentation/index.html#!using-xsl/compiling
[2] http://www.saxonica.com/documentation/index.html#!extensions/attributes/memo-function
[3] http://www.oxygenxml.com/doc/ug-editor/#topics/validation-actions-in-user-interface.html
[4] Section 7.31.13 "font", in the Extensible Stylesheet Language (XSL) Version 1.1 Specification [1]

[6] *Relax NG schema for XSL-FO*. Alexander Peshkov. XML Europe 2004. 18 - 21 April 2004. Amsterdam, Netherlands.
http://xep.xattic.com/xep/resources/validators/xmleurope2004-peshkov.pdf

[7] REx Parser Generator. Gunther Rademacher.
http://www.bottlecaps.de/rex/

[8] *Antenna House Formatter V6*. Antenna House Inc..
http://www.antennahouse.com/product/ahf60/ahf6top.htm

The application of Schematron schemas to word-processing documents

Document validation in a non-structured environment

Andrew Sales

Andrew Sales Digital Publishing Limited

<andrew@andrewsales.com>

Abstract

This paper will present Schematron as a portable, standards-based alternative to macros, demonstrating how it can be integrated into a word-processing template to alert authors and editors directly to content problems during capture.

It will demonstrate how business rules can be applied to a word-processing document held in one of the standard word-processing XML file formats using an ISO Schematron schema. These rules will comprise typical Schematron validation activity. Further, it will be shown how errors found in the document can be successfully merged back in situ into the original document, so that an editor can address the problem so located within the originating editing environment.

Keywords: Schematron, validation, OOXML, ODF

1. Background

As traditional print-based publishing has made the transition into the digital age, a convention has developed in some quarters of capturing or even typesetting content using word-processing applications.

These can present a convenient route to publication in the many instances where content derives (in the form of author manuscript) from the same word-processing package. It is also a relatively cheap and efficient one, demanding the now basic and widespread skills of styling a document to achieve the desired appearance.

As a result, typesetting workflows consuming these documents still exist, template-based workflows designed to capture structured data are still in place, and for some publishers large quantities of legacy data persist in word-processing formats only and require migration to XML to meet modern production demands.

2. Quality

During the long period (for some) of moving to a digital-first workflow, with publication of a single source of structured data in various renditions, it has become apparent to such publishers that the quality of their content no longer only resides in the appearance of the rendered product, but also in the quality of the data capture itself. The quality question has shifted from “Does my product look right?” to “Is my source markup sufficiently rich to service the outputs I wish to produce?” When generating XML markup from a word-processing source, the inevitable corollary is whether the document has been styled appropriately to drive good-quality data capture.

Consistently-applied styles are needed, which can be facilitated in part by a well-designed template. Badly-applied styles will, understandably enough, produce less than optimal results.

3. Approaches

The requirement to apply business rules to styled documents is not new. This was often done using macros to interrogate the underlying object model before Microsoft Office (OOXML) [1] and Open Office (ODF) [2] began exposing their respective file formats as XML. With the word-processing document now available as XML, other, native-XML validation approaches are viable and indeed attractive.

3.1. The case for Schematron

There are several benefits to preferring Schematron:

- compatibility of a native XML technology with the OOXML/ODF XML formats;
- the output validation format (SVRL [3]) can be used for onward analysis/processing;
- the technologist can concentrate on writing constraints using XPath (perhaps also re-using

favourite libraries or shared Schematron rulesets), rather than bespoke macro code;
- there is some scope for targeting different flavours of word-processing XML (with a little work);
- it is an international standard, whose *de facto* reference implementation is written in XSLT.

4. Types of rule

Let us look at some samples of common types of business rule expressed as Schematron constraints. The format to be constrained in this instance is WordprocessingML.

The natural-language business rule is given first, followed in each case by a Schematron implementation.

Note

These and other ISO Schematron examples omit the Schematron namespace for brevity. An XSLT 2.0 implementation is also assumed.

4.1. Unexpected styles

"All paragraph styles in the body of the document must be a member of a controlled list of styles."

```
<pattern id="unexpected-para-style">
<let name="allowed-para-styles"
  value=
  "('articlehead', 'bodytext', 'bibhead', 'bib')"/>
<rule context="w:p[not(parent::w:ftr)
and not(parent::w:footnote)
and not(parent::w:endnote)][w:r]">
<report
test="not(w:pPr/w:pStyle/@w:val =
  $allowed-para-styles)">
unexpected para style
  '<value-of select="w:pPr/w:pStyle/@w:val"/>';
expected one of:
  <value-of select="$allowed-para-styles"/>
</report>
</rule>
</pattern>
```

4.2. Unexpected sequence of styles

"The first bibliographic citation must be immediately preceded by a bibliography heading."

```
<pattern id="missing-bib-heading">
<rule
context="w:p[w:pPr/w:pStyle/@w:val='bib']
[not(preceding::w:p[w:pPr/w:pStyle/@w:val
  = 'bib'])]">
  <assert
    test="preceding::w:p[w:pPr/w:pStyle/@w:val
      = 'bibhead']">
    no bibliography heading found
  </assert>
</rule>
</pattern>
```

4.3. Formatting of datatypes, e.g. dates

"A date in a bibliographic citation must conform to the format YYYY-MM-DD."

```
<pattern id="bad-date">
<rule
  context="w:r[w:rPr/w:rStyle/@w:val='bibdate']">
  <assert test=". castable as xs:date">
  text styled as 'bibdate' must be in the format
    'YYYY-MM-DD';
  got '<value-of select="."/>'</assert>
</rule>
</pattern>
```

4.4. Co-occurrence constraints

"Every citation reference must have a corresponding citation number in the bibliography."

```
<pattern id="broken-citation-link">
<let name="citation-refs"
  value="//w:r[w:rPr/w:rStyle/@w:val='bibref']"/>
<rule context="w:r[w:rPr/w:rStyle/@w:val
  = 'bibnum']">
  <assert test=". = $citation-refs">
  could not find a citation reference to this
  citation:
  '<value-of select="."/>'</assert>
</rule>
</pattern>
```

These examples show that it is possible to express a range of constraints usefully with Schematron on WordprocessingML documents, just as it is on other types of XML document, provided that:

1. the expected disposition of styles supplies enough meaning to enable those constraints to be created in the first place; and
2. enough styling has been applied to produce sensible validation output.

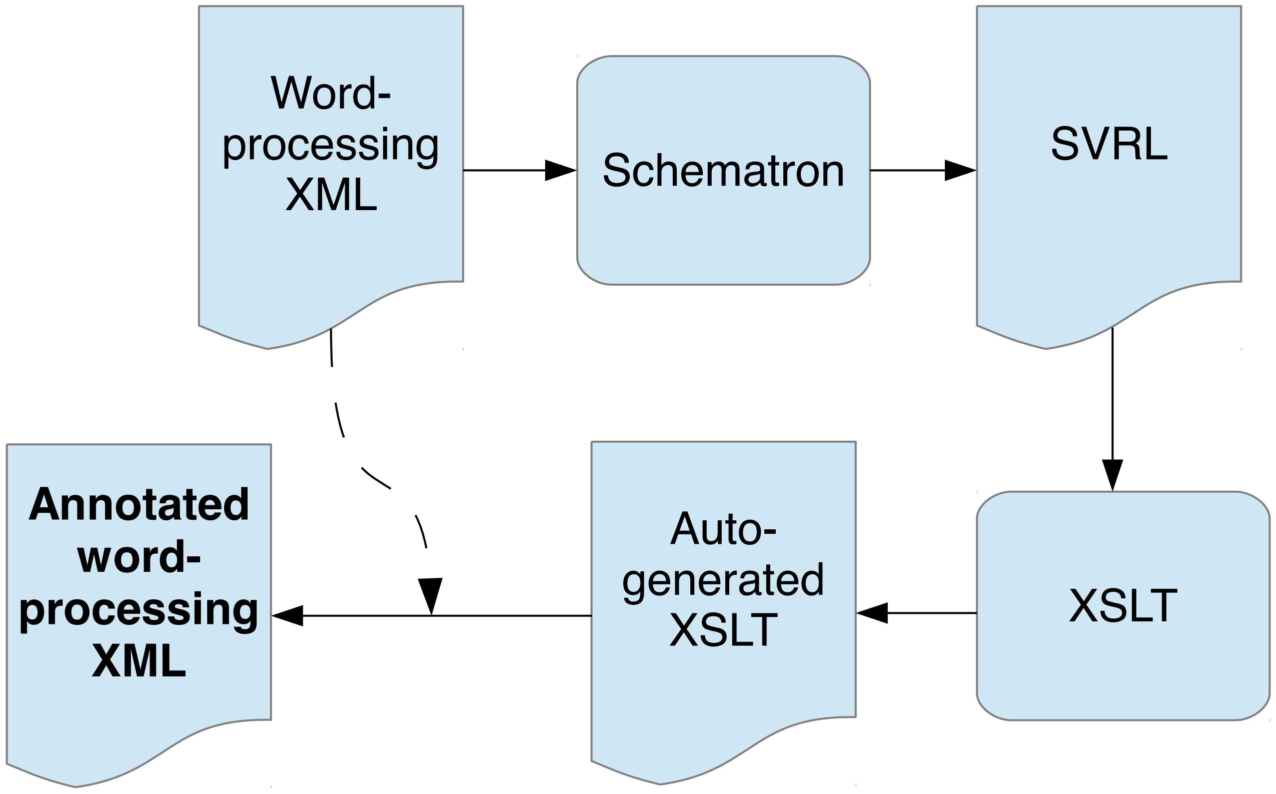
These are important, if obvious, limitations: a document whose contents have, for instance, a single default style applied to them will not be very amenable to this kind of validation.

