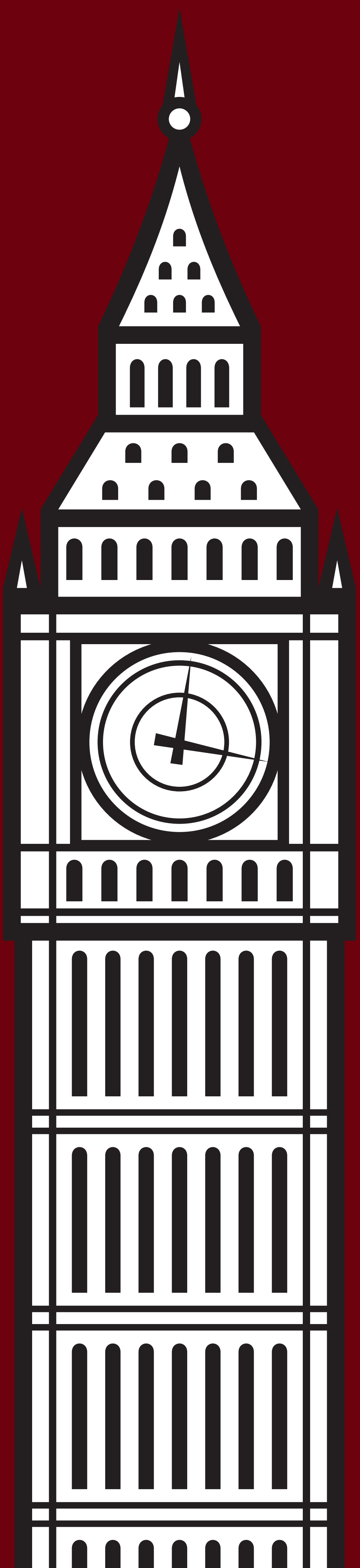


XML LONDON 2016

CONFERENCE PROCEEDINGS

UNIVERSITY COLLEGE LONDON,
LONDON, UNITED KINGDOM

JUNE 4-5, 2016

XML London 2016 – Conference Proceedings
Published by XML London
Copyright © 2016 Charles Foster

ISBN 978-0-9926471-3-1

Table of Contents

General Information

Date

Saturday, June 4th, 2016
Sunday, June 5th, 2016

Location

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

Organising Committee

Kate Harris, Socionics Limited
Dr. Stephen Foster, Socionics Limited
Charles Foster, Socionics Limited

Programme Committee

Abel Braaksma, AbraSoft
Adam Retter, Freelance
Charles Foster (chair)
Dr. Christian Grün, BaseX
Eric van der Vlist, Dyomedea
Geert Bormans, Freelance
Jim Fuller, MarkLogic
John Snelson, MarkLogic
Mohamed Zergaoui, Innovimax
Norman Walsh, MarkLogic
Philip Fennell, MarkLogic

Produced By

XML London (http://xmllondon.com)

Preface

This publication contains the papers presented during the XML London 2016 conference.

This is the fourth 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 4th and 5th June 2016 at the Faculty of Engineering Sciences (Roberts Building) which is part of University College London (UCL). The conference dinner and the XML London 2016 DemoJam is being held in the UCL Marquee located on the Front Quad of UCL, London.

— Charles Foster
Chairman, XML London

Dealing with unlimited XML feeds using XSLT 3.0 streaming

How to use streaming features to process uninterrupted and unlimited live feeds of XML data

Abel Braaksma

Exselt

<abel@exselt.net>

Abstract

Working with unlimited XML feeds poses specific challenges to processors. This paper will discuss those challenges and shows how to solve them using standardized techniques made available by XSLT 3.0 and the new streaming feature. We will see how to set up the processor, how to deal with buffering and flushing, how to handle errors and how to handle a simple example feed with RSS and Twitter.

Keywords: XML, XSLT, XPath, Exselt

1. Disclaimer

This paper discusses new features defined in XSLT 3.0 and XPath 3.0. The XSLT 3.0 specification [1] is a Candidate Recommendation, and information in this paper may be superseded by changes in future additions of this specification. You can track such changes through the publicly available bug reports [2]. [3] is a W3C Recommendation, this paper focuses on XPath 3.0 and not on new features introduced in [4]. Where appropriate, bugs of the specification that were recognized at the time of writing have been incorporated in the text.

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 [1], [3], [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

In XSLT 2.0 it was not trivial to process an uninterrupted live feed using a single stylesheet. XSLT 3.0 fills this gap with the introducation of *streaming* , which allows for any size of input document, including perpetual XML streams like data or news feeds.

The typical reception by the public of the concepts of streaming has been a combination of enthusiasm ("finally, we can deal with *Big Data* using XSLT") and of criticism ("all that new terminology, what is *posture* and *sweep* of a construct, what is *operand usage* and so on"). Most of these criticisms evolve around the fact that the only available documentation is a handful of papers (which often focus on the technical aspect of streaming) and the XSLT specification itself, which is written with processor implementors in mind, not the typical XSLT programmer. Tutorials are currently hard to find, though Dimitre Novatchev has done an excellent job of filling that gap with courses on XSLT 3.0 concepts and more advanced topics, available on Pluralsight, see [6].

This paper aims at filling at least a part of that gap. While last year's talk on XSLT 3.0 was about *The little things* [7] explaining many smaller but useful enhancements of the XSLT language, this year's talk and paper will be about practical application of certain advanced concepts with everyday scenarios. We will see that using these techniques, once undone of the complexity of the language descriptions in the specification, are quite easy to master. The result is, in most cases, a cleaner stylesheet that can run on any processor and with any size of input. The particular use-case we will look at is about a Twitter feed, though the methods and practices explained will be applicable to any uninterrupted stream of data.

3. Setting up the environment

This paper will assume you will be using a conformant XSLT 3.0 processor. While the specification is not final

doi:10.14337/XMLLondon16.Braaksma01

yet, it is in *Candidate Recommendation* status[1], which means that the specification is generally frozen except for fixing bugs that are found by implementors of the spec. Conformant here means: conformant to the latest version of the spec, or the *Candidate Recommendation* in particular. As of this writing I am aware of two processors that follow the specification as close as possible: Exselt and Saxon. Other processors have announced that they are in the process of supporting XSLT 3.0 but I am not aware to what level they will support the streaming feature. You can download Exselt from http://exselt.net and Saxon from http://saxonica.com.

Considering that streaming is a specific feature that a processor does not have to support, it may only be available in the more extended editions of your chosen processor. When you choose an edition, make sure to choose one that supports streaming. This may be a part of the Enterprise Editions of the processors, or available as an add-on to the license.

Where this text uses processor-specific options, it will use the syntax of Exselt. This is primarily related to commandline options, which are somewhat different between processors.

3.1. Quick guide to the commandline syntax of Exselt

This section will briefly explain parts of the commandline syntax of Exselt. It is not meant to be complete, but will give you enough ammunition to run the examples in this paper and to experiment a bit beyond that.

Other processors have similar syntax and commandline abilitites. You may even find that many options work the same way or use the same switches. For more information, refer to your processor's documentation.

Some commandline parameters can take either a URI or an XPath expression (with, by default, the XSLT stylesheet document root as context item, except for the `-xsl` commandline switch itself). Where the expression is ambiguous, for instance the commandline parameter-`xml:file.xml` is both a valid URI and a valid XPath expression, it will be treated as a URI. To have it treated as an XPath expression use the syntax -`xml:xpath=file.xml`, which in this case may not make much sense and return the empty sequence. You can use the syntax `-xml:uri=file.xml` to signify that it is an URI, but this is also the default.

You can use spaces in your XPath expression, but this may create an ambiguous commandline, in which case you should wrap the expression in single or double quotes.

Commandline parameters that do not take a URI always take an XPath expression, though in many cases this will be a numerical or string constant. For instance, -`param:$foo=12*12` will set parameter `$foo` to the integer `144`

Commandline parameters that take a namespace sensitive *QName* should either use the *EQName* syntax, or use the prefixes of the namespace bindings of the loaded stylesheet.

The following sections explain the commandline switches needed to run the examples. More commandline options are available, check documentation of your processor. The Saxon notes below are based on the official documentation found online.

3.1.1. Commandling switches for stylesheet invocation

`-xsl` Sets the principal stylesheet module or the top-level package. Takes a URI, a local file name, or an XPath expression as input. This parameter is always required.

Ex: `-xsl:twitter.xsl` will run that stylesheet from the current directory.

Saxon uses `-xsl` as well.

`-xml` Sets the input document, which means, it sets the *global context item* and the *initial match selection* to an instance of that document. Takes a URI, a local file name, or an XPath expression as input. This parameter is optional. If absent, the *initial match selection* and *global context item* will be absent unless you use `-ims` or `-gci`, which are more versatile.

Ex: `-xml:feed.xml`

Saxon uses `-s` instead, but syntax is the same. Saxon also supports selecting a directory for input.

`-o` Sets the URI for the principal output document, that is, the location of the output set by the unnamed `<xsl:output>`. Takes a URI, a local file name, or an XPath expression as input. If absent, `stdout` is used. The target location must be writable.

Saxon uses `-o` as well.

`-text` Sets the input document, but here sets the *global context item* and the *initial match selection* to a string as if `fn:unparsed-text` was called. Takes a URI, a local file name, or an XPath expression as input.

Saxon has no equivalent.

`-text-lines` Sets the input document for streaming of text files. That is, it will set the *global context item* to *absent* and the *initial match selection* to a sequence of

[1] See announcement on the xsl mailing list, https://www.oxygenxml.com/archives/xsl-list/201511/msg00025.html

strings as if the argument was called with the function `fn:unparsed-text-lines`. This mode of invocation was added since using this function inside your stylesheet is *not* necessarily streamable, but using it with this commandline argument is *always* streamable. This allows you to process large text documents using streaming. Takes a URI, a local file name, or an XPath expression as input.

Saxon has no equivalent, but you can reach the same behavior through the API.

`-gci` Sets the global context item. The g *lobal context item* can be different from the *initial match selection* (in XSLT 2.0 these were always the same). Takes a URI, a local file name, or an XPath expression as input, or the special value `#absent` to force it to be absent[1] or use the special variable `$ims` to access whatever the *initial match selection* is set to.

Ex: `-gci:"map { 'subject' : 'Opera' }"` will set the *global context item* to this map. sets the global context item to a map with one key, `'subject'` set to the string `'Opera'`.

Ex: `-gci:$ims[2]` will set the *global context item* to the second item in the *initial match selection* , which may be the empty sequence.

Saxon has no equivalent yet, but this may change due to this being a new addition to the spec.

`-ims` Sets the *initial match selection* [2], can be used instead of `-xml` to provide more control. They cannot both be present. This commandline switch allows for *any* item or sequence of items to become the initial match selection. Takes a URI, a local file name (in which cases it behaves similar to `-xml`), or an XPath expression.

Ex: `-ims:"1,2,3"` sets the *initial match selection* to a sequence of three integers, which will each be processed in turn.

Ex: `-ims:"collection('.?select=*.xml')"` will set the *initial match selection* to all documents in the current directory.

Saxon has no equivalent, though you can reach similar behavior through the API or by using the `-s` option with a directory. However, semantically these approaches are different.

3.1.2. Commandline switches for stylesheet invocation

The switches in this section are not required. If absent, the default behavior with an input document is to start with *template based processing* and the initial mode set to the default mode. Without an input document, the default is to start with *named template processing* , with the initial template set to the special value `xsl:initial-template`. Use the following commandline syntax to override the default behavior.

`-im` Sets the initial mode(s). If all of `-im`, `-it` and `-if` are absent and one of `-xml`, `-text`, `-ims`, or `-text-lines` is present, then defaults to the default mode as specified in the stylesheet, which itself defaults to the nameless mode. Takes a sequence of *EQNames* , each name separated by a comma, that correspond to modes in the stylesheet, or the special values `#default`, `#unnamed` or an XPath returning a sequence of *QNames* . The stylesheet is invoked with the same arguments for each mode in the sequence.

Saxon uses `-im` as well, but for namespaced *QNames* , you must use the *EQName* syntax. It doesn't accept the special names.

`-it` Sets the initial template(s), syntax and behavior is similar to `-im`. If neither of `-im`, `-it`, `-if`, `-xml`, `-text`, `-ims`, or `-text-lines` is present then default to `xsl:initial-template` [3]

Ex: `-it:main` to set the initial template to `main`.

Ex.: `-it:first,second,third` will run the stylesheet three times with each time a different initial template.

Saxon uses `-it` as wel, with the same restrictions as for `-im`.

`-if` Sets the initial function(s), syntax and behavior is similar to `-im`, except that it has no special values. To set parameters, use the nameless parameter syntax explained

[1] When using streaming with the `-xml` commandline option, by default you would set both the *global context item* and the *initial match selection* to the same document. But in many cases this is illegal with streaming. Adding `-gci:#absent` you can override this behavior by telling the processor to use no global context item at all. Alternatively, you can set it to a document, for instance a document with settings or configuration options, to be used with your streaming input, but itself to be read without streaming. To prevent such complex commandline scenarios altogether, you can force this behavior from within your stylesheet by using `<xsl:global-context-item use="absent" />`.

[2] The *initial match selection* and the *global context item* can now be set independently. This was done in XSLT 3.0 to allow for any input to be used as initial input for the matching templates: it can be a string, a sequence of dates, a document, a map or it can be absent. This by itself would create a controversy as to what the *global context item* (the item accessible with the expression `.` from global variable, parameter, accumulator and key declarations) should be if the match selection is more than one item. Hence, XSLT 3.0 processors allow you to set a *different* global context item. In Exselt, this can be achieved with using both `-ims` and `-gci`.

[3] An addition to XSLT 3.0 was to allow similar behavior to `int main()` in C or C++, in other words, a starting point where processing starts if no other arguments are present. For XSLT this is the template with the special name `xsl:initial-template`.

below, or use typical function-call syntax, as if you are calling the function from XPath.

Ex.: `-if:my:sum((1,2,3,4))` calls the `my:sum` function with a sequence of four integer.

Ex.: `-if:my:start` calls the `my:start` function, optionally with whatever you put in `-param`.

Ex.: `-if:my:add(12,30),f:test('foo'),x:start` will call the three functions `my:add`, `f:test`, `x:start` with other commandline parameters the same. The results will be concatenated as with typical sequence normalization.

Saxon does not yet have a commandline switch for invoking a function, but you can achieve the same result through Saxon's API.

`-param` Sets the global parameters, or the nameless parameters for the `-it` invocation. There are two distinct forms. Either `$var=expr` or `expr`, where `expr` is a regular XPath expression with other parameters, global variables, accumulator and even stylesheet functions in scope of the expression. This commandline switch can be repeated multiple times, where order may be important if you cross-reference parameters. The dollar sign is optional.

The order of nameless parameters must match the order of the stylesheet function declaration and can only be used with `-if`. The effective type of the param must match the type of the declaration.

This commandline switch can be used to set global parameters, inherited parameters from used packages, parameters of stylesheet functions, parameters of initial named templates, initial values for tunneled parameters for template invocation.

Ex.: `-param:"fourty" -param:42` sets the first nameless param to a string and the second to an integer.

Ex.: `-param:$foo=42` sets the parameter `$foo` to an integer.

Saxon uses a similar syntax, but without the `-param`, all parameters must come at the end of the commandline and take the syntax `key=value`. Use `?key=value` if you want the value to be interpreted as an XPath expression. Saxon does not allow the dollar sign to be prepended.

3.1.3. Commandline switches to manipulate streaming behavior

`-istream` If set to `yes`, or used without argument, forces the initial match selection to be loaded as streamed documents. In case you run it against a sequence of documents, each document will be streamed. This will raise an error if the initial mode is not streamable, that is, `xsl:mode` must have an attribute `streamable="yes"`. It is ignored when you use `-it` or `-if` invocation. This option is normally not required, unless to differentiate between the global context item and the initial match selection and which of the two are streamable, or when loading a collection of documents.

Saxon has no equivalent, though this may be possible through the API.

`-gstream` If set to `yes`, or used without argument, forces the global context item to be loaded as a streamed document. This means that each global `xsl:variable` and `xsl:param` and other declarations that access the global context item must be motionless. By default, the global context item is assumed to be non-streamable, but if you run your stylesheet with the `-xml` option and the initial mode or default mode is streamable, the global context item will also be streamed and above rules apply. Using this option with the `-gci` option you can initiate a transformation with a streaming initial match selection and a non-streaming, or streaming and different global context item. Note that, if the `xsl:global-context-item` is present and has the `streamable` property set either implicitly or explicitly, this option should either be omitted or set to the same value.[1].

Saxon has no equivalent, though this may be possible through the API.

`-xstream` Sets a special mode of operation for reading the stylesheet XML itself. It serves the situation where the data you want to stream is inside the stylesheet as *data elements* in the root. This mode assumes that the stylesheet, if read in memory at once, would be too large because of these data sections. Such a stylesheet *must* have all XSLT declarations *before* the data elements. All data elements *must* come *after* the last declaration and just before the closing `</xsl:stylesheet>` or closing `</xsl:package>`. This allows the compiler to read and compile the stylesheet or package without reading through all the data elements and without running out of memory.