5. Error reporting and visualisation

Word-processing packages are by nature presentation-driven, so it makes sense to display the problems found within the original document for editorial convenience where possible.

The image below shows an approach to achieving this, starting with the original document at top left and proceeding clockwise until the final step, where the original document is the input document for the final transformation.

Figure 1. Annotating word-processing documents with Schematron errors



Assuming a document containing errors which we have obtained from Schematron as an SVRL report, let us look at this process in more detail.

5.1. SVRL

We can take advantage of SVRL's `failed-assert` and `successful-report` elements, whose `location` attribute contains an XPath locator to the node at fault, e.g.

```
<svrl:failed-assert test=". castable as xs:date"
   location="/*:wordDocument[namespace-uri()=
   'http://schemas.microsoft.com/office/word/2003/wordml'][1]
   /*:body[namespace-uri()='http://schemas.microsoft.com/office/word/2003/wordml'][1]
   /*:sect[namespace-uri()='http://schemas.microsoft.com/office/word/2003/auxHint'][1]
   /*:sub-section[namespace-uri()='http://schemas.microsoft.com/office/word/2003/auxHint'][1]
   /*:sub-section[namespace-uri()='http://schemas.microsoft.com/office/word/2003/auxHint'][1]
   /*:p[namespace-uri()='http://schemas.microsoft.com/office/word/2003/wordml'][4]
   /*:r[namespace-uri()='http://schemas.microsoft.com/office/word/2003/wordml'][4]">
  <svrl:text>text styled as 'bibdate' must be in the format 'YYYY-MM-DD'; got 'February 2015'</svrl:text>
</svrl:failed-assert>
```

5.2. Auto-generated XSLT

From the XPaths supplied in SVRL, XSLT can be automatically generated to perform an identity transform to flag up errors in the original document.

The resulting, auto-generated XSLT contains an `xsl:key` matching each locator, whose paths are simplified so that each step contains a `QName` (the `key` name here is the ID of the respective SVRL `failed-assert` or `successful-report`):

```
<xsl:key name="d1e44" use="generate-id()"
  match="/w:wordDocument[1]/w:body[1]/wx:sect[1]/
  wx:sub-section[1]/wx:sub-section[1]/w:p[4]/
  w:r[4]"/>
```

This enables (in WordprocessingML) the insertion of start and end comment elements around the location of the problem[1] (`w:p` for paragraphs and `w:r` inline), and the text of the error message as the comment's content:

```
<xsl:template match="w:p">
<xsl:variable name="id" select="generate-id(.)"/>
<xsl:variable name="pos"
select="count( preceding::* )
  + count( ancestor-or-self::* )"/>

    <!--*COMMENT START MARKERS*-->
    <xsl:for-each select="key( 'd1e44', $id )">
        <xsl:call-template name="annotation">
            <xsl:with-param name="att-name"
            select="'Word.Comment.Start'"/>
            <xsl:with-param name="att-value"
            select="$pos"/>
        </xsl:call-template>
    </xsl:for-each>
    <!-- a further for-each select='key(...)'
      here for each error found -->
    <w:p>
    <xsl:apply-templates select="*"/>

    <!--*COMMENT END MARKERS*-->
    <xsl:for-each select="key( 'd1e44', $id )">
        <xsl:call-template name="annotation">
            <xsl:with-param name="att-name"
            select="'Word.Comment.End'"/>
            <xsl:with-param name="att-value"
            select="$pos"/>
        </xsl:call-template>
    </xsl:for-each>
    <!-- a further for-each select='key(...)'
      here for each error found -->

    <!--*COMMENT CONTENT*-->
    <xsl:for-each select="key( 'd1e44', $id )">
      <xsl:call-template name="insert-comment">
        <xsl:with-param name="id"
          select="$pos"/>
          <xsl:with-param name="message"
              select="'text styled as
&quot;bibdate&quot; must be in the format
&quot;YYYY-MM-DD&quot; got
&quot;February 2015&quot;'"/>
      </xsl:call-template>
    </xsl:for-each>
    </w:p>
</xsl:template>

<!--the same approach also applies to any w:r
  elements-->

<xsl:template name="annotation">
    <xsl:param name="att-name"/>
    <xsl:param name="att-value"/>
    <aml:annotation>
        <xsl:attribute name="w:type">
            <xsl:value-of select="$att-name"/>
        </xsl:attribute>
        <xsl:attribute name="aml:id">
            <xsl:value-of select="$att-value"/>
        </xsl:attribute>
    </aml:annotation>
</xsl:template>

<!-- N.B. template named 'insert-comment'
  omitted; see its output in the following section
-->
```

This XSLT has clearly not been finessed for elegance or efficiency. Templates matching the error XPath locators are a more natural choice; another alternative would be to use an extension functions to evaluate the XPaths reported in SVRL.

[1] ODF uses a similar approach with its `office:annotation` and `office:annotation-end`.

5.3. Annotated source document

When the automatically-generated XSLT is run on the original document, an annotated analogue of the source results.

The WordprocessingML contains for example this markup for each error identified using Schematron:

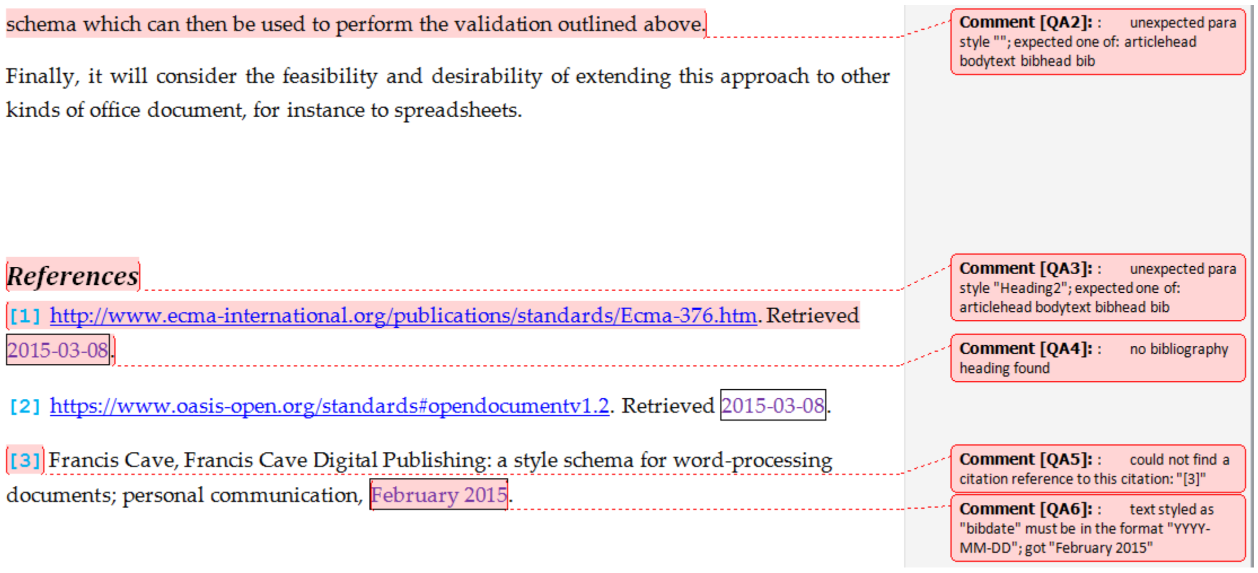
```
<!-- comment start marker -->
<aml:annotation w:type="Word.Comment.Start"
  aml:id="650"/>
<!-- original text where problem located: -->
<w:r>
  <w:rPr>
    <w:rStyle w:val="bibdate"/>
  </w:rPr>
  <w:t>February 2015</w:t>
</w:r>
<!-- comment end marker -->
<aml:annotation w:type="Word.Comment.End"
  aml:id="650"/>
<!-- comment content
  (i.e. Schematron error message) -->
<w:r>
  <w:rPr>
    <w:rStyle w:val="CommentReference"/>
  </w:rPr>
  <aml:annotation
    aml:author="QA" w:type="Word.Comment"
    w:initials="QA"
    aml:id="650"
    aml:createdate="2015-05-06T16:33:59.801+01:00">
    <aml:content>
      <w:p>
        <w:pPr>
          <w:pStyle w:val="CommentText"/>
        </w:pPr>
        <w:r>
          <w:rPr>
            <w:rStyle w:val="CommentReference"/>
          </w:rPr>
          <w:annotationRef/>
        </w:r>
        <w:r>
          <w:t>text styled as "bibdate" must be
          in the format "YYYY-MM-DD";
          got "February 2015"</w:t>
        </w:r>
      </w:p>
    </aml:content>
  </aml:annotation>
</w:r>
```

Using comments represents a relatively non-invasive way of inserting error information into the original. In both OOXML and ODF they reside in a different namespace from the "true" content of the document, with the added advantage that an editor can cycle through the errors using the application's in-built review tools.