Saxon has no equivalent.

`-sr` Sets the *StreamReader* to be used for streaming. The default StreamReader accepts XML. For the purpose of this paper, an additional StreamReader was added to read RSS feeds as an indefinite stream, that is, it will poll for new messages and not close the XML document. Message are read with most recent last. To use this StreamReader add `-sr:RssStreamReader` to the commandline.

Saxon accepts alternative *XmlReaders* , which you must enable on the *classpath* .

[1] A recent discussion in the XSL Working Group showed the necessity of such flexibility. At the moment of this writing, this bug is still open, but I assume that all combinations mentioned here will be valid according to the XSLT 3.0 specification, or allowed as extensions. See bug number 29499 in [2].

`-ff` Sets the *flush frequency* . The frequency is in Herz and takes the format of an `xs:decimal` literal or an XPath expression. For instance, setting `-ff:0.1` will flush every 10 seconds and `-ff:42` will flush 42 times per second. The default is `-ff:0.1`. The default can globally be overridden by setting the environment variable `EXSELT_FLUSH_FREQUENCY`. See also the section on flushing.

I've asked Saxon by mail on how to do this in Saxon and the answer was that it can *probably* be achieved by configuring a non-buffering output stream.

3.2. Running the examples

With the commandline arguments in place you should have enough information to play around with the examples or to experiment yourself. The examples can be used with any [8] stream. For the purpose of these examples, we use the Twitter RSS Feed Google script made by Amit Argarwal, see [9]. It requires no programming skills to set it up. It does require a Google account. Simply follow the instructions on the referred to web page[1].

To run a stylesheet with a streaming input document, perhaps the simplest is to use `xsl:stream` in your code. For simplicity and shorter examples, the code in this paper will assume (explained in the next section) that the *initial mode* is the nameless mode (this is the default since XSLT 1.0) and that this mode is explicitly set to be streamable using the declaration `<xsl:mode streamable="yes" />`. However, it is possible that your processor does not support all commandline options mentioned in the previous and this section. You can force the examples to work with such processors by using the `xsl:stream` instruction to load the external streamed documents.

The simplest way to call a stylesheet is to have no input document at all. The examples in this paper will *not* use this approach though. Add a named template such as the following (the name is pre-defined in XSLT 3.0) to your stylesheet:

```
<xsl:template name="xsl:initial-template">
  <xsl:stream href="streamed-source.xml">
    <!-- applies the principle node to the
         streamed default mode -->
    <xsl:apply-templates/>
  </xsl:stream>
</xsl:template>
```

Having this in place, where the `href` attribute points to your streamed source document, you can run the examples as well with the following simplified commandline, which will auto-select the default initial named template:

```
exselt -xsl:example.xsl
```

Most examples, unless otherwise mentioned, can be run by using the following commandline with Exselt. For the RSS examples to work with other processors, notably Saxon, you may need to create the equivalent of the StreamReader mentioned in the `-sr` commandline argument. At the time of this writing I do not know if Saxon supports indefinite streams, the question is still open.

```
exselt -xsl:example.xsl
   -xml:feedurl.rss
   -sr:RssStreamReader
   -gci:#absent
```

The last line in the above example forces the *global context item* to be absent. The default behavior of the `-xml` argument is, for backwards compatibility reasons with XSLT 2.0, to set both the global context item and the initial match selection to the same input document. But, as mentioned before, setting the global context item to a streamed document can be problematic, it is generally favorable to set it to the special value *absent* . A cross-processor solution is to simply always use the declaration `<xsl:global-context-item use="absent" />` in your code to prevent this from happening. Exselt will detect this scenario and will prime your stylesheet without the global context item set if you use `-xml` or `-ims` commandline options without a specific `-gci` option.

For examples that require a separate global context item, the following commandline can be used:

```
exselt -xsl:example.xsl
   -xml:feedurl.rss
   -sr:RssStreamReader
   -gci:"document('settings.xml')"
   -gstream:no
```

If you want to try the examples with multiple RSS feeds, which will mean they will be processed one after another requiring the first feeds to be terminal, you can use the following commandline, which uses the *initial match selection* to set multiple streams. This can also be achieved by code by using the `xsl:stream` instruction on each

[1] Prior to the conference, all examples will be made available on http://exselt.net, including a copy of these instructions in case Argarwal's blog disappears, and a ready-made twitter feed that can be configured with search criteria.

URI. In Saxon you can get similar behavior by using a directory search pattern.

```
exselt -xsl:example.xsl
-ims:"document('feed1.rss','feed2.rss','feed3.rss')"
  -istream:yes
  -sr:RssStreamReader
  -gci:"doc('settings.xml')"
  -gstream:no
```

Finally, if you want to experiment with the *global context item* itself being streamable, which requires the `xsl:global-context-item` to be present with an attribute `streamable="yes"`, you can use the following commandline:

```
exselt -xsl:example.xsl
   -xml:feedurl.rss
   -sr:RssStreamReader
```

The above will use the following defaults automatically:

```
exselt -xsl:example.xsl
   -xml:feedurl.rss
   -sr:RssStreamReader
   -gci:feedurl.rss
   -gstream:yes
```

In other words, it will use the feed from the `-xml` argument as the global context item. This allows you to access the streamed document from within global declarations like `xsl:accumulator` and `xsl:variable`. Be aware that this is an advanced concept and that you can only use motionless expressions in the global declarations (that is, the global declarations are not allowed to advance past the root element).

3.3. Boilerplate for the examples

The examples, for the sake of brevity, will omit boilerplate code. The templates, functions and other declarations should be put inside the following base stylesheet used for streaming. Without additions, it will act as an identity template:

```
<!--
   using xsl:package instead of xsl:stylesheet
   enforces that modes have to be declared, which
   prevents type errors
 -->
<xsl:package
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  expand-text="yes"
  version="3.0">

  <!-- sets the initial default
       mode to streamable -->
  <xsl:mode streamable="yes"
            on-no-match="shallow-copy"/>

  <!-- remove the xml decl in the output -->
  <xsl:output method="xml"
              omit-xml-declaration="yes"/>

  <!-- prevent the global context item
       to be there at all -->
  <xsl:global-context-item use="absent"/>

</xsl:package>
```

4. Understanding difference between global context item and initial match selection.

In a typical transformation scenario there's a triplet of a stylesheet, that contains the programming logic, an input document that needs to be processed, and an output document containing the result.

That is no different with streaming. However, you need to instruct the processor that you want to use streaming. This can be done by using `<xsl:mode streamable="yes" />`. This will set the initial mode to be streamable and when you call the stylesheet in the normal way, the processor will detect that it should read the input document using streaming.

However, in XSLT 2.0 there was always only one input document and the processor would present the stylesheet with a document node of that input document. In XSLT 3.0 this has changed. While the above is still the default for backwards compatibility, the input *sequence* can now be anything, ranging from an empty sequence, to a sequence of strings, dates, integers maps etc, to a sequence of a single or multiple documents, or even a mix of all of the above.

Since global variables and parameters and the like can access the context item, for instance with `<xsl:variable name="settings" select=".//settings" />` to retrieve an element named `settings` from the input document, the question rises, if the input contains multiple items and not necessarily documents, what is this context item set to?

To differentiate between the two, XSLT 3.0 introduces the *global context item* and the *initial match selection* . They do not have to be the same. It is up to the processor's API or commandline interface to set these to different values. It can be expected, but is by no means necessary, that if the input sequence, which is the *initial match selection* , is a sequence of more than one, that the first item will serve as the *global context item* .

The commandline reference summary in the previous section explains how to set this for Exselt, for Saxon you can set this by API only, as of yet there is no way to set this to different values using commandline parameters only.

4.1. Relation between streaming, global context item and initial match selection

When it comes to streaming, the processor can easily detect when there is a single input document and when the stylesheet has an `<xsl:mode streamabe="yes" />`. However, for the *global context item* this is not so trivial.

A streamable construct is a construct that has expressions or nested constructs that together are *guaranteed streamable* . This paper will not discuss the rules of guaranteed streamability, other papers are available for that, including some of myself. In case of the global context item, for a global declaration like `xsl:variable` to access a streamed input document, the processor must be informed about this by the use of `<xsl:global-context-item streamable="yes" />`, which in turn puts restrictions on the global declarations: they may only contain grounded, motionless constructs.

It is rarely necessary to use this mechanism, unless you want to maintain some information, for instance the input document URI, for reference in other constructs.

A more serious issue arises if you would access the global context item and you do not specify that it must be streamable. Suppose you want to read the settings of the input document as explained above. Such a construct would be illegal with streaming.

A scenario like that can be achieved by setting the global context item to a *different* document than the initial match selection and to override that it must *not* be streamed. You can get to this behavior by using the `-gci` and the `-ims` commandline switches together with the `-istream` and `-gstream` switches to instruct the processor that the global context item should, or should not be streamed. This scenario is most useful when the input document should be streamable, but the global context item should not.

Note that, if you set the global context item to a streamed document and you do not provide the same item in the initial match selection, that you cannot access anything else than the root element of that document. While an allowed scenario, in practice this is of little use.

5. Reading an uninterrupted stream

An uninterrupted stream, eternal stream, neverending stream is a stream that has a start, namely the moment the stream is requested, but no end. Processing such streams poses an extra challenge on processors because it requires them to do intermediate flushes, otherwise the result document will be created in memory, but never released, leading to an eternally running processor but no output document.

One of the simplest conceivable uninterrupted streams is a time ticker. For purposes of testing the examples in this paper I have created an online time ticker that can be found at http://exselt.net/time-ticker. The stream looks something like this:

```
<ticker>
  <time>2016-06-15T15:04:36+01:00</time>
  <time>2016-06-15T15:04:37+01:00</time>
  <time>2016-06-15T15:04:38+01:00</time>
  ...
```

Not a particularly entertaining stream, but it works as an example. It is somewhat similar to a *ping* command, that it will return an element each second and leave the connection open. It will never close the opening root tag.

You can process this stream using the example boilerplate code, store it as `timefeed.xsl`, without any additions and the following commandline:

```
exselt -xsl:timefeed.xsl
   -xml:http://exselt.net/time-ticker
   -gci:#absent
   -o:output.xml
```

You should now see a document `output.xml` that is growing each time a new element is read from the input stream. Since we use an identity template, seemingly nothing special happens and we output exactly the same as the input.

The added `-gci:#absent` is not required, but makes it clear that we do not want to set the global context item to the same as the stream, see previous section for a discussion.

Behind the scenes, this example uses the default matching templates for *shallow-copy* , which means that each node is processed and then its children are processed, and when there's no match, they will be copied. The behavior is similar to the identity template using `xsl:copy` on non-matched items. This is different from *deep-copy* , where the children are not processed separately and once there's no match, the whole node and all its children are copied to the output stream, similar to `xsl:copy-of` [1].

5.1. Elaborating on the time-ticker

Let's expand a bit on the previous example and do something a little bit more useful. Let's print the time in a more humanly readable format, by adding the following to the example:

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

<xsl:template match="time">
  <current-time>{
    format-time(., '[H01]:[m01]:[s01] [z]')
  }</current-time>
</xsl:template>
```

The output will now look something like:

```
<current-time>23:45:12 GMT+1</current-time>
<current-time>23:45:13 GMT+1</current-time>
<current-time>23:45:14 GMT+1</current-time>
...
```

We deliberately didn't output a root element, as we won't be able to close it anyway[2]. By creating a sequence of root elements, it may be easier to post-process this stream, but of course that is up to whatever application you want to process this further with.

Let's look at the example a bit more closely. The first template, which has no *mode* attribute so it sits in the unnamed mode, which is streamable because we use our boilerplate example, skips the root element and processes its children elements. This is called a *downward selection* or more properly, a consuming expression. In streaming, consuming expressions are allowed (they literally *consume* the input stream, i.e., they move the reading pointer forward through the stream data), as long as there is a maximum of one consuming expression per construct. All examples in this paper will use proper *guaranteed streamable* expressions, for a discussion of such expressions, several papers, talks and tutorials are available online, on overview of which can be found at http://exselt.net/streaming-tutorials.

The second template operates on `time` elements and formats them in a more humanly readable format. The curly brackets act as *text value templates* , which is similar to *attribute value templates* from XSLT 2.0 and can take any XPath expression, but can be applied to any place where a text node is created, provided you have the `expand-text="yes"` in a parent element. We set it already on the root `xsl:package`, so we can use this syntax everywhere.

This template, inside the *text-value template* uses the function `fn:format-time` with the *context item expression* `.`. This function consumes its first argument and formats it. Since this whole matching template has only one consuming expression, it is streamable.

6. Challenges with uninterruped streaming

Several challenges exist for processors that support uninterrupted streaming that are not directly addressed by the specification, which means the specification leaves it the API of the processor to address those challenges.

First of all, it is no requirement at all that processors are able of reading an uninterrupted stream. Supporting streamability means that processors can process arbitrary large XML documents, it doesn't specify anywhere what the maximum size is, though it implies that the size can be unlimited, that is, an eternal stream, instead of a large

[1] With uninterrupted streams it can be dangerous to use `fn:copy-of`, `xsl:copy-of` and similar instructions, especially when operated on the element that will never be closed, in this case the root element `ticker`. Since these *copy-of* instructions are supposed to read to the end of the element, and there is no end of an element, it will lock your processor until it is forcibly closed. It may also crash your processor, as essentially you are telling the processor that you can read the whole element in memory, which in this case clearly isn't possible, leading to an *out-of-memory* exception.

[2] It is possible that processors may provide a mechanism for closing the output stream when the stream is interrupted, but this is API design and out of scope of the specification. In fact, it more likely that the XmlReader you are using can close the stream neatly when it is interrupted, providing, in this case, the closing `</ticker>` to keep the XML well-formed.

document that is too large for memory, but nonetheless has a beginning and end.

Secondly, processors are not required, though encouraged, to provide limited buffering on the output stream. If a processor does not provide buffering, and your stylesheet is such that the output stream grows porportionally to the input stream, it will eventually, given enough time, run out of memory. Furthermore, in such cases the output will never appear physically, as it is maintained in memory and not flushed in-between.

A third challenge is how to deal with interrupting the unlimited stream. Suppose you would simply hit *Ctrl-C* on the commandline, the processor will crash with an error and what is written to the output may be a broken document. The API may in such case provide a mechanism to close the stream properly.

Several of those mechanisms have already been discussed above with the commandline reference (see the `-ff` switch to control the flush frequency). A further improvement could be to allow flushing at a number of elements or a number of bytes. Or to allow no flushing at all, but to wait until the stream is completed and force the processor to keep everything in memory.

At present, Exselt, and I believe Saxon too, does not provide a mechanism by default to break the stream in a neat way. However, it can be expected that such options will become available in the near future. For now it means that if you interrupt the stream forcefully, the behavior is processor-dependent. If a part of the stream is already flushed, at least that part will be readable.

6.1. Dealing with errors

XSLT 3.0 introduces `xsl:try/xsl:catch` for catching and recovering from errors. This instruction comes with the attribute `rollback` which takes a boolean value. If set to `"yes"` it instructs the processor to buffer enough of the output (and not flush it) so that it can recover from the error by rolling back. An alternative mechanism can be provided by use *safe points* or *recovery points* .

This approach is not very helpful for errors resulting in unlimited streams, as that would require unlimited buffering for the processor to recover from a potential error. On a smaller scale you can still use this though, for instance by using a try/catch around a leaf node, in our example we could do it inside the `time` element:

```
<xsl:template match="time">
  <xsl:try rollback="yes">
    <current-time>{
      format-time(., '[H01]:[m01]:[s01] [z]')
    }</current-time>
    <xsl:catch>
      <current-time>invalid time</current-time>
    </xsl:catch>
  </xsl:try>
</xsl:template>
```

This works as can be expected. Where we to use this without the rollback attribute, the processor may not be able to recover from the stream leaving the output in an inderminate state.

In case we would wrap the whole stream in a try/catch we would everntually run out of memory, because everything would need to be buffered to allow rolling back. In practice this mechanism is only useful on smaller elements that can easily be buffered.

7. Processing a twitter feed

Now that we have seen how a trivial uninterrupted feed can be handled, let's see how we can process an unlimited RSS twitter feed.

The URL created from the section on obtaining the Twitter feed as RSS will look something like *https://script.google.com/macros/s/AKfycbxSzab_rjrOSSF1s6N-C5kjXLdD0ZQZx-Zu3sqaeKS3Y38Bd6Y/exec?730658803637727232* (on one line without spaces).

Using the *RssStreamReader* mentioned before, with this URI, will create a stream of the following format:

```
<rss version="2.0"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xmlns:media="http://search.yahoo.com/mrss/">
  <channel>
    <title>Twitter RSS Feed ...</title>
    <!-- some information
            removed for readability -->
    <item>
      <title>
        <![CDATA[Abel Braaksma: @gimsieke lol,
        you got me, I thought it was
        http://twitter.com&nbsp;
        acting up after my browser
        crashed and restarted ;p]]>
      </title>
      <pubDate>2016-02-20T11:10:23+0000</pubDate>
      <author>Abel Braaksma</author>
      <!-- some info removed for readability -->
      <description>
        <!-- descr in esc. XHTML markup -->
      </description>
    </item>
    <item> ... next item
```

If you open the URI with a browser, it will show the most recent first. When you open it with the *RssStreamReader* the most recent item will come last. Whenever a new tweet is sent to this Twitter account, the reader will read it and append it to the input stream.

Using the commandline syntax discussed earlier we can read this stream as an uninterrupted stream. The closing `</rss>` will not be sent to the processor using our reader, making it easier to deal with the stream.

7.1. Getting the Twitter descriptions

To get started, let's take our boilerplate code and try to output only the description:

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

<xsl:template match="item/description">
  <div>
    <xsl:copy-of select="parse-xml-fragment(.)"/>
  </div>
</xsl:template>
```

This example skips over everything and then loads the *XML-as-string* using the new XPath 3.0 function `fn:parse-xml-fragment` [1] to interpret this escaped XML as proper XML and output it.