5.4. Visualisation

The Word rendition then contains comments flagging up each error:

Figure 2. Word rendition of Schematron errors as comments



schema which can then be used to perform the validation outlined above.

Finally, it will consider the feasibility and desirability of extending this approach to other kinds of office document, for instance to spreadsheets.

References

[1] http://www.ecma-international.org/publications/standards/Ecma-376.htm. Retrieved 2015-03-08.

[2] https://www.oasis-open.org/standards#opendocumentv1.2. Retrieved 2015-03-08.

[3] Francis Cave, Francis Cave Digital Publishing: a style schema for word-processing documents; personal communication, February 2015.

Comment [QA2]: : unexpected para style ""; expected one of: articlehead bodytext bibhead bib

Comment [QA3]: : unexpected para style "Heading2"; expected one of: articlehead bodytext bibhead bib

Comment [QA4]: : no bibliography heading found

Comment [QA5]: : could not find a citation reference to this citation: "[3]"

Comment [QA6]: : text styled as "bibdate" must be in the format "YYYY-MM-DD"; got "February 2015"

A limitation is that the error must "hang off" something that is visible in the rendered document. For example, a rule that stipulates a particular paragraph style must be present may have the element representing the document body as its context – something which is not visible when rendered. In these cases, the Schematron `rule/@context` should be adjusted so that e.g. the first paragraph in the document is targeted. Alternatively, such rules can be post-processed at the SVRL stage to use renderable locations.

6. Simplification

Writing XPath-based Schematron rules for flat structures is reasonably tedious work: in document-based XML the structure broadly reflects the meaning of the content, whereas a word-processing document is essentially a presentation-focused succession of (paragraph and character) styles, tables and other objects.

Some options are available to make this task less onerous.

6.1. Abstraction

Abstract patterns and rules are a well-known way of maximising re-use in Schematron.

In this context, because of the "flat" structure of the documents, many of the constraints apply to relationships on the preceding and following axes, and so these can be abstracted to make conveniently reusable rules. For instance, the business rule *"The first bibliographic citation must be immediately preceded by a bibliography heading"* may become:

```
<pattern id="expected-preceding-style"
  abstract="true">
    <rule context="w:p[w:pPr/w:pStyle/@w:val
      = $context-style]
        [not(preceding::w:p[w:pPr/w:pStyle/@w:val
          = $context-style])]">
        <assert test="preceding::w:p
        [w:pPr/w:pStyle/@w:val
            = $expected-preceding-style]">
            first occurrence of style
'<value-of select="$context-style"/>' has no
preceding style '<value-of
select="$expected-preceding-style"/>'
        </assert>
    </rule>
</pattern>

<pattern id="missing-bib-heading"
  is-a="expected-preceding-style">
    <param name="context-style" value="'bib'"/>
    <param name="expected-preceding-style"
      value="'bibhead'"/>
</pattern>
```

where the abstracted rule can be used for any such case, and the concrete rule simply passes the relevant style names as parameters.

Using this technique can result in a ruleset with surprisingly few unabstracted rules.

6.2. Simplified source

Another approach is to simplify the source format prior to Schematron validation. OOXML in particular is verbose and benefits from this treatment. Consider this simplification of an OOXML document, made using XSLT.

```
<doc>
  <sect>
    <p style="articlehead">The application of Schematron schemas to word-processing
      documents</p>
    <p style="bodytext">As traditional print-based publishing has made the transition into the
      digital age, a convention has developed in some quarters of capturing or even
      typesetting content using word-processing applications.</p>

    <!-- lots more here... -->

    <p style="heading 2">References</p>
    <p style="bib"><span style="bibnum">[1]</span>
      <url address="http://www.ecma-international.org/publications/standards/Ecma-376.htm"
          >http://www.ecma-international.org/publications/standards/Ecma-376.htm</url>.
      Retrieved <span style="bibdate">2015-03-08</span>.</p>
    <p style="bib"><span style="bibnum">[2]</span>
      <url address="https://www.oasis-open.org/standards"
          >https://www.oasis-open.org/standards#opendocumentv1.2</url>. Retrieved <span
      style="bibdate">2015-03-08</span>.</p>
    <p style="bib"><span style="bibnum">[3]</span> Francis Cave, Francis Cave Digital
      Publishing: a style schema for word-processing documents; personal communication,
      <span style="bibdate">February 2015</span>.</p>
    <p style="footer">Andrew Sales Digital Publishing Limited 8<sup>th</sup> March 2015</p>
  </sect>
</doc>
```

Here the document structure is boiled down to its essence[1]. Built-in inline formatting (bold, italic, hyperlinks etc.) is simplified. Schematron rules can be expressed more succinctly because namespaces have been removed and paragraph and run properties shifted into attribute values.

For example, the date datatype rule above can now become:

```
<pattern id="bad-date-simplified">
    <rule context="span[@style='bibdate']">
        <assert test=". castable as xs:date">
text styled as 'bibdate' must be in the format
'YYYY-MM-DD'; got '<value-of select="."/>'
        </assert>
    </rule>
</pattern>
```

The additional, simplifying step of course means that we are at one remove from the original and no longer have access to the location of the original problem if we are to display that error *in situ* when rendered.

[1] In fact, it is much more similar in markup style to ODF.

This can be overcome by storing the XPaths to the original nodes, here as the first processing instruction child of the simplified node (extra whitespace for readability):

```
<p style="bodytext">
  <?src-xpath-loc /w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-section[1]/w:p[6]?>
The requirement to apply business rules to styled documents is not new. This was often done
using macros to interrogate the underlying object model before Microsoft Office (OOXML)
<span style="bibref">
  <?src-xpath-loc /w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-section[1]/w:p[6]/w:r[2]?>[1]</span>
and Open Office (ODF)<span style="bibref">
  <?src-xpath-loc /w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-section[1]/w:p[6]/w:r[4]?>[2]</span>
began exposing their respective file formats as XML. With the word-processing document
being edited now available as XML, other, native-XML validation approaches are viable
and indeed attractive.</p>
```

A further complication is that the auto-generated annotating stylesheet must now retrieve these XPaths, by looking them up based on the SVRL XPaths pointing to locations in the simplified markup.

Using simplified source markup also introduces the possibility of making Schematron rules more format-agnostic. If OOXML and ODF can be simplified and aligned to conform to the same model using XSLT, rules can be written in an interchangeable way. This does ignore the logical differences between the two formats, but if your business rules target something specific to one format, you would probably write rules targeting that format anyway.

7. Further simplification

Even with these kinds of simplification available to ease the task, we are basically still writing *rules* to validate documents. A more natural fit for a "conventional" XML document would be a schema. Can a domain-specific language be derived for word-processing documents, to express their expected structure more declaratively?

Some work has also been done to derive an XML-based schema for a document's expected disposition of styles [4]. The schema is authored in RELAX NG and specifies the allowed structures for a document (it is based on OOXML only so far) at an abstract level. An instance valid to this schema therefore specifies in schema-like language (it looks similar in some respects to RELAX NG itself, having occurrence indicators and group/sequence structures, but with document-specific enhancements) the expected pseudo-structure of a word-processing document. For instance, a simplistic "style schema" for an extended abstract might be:

```
<StyleSchema
xmlns="http://ns.franciscave.com/styleschema">
    <Start>
        <Document>
            <Ref name="articlehead"/>
            <OneOrMore>
                <Ref name="bodytext"/>
            </OneOrMore>
            <Optional>
                <Group>
                    <Ref name="bibhead"/>
                    <OneOrMore>
                        <Ref name="bib"/>
                    </OneOrMore>
                </Group>
            </Optional>
        </Document>
    </Start>
    <Define name="articlehead">
        <Para styleID="articlehead"/>
    </Define>
    <Define name="bodytext">
        <Para styleID="bodytext"/>
    </Define>
    <Define name="bibhead">
        <Para styleID="bibhead"/>
    </Define>
</StyleSchema>
```

This is a more declarative way to express what is expected, and looks familiar from the schema-writing perspective. `Start` defines the starting point for a validator, and `Document` contains the structures allowed in a document valid to this schema.

Element `Para` specifies the style name expected at this point, through its `styleID` attribute. This element's content model is given in the RELAX NG schema as:

```
<element name="Para">
  <zeroOrMore>
    <choice>
      <ref name="Drawing"/>
      <ref name="DocProperty"/>
      <ref name="Text"/>
      <ref name="Tab"/>
      <ref name="Bookmark"/>
      <ref name="Comment"/>
      <ref name="ParaAnyOf"/>
    </choice>
  </zeroOrMore>
  <attribute name="styleID">
    <text/>
  </attribute>
</element>
```

therefore the expected content of the paragraph can also be specified at this level. For instance, to say that the bibliography heading should always be "References":

```
<Define name="bibhead">
    <Para styleID="bibhead">
      <Text>References</Text>
    </Para>
</Define>
```

`Text` also has the `styleID` attribute available, so it is also possible to specify expected inline styles.

Apart from the declarative strengths of a schema language, this approach also has the potential to specify the expected content of a document in a format-agnostic way (i.e. independent of the OOXML or ODF dialect used). A notable limitation is that there is no facility currently to express datatypes.

7.1. Auto-generated Schematron

It is also possible to go some way in using XSLT to generate the kind of Schematron rules we started out with.