The result is that on each flush, an element `<div>` is added with as its content the tweet from the current feed. Those `div` elements could be appended to any existing HTML page, for instance by using AJAX technology.

7.2. Processing the result

Since we are dealing with uninterrupted feeds, the result will also be an uninterrupted feed. To process this further, we need to feed it to a system that can process such streams. Since we are currently outputting XML this may not be trivial and moves the burden of processing an unlimited stream to the next tool in the chain.

An easier way out is to use the `xsl:message` mechanism of XSLT. This creates a document node each time it is called and this document is fed to a *message listener* . Both Saxon and Exselt provide an easy interface to create a listener through the API which can react to new messages whenever they appear and process them further, for instance by sending them to a database, making a *push message* to a mobile phone app or turning it into a new feed for a web page.

The example above would look as follows when using this technique:

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

<xsl:template match="item/description">
  <xsl:message>
    <div>
      <xsl:copy-of select="parse-xml-fragment(.)"/>
    </div>
  </xsl:message>
</xsl:template>
```

7.3. Accumulating data of the Twitter feed

Just processing a Twitter feed as explained above is not particularly challenging and can be achieved by a myriad of other techniques as well. But what if you want to accumulate data, let's say the *nth message* send in this feed?

[1] A similar function exists that is called `fn:parse-xml`, but that requires a single root node. This is not guaranteed with this Twitter feed, so we use `fn:parse-xml-fragment`, which wraps it in a document node.

This cannot be done by using standard XSLT 2.0 mechanisms. This was recognized when XSLT 3.0 introduced streaming and the `xsl:accumulator` instruction was added for this purpose. An accumulator is created declaratively and globally and does what its name implies: it accumulates data that is passed through as a stream. You can look at is as a way of appending meta-data to elements and other nodes that pass through while streaming, which can be read out at a later stage using the functions `fn:accumulator-before` and `fn:accumulator-after`.

The reason this cannot be done is because in streaming, you cannot *look back* or *peek forward* . In other words, you simply cannot do something like `count(preceding-sibling::*)`, which might suffice here to simply count all the tweets that have been passed through up until the current tweet.

The following example shows how we can count with an accumulator, expanding on our existing example:

```
<xsl:accumulator name="count"
                 streamable="yes"
                 initial-value="0">
  <xsl:accumulator-rule match="channel/item"
                        select="$value + 1"/>
</xsl:accumulator>

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

<xsl:template match="item/description">
  <xsl:message>
    <div>
      <p>This is message {
        accumulator-before('count')
      }</p>
      <xsl:copy-of select="parse-xml-fragment(.)"/>
    </div>
  </xsl:message>
</xsl:template>
```

As long as we do not restart or interrupt the stream, the accumulator will continue to add 1 each time it encounters a new `<item>` and we can access this value by using the `fn:accumulator-before` function. This function is *motionless* , which means we can use it as often as we want inside a single construct. In fact, the accumulator itself must be motionless as well.

7.4. Accessing the global context item to set defaults for the accumulator

The previous section showed how to count messages. But what if we know the stream will not start with showing all the past messages, but will start with the current message, and we have a different XML feed that will return the total messages processed?

There are several approaches to this, the approach I will present here is by using a settings document that contains the necessary metadata and set this as a *non-streamable* global context item. Use the `-gstream:no -gci:settings.xml` as additional commandline parameters.

Suppose we feed our processor an additional settings document that, for the purposes of keeping the example brief, looks as follows:

```
<settings>
  <!-- set how many have been processed already -->
  <tweet processed="325" />
</settings>
```

With the commandline parameters specifying that the the global context item is not streamed (if it were streamed it would be illegal unless we also have the `xsl:global-context-item` with `streamable="yes"` present as mentioned previously) we can update our accumulator as follows:

```
<xsl:accumulator name="count"
  streamable="yes"
  initial-value="xs:integer(
                   ./settings/tweed/@processed)">
  <xsl:accumulator-rule match="channel/item"
                        select="$value + 1"/>
</xsl:accumulator>
```

Normally, if the global context item were indeed streamable and we had added the `xsl:global-context-item` to make the stylesheet valid, we would not be able to use the expression above, because all expressions that access the global context item must be motionless. Traversing down the tree is *consuming* , which is not allowed.

Since we specified explicitly that we can read the settings document in one go, without streaming, the consuming expression is of no influence to the streamability and the output will start counting with the number `325` as expected.

7.5. Expanding on the Twitter feed example

We have seen some trivial examples that use an RSS feed as unlimited XML stream. The examples here were necessarily trivial to show how this process works. To expand on these examples, for instance by adding layout, more information, process timestamps of the feed etc., you can simply append the necessary matching templates. Keep it simple, as each template is allowed at most one downward expression[1].

To expand on the accumulator, for instance to provide a character-average, or other calculations, you can add new accumulators or update the existing ones. While doing this, make sure that the expressions you use do not consume the tree. This can be tricky at times, but you can circumvent more complex scenarios by referencing accumulators from each other to create more complex expressions and to refer or combine calculations.

A more elaborate example that shows this and other techniques is available online from http://exselt.net/papers and will also be presented at XML London 2016.

8. Conclusion

While XSLT 3.0 does not directly address unlimited XML feeds, the streaming feature comes with enough capabilities for processors to support it without really resorting to processor-specific extensions. While some of this paper discussed processor-dependent ways of invoking stylesheets, the principles shown here can be applied with any capable processor, provided they support intermediary flushing.

I've kept the examples necessarily brief to focus on the matter at hand: uninterruped, unlimited streaming. We've seen that we can process a feed of timestamps, a Twitter feed or essentially any feed. Using techniques presented here you can process XML streams that are indefinite, a capability that was not available in XSLT 2.0, but could now become a mainstream approach for the many live feeds that we work with everyday on our phones, websites or apps.

Bibliography

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

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

[3] *XML Path Language (XPath) 3.0, W3C Recommendation 08 April 2014*. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. http://www.w3.org/TR/xpath-30/

[4] *XML Path Language (XPath) 3.1, W3C Candidate Recommendation 17 December 2015*. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. Discussed version: http://www.w3.org/TR/2015/CR-xpath-31-20151217/, Latest version: http://www.w3.org/TR/xpath-31/.

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

[6] *XSLT 3.0 new features*. Dimtre Novatchev . 2014. Pluralsight. https://www.pluralsight.com/courses/xslt-3-0-whats-new-part1

[7] *It's the little things that matter*. XML London. 2015. Abel Braaksma. doi:10.14337/XMLLondon15.Braaksma01

[8] *RSS 2.0, Really Simple Syndication.*. Dave Winer. http://cyber.law.harvard.edu/rss/rss.html

[9] Creating an RSS feed from a Twitter query. Amit Agarwal. 2016. Labnol. http://www.labnol.org/internet/twitter-rss-feed/28149/

[1] If you need multiple downward expressions, you can use the instruction `xsl:fork`, which creates a *forked stream* , that is, you can have any number of downward selections in the children of that instruction and they may even overlap, the processor will create multiple read pointers to deal with this.

XML, blockchain and regulatory reporting in the world of finance

Combining the strengths and weaknesses of mature and modern technologies to drive innovation and regulatory compliance

Lech Rzedzicki

Abstract

The regulatory burden for financial institutions makes it a hard environment to innovate, yet the unfavourable market conditions mean that banks must adopt the latest technologies to cut costs and keep up with the market demands.

This paper aims to show a technical proof of concept of how to combine the seemingly opposite goals of using a combination of tried and tested XML technologies such as XSLT and Schematron in conjunction with experimental distributed ledger technology to drive both regulatory compliance and implement innovative features such as inter-bank trade settlement using blockchain technology.

1. Background

1.1. XML in regulatory reporting

XML 1.0 is by now a very mature standard. As most of the audience will know, the standard hasn't changed in 20 years. XML 1.1 hasn't been widely adopted and no one is contemplating XML 2.0 to replace XML 1.0 anytime soon.

The strength of that is stability, great tool support, human readability and the reach.

This means XML still is, probably more than before, a good archival and reporting markup. Worst case scenario is someone will open it in 20 years time in a text editor and it will still be readable.

Enter the magical world of finance where common sense quickly disappears and is replaced by exotic, derivative products such as index variance swaps, whose main goal is to depart the client from their money.

It is hard to prove whether the complex financial product was engineered as a way to improve the bottom line of financial institutions or are truly filling a market need such as shielding clients from the volatility of the equities markets or commodity prices.

What is universally true however is that the products are complex and it is often hard to figure out who is exposed to risks and to what extent.

As a result various regulatory bodies, such as the FCA have started requiring more and more reporting on the trades, positions and risks.

As it happens XML is a great fit for this purpose and XML (and more specifically XML standard called FpML) has been a de facto standard for regulatory reporting to the extent that the regulatory bodies require that the reporting be done in XML

Regulatory reporting is one of the major considerations and driving forces for any established or upcoming financial organisations in 2016.

1.2. Fintech innovation and blockchain

Another trend in 2016 is the need to radically disrupt or reinvent and optimise financial services institutions.

Even armed with a full suite of exotic financial products and masses of individual and institutional clients, in an era of negative interest rates and fierce competition from non-traditional players such as supermarkets and tech companies, financial institutions find that a combination of low yields and high regulatory costs make it impossible to retain status quo and still provide sufficient returns back to clients and shareholders.

Executives in financial institutions are desperately looking for solutions and for many they have found the holy grail by the name of blockchain.

Originally devised by the mysterious Satoshi Nakamoto for the purposes of using with bitcoin, the concept of a distributed ledger is a powerful one.

Blockchain or distributed ledger uses solid and well known cryptography around factoring large prime numbers to make it hard to calculate and verify transactions on the ledger and mathematically nearly impossible to alter them by any single contributor.

doi:10.14337/XMLLondon16.Rzedzicki01

By now the technology has enjoyed a massive wave of early adopter hype and financial institution have been heavily looking at using it for trade settlement and smart contracts.

A single project called r3cev alone has gathered over 40 international financial institutions and has recently completed the early tests with 11 of them.

It is therefore an excellent example of an innovative technology that many financial institutions would like to try and we will be using it in the paper to showcase how innovation, XML and regulatory reporting can play together nicely.

1.3. Distributed ledgers history

Ledgers have been used throughout human history, from Sumerian records of harvests and crop usage 4500 years ago through to the tracking of bad debts by the Medici bank in the 14th and 15th Century. Today's modern double-entry ledgers can trace their roots back to early Islamic records in 7th Century and were first publicly codified in Venice in 1494 by Friar Luca Pacioli based on the practice of Venetian merchants of the Renaissance. Distributed ledgers represent the next evolution of these devices.

1.4. What is a distributed ledger?

A distributed ledger is essentially an asset database that can be shared across a network of multiple sites, geographies or institutions. All participants within a network can have their own identical copy of the ledger and any changes to the ledger are reflected in all copies in minutes, or in some cases, seconds. The security and accuracy of the assets stored in the ledger are maintained cryptographically through the use of 'keys' and signatures to control who can do what within the shared ledger. Entries can also be updated by one, some or all of the participants, according to rules agreed by the network.

It is important to understand that typically the design is such that you can only add new entries to the ledger and can not remove previous entries. If you wish to amend entries, you add a new, corrected entry and a reference the old one.

This also means that the ledger, whether it is stored as a file or in a database, will grow over time which is both a strength and a weakness. It makes it almost impossible to falsify previously added information, but it also makes it hard when you truly need to change the information – a person changes name, or there was any other factual error when entering the information on the blockchain.

1.5. What can you use distributed ledgers for?

There are three core use cases where distributed ledgers can be used:

- To secure an enormous range and type of value transactions – from simple money transfers, of which bitcoin provides the best exemplar, to complex registration of asset ownership and transfer such as in the financial services industry.
- To provably convey trust between parties and verify provenance, reputation or competence – certification of university qualifications, the status of components within a complex supply chain such as an aircraft engine manufacture.
- To serve as the backend execution environment, a distributed "computer in the cloud", where algorithms are executed on the blockchain rather than on any particular machine.

There is also a number of areas where people see blockchain as the holy grail and mistakenly think it can be applied to anything from preventing diamond forgeries to solving blood bank reserves inefficiencies.

As a general rule, whenever there is a part of the system that can not be put "on the blockchain", for example physical assets, or 3rd party IT systems, a blockchain solution is unlikely to work as information can be changed easily outside the blockchain.

1.6. Blockchain challenges

There are currently a number of fundamental challenges with blockchain based distributed ledgers, particularly around scalability and privacy of information on a public blockchain (many of these are resolved in a private blockchain systems).

The *scalability challenges* are numerous - the transaction limits for blockchain based systems are low (20 per second for Ethereum, a common next generation blockchain system), the energy requirement for proof of work based consensus systems is huge (the Bitcoin network is estimated to use as much energy as the republic of Ireland), although distributed amongst many parties, the requirement for each node to hold a copy of the full ledger means the cost of storing data on the blockchain is computationally high (although this is mitigated by other distributed storage systems such as IPFS).

Privacy challenges exist as a result of each node holding a copy of the ledger – all transactions are visible to all parties currently unless encryption is used on the data. The immutability of the blockchain also raises a potential legislative challenge around the European “right to be forgotten”. How do you erase records on the blockchain?

Due to the above concerns, usually the blockchain stores just a hash – a signature of the transaction or information and the private information is stored separately in a wallet and protected by a private key. This is turn means that if a private key is lost, it makes it impossible to verify any previous or future information for that key.

Distributed ledger can only be used to verify the information on the blockchain. If the data is falsified or tampered with before or outside of blockchain, a distributed ledger can only verify that the data is not on the ledger. An example here is diamonds and Everledger. Everledger takes several measurements of the diamonds (the way it is cut, the colour, size etc) and puts this information on the distributed ledger. If the diamond is lost or stolen and surfaces again, it can be easily identified by taking the measurements again and comparing the database stored on the distributed ledger. This parallel verifiable digital certification on the blockchain can assure buyers and sellers of the provenance of the item.

Unfortunately, all it takes for a savvy thief to avoid this is to change the qualities of the diamond slightly – in the process the diamond might lose some value, but it is very likely that a thief will prefer that to getting caught. To sum up and generalise this issue, any inputs or outputs that are not on the blockchain are vulnerable, hence the movement to smart contracts and trying to do as much as possible inside the blockchain.

More research and good education for decision makers is key here – this will prevent using blockchain where it is not appropriate – for example where data changes often, or high performance or efficiency is required. Many of these difficulties are the subject of active research projects, and in the UK, EPSRC are launching a call this summer to support the research further with up to 10 small scale project grants with a total of £3M.

1.7. The opportunity for distributed ledgers

Despite all the identified challenges, the opportunities for distributed ledgers are potentially huge. Systems such as Ethereum, Eris, NXT, R3CEV, with integrated smart contracts, offer the ability to place complex business logic on a public or private system where they can be triggered by transactions – contract signing, approval of work, external events etc.

The most direct beneficiary of distributed ledgers technologies are the platform developers (initially developing the private blockchain solutions) and the application developers. In common with many open source companies, the platform business model is to sell proprietary value around an open source core. One analogous example is Red Hat – a linux distribution provider that offers their core product free of charge and then offers consultancy, integration and development services to enterprises – recently they were valued at $2B.

Assuming there is large scale adoption, application developers stand to create value in the same way that the app ecosystem has developed on the Apple and Android mobile platforms.

Distributed ledgers allow for complex business processes between parties who do not implicitly trust each other to be automated and hence significant cost savings to be achieved in many sectors. The classic example is financial instrument trade reconciliation but supply chain is another commonly talked about use case. In many of the use cases, the core focus is about cost reduction through the removal of unwanted middlemen or through the reduction of duplicated effort across untrusted parties. Distributed ledgers may also improve transparency and efficiency, by ensuring that regulators and other third parties have full, real time views of transactions.

1.7.1. Smart contracts and Distributed Autonomous Organisations

Smart contracts are beyond the scope of this short presentation, but many in the world of blockchain speak about distributed autonomous organisations (DAO), written in code and deployed on the blockchain these lend themselves strongly to new business structures or digitisation of e.g. cooperatives. How these will develop is unknown at this point, but initial DAO’s have raised millions of pounds in blockchain crowdsales.

An example of that is Ethereum, where any computation cost can be covered by spending a virtual currency – Ether. Software developers wishing to execute their programs on Ethereum network can choose to outright buy the Ether computing units or provide the computational resources themselves and even sell the excess power in exchange for Ether/money. Such an infrastructure is somewhat similar to cloud services provided by Amazon or Google, where the price is set by the factors such as electricity costs and demand, but with distributed ledger, it is much more fair to smaller players and there is no single entity that can control the network and switch off an application.

The existence of such networks enables the execution of "Smart Contracts" - autonomous code running on the blockchain (as opposed to a single machine). Given certain conditions, the code can execute automatically.

A simple example here is bond coupon payment – in a typical bond issue, the buyers buy the bond for 100% of the price and the issuer repay the bond in instalments.

Traditionally this involves issuing paper certificates of bond ownership and manually sending money via cheque or bank transfer every month and keeping track of what's been repaid etc. Assuming that all the participants – bond issuer and the buyers are also participating in the same distributed ledger network that is capable of executing smart contracts and sending virtual currency, the process can be simplified vastly, possibly even removing the need for intermediaries such as banks (for sending money) and law firms (for writing up the contracts). In such a scenario a bond issuer, would issue a smart contract to be viewed, audited, verified and accepted by the buyers and upon accepting, the funds would be automatically transferred to the issuer. Likewise every instalment the issuer would automatically pay back the coupon payments.

Smart contracts can enable a whole range of scenarios, from distributing aid money, through voting, secure communications and probably a number of areas that have not been discovered yet. Still the challenges described above remain.

The successful exploitation of smart contracts requires solving the technical challenges, but also changing laws and regulation and ensuring that the disruption caused by the automation has a net positive effect on societies and the economy.

2. Thesis

To many inside and outside the finance industry, the suffocating combination of a low yielding market and the burden of regulatory reporting may feel like a fatality combo from Mortal Combat.

It isn't and like many combos in fighting games it can be blocked or better yet countered.

What this paper describes is a proof of concept technical solution to solve two seemingly opposite problems - use tried and tested XML technologies to solve compliance issues for financial institutions while at the same time allow for innovative technologies such as distributed ledger to be used alongside.

XML is a very mature and stable standard. It has built a great ecosystem of technologies that enable financial institutions to reliably solve their regulatory reporting requirements.

XML does allow mixing in with modern ideas such as distributed ledger or modern DevOps stacks such as Docker and modern noSQL solutions such as Marklogic and in fact makes it easy to do so, thanks to a few surrounding standards such as XPath, XSLT, XQuery or Schematron.

The aim of this presentation is to show that XML is a great fit for financial institutions to deliver both the business as usual activities such as regulatory reporting and to explore new areas such as distributed ledger at the same time.

3. Technical Description

This section describes a sample journey of a financial transaction - a client requests FX swap between a client and a bank, the bank executes the transaction, which is then published to the distributed ledger and the authenticity is jointly verified on the distributed ledger by the client, bank and the regulatory body.

To best illustrate this, there are three separate instances running blockchain - one for the client, one for the bank and one for an imaginary regulatory body called the Fictional Compliance Authority.

We aim to show how to add a transaction, how to sign it, add it to the blockchain, how to prove the authenticity of a given transaction, how to run a few basic tests for the correctness of the message (using Schematron), what happens when a bank or a rogue party tries to falsify data on a blockchain and finally showcase a few good use cases where XML technologies show their strengths - transforming from raw CSV input to FpML, transforming from FpML to a hash and plain text ledger, generating reports and graphs.

3.1. Infrastructure - Stone Circle

Kode1100 blockchain proof of concept uses internally developed infrastructure inspired by the Stone Circles found in Gambia, West Africa.

Built thousands of years ago that continue to stand today, and are a good symbol of stability and reliability.

Financial institutions have stringent requirements about robustness of the infrastructure. As result they often go for the tried and tested technology as opposed to the cutting edge. Our early proof of concept was a setup of Docker containers running Python code + open source database. None of that proved to be a good fit and the following setup more accurately depicts the needs and wants of the IT managers at the financial institutions that we talked to:

- Operating System: Digital Ocean Virtual Machines running CentOS 7 Linux. No special requirements here, as long as the Operating System is capable or running Java.

  Possible environments where this proof of concept can run (with slight modifications) are: raw Linux Debian or Ubuntu, Linux on top of Docker, Amazon AWS, Azure etc.
- Blockchain and "glue" code - Java EE 7. We opted for Oracle as this caused least amount of problems and mimics the bank environments that we know of. As discussed earlier implementation for other platforms and languages are possible - for example, we have started our proof of concept using Python and most of Bitcoin code is written in C.
- Marklogic 8. It is being described as the enterprise noSQL database, has excellent XML technologies support and replication features that we needed to implement anyway. Having said that, is it possible to use another data persistence mechanism, with additional work.

3.1.1. Stone Circle

A cluster or three or more Linux nodes each consisting of a Marklogic database, a user interfaced called Stoneface and a Stoneman.

We chose to run three nodes: chain1.kode1100.com, chain2.kode1100.com and chain3.kode1100.com to represent three types of institutions - someone initiating a transaction (the client), someone facilitating, executing, and reporting the transaction (the bank), and an independent regulatory body which we called FCA (Fictional Compliance Authority).

To simplify these will be referred to as node 1-3 or client, bank and FCA respectively.

The minimum setup is a single node and there is no theoretical limit to the maximum amount of nodes, although some steps, such as reaching consensus within the network will take longer as the network size increases.

To accurately mimic a realistic setup that could actually reliably work in production for a major financial institution, we have configured a high availability MarkLogic cluster, where both the configuration, transactional data and the blockchain itself are protected from failures using Marklogic enterprise features such as clustering and automatic failovers.

In addition to that, it possible and very easy to configure additional, automatic cluster replication, to ensure additional availability in different data centres or different time zones.

3.1.2. Stoneface

A JSF Primefaces interface allowing clients to interact with the Stone Circle via graphical, browser user interface.

It allows for uploading, validating files, verifying hashes etc.

4. The Stone Circle algorithm

The Stone Circle algorithm is a proprietary private consensus algorithm that generates in the end a stone hash to further prove a node is in fact one of the trusted delegate nodes such that it helps prevent man in the middle attacks.

The basic cryptography is based on the original bitcoin protocol (as described in the Bitcoin research paper by Satoshi Nakamoto) with a number of adjustments and improvements. That basic cryptography is using well understood and widely used elliptical curve mathematics and prime number factoring to make any attempts at forging mathematically impossible using current computer technology.

The improvement in the algorithm, include but are not limited to the following:

- Adjustable difficulty of the hashing function strength. In bitcoin and pretty much all blockchain solutions to date, the security and immutability of the blockchain signature comes with a significant computational cost of having to run hashing functions multiple times (and often discard the results due to race conditions)

  The stone circle algorithm allows using different hashing implementations, for example SHA256 instead of SHA-512 to effectively double the performance and slash in half the resource usage for signing the messages. This still results in more than sufficient security of the blockchain due to the multiplication effects of adding more transactions.
- Ability to disable coin mining. In a private blockchain setup it is unnecessary to reward participants for signing the messages. In this particular example it is simply a regulatory requirement.

  By removing the coin mining parts, we are again massively reducing computational waste of calculating who gets the reward and the waste of discarding any computations of those who don't get the reward.

- Using ethnographic research to avoid typical human biases against entropy.
- Using tested and proven database technology for storage, reliability and availability.

4.1. Stones

Stones are similar to blocks in bitcoin. They hold the prev stone hash, the file hash and the current stone hash 'this' as well as the time in milliseconds the file was created and the public key of the user who uploaded the block or an agreed upon identification between parties utilizing the system. In these example for security reasons the 'this' hash is simply the hash of the prev + file rather then stone hash generated by the stone circle algorithm.