The first implied rule in the style schema above is that the first style in the document should be "articlehead". Consider the following naive XSLT:

```
<!-- Ref in first position in Document -->
<xsl:template match="sts:Document/sts:Ref[1]">
<sch:pattern id="{local-name()}-
  {count(preceding::sts:Ref)}">
<sch:rule context="w:body//w:p
  [not(preceding::w:p[ancestor::w:body])]">
<!-- first para in body -->
<sch:let name="style-name"
  value="w:pPr/w:pStyle/@w:val"/>
<sch:assert test='$style-name = {@name}"'>
  expected first para to be styled
'<xsl:value-of select="@name"/>'; got
  '<sch:value-of select="$style-name"/>'
</sch:assert>
<xsl:choose>
<xsl:when test="following-sibling::*[1]
  [self::sts:OneOrMore]">
<xsl:variable name="next-style"
  select="following-sibling::*[1]/sts:Ref"/>
<sch:assert
  test="following::*[1][self::w:p]
  /w:pPr/w:pStyle/@w:val = {$next-style/@name}">
expected style
  '<xsl:value-of select="$next-style/@name"/>'; got
      '<sch:value-of select="."/>'</sch:assert>
</xsl:when>
<!-- etc. -->
</xsl:choose>
</sch:rule>
</sch:pattern>
</xsl:template>
```

With the caveat that this is part of an early stage of a work-in-progress, and targets only a simple subset of what the style schema can express, there is potential at least to author some of the constraints more declaratively and generate Schematron rules automatically to express these checks. The output is as follows:

```
<pattern id="Ref-0">
  <rule
    context="w:body//w:p[not(preceding::w:p[
                                 ancestor::w:body])]">
     <let name="style-name"
      value="w:pPr/w:pStyle/@w:val"/>
    <assert test="$style-name =
      'articlehead'">expected first para to
      be styled  'articlehead'; got
      '<value-of select="$style-name"/>'</assert>
    <assert test="following::*[1][self::w:p]
/w:pPr/w:pStyle/@w:val = 'bodytext'">expected style
'bodytext'; got '<value-of select="."/>'
    </assert>
  </rule>
</pattern>
```

8. Further applications

The OOXML and ODF families of schemas of course cover a wider variety of office documents than just word-processing ones.

Business rules could apply to many of these with a strong textual component, from the relatively simple – ensuring every slide in a presentation bears the corporate logo – to verifying the consistency and integrity of spreadsheets.

Because spreadsheets are often used in a way that overloads their original, intended purpose, e.g. to track or specify requirements, or capture a report from an automated process, much importance can be invested in them and their maintenance. Like other valuable (perhaps curated) documents, they may require validation beyond the in-built tools of Excel and Calc.

It is possible to validate spreadsheets and present faults as annotations to the user at render-time in much the same way as for word-processing XML.

Example 1. The comment model in SpreadsheetML

```
<Cell>
 <Data ss:Type="String">bad data</Data>
 <Comment ss:Author="Author">
  <ss:Data xmlns="http://www.w3.org/TR/REC-html40">
   <B>
    <Font html:Face="Tahoma" x:Family="Swiss"
      html:Size="8"
      html:Color="#000000">Author:</Font>
   </B>
   <Font html:Face="Tahoma" x:Family="Swiss"
     html:Size="8"
     html:Color="#000000"
     >&#10;a problem here</Font>
  </ss:Data>
 </Comment>
</Cell>
```

This is a trivial example, but shows how the commenting style differs from WordprocessingML: the `Comment` is a wrapper element, since a single comment can only appear within a single cell, and HTML can be used to style its text.

At some point, however, particularly if many business rules are used to layer validation onto the spreadsheet, the question is whether a spreadsheet is still an appropriate choice for capturing the information in the first place.

9. Conclusion

When data quality matters and XML is captured in a non-native XML environment, tools like Schematron can still be used to interrogate underlying XML formats. Business rules expressed as Schematron can be written laboriously in full, or with some effort abstracted at the rule or the content level. A further abstraction to create a DSL for word-processing styles is useful in this respect, and it is possible to get some of the way towards validating a document against such a schema, again using Schematron, this time generated automatically.

It is often beneficial to present data errors to authors and editors unfamiliar with XML in a more accessible way. Schematron, along with some XSLT, enables the presentation of these errors in the context of the original document. It would be interesting to see applications of this technique to other types of office document.

Bibliography

[1] *Office Open XML*. Microsoft Corporation. ECMA.

[2] *Open Document Format*. OASIS. OASIS.

[3] *Schematron Validation Reporting Language*. ISO/IEC. ISO/IEC 19757-3:2006.

[4] *A style schema for word-processing documents*. Francis Cave. February 2015. Personal communication.

XML Interfaces to the Internet of Things with XForms

Steven Pemberton

CWI, Amsterdam

Abstract

The internet of things is predicated on tiny, cheap, lower power computers being embedded in devices everywhere. However such tiny devices by definition have very little memory and computing power available to support user interfaces or extended servers, and so the user interface needs to be distributed over the network.

This paper describes techniques using standard technologies based on XML for creating remote user-interfaces for the Internet of Things.

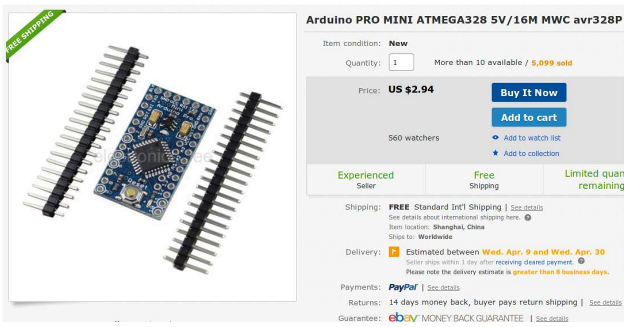
1. Introduction

Moore's Law is alive and well; to use the quote attributed to Mark Twain, the reports of its death have been greatly exaggerated. This year the 50th anniversary of Moore's Law was celebrated, which means since Moore's original paper was published, there have been 33⅓ iterations of the law, which represents an improvement factor of ten thousand million since 1965.

As an excellent test point of Moore's Law's continuation, in February this year, almost exactly three years after the announcement of the first version, version 2 of the Raspberry Pi computer was announced. Moore's Law leads us to expect that every eighteen months you can get twice as many components per unit of surface area on an integrated circuit at the same price. (Note that, for instance, it doesn't say anything about expected clock speeds of computers). Since three years is exactly two cycles of Moore's Law, does the new Raspberry Pi deliver a four-fold improvement? Well, it is reportedly six times faster, has four times as many cores, four times as much memory, and twice as many USB ports as the original, all for the same price. Moore's Law has apparently done some pretty good work.

Moore's Law has three parameters apart from time: price, size, and number of components. Hold any one of these constant, and the other two can vary accordingly. So apart from a Raspberry Pi that is the same price and size, but is better endowed, you can also reduce the price and size to get a less-powerful but nevertheless functional computer. This has been observable since the introduction of the first commercial computers in the 50's: with each order-of-magnitude decrease in price of computers, a new generation of computers has emerged, that gets used in a different sort of way. In the 50's you had mainframes that cost of the order of millions, in the 60's and 70's, minicomputers, of the order of 100,000; in the 70's and 80's, workstations, of the order of 10,000, and then starting in the early 80's the first home computers and laptops, in the order of thousands. Now we have netbooks and tablets of the order of hundreds, and an emerging class of computers, like the Arduino and the Raspberry Pi that cost of the order of tens (and since we're talking orders of magnitude, it doesn't matter if we're talking dollars, pounds, or euros, since they are all roughly of the same value).

Figure 1. Arduino



Arduino PRO MINI ATMEGA328 5V/16M MWC avr328P

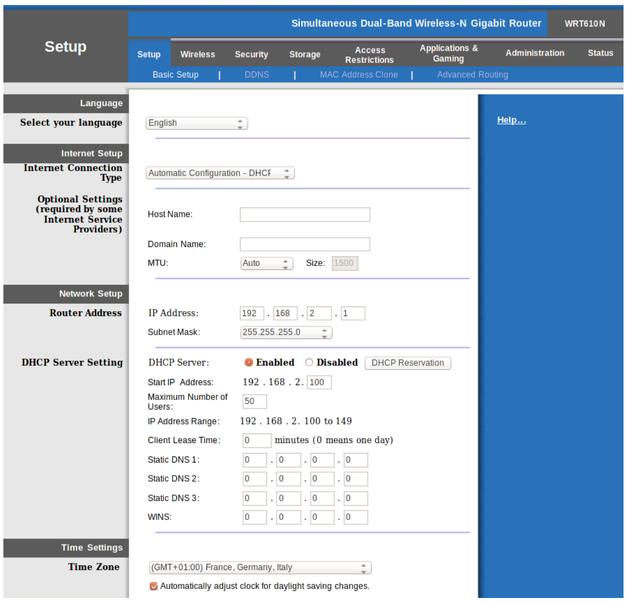
Item condition: New
Quantity: 1 More than 10 available / 5,099 sold
Price: US $2.94
560 watchers
Shipping: FREE Standard Int'l Shipping | See details
See details about international shipping here.
Item location: Shanghai, China
Ships to: Worldwide
Delivery: Estimated between Wed. Apr. 9 and Wed. Apr. 30
Seller ships within 1 day after receiving cleared payment.
Please note the delivery estimate is greater than 8 business days.
Payments: PayPal | See details
Returns: 14 days money back, buyer pays return shipping | See details
Guarantee: ebay MONEY BACK GUARANTEE | See details

Recently the first computers of the order of one unit of currency have been appearing, such as the Arduino mini shown in Figure 1, “Arduino”.