```
<block version="1">
  <this>500251402261245FCB870657050AB1CAA5A5F137E25A77B5861EDD38964ED727</this>
  <prev>GENSIS893583B63FF73B0474CB42A1CBE7A96E1D8CE52854B4C876026BA453F</prev>
  <file>58D226C6016DCE5B25133D7388FFE29757E5476609FFC3B9BE988B3FF8D2DF3D</file>
  <id>PUBLIC_KEY_USER_ID</id>
  <time>1462288317220</time>
</block>
```

4.2. Stone Chain

A chain consisting of references to stones via the stone hash, a genesis stone hash which can be a hash of any content and a count 'c' to have a cheap method to determine the position in the stone chain.

```
<chain>
  <block c="2" hash="B37EF2958FE7B62C0D1532E394895FD51F7053FA8B1457ABAF01A7139F905AE5" type="top"/>
  <block c="1" hash="500251402261245FCB870657050AB1CAA5A5F137E25A77B5861EDD38964ED727"/>
  <block hash="GENSIS893583B63FF73B0474CB42A1CBE7A96E1D8CE52854B4C876026BA453F" type="gensis"/>
</chain>
```

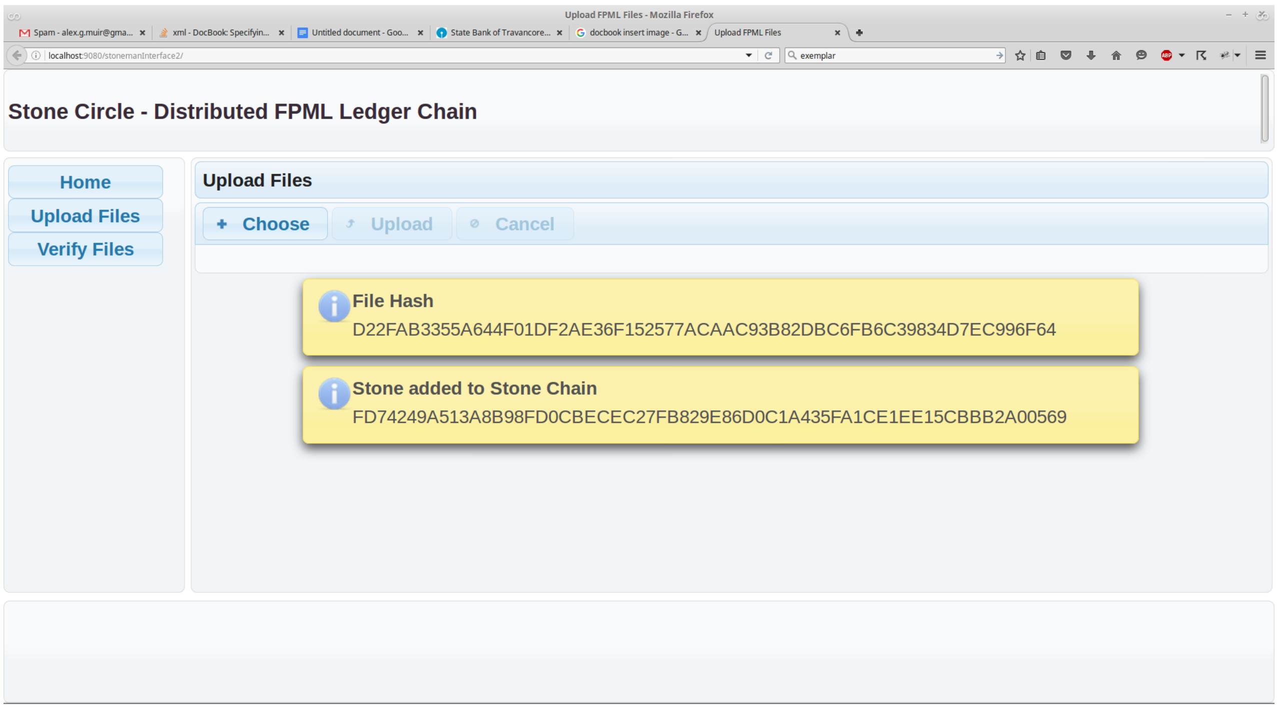
5. Blockchain operations

5.1. Adding a new transaction

5.1.1. File Upload

When a client uploads a file to the Stone Chain system through the Stoneface, the system can be configured to permanently or temporarily store the file, until it's existence is preserved in the Stone Chain, in a Marklogic database. A SHA256 hash of the file is created and used to store the file as filehash.xml. The file hash is broadcast to each Stoneman who each listen to JMS queue.

Figure 1. File Upload


Upload FPML Files - Mozilla Firefox
localhost:9080/stonemanInterface2/
Stone Circle - Distributed FPML Ledger Chain
Home
Upload Files
Verify Files
Upload Files
Choose
Upload
Cancel

File Hash
D22FAB3355A644F01DF2AE36F152577ACAAC93B82DBC6FB6C39834D7EC996F64

Stone added to Stone Chain
FD74249A513A8B98FD0CBECEC27FB829E86D0C1A435FA1CE1EE15CBBB2A00569

5.1.2. Preliminary Chain

Each Stoneman updates it's own preliminary chain based on a first come first serve document queue and broadcast the Stone Hash to the other Stonemen before continuing to process more files. A file can be upload onto each cluster from various clients producing documents on system and it's possible that two files are uploaded simultaneously and a conflict will need to be resolved. A timestamp created during file processing resolves conflicts followed by server priority should the rare situation of two files uploaded at the exact same milliseconds on each server.

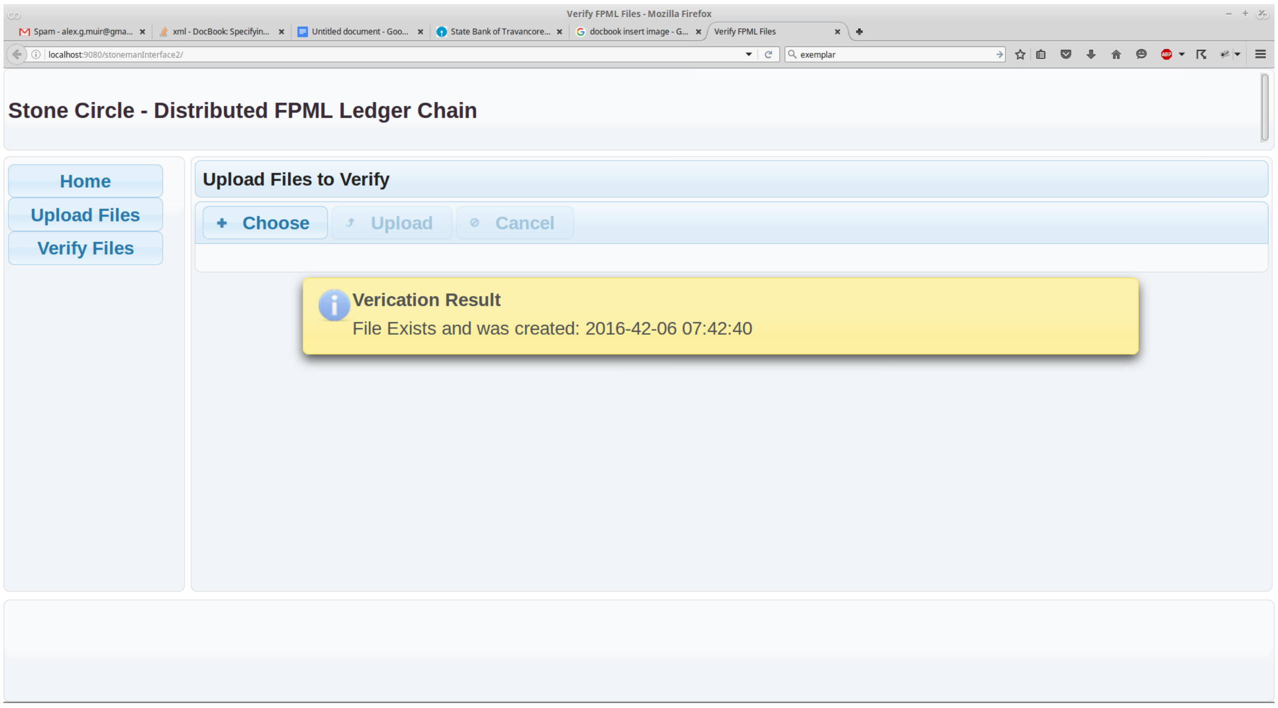
5.1.3. Main Chain

When each Stoneman broadcasts the same Stone Hash for the last file added to preliminary chains, then agreement has been reached and the Block is added to the Stone Chain.

5.2. Verifying the hash for a given file

The Stoneface allows clients to upload files to verify their existence on the chain. If the file exists the user gets a message consisting of the date the file was added to the chain.

Figure 2. Verification Files Exists

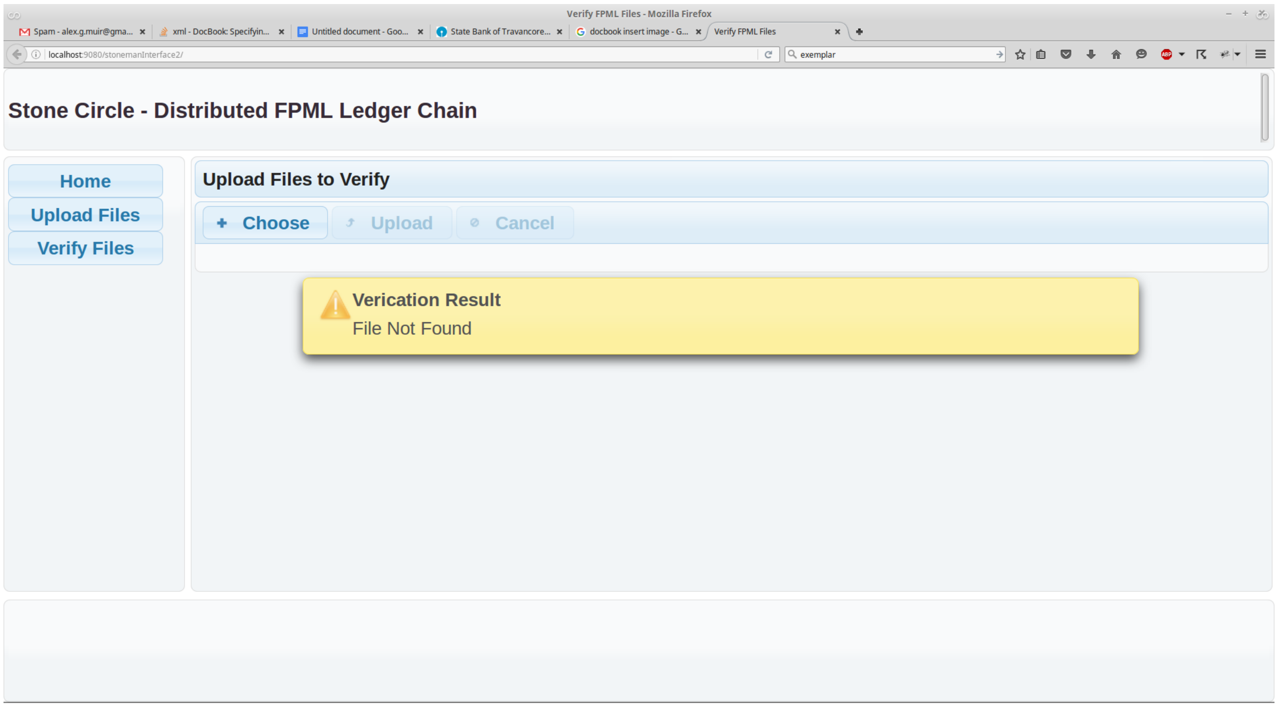

Verify FPML Files - Mozilla Firefox
localhost:9080/stonemanInterface2/
Home
Upload Files
Verify Files
Choose
Upload
Cancel

Stone Circle - Distributed FPML Ledger Chain

Upload Files to Verify

Verication Result
File Exists and was created: 2016-42-06 07:42:40

Figure 3. Verification File Does Not Exist


Verify FPML Files - Mozilla Firefox
localhost:9080/stonemanInterface2/
Home
Upload Files
Verify Files
Choose
Upload
Cancel

Stone Circle - Distributed FPML Ledger Chain

Upload Files to Verify

Verication Result
File Not Found

6. Summary and the future

We have showed that XML is still a good fit for business as usual activities such as trade onboarding and trade reporting. We also showed how strengths of the XML ecosystem can work together with new and emerging technologies, using distributed ledger as an example.

The future steps in this area will largely be market driven, possible next steps include open sourcing some or all of the blockchain code for the community to use,

It is also fairly easy to evolve the Stone Circle algorithm blockchain code to enable distributed, verifiable code execution.

Such a development would then allow applications such as smart contracts and autonomous distributed organisations, which are beyond the scope of this presentation, but we are more than happy to discuss them.

Pioneering XML-first Workflows for Magazines

Dianne Kennedy

Idealliance

<dkennedy@idealliance.org>

Abstract

Most of the publishing world has long embraced the value of structuring content using SGML/XML in the form of DocBook, DITA or any number of vertical, standardized publishing markup schemes. Yet those who publish magazines have stubbornly remained in a world where page-layout-based workflows predominate. While this remains true today, for the first time magazine publishers are flirting with a change to their traditional publishing workflow that would employ XML-first content creation. In this session you will gain an understanding of the two worlds of publishing, why magazine publishers have been so reluctant to embrace XML technologies and the emerging trends that may bring magazine publishers into the XML publishing domain.

Keywords: magazine publishing, PRISM metadata, content-based publishing, design-based publishing, RDF, XML authoring, HTML5, AMP Project, metadata

1. Introduction

In the early 1980's, when publishing moved from linotype to computer assisted composition, the publishing world was split into two different and distinct worlds. These worlds, content-based publishing and design-based publishing, adopted different tool sets, different workflows and have clearly different philosophies.

1.1. Content-Based Publishing

The world of "content-based" publishing includes technical publishing, reference publishing, educational publishing and many more kinds of publishing where delivering the content is the whole point of publishing. Content-based publications are often quite lengthy, being made up of hundreds and often thousands of pages of highly structured content. Technical publishing encompasses technical documentation such as maintenance and operational technical manuals and has the clear purpose of communicating content. Reference publishing such as legal publishing, financial reports and product information along with content that is assembled from databases such as course catalogs, racing forms and even television show guides is also content-based publishing.

Publishers with large volumes of structured content have often adopted publishing tools and systems based on *ISO 8879: SGML* and later on W3C's XML markup language. Content can be created in XML first and stored in XML content management systems that are modeled on the XML document structure. Content can then be assembled from the XML content repositories and the layout and styling are commonly automated through the use of high-speed computerized pagination engines. In the world of content-based publishing, content formatting is simply a way to make the content more easily digestible. Content-based publishing is characterized by applying style sheets, templates or scripted composition algorithms designed to enable a reader to quickly access and understand the information in lengthy publications.

Note

My background was firmly in the world of content-based publishing. In the mid 1980's I worked for Datalogics and participated in the first working SGML-based composition systems for the United States Air Force. Later I participated in the CALS SGML document design efforts, chaired the SAE J2008 SGML design work and participated with the Air Transport Association in the design of the ATA 100 SGML tag set. At that time I believed that all content could be structured in a standardized way. But in 2003, when I came to Idealliance I was shocked to find out that a whole different world of publishing existed; one where content was not structured and where organizations firmly resisted the mere idea of standardizing structures that made up a magazine.

1.2. Design-Based Publishing

A second and very different world of publishing is "design-based" publishing. Most magazines fall into this category along with highly designed books such as cookbooks, travel books and other "coffee table" books. When design comes first, the art director usually develops the concept and design before any content is created. For magazines, using standard style sheets that give each issue the same look month after month simply would not do. Articles within a magazine issue often have their own unique design as well. And because design comes first, content structures cannot be effectively standardized. Hence, very few publishers of design-based publications have adopted "XML-first" workflows. And in fact, until recently, very few members of this community employed XML as a publication source at all.

2. Metadata for Magazines

Even though magazine publishers firmly resisted any pressure to standardize the content and data structures within their publications, they did respond to the pressures to develop mechanisms that would enable them to manage content assets and produce publication products more efficiently. Urgency to develop a standard, technical infrastructure increased as strategists began to predict that content distribution would likely shift from print to the Web and to new digital platforms in the near future. At the same time the W3C was developing XML, a separate W3C initiative, focusing on descriptive metadata, known as RDF (Resource Description Framework) [1], caught the interest of magazine publishers.

In late 1999, magazine publishers and their content technology partners came together as an Idealliance working group to standardize metadata to enable content/asset management. Founders of PRISM included Linda Burman, a consultant to Idealliance and Ron Daniel, a technologist who also served as co-chair for the Dublin Core Data Model Working Group [2] and an active participatant in the W3C RDF effort. Since it's founding the PRISM Working Group has included representatives from over 80 organizations and more than 250 individuals. The concept behind PRISM was simple. Since the magazine community could not standardize the structures that make up an article, they would focus on defining standard metadata vocabularies that could provide a techincial infrastructure for magazine content.

Work on the first version of PRISM, Publishing Requirements for Industry Standard Metadata, began in 1999 and PRISM Version 1.0 [3] was published in 2001. Initially the scope of PRISM was metadata for print-based periodicals. PRISM was built upon, and extends the Dublin Core metadata set. PRISM employs Dublin Core publishing metadata fields where appropriate and defines additional metadata specific to periodical publications.

In 2008 PRISM 2.0 was published. This was the first major revision to PRISM and the scope of PRISM descriptive metadata was expanded to include both print and online periodical content. PRISM 2.0 [4] provided a number of additional special-purpose metadata vocabularies modularized by namespace. This allowed each publisher or user to select the modules that fit their unique business requirements. PRISM 2.0 included periodical metadata (prism:), image metadata (pim:), recipe metadata (prm:), crafts metadata (pcm:), usage rights metadata (pur:) and contract management metadata (pcmm:) [5].