2. User Interfaces for Devices

One of the unanticipated successes of HTML was in its adoption for controlling devices with embedded computers, such as home wifi routers. To make an adjustment to such a device, you direct your browser to the IP address the device is running from, and a small webserver on the device serves up webpages to you, that allow you to fill in values, and submit them to change the workings of the device.

doi:10.14337/XMLLondon15.Pemberton01



Language
Select your language: English

Internet Setup
Internet Connection Type: Automatic Configuration - DHCP

Optional Settings (required by some Internet Service Providers)
Host Name:
Domain Name:
MTU: Auto Size: 1500

Network Setup
Router Address
IP Address: 192 . 168 . 2 . 1
Subnet Mask: 255.255.255.0

DHCP Server Setting
DHCP Server: [✓ Enabled] Disabled
Start IP Address: 192 . 168 . 2. 100
Maximum Number of Users: 50
IP Address Range: 192 . 168 . 2. 100 to 149
Client Lease Time: 0 minutes (0 means one day)
Static DNS 1: 0 . 0 . 0 . 0
Static DNS 2: 0 . 0 . 0 . 0
Static DNS 3: 0 . 0 . 0 . 0
WINS: 0 . 0 . 0 . 0

Time Settings
Time Zone: (GMT+01:00) France, Germany, Italy
[✓] Automatically adjust clock for daylight saving changes.

However, the form-filling facilities of HTML are rather meager: you can fill in values, and submit them, but there is little checking possible on the client side, imposing a duty on the server to check values, and construct error pages that are sent back to the client asking for values to be corrected should they be wrong.

However, the tiny computers that are and will be embedded and form part of the internet of things typically have memory in kilobytes, not megabytes, and certainly don't have the power to run a webserver that can serve and interpret webpages; therefore a different approach is called for.

One way is for the devices to serve up just the data of the parameters, and accept new values for them, so that the values can be injected into a remote interface served from elsewhere.

3. XForms

One technology suitable for just such usage, XForms, is a standard developed at W3C [1]. XForms is a technology that was originally designed for improving the handling of forms on the web. It has two essential parts. The first part is the model, that specifies details of the data being collected, where it comes from, its structure, and constraints; it allows combining data from several sources, and submitting data to different places.

The second part of XForms is the user interface, that displays values, and specifies controls for collection, modification, and submission of the data described, in a device-independent way.

XForms has already been used for a number of years to control devices in this way at many petrol stations in the USA. Each device, storage tank, petrol pump, cash register, and so on, contains a simple server that delivers its data as XML instances. XForms interfaces are then used to read and combine these values, and update control values (for instance the price of fuel being displayed on pumps and charged at tills).

4. Example: A Thermostat

As an example of how it could be used, Nest, a well-known producer of internet thermostats, has published the data-model interface to its devices[1]. A simple interface to this could look like this:

```
<model>
  <instance id="thermostat"
            resource="http://thermostat.local/"/>
  <bind ref="ambient_temperature_c"
        type="decimal" readonly="true()"/>
  <bind ref="target_temperature_c"
        type="decimal"/>
  <bind ref="target_temperature_f" type="decimal"
   calculate="../target_temperature_c*9 div 5+32"/>
  <submission
    resource="http://thermostat.local/data"
    method="put" replace="instance"/>
</model>
```

Here we see an instance that contains the data obtained from the thermostat, and three binds that assign properties to the data, in this case types, the property that the ambient temperature value is read-only, and a calculation that relates the values of the target temperature in Fahrenheit and Celsius, which ensures that whenever the Celsius value is changed, the Fahrenheit value automatically changes with it.

The submission element specifies where the data is to be submitted, and what to do with the reply, in this case that it is data that replaces the instance values.

A nice feature of this is that even if Nest changes the data structure returned by the thermostat, as long as the names of the elements used here remain the same, this interface will continue to work.

5. Display Values

For a user interface for the thermostat, we need some extra local data values. In particular we want to offer the user the choice between Fahrenheit and Celsius in a single control. For this we need to add an extra instance to the model for the display values:

[1] Nest API Reference - https://developer.nest.com/documentation/api-reference

```
<instance id="display">
  <data xmlns="">
    <temperature/>
    <target>20</target>
    <scale>C</scale>
  </data>
</instance>
<bind ref="instance('display')/temperature"
  type="decimal"
  calculate="if(../scale='C', instance('thermostat')/ambient_temperature_c,
                              instance('thermostat')/ambient_temperature_c * 9 div 5 + 32"/>
<bind ref="target" type="decimal"/>
```

Here we specify that the displayed temperature is related to the data from the device, but with a conversion if the user chooses for the Fahrenheit scale. Similarly, we have to add a relation back to the thermostat instance, so that the input required temperature is converted to Celsius if necessary:

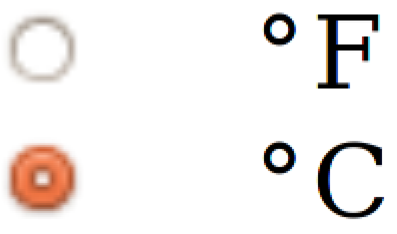
```
<bind
  ref="instance('thermostat')/target_temperature_c"
  calculate="if(instance('display')/scale='C',
              instance('display')/target,
             (instance('display')/target - 32) *
             5 div 9")/>
```

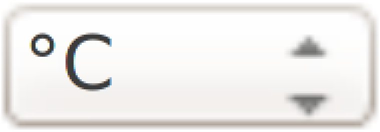
6. The User Interface

XForms controls are specified in a device independent manner, that only describes what they are meant to achieve (for instance "pick one value from this list") and not how to do it (using radio buttons, using drop-downs etc.) This makes it easier to adapt the interface to different devices, screen sizes, etc., while still allowing the use of specific interfaces, such as radio buttons, via style sheets. For instance, to specify a control that allows the user to chose the temperature scale, we specify

```
<select1 ref="instance('display')/scale"
         label="Scale">
  <item label="°C" value="C"/>
  <item label="°F" value="F"/>
</select1>
```

This specification allows several different possible controls, for instance as a drop-down, or using radio buttons, depending on style-sheet options:





Similarly, the range control specifies an input for a number, that allows different styling options, such as nudge buttons, a slider, or a dial; a step attribute specifies the granularity of the changes:

```
<range ref="instance('display')/target" step="0.5"
        start="0" end="30"/>
```

Of course, these start and end values are in Celsius, and we want to specify the limits in terms of the scale used. However, any attribute can have a calculated value, using attribute value templates:

```
<range ref="instance('display')/target" step="0.5"
  start="{instance('display')/start}"
  end="{instance('display')/end}"/>
```

and add these values to the display instance:

```
<instance id="display">
  <data xmlns="">
    <temperature/>
    <target>20</target>
    <start/>
    <end/>
    <scale>C</scale>
  </data>
</instance>
<bind ref="instance('display')/start"
      calculate="if(../scale='C', 0, 32"/>
<bind ref="instance('display')/end"
      calculate="if(../scale='C', 30, 90"/>
```

7. Submitting Data

Normally in a form-based interface, there is an explicit [submit] button or similar that indicates you are ready with the data and want to submit it to be used.

For instance, in XForms, you would typically have details about where the data is to be submitted, in the form of a submission element in the model, as above:

```
<submission resource="http://thermostat.local/data"
            method="put" replace="instance"/>
```

This specifies the URL that the data is to be submitted to, the method to be used (PUT in this case), and what to do with the result. In the case of the thermostat, the state of the internal values are returned again, and these are just used to overwrite the values in the instance.

Then in the user interface, there would be a submit control, that initiates the submission, normally displayed as a clickable button:

```
<submit label="Submit"/>
```

However, typically in a direct-manipulation style interface such as a thermometer, there is no moment that you explicitly submit the data: it just happens. To effect this in XForms, the submission has to be done automatically. This can be done by using the standard event mechanism inherited in XForms from DOM-based systems [2], and using XML Events [3] to listen for events, and react to them.

One of the events that XForms generates is the xforms-value-changed event, which is generated everytime a value is changed in an instance by a control. There are several ways of specifying this, but the most direct is to include, as a child of the control, an action that responds to the event:

```
<range ref="instance('display')/target" step="0.5"
       start="{instance('display')/start}"
       end="{instance('display')/end}">
  <action ev:event="xforms-value-changed">
    <send/>
  </action>
</range>
```

This says that whenever the `<range>` control receives the xforms-value-changed event because the bound value has been changed, then the `<send/>` action is initiated, which causes the submission to do its work. When an `<action>` element only has one child like this, then it can be contracted:

```
<range ref="instance('display')/target" step="0.5"
       start="{instance('display')/start}"
       end="{instance('display')/end}">
  <send ev:event="xforms-value-changed"/>
</range>
```

8. Polling

As a result of submitting data as shown above, the thermostat returns the current values in its internal state, including the currently measured temperature, which then gets displayed.