In addition to publishing standard metadata sets, or taxonomies, the PRISM Working Group has developed over 40 controlled vocabularies [6] including vocabularies for periodical content types, article genres, content presentation types, issue types, publishing frequency types, role types for creators and contributors . . .

Strange as it might seem, PRISM V1.0 was developed without a specific use case in mind. At the time we envisioned that PRISM may be instantiated using either RDF or XML and used for many purposes. Three years after release of PRISM, magazine publishers developed the first industry use case for PRISM. The use case was to deliver magazine articles to aggregators following publication of the magazine or newsletter. The use case, known as the *PRISM Aggregator Message,* or PAM, required the use of PRISM metadata along with encoding of article text in XHTML. In the years after publication of PAM [7], most magazine publishers in the US came to use it as the standard format to deliver content to their aggregation business partners.

It is important to understand that generating PAM XML was, and remains, a post publication process. The magazine publishing workflow is typically focused on the production of the print product. Once the PDF is produced and sent to the printers, the source is either converted or rekeyed into the PAM XML message format to deliver to aggregators. In addition to sending articles to aggregators, some publishers began to use PAM as an archive format as well. Time, Inc., for example, invested in the conversion of all previously published Time Magazine content into PAM in order to establish a content archive to serve as an editorial research database. It seemed that while most magazine publishers came to value XML-encoded content, they were still trappped in their traditional design-first workflows.

3. The Impact of the iPad

A potential sea change for magazines and other design-based publications came in 2010 with the launch of the iPad. Suddenly publishers were called on to deliver their publications digitally instead of exclusively in print. Until this point, design-based publications were still able to cost justify their labor-intensive design-based publication process because their print revenues supported the expense. But the launch of the iPad meant producing well designed publications, not only in print, but on a growing number of digital devices with different resolutions, aspect ratios and sizes. This was tremendously challenging and expensive. Idealliance members launched the nextPub Council to develop a strategy for the efficient cross platform publishing of magazine content in the fall of 2010. After a six month study the nextPub Council recommended that magazine publishers shift to an XML-First publishing model. To support this move, Idealliance extended PRISM to develop the *PRISM Source Vocabulary* , or PSV [8]. The PSV tagline was "The Source is the Solution." The PSV XML schema employed PRISM / Dublin Core publishing metadata along with semantic HTML5 for rendering on tablets and mobile devices.

Once again, the tradition of design-first publishing prevented the fulfillment of the vision to create magazine source content in XML. In part the reason was institutional. It was very difficult to convince the art directors managing magazine production to move to a content-centric workflow. The move to XML content creation for magazines was also limited by the publishing tools available at the time and the skill set of the editorial and production staff. Since the predominant publishing tools used by magazine publishers were page layout based, i.e. design oriented, and the investment in them was so great, most magazine publishers opted to remain with design-based content creation workflows. And instead of creating articles in XML, magazine publishers simply decided to repurpose their layouts as page replicas (PDFs or page images) on tablets, thus protecting their tradition of uniquely crafted, high-quality publishing design and layout.

It is important to understand that publishing print replicas of magazines on tablets was generally a disappointment. Once the novelty of interactivity wore off, an ongoing readership could not be maintained. And publishers never figured out how to make a profit on tablet-based magazine content. As new, smaller tablets came to market, presenting print-size pages on small screens became a serious issue for readers as well.

4. Mobile Comes of Age

Today, more than 5 years after the impact of the iPad on the magazine marketplace, a new disruptive force, even more significant than tablets, is demanding that design-based publishers finally shift to XML-first workflows. That force is the rapid shift toward content consumption on smart phones along with the increasing demand for "360 degree" or "integrated access" to magazine content across media platforms, digitally and in print. Today's readers will no longer accept a frustrating, slow, clunky digital reading experience. As a result major US publishers are beginning to shift away from presentation of page replicas and toward HTML5-based viewing technologies. And because presentation speed is becoming increasingly critical, the lightweight *Accelerated Mobile Pages* project (AMP) [9] is gaining traction.

5. Developing a Magazine Authoring Schema

As new tools to author structured magazine content come to market, magazine publisher's still have to decide on an authoring schema. Since the industry has invested so much in organizing their content using PRISM metadata that seems to be a natural fit. But again the issue comes down to the correct authoring schema. Some initial work has been done on that front.

In the summer of 2014, usability testing was conducted using the PSV schema. This schema was designed to store source magazine article metadata and text in semantic HTML5. At first glance, it seemed a natural fit. But usability testing quickly lead to the conclusion that a robust archive format was not ideal for authoring. The HTML5 model offered just too many options that most authors would not use, and in fact had a severe impact on the usability of the schema.

For the next iteration, Idealliance tested a schema that allowed for key fields of PRISM metadata along with a simplified version of HTML5 that contained only those elements that an author was likely to use. In order to test this iteration of the authoring schema, we utlized an MSWord plugin from Ictect [10] that had been specially configured to support this version of the schema. Several rounds of testing with the schema led to further modifications.

The work to develop a magazine authoring schema by Idealliance is ongoing. New metadata fields to embed licensing terms such as content expiration dates and geographic display access are being added. AMP enablement is being studied as well. Ongoing work can be tracked on the Idealliance PRISM website [11]

Bibliography

[1] *RDF*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Resource_Description_Framework

[2] *Dublin Core*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Dublin_Core

[3] *PRISM V1.0*. Idealliance, Inc..
http://www.prismstandard.org/specifications/1.0/prism1.0.htm

[4] *PRISM V2.0*. Idealliance, Inc..
http://www.prismstandard.org/specifications/2.0/

[5] *PRISM Taxonomies*. Idealliance, Inc..
http://www.idealliance.org/specifications/prism-metadata-initiative/prism/specifications/taxonomie

[6] *PRISM Controlled Vocabulary Specification V3.0*. Idealliance, Inc..
http://www.prismstandard.org/specifications/3.0/PRISM_CV_Spec_3.0.htm

[7] *PRISM Aggregator Message Guide*. Idealliance, Inc..
http://www.prismstandard.org/specifications/3.0/PAM_Guide_2.2.htm

[8] *PRISM Source Vocabulary Specification*. Idealliance, Inc..
http://www.prismstandard.org/specifications/psv/1.0/

[9] *Accelerated Mobile Pages Project*. Google.
http://www.ampproject.org

[10] *Intelligent Content Architecture*. Ictect, Inc..
http://www.ictect.com

[11] *PRISM Website*. Idealliance, Inc..
http://www.prismstandard.org

CALS table processing with XSLT and Schematron

Nigel Whitaker
<nigel.whitaker@deltaxml.com>

Abstract

CALS tables are used in many technical documentation standards. There are OASIS specifications for CALS tables which include a number of semantic rules to ensure table validity. This paper reports on some of our experiences with CALS table processing and validation. We implemented the majority of our validation code in XSLT and have needed to carefully consider performance when handling large tables of several thousand rows.

We have experimented with a number of new XSLT features when addressing performance issues and will report on our experiences. In addition to processing tables we wished to improve the quality of CALS tables that we would meet and which our users/customers would produce (we wished to rid the world of bad tables!). For this we have used schematron to check and report the validity of tables in a user friendly way. We met a number of obstacles and will report on these and our solutions ('work-arounds') in this paper.

1. Background

CALS tables are used in many technical documentation standards such as DocBook, DITA and s1000d. Unfortunately these document standards refer to slightly different versions of the CALS specification (the 'exchange subset' and the full specification [1]) and they have slightly different messages about validity.

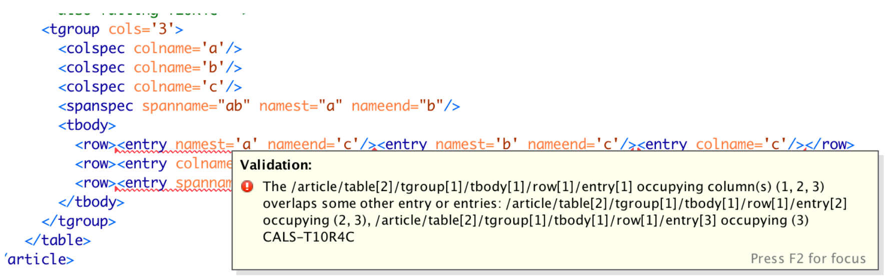
Different XML tools and processors (such as XSL:FO to PDF engines) enforce different subsets of the semantic rules. Our goal for comparison processing was to generate a valid result if the inputs were valid. In order to do this we needed a way of checking validity. We looked for something that we could reuse as part of our test processes and also internally as part of the processing of the tables. Table processing is a widely used example in schematron [2] tutorials and blog postings, however we did not find anything that was close to being a complete implementation of the CALS specification(s). We therefore decided to construct one.

Our CALS validity code is Open Source software (originally available on GoogleCode, now GitHub [3]). We provided it as open source as a number of collaborators were asking us about it. Additionally, customers were questioning our error/warning reporting and not fully understanding the nature of the validity issues. Using schematron allows us to provide good diagnostics to the user, typically in an XML editor or authoring tool, and we hope this would help reduce the number of bad tables, finding the structural and other table errors as they are created. Providing Open Source code would improve use and adoption. The oXygenXML editor supports schematron validation and Figure 1, "Example of table validation error" shows how an error is reported.

2. Introduction to table validity

CALS tables allow for 'wide' cells or entrys. Rather than a regular grid there are cells that 'span' a number of grid positions. Some simpler table models such as those for HTML tables use attributes with integer dimensions to

Figure 1. Example of table validation error