Of course, you want to continue to display the current temperature, even if the user hasn't changed anything via the interface. To achieve this, the data has to be periodically polled. This can be done also using the event mechanism, by listening for timing events: at start up you initiate a timer, and then listen for the event to go off. When it goes off, you respond, and then re-initiate the timer:

```
<action ev:event="my-timer">
  <send/>
  <dispatch name="my-timer" delay="20000"
            targetid="parent"/>
</action>
```

This `<action/>` element can go anywhere, as long as its parent element has id 'parent' (as named in this case) . The delay is specified in milliseconds, so in this case, every 20 seconds the thermostat is polled for its current values.

The only other thing that has to be done is to start off the initial timer, by listening for the xforms-ready event, which is dispatched when an XForm starts up:

```
<action ev:event="xforms-ready">
  <dispatch name="my-timer" delay="20000"
            targetid="parent"/>
</action>
```

As in the earlier case, this can be shortened:

```
<dispatch ev:event="xforms-ready" name="my-timer"
          delay="20000" targetid="parent"/>
```

Since the xforms-ready event is dispatched to the `<model>` element, this `<dispatch/>` element should be a direct child of it, and since it doesn't matter where the other action is placed, it can also be placed there:

```
<model id="model">
  ...
  <dispatch ev:event="xforms-ready" name="my-timer"
            delay="20000" targetid="model"/>
  <action ev:event="my-timer">
    <send/>
    <dispatch name="my-timer" delay="20000"
              targetid="model"/>
  </action>
</model>
```

Of course, the delay value doesn't have to be hard-wired like this, but can also be stored in an instance, and accessed from there:

```
<model id="model">
  ...
  <dispatch ev:event="xforms-ready" name="my-timer"
  delay="{instance('display')/poll-interval}"
  targetid="model"/>
  <action ev:event="my-timer">
    <send/>
    <dispatch name="my-timer"
      delay="{instance('display')/poll-interval}"
      targetid="model"/>
  </action>
</model>
```

9. Repetition

In many applications, there can be a variable number of values for a particular field; for instance in a router, there can be several rules for firewall exceptions. Traditionally an interface is used that offers several blank entries to be filled in. However XForms offers a dynamic control that grows and shrinks with the number of entries, and allows entries to be added and deleted:

```
<repeat ref="firewall/rules" label="Exceptions">
  <output ref="./port" label="Port"/>
  <output ref="./url" label="URL" />
</repeat>
```

10. Multilingual Interfaces

Every XForms control has a label. Of course, it is good to be able to offer an interface in the language of the user. Attribute value templates make this almost trivially easy: you create an instance to hold the messages and labels:

```
<instance id="label"
  resource="http://example.com/labels-en.xml" />
```

which can have a structure like:

```
<labels lang="en">
  <submit>Submit</submit>
  <help>Help</help>
  <scale>Scale</scale>
  ...
</label>
```

and then reference these in the controls:

```
<select1 ref="instance('display')/scale"
         label="{instance('label')/scale}"> ...
```

Changing the language is then a simple case of having a control that selects the language wanted:

```
<select1 ref="instance('lang')/language"
         label="{instance('label')/language}"> ...
```

and when an xforms-value-changed happens on this control, the value chosen can be submitted, and the labels instance replaced with the returned instance.

These also has the advantage that lables are not hard-wired in the application, and can be updated on the fly. And of course the languages available can also be provided by an external instance, so that new languages can be added on the fly.

11. Experience

XForms has been used in many projects connecting to devices, including some very large projects of many person-years. Experience has repeatedly shown that the time needed to implement such projects is about one tenth of equivalent projects done using traditional programming methods. This advantage can largely be ascribed to the declarative nature of XForms, so that much administrative code that is normally needed in programs is not needed in XForms, since the system ensures that invariants are kept up to date.

12. Specifications and implementations

The current official version of XForms is XForms 1.1 [4], though XForms 2.0 is in preparation and close to completion [5]. There are several implementations available, that work both server-side and client-side, both commercial and open-source. There is a tutorial [6] and a quick reference [7] available.

References

[1] Micah Dubinko, Leigh Klotz, Roland Merrick, and T. V. Raman. *XForms 1.0*. World Wide Web Consortium (W3C). 14 October 2003.
http://www.w3.org/TR/2003/REC-xforms-20031014/

[2] Tom Pixley. *Document Object Model (DOM) Level 2 Events Specification*. World Wide Web Consortium (W3C). 13 November, 2000.
http://www.w3.org/TR/DOM-Level-2-Events/

[3] Shane McCarron, Steven Pemberton, and T. V. Raman. *XML Events*. An Events Syntax for XML. World Wide Web Consortium (W3C). 14 October 2003.
http://www.w3.org/TR/2003/REC-xml-events-20031014/

[4] John Boyer. *XForms 1.1*. World Wide Web Consortium (W3C). 20 October 2009.
http://www.w3.org/TR/xforms/

[5] John Boyer, Erik Bruchez, Leigh Klotz, Steven Pemberton, and Nick Van den Bleeken. *XForms 2.0*. World Wide Web Consortium (W3C).
http://www.w3.org/MarkUp/Forms/wiki/XForms_2.0

[6] Steven Pemberton. *XForms for HTML Authors*. World Wide Web Consortium (W3C). 27 August 2010.
http://www.w3.org/MarkUp/Forms/2010/xforms11-for-html-authors/

[7] Steven Pemberton. *XForms 1.1 Quick Reference*. World Wide Web Consortium (W3C). 29 November 2010.
http://www.w3.org/MarkUp/Forms/2010/xforms11-qr.html

Diaries of a desperate XProc Hacker

Managing XProc dependencies with depify

James Fuller

MarkLogic

`<jim.fuller@marklogic.com>`

Abstract

XProc [1] is a powerful language providing a facade over many XML technologies which can make managing that 'surface area' difficult. XProc v1.0 also presents difficulties to the new user as it has a learning curve which forces developers to learn many concepts before they can be productive in the language.

This paper identifies some of the sources of despair for todays Desperate XML Hacker illustrates how XProc and depify [2], a modest package manager for XProc, can help make developing and maintaining XProc development easier.

Keywords: XML, XProc, XML Calabash, depify

1. Overview

For years in Perl, being known as a D.P.H. (desperate Perl Hacker) was a badge of honour of sorts but looking back I am not certain this ever carried over to the XML domain.

Granted XML may have the odd idiom (ex. Muenchian Method[6]) but many of the core technologies are just too complex a powertool to fit into the concise one liners we see in the healthy flora that grew up around Perl[1] (with sed[2], awk[3], and friends)

That's not to say that the same kind of clever, smart and efficient solutions are not possible, it's just that they tend to occur in a different form and level then a dense and opaque one liner.

What this paper's focus is on is the Desperate XML Hacker's similar plight, where acts of desperation are employed when building, maintaining and creating software solutions that solve real problems.

Figure 1. obligatory XKCD [3] comic

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

The term 'Desperate' is derived from the latin dēspērātus which has a number of definitions;

- Being filled with, or in a state of despair; hopeless. I was so desperate at one point, I even went to see a loan shark.
- Without regard to danger or safety; reckless; furious.a desperate effort
- Beyond hope; causing despair; extremely perilous; irretrievable. a desperate disease; desperate fortune
- Extreme, in a bad sense; outrageous.
- Extremely intense. wikipedia definition

Wikitionary definition of 'Desperatus' [4]

These definitions apply to a D.X.H. employing acts of desperation in trying to get something working. Seemingly bright, creative and smart programmers will try 'anything' to transition from some current problem state, however reckless or outrageous at the most intense hopeless and perilous times.

We introduce how XProc can transform a D.X.H. into a Delighted XProc Hacker (sic) and illustrate

[1] Perl - http://en.wikipedia.org/wiki/Perl
[2] sed - http://unixhelp.ed.ac.uk/CGI/man-cgi?sed
[3] awk - http://unixhelp.ed.ac.uk/CGI/man-cgi?awk

concepts and related tools (like depify package management) which serve to reduce an XML developers despair.

1.1. Why XProc

XML developers use a hodge podge of tools to control and orchestrate XML processing, such as;

- Shell scripting (bash, etc)
- Build tools (ant, make, etc)
- XSLT
- XML databases (via stored proc eg. XQuery)
- Code (main class, SAX pipelines, etc …)

Providing a simplified execution entry point is a good thing for users of a solution though all of the above approaches have trade offs to consider.

Here are some adhoc observations of the most common forms of despair motivating an XML developer to choose any of the above solutions to control their XML processing.

1.1.1. Adoption of a large set of dependencies and software

One informal measure of a programmer's desperation is by the number of dependencies they are willing to inherit into their project.

In the Perl world, there are plenty of examples of desperation on display when you install a single innocent module with CPAN, only to watch in horror as it pulls down a few hundred other CPAN modules with each of those modules in turn pulling down ever more dependencies.

I do not claim to have a solution to minimise dependencies in your software but if your language requires a lot of third party libraries and modules just to be useful, then you need to ask yourself if the language is significantly abstracted to model and solve problems that interest you.

While it is impossible to define a static limit to the number of dependencies, clearly a developer should manage dependencies logically and strive to keep them to an absolute minimum.

Modern package management systems provides developers the means to tame dependencies but paradoxically this same capability makes it easy to accrete dependencies.

1.1.2. Not the right tool for the job

Using software not fit for purpose is a common desperate means to an end.

The canonical example is the usage of GNU make[1] with Makefiles to control processing which has a distinct whiff of desperation just for the fact that GNU make is a build tool and should not find itself as any part of your runtime solution.

Make's sweet siren call of recursive Makefile processing and conditional execution based on a file's timestamp value may seem reasonable at the time but problems quickly mount up. GNU make chatty output (where in unix 'silence is golden' is the golden rule) can unnerve users. However there are larger issues, for example, timestamps may seem like a great way to conditionally process files but can be unreliable at scale (yes time can go backwards).

Similarly, bending existing XML technologies, like XSLT or XQuery, to control your solution's processing presents many challenges. Any non-trivial processing scenario comes with requirements that these technologies were never meant to handle and you will end up doing contortions to address the gaps in functionality.

1.1.3. Java invocation adds a layer of abstraction

The java stack provides a mature and robust set of XML capabilities though often we see solutions created with Java that must take care of the 'feed and care for' of the Java VM. Users of a solution developed with Java must know how to invoke the underlying technology from Java with attendant Java Main method, properties, jvm switches and classpath.

The benefit of reuse with Java outweighs most of the negatives, though if you want users to be able to cleanly and quickly use your XML processing solution one needs to provide an entry point which is easy to invoke, abstracting away the complexity of the invocation itself while providing sufficient configurability.

Often tools used during development, such as Java build tool Apache Ant[2], provide end users a good enough 'run wrapper'. Apache Ant is a wonderful example of a tool applied far beyond its original intended purpose though clearly it's less then ideal to use a build tool to achieve this.

1.1.4. Hard to maintain

Makefiles or shell scripting have always been difficult to maintain but what if we consider some higher order scripting language (Python, etc).

[1] GNU make - http://www.gnu.org/software/make/
[2] Apache ANT - http://en.wikipedia.org/wiki/Apache_Ant

Faster code development is made possible with scripting languages because they take care of complex sundries such as memory management and garbage collection but with power comes well-known sacrifices like slower runtime performance.

Using a scripting language provides XML developers with great flexibility though also comes with a few enigmas;

- Create bugs faster
- Avoid or delay critical design decisions
- Incrementally develop code to fulfil tactical needs

There are some better designed scripting languages which help force the programmer to make better upfront design decisions but there is no replacement to well considered up front design.

Providing script run wrappers does not insulate you from the need to design and maintain those wrappers. When presented between a choice between 'hard to maintain' versus quick to develop its always best to choose the route that simplifies maintenance.

For many D.X.H. the act of maintenance seems like a task done far into the future, though its been shown that the ability to easily refactor code is one route to successful software [20] and directly related to the design decisions made today.

1.1.5. Not enough time

While no discipline can assume 'unlimited time' in fulfilling their required goals, few have to calculate the intersection of the time dimension in as many challenging ways as a programmer is commonly asked.

Many of the aforementioned approaches are chosen because they pre-exist, are already being used or there is little time to properly assess the right way forward. In this scenario, its easy to avoid 'doing things the right way' as there is can be no visible impact on the end users usage.

Desperation due to time starvation forces the D.X.H. into the risky proposition of building up significant technical debt for immediate gains today. Put another way, if the D.X.H. has no time to choose and develop the right way, how will they ever have enough time to develop doing it the wrong way?

1.1.6. Why XProc again ?

XProc reduces all the above forms of desperation transforming the D.X.H. into a Delighted XProc Hacker. Its been designed from the ground up to be your default entry point/run wrapper and controller to using XML technologies in processing pipelines.

While using XProc itself may come with some of its own acts of desperation, they pale in significance compared to the issues brought on using the other approaches. XProc is a domain specific language (DSL) designed to control your XML processing. Apart from hand tuned, manually crafted code it's the best thing for controlling your XML processing and you should be using it today.

1.2. Why Dependency Management?

D.X.H. will already use package managers to manage their environments dependencies and its highly likely they also use package managers [1] [2] [3] to manage language dependencies.

XProc presents a unique challenge for existing package managers as its primary extension mechanism is in the definition of custom steps. A custom step can be created with pure XProc or be implemented as an extension step to the underlying XProc processor. Custom steps will need to include its step signature definition so it can be used as well as any ancillary dependencies.

This approach to extension means that the core XProc language itself is spartan and highly generic presenting challenges for new users attempting to do simple things. Often a D.X.H. practices acts of desperation in learning XProc itself.

[1] Node Package Manager - https://www.npmjs.com/
[2] Maven Central Repository - http://search.maven.org/
[3] Bower - http://bower.io/

Take the following stylized XProc pipeline which processes a collection of XML files with XSLT.

```
<p:pipeline>

  <p:directory-list include-filter=".*\.xml$">
    <p:with-option name="path" select="$testdir"/>
  </p:directory-list>

  <p:for-each>
    <p:iteration-source
      select="/c:directory/c:file"/>
      <p:variable name="namein"
                  select="/c:file/@name"/>
      <p:load>
        <p:with-option name="href"
          select="concat($testdir, $namein)"/>
      </p:load>

      <p:xslt>
      ...
      </p:xslt>

      <p:store>
      ...
      </p:store>
  </p:for-each>
</p:pipeline>
```

Which forces the XProc author to learn many concepts before becoming useful.

- list files with p:directory-list
- iterate with p:for-each
- perform xslt step
- save output with p:store step

Future versions of XProc will make this particular scenario easier but serves as a good example where a custom step could be developed to abstract away the complexity, as shown in the following listing.

```
<p:pipeline>
  <my:customXSLTProcessStep dir=".*\.xml$"
    xslt="mystylesheet.xsl"/>
</p:pipeline>
```

Lastly, XProc needs to handle ancillary dependencies, such as an XSLT stylesheet or XQuery module which have no convention in existing package management systems. Depify attempts to provide simple transitive dependency analysis to pull down such dependencies to satisfy custom XProc step need.

1.3. XML Calabash and depify solution

XML Calabash [5] is the reference implementation for XProc and is built using Java.

Depify [2] is a modest package management system, based on a github repository which provides custom step dependency management for XML Calabash. It is specifically designed to make it easy to develop and distribute custom steps built for XML Calabash. Depify's assumptions of usage of XProc with XML Calabash is a form of 'convention over configuration'.

XML Calabash extension step mechanism provides a reasonable set of defaults which can be assumed by the package manager, making it easy to reuse and distribute custom steps when building XProc solutions.

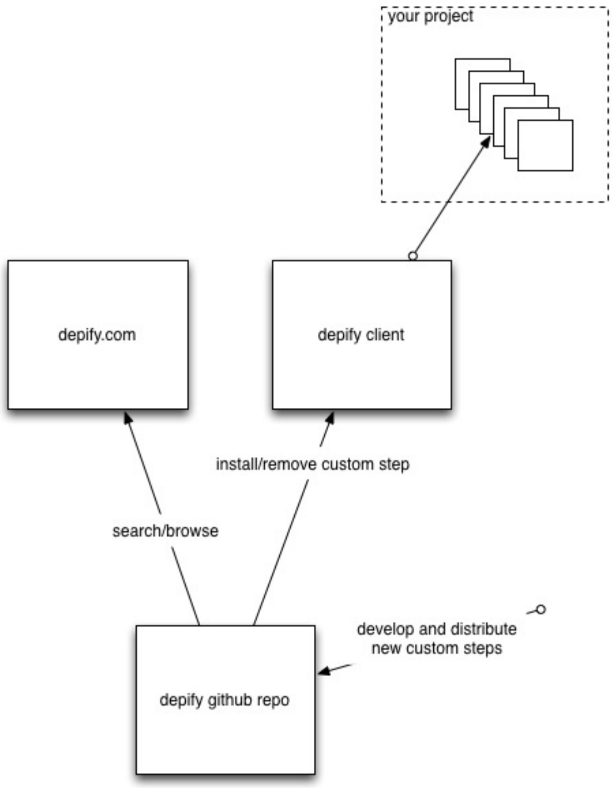
This is the route by which the D.X.H. transforms into the Delighted XProc Hacker.

2. Technical

2.1. Architecture Overview

Depify has three components that comprises its high level architecture.

Figure 2. depify architecture



- depify github repo[1] – contains metadata containing details of your step implementation

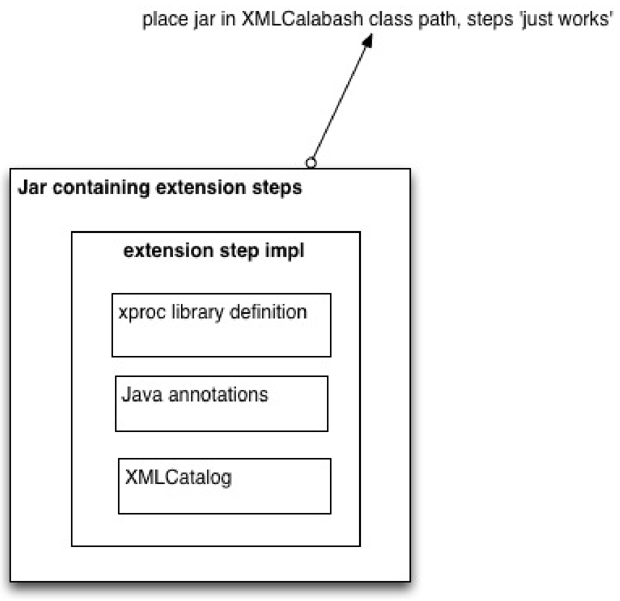
[1] Depify Packages github repo - https://github.com/depify/depify-packages

- depify client[1] – client written with XProc that communicates with depify github repo and pulls down and installs custom steps
- depify website [2] - searchable interface for discovering new custom steps

We will describe the detail of a depify package in the following section.