```
    <tgroup cols='3'>
      <colspec colname='a'/>
      <colspec colname='b'/>
      <colspec colname='c'/>
      <spanspec spanname="ab" namest="a" nameend="b"/>
      <tbody>
        <row><entry namest='a' nameend='c'/><entry namest='b' nameend='c'/><entry colname='c'/></row>
        <row><entry colname
        <row><entry spannam
      </tbody>
    </tgroup>
  </table>
'article>
```

Validation:

The /article/table[2]/tgroup[1]/tbody[1]/row[1]/entry[1] occupying column(s) (1, 2, 3) overlaps some other entry or entries: /article/table[2]/tgroup[1]/tbody[1]/row[1]/entry[2] occupying (2, 3), /article/table[2]/tgroup[1]/tbody[1]/row[1]/entry[3] occupying (3)
CALS-T10R4C

Press F2 for focus

describe the wide/tall cells. CALS tables use the *@morerows* attribute with an integer for vertical spanning, but use a more complex structure for horizontal spanning. A column specification section is declared and then the later cells reference these *colspec* and similar elements by name in order to configure their spanning.

The CALS rules cover a number of areas. Here are some examples.

2.1. Context constraints

The CALS specification include a number of constraints about the context in which an attribute can occur. For example, a wide (horizontally spanning) `entry` can be specified by referring to the start and ending `colspecs` (the `@namest` and `@nameend` attributes). It is also possible to define a `spanspec` element at the same time as the colspecs which itself refers to the start and end colspecs. It is permissible for an entry to refer the the `spanspec` (using `@spanname`) in the main body of the table. However using a `spanname` attribute is not allowed in the context of the `thead` and `tfoot` elements used to define the headers and footers of a table.

2.2. Referential integrity constraints

References to `colspec` and `spanspec` elements from within a table have to be validated. These references have a per-table scope and therefore do not make use of the `id` or `xml:id` mechanisms.

2.3. Structural constraints

There are a number of structural constraints in the CALS spec. These tend to be orientated to making the table regular (so that for example each `row` has the correct number of columns occupied or that a `morerows` vertical span does not extend beyond the end of the table. Other constraints are used to ensure correct left-right ordering and to prevent overlap.

As discussed above attributes can be used to specify wide table entrys[1]. They can also be used simply position an entry in the table. Some editors and authoring tools will refer to a `colspec` in every entry in the table. However, its also possible to use an entry without specifying its position in which case its position has to be inferred and this style is used by other tools. The rules are quite complex and consider the position of the preceding `entry` element but also tall entrys that use `morerows` attributes in the rows above. The situation may be quite recursive in that an entry will use `morerows` but in order to work our which columns it occupies then it's necessary to look to all the entrys to its left to see if they define their position or infer their position by default rules and its also necessary to look at rows from above that 'straddle' or 'overlap' into the current row at positions before where the entry may be placed by the default rules.

3. XSLT processing of CALS tables

3.1. Row distance calculation

In calculating and checking for vertical spanning we originally wrote code that measured the distance between rows. The initial naive implementation is shown in Figure 2, “Original row-distance code”.

Figure 2. Original row-distance code

```
<xsl:function name="cals:row-distance"
              as="xs:integer">
  <xsl:param name="r1" as="element()"/>
  <xsl:param name="r2" as="element()"/>
  <xsl:sequence
    select="abs(count($r1/preceding-sibling::*:row) -
            count($r2/preceding-sibling::*:row))"/>
</xsl:function>
```

This is code at least O(n) and the calling code, discussed later, made for O(n^2) complexity. From our analysis we knew this function was called a lot, over 10^8 invocations in one of our tests.

Customer feedback reported performance concerns and the need to process larger tables. As it was a known 'hot-spot' we concentrated on optimizing this function. This was around the time of Saxon version 9.3 and maps were used[2]. This optimized code is shown in Figure 3, “Optimized row-distance code”.

[1] As `entry` is an element name we have chosen to deliberately misspell this plural form.
[2] An accumulator would now be a better choice and would avoid the use of generate-id to form the map key.

Figure 3. Optimized row-distance code

```
<xsl:variable name="rowtopos"
              as="map(xs:string, xs:integer)">
  <xsl:map>
    <xsl:for-each select="//*:row">
      <xsl:map-entry key="generate-id(.)"
        select="
          count(./preceding-sibling::*:row) + 1"/>
    </xsl:for-each>
  </xsl:map>
</xsl:variable>
<xsl:function name="cals:row-distance"
              as="xs:integer">
  <xsl:param name="r1" as="element()"/>
  <xsl:param name="r2" as="element()"/>
  <xsl:sequence select="
    abs(map:get($rowtopos, generate-id($r1)) -
    map:get($rowtopos, generate-id($r2)))"/>
</xsl:function>
```

This optimization reduced a 5 minute plus runtime to the 15-20 second range. For further details see Section 3.6, “Performance summary”.

3.2. Vertical column infringement processing

The code shown in Figure 4, “vertical infringement code” was used to calculate which columns of a row are overlapped or infringed from above.

Figure 4. vertical infringement code

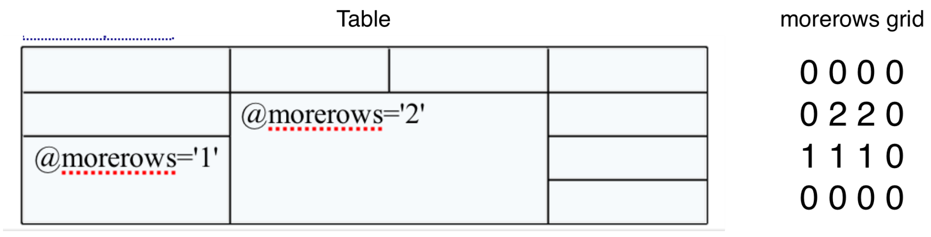
```
<xd:doc>
  <xd:desc>
    <xd:p>Describes how a table row is
          spanned from above.</xd:p>
    <xd:p>This result is a set of columns which are
          overlapped from above in the row
          specified as an argument.  The 'set' is
          really a sequence and may be out of
          order, eg:  (3, 2).</xd:p>
  </xd:desc>
  <xd:param name="row">A table row</xd:param>
  <xd:return>A sequence of integers specifying
             which columns are spanned or
             'infringed' from above</xd:return>
</xd:doc>

<xsl:function name="cals:overlap2"
              as="xs:integer*">
  <xsl:param name="row" as="element()"/>
  <xsl:sequence select="
  for $r in $row/preceding-sibling::*:row return
    let $row-distance :=
      cals:row-distance2($r, $row) return
      for $e in $r/*[@morerows] return
      if (xs:integer($e/@morerows) ge $row-distance)
        then cals:entry-to-columns($e)
        else ()"/>
</xsl:function>
```

The above code reflects our XSLT 2.0 training and experience. For each row we look upwards and see if any of the rows above could infringe the current row. This process is O(n) and makes repeated use of the `row-distance` function above. It also uses the `entry-to-columns` function which for a table entry reports which columns are occupied.

There are several issues we knew about when writing this code that we were aware could cause performance issues:

- We will be using this function repeatedly and each time it is called it will look all the way back through the table.
- It looks all the way back to the top of the table since theoretically it is possible for the first row to have a morerows span to the end of the table. In common cases it's likely that spanning would be short, however doing an analysis of the maximum morerows value and the using this to optimize the code would have made some complex code even more complicated and difficult to maintain.

Figure 6. Example table and morerows grid



3.3. Forward looking morerows processing

The processing of morerows attributes could be attempted in forward looking manner. An `xsl:iterator` was used to make a morerows calculation using a sequence of integers. Each member of the sequence would store the current morerows value for its corresponding column as the iterator would allow it to be decremented as each subsequent row was processed, provided it did not also use morerows.

Figure 5. The morerows iterator

```
<xsl:iterate select="$tgroup/*:row">
  <xsl:param name="morerows" as="xs:integer*"
  select="for $i in 1 to $tgroup/@cols return 0"/>
  <xsl:param name="grid" select="map{}"
    as="map(xs:integer, xs:integer*)"/>
  <xsl:on-completion select="$grid"/>
  <xsl:variable name="rowmap"
                as="map(xs:integer, xs:integer)">
    <xsl:map>
      <xsl:for-each select="entry[@morerows]">
        <xsl:variable name="coveredCols"
          as="xs:integer+" select="
             cals:entry-to-columns(., $morerows)"/>
        <xsl:sequence select="
          map:merge(for $i in $coveredCols return
          map:entry($i, xs:integer(@morerows)))"/>
      </xsl:for-each>
    </xsl:map>
  </xsl:variable>
  <xsl:next-iteration>
    <xsl:with-param name="morerows"
    select="for $i in 1 to count($morerows) return
           max(($morerows[$i]-1, $rowmap($i),0))"/>
    <xsl:with-param name="grid"
           select="map:merge(($grid, map:entry(
             count(preceding-sibling::*:row)+1,
             $morerows)))"/>
  </xsl:next-iteration>
</xsl:iterate>
```

In Figure 5, “The morerows iterator” the `morerows` param stores the sequence that is adjusted as each row is iterated over. Additionally, we wanted to store these values and the `grid` param records each of the `morerows` calculations using a map where the map key is an integer corresponding to the row number.

The `rowmap` calculates the morerows values declared in the current row (it maps from column number to the morerows value). In order to do this knowledge of the morerows spanning from above is needed to determine the column positions of any entries which do not define their columns by explicit reference to colspecs.

An example table and the corresponding morerows grid is shown in Figure 6, “Example table and morerows grid”.

The iterator that has been developed now needs to be used. If our intention was, for example, to produce a normalized form of the table then it should be possible to use the iterator in an `xsl:template` matching the `tgroup` or `table`. However, we are keen to preserve the ability to have checking for individual entries in the table and error reporting specific to those entries. We could use the iterator in a function after finding the parent table, but then we would be iterating repeatedly over the same table. In order to use the same data we then looked at accumulators.

3.4. Caching the table data for use in schematron

After discounting using the iterator in a template we tried to find ways of keeping the data available. Thinking in terms of imperative programming you would write:

```
if (empty map) then {
  construct map; // done once
}
return lookup data in map;
```

However this doesn't fit well with either XSLT or schematron. We found a solution using the new `xsl:accumulator` mechanism (once we realized an accumulator doesn't actually need to accumulate anything!).

If we put our accumulator in a function, as indicated in Figure 7, “Function with iterator”, we can then use the iterator in an accumulator as shown in Figure 8, “Storing data in an accumulator”. Another accumulator is used to provide a mapping from the rows to the row numbers as this is what is used to index the map representing our grid data. The use of a sequence in this accumulator is designed to support the possibility of nested tables.

Figure 7. Function with iterator

```
<xsl:function name="cals:generate-morerows-data"
      as="map(xs:integer, xs:integer*)">
  <xsl:param name="tgroup" as="element()"/>
  <xsl:iterate select="$tgroup/*:row">
    <xsl:param name="morerows" select="
    for $i in 1 to $tgroup/@cols return 0"
    as="xs:integer*"/>
    <xsl:param name="grid"
               as="map(xs:integer, xs:integer*)"
               select="map{}"/>
    <xsl:on-completion select="$grid"/>
    ...
  </xsl:iterate>
</xsl:function>
```

Figure 8. Storing data in an accumulator

```
<xsl:accumulator name="table-spanning"
    as="map(xs:integer, xs:integer*)"
    initial-value="map{}">
  <xsl:accumulator-rule match="*:tgroup"
    phase="start"
    select="cals:generate-morerows-data(.)"/>
</xsl:accumulator>

<xsl:accumulator name="row-number" as="xs:integer*"
    initial-value="()">
  <xsl:accumulator-rule match="*:tgroup"
    phase="start" select="(0, $value)"/>
  <xsl:accumulator-rule match="*:tgroup"
    phase="end" select="tail($value)"/>
  <xsl:accumulator-rule match="*:row"
    select="head($value)+1, tail($value)"/>
</xsl:accumulator>
```

3.5. Using the accumulator data in schematron

We can now use the data in various ways. One technique is to create functions that use the data by calling the `accumulator-after` or `accumulator-before` functions. It is also possible, when using the XSLT query binding and foreign element support, to use the data directly in schematron. In order to check CALS tables there are three assertions that check the table structure that operate on every table entry. We can use some foreign XSLT code at the the schematron rule level so that they are available to all of the assertions. The code is quite long so only the basic mechanism is shown in Figure 9, “Accessing accumulator data from schematron”.

Figure 9. Accessing accumulator data from schematron

```
<pattern id="p-structure">
  <rule context="*:entry">
    <xsl:variable name="row" as="element()"
      select="ancestor::*:row[1]"/>
    <xsl:variable name="table-data"
      as="map(xs:integer, xs:integer*)"
      select="$row/ancestor::*:tgroup[1]/
      accumulator-after('table-spanning')"/>
    <xsl:variable name="row-number" as="xs:integer"
        select="$row/
                 accumulator-after('row-number')"/>
    <xsl:variable name="morerows" as="xs:integer*"
        select="$table-data($row-number)"/>
    <assert
      test="... cals:entry-to-columns(., $morerows)
         ... ">...</assert>
    <assert test="...">...</assert>
    <assert test="...">...</assert>
  </rule>
</pattern>
```

Every table entry will invoke the assertions above, but in each case we rely on the pre-calculated or accumulated data. This should change an O(n^3) complexity problem into one closer to O(n).

3.6. Performance summary

Two customer support cases presented particular problems with table performance. The earlier row-distance optimization with maps gave performance improvements at the time, but there were still performance doubts and concerns. The optimization work with iterators and accumulators discussed here have provided dramatic performance improvements as seen in Table 1, “Performance data”.

The results discuss run time improvements. There was no obvious change in memory consumption.

Table 1. Performance data

	case 1	case 2
description	Standards documentation	Semiconductor data
file format	DocBook 5	DITA 1.1
file size	8.7MB	574KB
tree details	459262 nodes, 656993 characters, 208 attributes	31670 nodes, 41285 characters, 10384 attributes
table count	15	3
average row count	1083	1149
largest table (rows)	2032	1552
Saxon EE 9.7.0.4 performance: Apple iMac, 3.2 GHz Intel Core i5, 24GB, MacOS 10.11.4, Java 1.8.0_74 64 Bit Server VM		
original runtime	greater than 11 hours	348.49s
row-distance map optimized runtime	1557.52s	16.09 s
iterator/ accumulator optimized runtime	2.20s	0.152s

4. Schematron processing

We have two ways of using the schematron file. The first, standard, way is to use it as a file checker, perhaps inside an XML editor or authoring tool as discussed earlier. Additionally we are interested in using checking code as part of our comparison products which required some modifications to the to the standard schematron processing model and tool-chain. As schematron processing is based on XSLT (the schematron is 'compiled' to XSLT using XSLT) we've taken the approach of using further XSLT to modify the generated checker to satisfy our requirements. This approach works while the generated checker is stable, but could cause us problems if new version of the schematron code, such as the 'skeleton', is released. One reason for describing our requirements and changes here is to gauge if there is wider interest in them. If so, we could see if there is acceptance to have them incorporated into the official release.

4.1. Schematron phases

When developing the CALS validity constraints a misconception about schematron phases was not noticed until the final stages of testing. We followed a development model similar to that of traditional compilers where checking is performed in stages. Similar to a compiler constraints were grouped into categories such as: context, reference and structure. When developing the structural constraints referential integrity was assumed. We developed and tested those constraints first and assumed the implementation wouldn't run those constraints if the referential integrity constraints fail. Instead schematron phases seem to be something that is left to user control - the user is often presented with a list of phases and asked which to run.

Alternatives were considered. Adding conditional code to the later constraints to test the earlier one and therefore prevent failures in the execution of complex structural constraints was a possibility, but would add more complexity and make the code harder to maintain. We decided to modify the execution model of the generated schematron checker. The XSLT which is generated uses modes to (which are then template-applied to the root node '/' in turn) which correspond to the various phases. A very simple, but naive, initial modification used saxon:assign to record any failure in a global variable which was then tested between the apply-templates for the various stages.

While the saxon:assign approach worked it was not elegant. Advice was sought from the schematron-users email list and David Carlisle suggested an alternative algorithm using a nested structure of xsl:if tests, variables and apply-template instructions.

The new xsl:try/xsl:catch mechanism may provide a better mechanism for our preferred phasing and may be investigated.

4.2. Issues with schematron granularity

Current Schematron tools focus on validating an entire XML file. This is what is needed in an editor or authoring tool. For use in our comparator products we needed something slightly different. When comparing and aligning a table we make decisions on how the result will be represented partly based on whether the input tables were valid. The requires a knowledge of the validity of a table (we add a valid attribute during input processing) rather than the whole file. Schematron defines an XML format, the Schematron Validation and Reporting Language (SVRL), to describe the validation results. Unfortunately this format is a flat list of validation results. While it does contain XPaths which point to the location of failures it is a hard process to group these and associate them with source tables. We took a different approach and again adjusted (by post processing the generated XSLT) the control flow so that rather than rely on template matching form the root or / element downwards a set of phases are applied to each table and then processing moves to the next table.

If other schematron users have similar requirement it may be appropriate to share code, approaches or discuss alternatives further.

5. Acknowledgements

The author has been assisted by colleagues at DeltaXML Ltd who have also contributed to the code and associated tests. This is why the first-person form 'we' is used in this paper.

We would also like to thank Dr John Lumley who has been assisting DeltaXML with performance profiling and optimization approaches with our XSLT code. He suggested that an iterator would improve the performance of the CALS checking code and the results presented above confirms this.

David Carlisle and other members of the schematron-users email list have helped with the schematron issues we have discussed above and we are very gratefully for their advice.

References

[1] Harvey Bingham (ed.) *CALS Table Model Document Type Definition. OASIS Technical Memorandum TM 9502:1995.* . OASIS Inc.. 1995.
https://www.oasis-open.org/specs/tm9502.html

[2] *ISO/IEC 19757-3:2006 Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron.*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40833

[3] *CALS Table Schematron. Github repository.*
https://github.com/nigelwhitaker/cals-table-schematron

Language-aware XML Aggregation

Malte Brunnlieb

Capgemini Deutschland GmbH, Germany

Technical University of Kaiserslautern, Germany

`<m_brunnl@cs.uni-kl.de>`

Steffen B. Holzer

Technical University of Kaiserslautern, Germany

`<holzer@rhrk.uni-kl.de>`

Abstract

Looking at version control systems, XML data view updates, or simply code generation, merging XML based documents has become an omnipresent challenge these days. Each problem domain comes with its own facets of merge algorithms, like 2-way merges, 3-way merges, and patch description & applications. In this paper, we will focus on the problem domain of code generation applying a 2-way merge with patch semantics. As XML DTDs and XSDs specify the syntax of XML languages, document merges become even harder when language semantics are taken into account. Additionally, due to the fact, that one XML document can be based on multiple XML languages separated by namespaces, an appropriate merge algorithm has to focus on each XML language specification rather than just considering XML meta-language syntax. For this need, we developed a XML-based description language called MergeSchema. It is designed for controlling a 2-way XML merge to also cover the semantics of a XML-based language during merge processing.

Keywords: XML Aggregation, MergeSchema, 2-way merge, Incremental Code Generation

1. Motivation

The need of merging XML documents is quite old and omnipresent these days. As examples just take commonly known version control systems or data view updates [Abiteboul2001]. Already available approaches can be classified into 2-way merge vs. 3-way merge techniques. In a 3-way merge the two documents to be merged are derived from a known common base document. Thus, a merge algorithm can take additional information into account by comparing each document to the common base document. Such information for example cover the knowledge about deletions, additions, or updates. In contrast to that, a 2-way merge focuses on the merge of two documents, which may not be derived from a common base document. Thus, no further information are available and the two documents to be merged can just be processed in an aggregative manner by taking language specifications into account.

Merge techniques can also be classified into structural vs. line-based techniques. Whereas commonly used version control systems try to merge conflict based on line-based algorithms, only a view tools already provide structural merge techniques resulting in much better merge results due to the processed knowledge of the document's language. However, utilizing the XML meta-language as the basis for structural merge algorithms will easily result in unintended document merges as the concrete XML-based language of the documents is not considered at all. Therefore, there are different tools at least including DTD or XSD validation to not result in syntactically wrong merge results. But, as DTD and XSD just cover the specification of the syntax of a language, we experienced that the merge results are of a very generic nature. Especially in patch semantics, it would be much more beneficial to consider even the semantics of a document or architecture-driven coding conventions to generate merge results of higher quality specific to different use cases.

Our approach will focus on the lack of language specific information during a 2-way structural merge by providing an additional XML document named MergeSchema to describe further merge relevant language characteristics of XML-based languages, whereas each MergeSchema specifies merge properties for exactly on XML language. Furthermore, we

doi:10.14337/XMLLondon16.Brunnlieb01

implemented a merge algorithm, which can process multiple MergeSchemas to process a 2-way structural merge of documents by a XML language aware algorithm.

1.1. Outline

The reminder of the paper is structured as follows. First, we will shortly introduce the context as well as the background of this work to fully understand the motivation behind the MergeSchema specification needs. Following, we will discuss the requirements on the notion of equality of XML nodes as well as on the aggregation needs of XML nodes. As a third step, we will present the MergeSchema and its specification to influence the merge in use case proper way. Furthermore, we will shortly describe additional features covered in the merge algorithm itself regarding the alignment of the merge result. Coming to a conclusion, we will discuss the limitations as well as our experiences with the approach.

2. Patch Semantics in a Generative Context

Our approach of a 2-way structural XML merge has been developed in the context of code generation, in specific incremental code generation [Brunnlieb2014]. Incremental code generation essentially focuses on the generation and integration of small code fragments specified by AIM Patterns [Brunnlieb2016]. AIM Patterns are architecture dependent design patterns designed to be applied on an existing code base. This especially implies the generation and merging of new code into the current code base's code artifacts. AIM Patterns even specify the code artifacts to integrate new code to, which is basically driven by architectural code conventions [Brunnlieb2016].

As an example in the context of XML languages, the application of an AIM Pattern might result in the generation of a new XML document, which has to be deployed to a specific code artifact (i.e. XML file). If the code artifact does not already exist in the specific deployment path, a new XML document will be deployed to the specified path. However, if the code artifact does exist, the generated XML document has to be merged with the existing XML document. In addition, there are high requirements on the readability of the merged documents in the context of interest as the merged documents are basically designed and maintained by humans.

In the case of structurally merging the generated document to an already existing one, the generated document is further called the `patch`, whereas the existing document will further be referenced as the `base`. Thus, the patch encodes the semantics of providing an update to the base document. This is especially is different to general 2-way merge approaches as we inject further semantics about the relationship of the two input documents into the merge algorithm. Furthermore, the patch semantics in this context have to be considered as of an additive nature as the patch does not provide any meta information about the addition, removalm or update of XML nodes at all.

3. Requirements of a Language-aware Merge

There are basically two challenges for 2-way structural merge algorithms. First, equivalent document fragments have to be found and second, the operation to be performed has to be determined. As we are focusing on a merge algorithm to be usable for different XML-based languages, we even have to handle different notions of equality of element nodes. In the following, we discuss the notion of equality in more detail as well as different operations for aggregation.

3.1. Equality of nodes

Discussing the notion of equality of element nodes, the most trivial notion of equality is based on the recursive identity of all attributes and children nodes of an element. Another trivial notion of equality of element nodes can be fixed by relying on the `id` attribute already provided by the XML meta-language. However, in practice the identifying attribute does not always have to be named `id`. As an example, it might be also aliased by additional attributes as done in the specification of Spring Beans[1] by the `<bean>` element. A `<bean>` element provides the additionaly `name` attribute as an alias for the `id` attribute. Thus, it enables the developer to use characters not contained in the DOM ID type within the identifying string. Summarizing, a XML language-aware merge algorithm essentially should be able to analyze equality of elements based on arbitrary attributes.

Furthermore, the notion of equality can be easily also discussed on an element's children like any elements or

[1] https://www.springframework.org/schema/beans/spring-beans-4.0.xsd

textual nodes. As a first example, there is the significance of the order of child nodes. Considering Example 1 and Example 2 show different documents with different semantics indicating the importance of the consideration of node ordering.

Example 1. DocBook section

```
<section>
  <para>for all men</para>
  <para>it exists one woman</para>
</section>
```

In Example 1 consider docbook's semantics of the node `<section>` when switching the `<para>` elements. The semantics of the resulting document will change. In this specific example the ordering of `<para>` nodes is crucial for the semantics of the `<section>` node.

Example 2. Data store

```
<data>
  <entry id="1">...</entry>
  <entry id="3">...</entry>
  <entry id="2">...</entry>
</data>
```

In contrast to Example 1, consider the simple XML data store in Example 2. The order of child elements in this case does not have any impact on the semantics of the `<data>` node. However, the information about the nodes semantics are not specified in the language specifications as e.g. in DTDs or XSDs. For the design of a proper merge algorithm, these semantics have to be considered to enable a proper document merge.

Example 3. Example: HTML table

```
<table>
  <tr><th>Cell A</th><th>Cell B</th></tr>
  <tr><td>        ....        </td></tr>
  <tr><td>        ....        </td></tr>
</table>
```

Of course, child elements contain even more information than just encoding information in its order. In Example 3 a HTML table is shown. Since it lacks an `<id>` or any other attribute, the notion of equality has to be based on other properties than already discussed. Lets assume for our personal needs, we want to merge two HTML documents containing a HTML table, whereas the tables are designed for the same purpose but the data rows differ. Given this semantics, we could use the column headers of the table as the most identifying property. To access this information, the equality analysis again has to retrieve information from the `<table>` elements children to obtain the notion of equality of the `<table>` element, i.e. the first table row. However, the notion of equality might also refer to any other child nodes or even attributes of them. This rather individual and more complex notion of equality indicates the complexity a generic merge algorithm has to deal with to enable proper 2-way structural merges for different XML languages.

3.2. Node Accumulation & Aggregation

After gathering the basic requirements, which can be easily enriched by further more complex examples, the next step is to discuss the different facets and restrictions on aggregating and accumulating element nodes as well as non-element nodes. Again, we provide different examples to visualize and gather the needs of different use cases of a 2-way merge in a generative context.

Example 4. Node multiplicity

```
<!-- 1st input -->
<root>
  <child id="1"/>
  <child id="3"/>
</root>
```

```
<!-- 2nd input -->
<root>
  <child id="1"/>
  <child id="2"/>
</root>
```

```
<!-- DTD -->
<!ELEMENT root (child)*>
<!ATTLIST child id ID>
```

```
<!-- intended result -->
<root>
  <child id="1"/>
  <child id="3"/>
  <child id="2"/>
</root>
```

Considering Example 4, both input documents contain an element that isn't present in the other one. Since the document definition allows `<root>` to contain an arbitrary number of `<child>` elements, the merge result should contain both relying on simple data storage semantics. In addition, the `<child>` elements with `id="1"` of both input documents should be matched and thus should appear only once in the result.

Example 5. Unique nodes

```
<!-- DTD -->
<!ELEMENT root (child)>
<!ATTLIST child id ID>
```

```
<!-- 1st input -->
<root>
  <child id="1"/>
</root>
```

```
<!-- 2nd input -->
<root>
  <child id="2"/>
</root>
```

Example 5 describes a simlar case, but in contrast to Example 4, the `<child>` element can only occur once per document due to the changed language specification. Thus, simply accumulating the `<child>` nodes will not yield a valid result document regarding the language specification. Injecting additive patch semantics here while considering the first input as the base document and the second input as the patch document, the attributes value of the patch could be interpreted as an update for the matching attribute in the base document. However, it might be even more likely, that the base document should be left untouched in such case of a conflict. This obviously highly depends on the use case. Summarizing, there is a need of parameterizing the conflict handling to at least be able to prefer the base or patch to support different use cases for the structural merge algorithm.

Example 6. Accumulation of non-element nodes

```
<!-- Base document -->
<element id="1" type="a">
  lorem
</element>
```

```
<!-- Patch document -->
<element id="1" useage="b">
  ipsum
</element>
```

```
<!-- possible result -->
<element id="1" type="a" useage="b">
  lorem
  ipsum
</element>
```

Next to the handling of elements, especially in additive patch semantics, it would also make sense to accumulate non-element nodes to add or update new attributes or text nodes to already available nodes in the base documents. Example 6 shows a simple example of a base document and a patch document. Each containing an attribute that is not present in the other one. Merging base and patch, it can be assumed that both attributes are intended to occur in the resulting document. The even more interesting question raised in Example 6 is how to handle text nodes. In this example many different results can be discussed to be valid dependent on the document's semantics or even dependent on the intention of the merge. It can be easily seen, that the complexity of merging full text properly is very high and we are not able to takle this issue here exhaustively. However, especially in our daily context of XML used as specification language for framework configurations, you most often will find no full text in text nodes. Thus, we will focus on the simple use case of the merging text nodes like values of attributes. This indicates text nodes to just describe a simple structured value, which can be easily interpreted. By this, we can discuss different proper merge techniques of text nodes like appending text with or without a separator or even replace text nodes entirely in the base document by the patch's corresponding text node.

3.3. Result of the analysis

Summarizing, a generic 2-way XML merge algorithm just considering the XML meta-language specification will not be able to merge different XML-based languages properly. The same holds generic 2-way merge algorithms just considering the language specifications e.g. DTDs and XSDs as we observed, that the available XML language specifications do not cover all important properties for making a proper merge feasible. We observed, that the semantics of the language have to be considered to provide proper merge rules.

4. The MergeSchema

To overcome the lack of information according to the notion of equality of nodes as well as according to the individual merge operation to be performed, our approach focuses on the specification of further merge relevant information in a seperate language specific document further referenced as MergeSchema. In the following section the important parts of the MergeSchema language will be presented essentially again on the basis of different examples. The MergeSchema itself is specified as a XML-based language[1] as well.

4.1. Building the MergeSchema

The basic idea of specifying different notions of equality for different element nodes is based on XPath expressions. Thus, the assessment of the equality of two element nodes will be done by evaluating and comparing the corresponding XPath expression on both element nodes to be compared.

Example 7. MergeSchema for Example 4

```
<merge-schema>
  <handling for="child">
    <criterion xpath="./@id"/>
  </handling>
</merge-schema>
```

As a first simple example consider the MergeSchema in Example 7 provided for the input documents presented in Example 4. The `merge-schema` contains one merge rule described by the `<handling>` element. The `handling` describes a simple equality expression for `<child>` elements, which are specified to be equal if their `id` attribute's values are equal. The latter is described in the match `<criterion>`, which `xpath` expression points to the child's `id` attribute node ("./" points to the current node, "@" to an attribute of that node). A handling specification for the `<root>` element is not necessary. It is always assumed that the document's root elements match according to their full qualified element name with each other. Otherwise, the 2-way merge will not be considered to be possible at all. Utilizing this MergeSchema the `<child id="1"/>` elements from Example 4 of the base and patch will be matched with each other. The remaining `child` element can simply be accumulated since no further match is found.

If no `handling` is specified for an element, element nodes will be compared due to their recursive identity, meaning comparing all attribute nodes as well as child nodes value by value. In contrast to that, you can also imagine elements to just occur once as specified in the language specification or simply as of code conventions of a target code architecture. As an example take the `<title>` element of the docbook specification. To ensure this, "./true()" can be used as a `criterion's xpath` expression, which simply always evaluates to true. This will lead to a match of any `title` element between base and patch under an equally identified parent element. Treating each `title` element to be equal for the merge algorithm, there are two use cases coming up with the patch semantics for processing the merge. On the one hand, you could imagine a prefer base approach, which simply discards the title element from the patch and leaves the matching base's `title` element untouched. On the other hand, you can discuss it the other way around, treating the patch as an update for the base resulting in the patch's `title` element to occurr in the result. However, both use cases can be easily motivated in a generative context and thus should be supported by a proper merge algorithm. For now we do not cover this as a property to be specified in the MergeSchema. However, it is configurable as a global parameter of our implementation of the merge algorithm allowing to prefer the base's or patch's value.

[1] https://github.com/may-bee/lexeme/blob/development/src/main/resources/xsd/merge-schema.xsd

Example 8. Individual merge of HTML tables

```
<!-- Base -->
<table>
  <tr>
    <th>Group No.</th>
    <th>Attendees</th>
  </tr>
  <tr>
    <td>1</td>
    <td>John Doe</td>
  </tr>
  <tr>
    <td>2</td>
    <td>Joe Bloggs</td>
  </tr>
</table>
```

```
<!-- Patch -->
<table>
  <tr>
    <th>Group No.</th>
    <th>Attendees</th>
  </tr>
  <tr>
    <td>1</td>
    <td>G. Raymond</td>
  </tr>
</table>
```