The architectural components XML Calabash employs at runtime to achieve custom extension steps are as follows;

Figure 3. XML Calabash extension step runtime architecture


place jar in XMLCalabash class path, steps 'just works'
Jar containing extension steps
extension step impl
xproc library definition
Java annotations
XMLCatalog

A jar that works by just dropping onto XML Calabash classpath is elegant as the extension step can be distributed using existing distribution mechanisms (ex. gradle with maven) though depify provides the additional advantage of configuration which allows the step to be immediately usable in an XProc pipeline.

2.2. Points of Interest

There is not enough scope for this paper to go into deep technical detail of every component though I've presented a few highlights below.

2.2.1. depify github repo

The depify github repo contains metadata that represent each depify package with new packages added as the result of a pull request.

The following is an example of metadata that must be defined for a package to be usable by depify.

```
<depify xmlns="https://github.com/depify"
  name="xmlcalabash-ext-step-java"
  version="1.0"
  repo-uri="https://github.com/xquery/xmlcalabash-ext-step-java"
  keep-fresh="true">
  <title>xmlcalabash-ext-step-java</title>
  <desc>example impl of xproc extension step library, in java, for XML Calabash.</desc>
  <license type="?"/>
  <author id="xquery">Jim Fuller</author>
  <website>https://github.com/xquery/calabash-java-step-example</website>
  <xproc version="1.0"
      ns="http://example.org/xmlcalabash/steps"
      jar="example-library-ext.jar/example-library-ext.jar"
      library-uri="!/example-library.xpl">
      <!--catalog name="urn:example-library" jar="example-library-ext.jar" uri="/example-library.xpl"/-->
  </xproc>
</depify>
```

[1] Download depify client - https://github.com/depify/depify-client/releases

The depify package metadata format is described with a schema[1] and minimally must define a custom step's canonical name, version and repo uri.

It is possible to provide a zip archive for repo-uri attribute value but it is preferred to provide a github repo that contains the custom step. Depify leverages github's release features to be able to retrieve specific versions of a package.

With every new commit to the repo a travis build[2] takes care of generating the public package repository.

2.2.2. depify client

The depify client provides a command line interface for installing and removing depify packages.

```
$ depify help

depify 1.0 | copyright (c) 2015 Jim Fuller |
             see https://github.com/depify

usage: depify [install|remove|list|info|search|
                xproc|catalog|library|upgrade|help]
              [package name] [package version]
```

install package

```
$ depify install xprocdoc
```

remove package

```
$ depify remove xprocdoc
```

info package

```
$ depify info xprocdoc
```

list installed packages

```
$ depify list
```

search all packages

```
$ depify search xproc
```

generate xmlresolver catalog

```
$ depify catalog
```

generate xproc library

```
$ depify library
```

reinstall all packages

```
$ depify install
```

initialize .depify

```
$ depify init mypackage 1.0
```

upgrade depify client

```
$ depify upgrade
```

help with depify client

```
$ depify help
```

Depify is itself written in XProc and ships with latest version of XML Calabash, you may also integrate it into your own XProc pipelines.

```
<depify:depify>
  <x:option name="command" select="'install'"/>
  <x:option name="package"
            select="'xmlcalabash-ext-step-java'"/>
  <x:option name="version" select="'1.0'"/>
  <x:option name="app_dir" select="'.'"/>
  <x:option name="app_dir_lib" select="'lib'"/>
</depify:depify>
```

2.2.3. depify.com

With every new commit to depify package repository a searchable website, depify.com [2] is updated.

Depify.com is published to its associated github pages and leverages the use Saxon-CE[3] to deliver all functionality.

2.2.4. XML Calabash extension mechanisms

To implement a step in XML Calabash one needs to

- define a class that inherits DefaultStep and lives in package as a separate jar

[1] Depify Metadata Schema - https://github.com/depify/depify-packages/blob/master/etc/depify.rng
[2] Depify Packages Travis - https://travis-ci.org/depify/depify-packages
[3] Saxon CE - http://www.saxonica.com/ce/index.xml

- define annotations that represent step name and namespace

```
package com.example.library;

import com.xmlcalabash.library.DefaultStep;
import com.xmlcalabash.core.XProcConstants;
import com.xmlcalabash.core.XMLCalabash;
import com.xmlcalabash.io.WritablePipe;
import com.xmlcalabash.core.XProcRuntime;
import com.xmlcalabash.util.TreeWriter;

import net.sf.saxon.s9api.SaxonApiException;
import net.sf.saxon.s9api.XdmNode;
import com.xmlcalabash.runtime.XAtomicStep;

@XMLCalabash(
name = "ex:hello-world",
type = "{http://example.org/xmlcalabash/steps}hello-world")

public class HelloWorld extends DefaultStep {
  private WritablePipe result = null;

  public HelloWorld(XProcRuntime runtime, XAtomicStep step) {
    super(runtime,step);
  }

  public void setOutput(String port, WritablePipe pipe) {
    result = pipe;
  }

  public void reset() {
    result.resetWriter();
  }

  public void run() throws SaxonApiException {
    super.run();

    TreeWriter tree = new TreeWriter(runtime);
    tree.startDocument(step.getNode().getBaseURI());
    tree.addStartElement(XProcConstants.c_result);
    tree.startContent();
    tree.addText("Hello World");
    tree.addEndElement();
    tree.endDocument();
    result.write(tree.getResult());
  }
}
```

```
M Filemode      Length  Date         Time      File
- ----------  --------  -----------  --------  -----------------------------------------------------
  drwxr-xr-x         0   8-Mar-2015  10:43:38  META-INF/
  -rw-r--r--       843   8-Mar-2015  10:43:38  META-INF/MANIFEST.MF
  drwxr-xr-x         0   8-Mar-2015  10:43:38  com/
  drwxr-xr-x         0   8-Mar-2015  10:43:38  com/example/
  drwxr-xr-x         0   8-Mar-2015  10:43:38  com/example/library/
  -rw-r--r--      2062   8-Mar-2015  10:43:38  com/example/library/HelloWorld.class
  drwxr-xr-x         0   8-Mar-2015  10:43:38  META-INF/annotations/
  -rw-r--r--        31   8-Mar-2015  10:43:38  META-INF/annotations/com.xmlcalabash.core.XMLCalabash
  -rw-r--r--       294  19-Feb-2015  15:41:00  example-library.xpl
- ----------  --------  -----------  --------  -----------------------------------------------------
                  3230                         9 files
```

The jar containing the compiled java custom step you will need to include the XProc library that defines the custom step's signature.

This library just contains the step signature declaration.

```
<p:library version="1.0"
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:c="http://www.w3.org/ns/xproc-step"
  xmlns:ex="http://example.org/xmlcalabash/steps">

  <p:declare-step type="ex:hello-world">
    <p:output port="result"/>
  </p:declare-step>

</p:library>
```

There already exists many XML Calabash java extension steps, all available for installation by depify today.

- xmlcalabash1-asciidoctor
- xmlcalabash1-xmlunit
- xmlcalabash1-xcc
- xmlcalabash1-rdf
- xmlcalabash1-print
- xmlcalabash1-plantuml
- xmlcalabash1-metadata-extractor
- xmlcalabash1-mathml-to-svg
- xmlcalabash1-ditaa
- xmlcalabash1-deltaxml

3. Summary

The usage of XML Calabash with depify leverages the development and distribution of custom step libraries.

Unsurprisingly, enabling XProc's primary extension mechanism makes XProc itself easier to use.

Additionally, developing XML Calabash custom steps allows for distribution with pre-existing deployment mechanisms (maven central repository).

Bibliography

[1] *XProc*. An XML Pipeline Language. 11th May 2010. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xproc/

[2] *depify.com*.
http://depify.com/

[3] *Regular Expressions (This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License.)*. xkcd.com.
http://xkcd.com/208

[4] *definition of 'desperatus'*. Wiktionary, The Free dictionary.
http://en.wiktionary.org/wiki/desperatus#Latin

[5] *XML Calabash*. An implementation of XProc: An XML Pipeline Language..
http://xmlcalabash.com/

[6] *Muenchian grouping method*. Wikipedia.
http://en.wikipedia.org/wiki/XSLT/Muenchian_grouping

[20] *A case study on the impact of refactoring on quality and productivity in an agile team* . Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. Springer Berlin Heidelberg.
http://www.researchgate.net/profile/Giancarlo_Succi/publication/221200711_A_Case_Study_on_the_Impact_of_Refactoring_on_Quality_and_Productivity_in_an_Agile_Team/links/0046351f7fbd1e7a41000000.pdf

Charles Foster

XML London 2015
Conference Proceedings

Published by
XML London

103 High Street
Evesham
WR11 4DN
UK

This document was created by transforming original DocBook XML sources into an XHTML document which was subsequently rendered into a PDF by **Antenna House Formatter**.

1st edition

London 2015

ISBN 978-0-9926471-2-4