```
<!-- MergeSchema -->
<merge-schema>
  <handling for="table">
    <criterion xpath="./tr/th"/>
    <handling for="tr">
      <criterion
        xpath="./td[1]/text()"/>
      <handling for="td"
        attachable-text="true"/>
    </handling>
  </handling>
</merge-schema>
```

```
<!-- Result -->
<table>
  <tr><!-- ==base --></tr>
  <tr>
    <td>1</td>
    <td>John DoeG. Raymond</td>
  </tr>
  <tr>
    <td>2</td>
    <td>Joe Bloggs</td>
  </tr>
</table>
```

Example 8 introduces a more complex example of merging two HTML tables in a very specific way. The base document specifies a double-columned table listing groups with their identifier and attendees. The intention of the patch is to add a new attendee to an already existing group by enriching the `Antendees` cell's value of the corresponding group by an additional person's name. Therefore, we describe the data to be added in the patch, in specific, the already known table and the group with the new attendee's name. To merge base and patch documents as intended, the merge schema has to be specified as follows. First, we identify the `table` by their column headers (`xpath="./tr/th"`) as already been discussed in Example 3. Next, we have to identify the rows (`tr`) by their first column's value respectively `Group No.`(`xpath="./td[1]/text()"`). Given that, the merge algorithm will find the matching rows and compare its children, whereas the first column will obviously been detected as identity and left unchanged. However, the merge of the second column will result in a conflict unless a possibility of conflict resultion is given. Besides prefering one of the documents, there is also the ability to accumulate the patch's text nodes to the base's one's by adding a `handling` for `td` elements allowing text node attachments (`attachable-text="true"`). The current pitfall here, is the fact that the text nodes are simply accumulated and thus not appendend with any possibly custom text separator. This already shows one of the shortcomings of the current implementation. For the moment (custom) separators for appending text is just supported for attributes due to technical reasons of the algorithm implementation.

Finally, working with XML we also have to discuss XML namespaces. Namespaces are used for language separation in XML documents as one XML document can contain elements from multiple XML-based languages. With the introduction of XML namespaces, the distinction between local and full qualified names of element nodes has to be considered. While the full qualified name looks like `nsp:tag-name` the local name is just the `tag-name` and `nsp`: the abbreviation of its namespace. To overcome the issue of non unique local names when using different XML namespaces in a document, we restrict each MergeSchema to one XML namespace. Thus, we analogously separated concerns as already done by the introduction of different XML namespaces respectively languages. As a consequence, the merge algorithm will have to adduce the correct MergeSchema dependent on the current element's namespace to gather further merge relevant information for each language.

Example 9. Consideration of XML namespaces

```
<!-- Base -->
<a:root xmlns:a="firstNameSpace">
  <a:child id="1"/>
  <b:root xmlns:b="secondNamespace">
    <b:child id="1"/>
  </b:root>
</a:root>

<!-- MergeSchema firstNamespace -->
<merge-schema>
        <definition
    namespace="firstNamespace"/>
  <handling for="child">
    <criterion xpath="./@id"/>
  </handling>
</merge-schema>
<!-- MergeSchema secondNamespace -->
<merge-schema>
  <definition
    namespace="secondNamespace"/>
  <handling for="child">
    <criterion xpath="true()"/>
  </handling>
</merge-schema>
```

```
<!-- Patch -->
<a:root xmlns:a="firstNameSpace">
  <a:child id="2"/>
  <b:root xmlns:b="secondNamespace">
    <b:child id="2"/>
  </b:root>
</a:root>

<!-- Result -->
<a:root xmlns:a="firstNameSpace">
  <a:child id="1"/>
  <a:child id="2"/>
  <b:root xmlns:b="secondNamespace">
    <b:child id="2"/>
  </b:root>
</a:root>
```

Considering Example 9, the `child` element is present in two different languages introduced by the XML namespaces `firstNamespace` and `secondNamespace`. Different languages may most commonly introduce different semantics of poentially equally named elements. To provide a MergeSchema for each of the languages, we introduce the `definition` element to the `merge-schema` root indicating the `namespace` the MergeSchema corresponds to. Example 9 specifies the `child` element of the `firstNamespace` to match on its `id` attribute. In contrast to that, the `child` element of the `secondNamespace` is specified to match any `child` element of the same language, i.e. is forced to occur only once. To gain the result as shown in Example 9, we in addition have to assume a global conflict resolution of *patch wins* . This simple example already indicates quite well, that by considering the elements' language semantics, we were able to merge the documents more properly than just assume the same semantics for each XML element regardless its language.

4.2. Nesting of Handlings

The XML language definition allows a distinction between the type and the name of an element. The name and type information of an element is specified in the corresponding XSD or DTD. But especially, in an XSD the mapping between name and type is not bidirectional. This introduces mainly two issues with XSDs related to the specification of the MergeSchema presented so far. First, the name of an element needs only be unique under any parent element and second, types can be shared between multiple namespaces due to type reuse and extension. In order to overcome the second problem, we focus on element names rather than on the types to connect a MergeSchema's `handlings` to. Thus, we were able to ensure that a MergeSchema just covers the specification of the corresponding language rather than also covering the handling of types inherited from different languages. To also takle the first problem also, the MergeSchema specification allows nesting of `handlings` to specify the `handling's` context to be valid for. As given by the XML specification the context basically can be summarized as the axis of ancestors of a node.

Introducing nesting immediately introduces the need of semantics for overriding `handlings` and their visibility for application. A `handling` is visible and thus applicable for all its sibling `handlings` and all `handlings` on the descendant axis. However, by specifying `handlings` for the same element names on the same descendant axis, the most specific `handling` in each context will visible and thus applicable dependending on the current context of the node to be processed by a merge algorithm.

Example 10. Nesting of handlings

```
<merge-schema>
  <handling name="a"/>
  <handling name="b">
    <handling name="c"/>
    <handling name="d">
      <handling name="a"/>
    </handling>
  </handling>
</merge-schema>
```

For a better understanding consider Example 10. Assuming the merge algorithm already merged `<b>` or `<c>` elements and stepping down on the descendant axis encounters an `<a>` element, we would apply the first `handling` declaration. However, if the merge algorithm would have merged a `<d>` element and stepping down on the descendant axis encountering an `<a>` element, we would apply the fifth `handling` since it overrides the first one.

4.3. Reduction of Redundancy

MergeSchemas obviously can become quite large if considering complex XML languages. Often, many elements can be identified with the same `xpath` statement or merged with any specific default configuration. Some but not all XML languages use XML's id attribute as identifier for elements, e.g. reconsider the language for Spring Bean declaration discussed in Section 3.1. Such notion of equality might even hold for multiple elements of the smae language and thus will result in many duplicates of `criterion` specifications. Therefore, the `<default-criterion>` element has been introduced as a child of the `merge-schema` root to be able to change the default criterion with `xpath="./*"` (recursive equality) to anything more suitable specific to each language.

4.4. Further MergeSchema Features

Similiar to many other concepts around 20% of the language specification of MergeSchemas already covers around 80% of use cases. Describing the whole language specification of MergeSchemas would exceed the scope of this paper. Therefore, we briefly want to point out also implemented concepts we could not cover in this scope, but which have been experienced to be necessary for real world application.

Ignore order on matching criterion's XPath results: Evaluating the `xpath` expression of a `criterion` may result in an ordered list of nodes. By default, this list is compared to another list by considering the the order as well. However, as indicated in Section 3, ordering of nodes has not to be considered in each language.

Support for XSD type inheritance: As shortly discussed in Section 4.2, XML introduces types and element names which are losely coupled. This may lead to redundancy e.g. of `criterion` specification in the MergeSchema as of the fact, that the MergeSchema refers to element names rather than to XML types but you may want to specify criterions not in each element anew. However, this is a very advanced topic, which is just partially adressed in our approach.

Connect multiple namespaces to one MergeSchema: Most commonly, XML namespace aliases for the latest release of evolving language specifications are introduced next to the XML namespaces for each released version of a XML language. Thus, the latest released language specification can be referenced by two different XML namespaces. However, you do not want to specify two equivalent MergeSchemas for it, but just connect a MergeSchema to different namespaces.

Conditional merge operations: In some use cases, the merge operation for elements has to differ in a specific context, e.g. the merge operation should be different for the first element respectively the subsequent elements with the same name. Therefore, we introduced some kind of `where` clauses to activate/deactivate `handlings` in specific contexts.

Value precedence specification: Especially attributes can be restricted to some enum values, which in case of a conflict during merge should not just processed stupidly based on a global conflict handling. Moreover, the merge algorithm should follow an individual precedence of the values specified in the MergeSchema.

5. LeXeMe

To enable structural 2-way merging of XML documents based on the previously introduced MergeSchema, we developed the Java library LeXeMe[1] to be used in a company internally developed and maintained code generator [Brunnlieb2014]. As of the complexity of the overall MergeSchema language specification, the following just focuses on an additional feature introduced by the merge algorithm itself rather than explaining how the algorithm implements the previously discussed language semantics of the MergeSchema in detail.

5.1. Preserving the Document Order

The merge algorithm implemented in LeXeMe mainly introduces one important additional feature to gain better merge results. Remembering the context considerations for our approach, the XML documents to be merged are also manipulated by humans. Thus, the merge result should be readable by humans easily. This essentially means, that it is important to preserve the ordering of the documents XML child elements to not destroy the humans mental model. Thus, the LeXeMe merge algorithm tries to preserve as much ordering in the document as possible to produce more predictable results for humans. The following listing of pseudo-code describes a simplified version of the merge algorithm to merge element nodes.

```
mergeElement(base, patch)
  FORALL elements IN patch.ChildElements AS pElem
    IF (pElem.findMatch(base.ChildElements) AS bElem) THEN
      // special case: child nodes list starts with text nodes
      tNodes = text nodes patch.ChildNodes starts with
      IF (tNodes not empty && pElem is first of patch.ChildElements) THEN
        Add tNodes to base before base.bElem
      // recursively merge pElem and bElem
      base.bElem = mergeElement(bElem, pElem)
      // handle subsequent text nodes
      Add all subsequent text nodes of pElem after bElem
      Remove tNodes and pElem from patch
  // handle non matching nodes
  IF (patch.ChildElements is not empty)
    Try to align and merge remaining patch elements to base
    Accumulate remaining nodes
RETURN base
```

The listing describes a method called `mergeElement`, which takes a `base` element node as well as a matching `patch` element node as inputs. For simplicity reasons, we ignored all the processing steps handling the corresponding MergeSchema specification and conflict handling. The algorithm starts with entering the root nodes of the base and patch, whereas it is assumed as a precondition for the shown function, that the root nodes match each other, i.e. have the same element name and namespace.

The first part of the algorithm (line 2-12) tries to find a matching child element node (`bElem`) for each `patch` element child (`pElem`) in the list of element children of `base`. If there is a matching base child `bElem`, the algorithm merges `pElem` and `bElem` recursively (line 9). Furthermore and even more important, the text nodes are merged based on the location of `bElem` as a sophisticated comparison and alignment of text nodes is a non-trivial problem not covered. Thus, we chose to stick each list of sequent text nodes to their next element predecessor. Given that, all immediately subsequent text nodes of `pElem` will be merged right after `bElem` to the base (line 11). As we are just consider subsequent text nodes, the exceptional case of a list of children starting with text nodes is considered in lines 5-7.

[1] Language-aware XML Merger; https://github.com/may-bee/lexeme

After processing all matched child elements of `patch` and their subsequent text nodes, the merge of the non-matching child elements of `patch` as well as their subsequent text nodes have to be merged as well. To find a suitable place for the remaining elements, we memorize the siblings of each child element of `patch`. Based on that knowledge, the algorithm tries to align and merge the remaining element nodes of `patch` to base. Surely, it is not trivial to preserve all orderings of the `patch`'s children in the `base` as well. Therefore, we simply implemented a first come first serve approach discarding non-preservable alignment needs of further nodes and thus prevent the algorithm from struggling with a general optimization problem. Finally, the remaining child elements and its text nodes, which could not be aligned and merged until now, are accumulated at the end of the `base` (line 16).

Summarizing, this part of the complete LeXeMe implementation is obviously a best effort approach, which essentially works with non-ordered XML documents best, just preserving some of the ordering within the document in the resulting document. This algorithm has not been designed to preserve the semantics of e.g. xHTML documents entirely. We claim, that this is not easily manageable as it will introduces the need for merging full text, which finally will lead to linguistic analysis.

5.2. Limitations to the Implementation

The merge algorithm implemented by LeXeMe assumes the root elements of the two input XML documents as mergable, i.e. declaring the same full qualified element name. Given that, the recursive merge processing is able to start and produce a result. Until now, the merge algorithm just aggregates nodes based on the MergeSchema and not based on the DTD or XSD knowledge as well. This especially includes the multiplicity constraints of a XSD, which can be more fine-grained than it can be described in a MergeSchema (1 to unbound). LeXeMe only validates the merge result against its DTD or XSD detecting wrongly accumulated nodes or attribute values. Furthermore, LeXeMe tries to preserve the document structure of the base as explained in the previous section. However, merging ordering sensitive documents as e.g. xHTML will remain unsolved taking the entire document semantics into account. At least our approach may produce non-intended results in its current implementation. Finally, as we introduced by the context the implementation is operating in, LeXeMe is not able to perform any deletions in the resulting document as the MergeSchema as well as the two documents simply do not provide any information to this regards.

6. Industrial Experiences

Compared to the previous implementation of a generic structural xml merger based on the XMLMerge [XMLMerge] framework, the structural merge has become a lot more powerful with the LeXeMe. Although we tuned the generic XML merge algorithm a lot to make it even more smarter, we quickly observed multiple conflicts in our needs to merge different XML-based languages properly. As indicated, the original motivation for structurally merging XML files comes from the need of adding new XML contents to already existent XML documents. Especially in the context of Java applications, there are a bunch of frameworks, which can be configurated by using XML documents for various use cases. As an example, there are multiple XML documents to configure the component framework Spring. So we implemented some mergeSchemas[1] to be able to merge such configuration files even better than with generic XML merge approaches. We successfully used LeXeMe in a company internal generator implementation to add new `<bean>` declarations, add new `<property>` elements to already existing bean declarations, as well as update different XML attributes accordingly. Furthermore, we were able to generate additional XML code to Java persistence configurations[2] as well as to Dozer Mapper configurations[3].

7. Conclusion

First, we discussed different examples of merging two XML documents to underline the different needs to be takled by a structural 2-way merge algorithm. Based on that, we developed a language specific MergeSchema, which encodes different rules derived from the language's semantics to handle structural merges properly for different XML-based languages. Finally, we described the core idea of the merge algorithm considering the MergeSchema. We showed, that structural 2-way merges for XML documents can be much more effective if

[1] https://github.com/may-bee/lexeme/tree/development/src/main/resources/mergeSchemas

[2] http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/persistence/index.html

[3] http://dozer.sourceforge.net

considering further semantics of the language to be merged. However, specifying a MergeSchema for each XML based language to be supported for 2-way merging introduces additional effort. We further indicated different limitations of our approach as e.g. the arbitrary complex problem of merging text nodes has not been covered in our approach as we mainly focused on the structural merge rather than on the merge of text nodes potentially containing full text.

Finally, we identified the following future works. We identified the need to also process information from already available language specifications like DTD or XSD right during the merge to not end up in invalid results according to the language specification. This especially adresses the XSD as it not only provides more fine grained multiplicity constraints, but also gives hints about the validity of attribute values during merge. As another topic of future work, LeXeMe currently provides a parameter for conflict resolution globally indicating the algorithm to prefer the base or patch values when detecting non-resolvable conflicts. It might be more suitable to provide such kind of conflict resolution on a more fine-grained level specified per element in the MergeSchema. Thus, MergeSchemas could not only be used for language specific 2-way XML merge, but also for an use case specific document merge.

8. Related Works

DeltaXML [LaFontaine2002] has been developed for merging documents by first deriving so called delta documents from two XML documents to be merged. Such delta document can then be further processed to show the differences or to be executed by a structural merge algorithm. This already is one of the major differences between LeXeMe and DeltaXML as DeltaXML determines a delta document of the inputs encoding all information of all input documents. Furthermore, DeltaXML just processes the XML meta-language and does not consider any further language specific merge properties during a merge. DeltaXML just provides a trivial matching algorithm and just provides a manual approach to influence element matching via `deltaxml:key` attributes by manipulating the input documents.

XMLMerge [XMLMerge] was mainly one of the open source libraries takling the general issue of merging XML documents in a 2-way fashion. However, programmatically adjusting the merge algorithm to some language needs will immediately collide with any other languages' needs. Diving deeper into the libraries API it may be also possible to encode an appropriate merge behavior for each XML language. However, you will have to change your programs implementation on changes or different merge needs, which makes the approach highly inflexible. To overcome this, we chose a more declarative approach to easily change merge properties without having to change the implementation of our merge algorithm.

Our approach and the specification of the MergeSchema is somehow influenced by the work of Tufte and Maier [Tufte2001]. They introduce a so called Merge Template, which purpose is quite similar to our MergeSchema. One of the basic differences is the context of application. The Merge Template is designed to accumulate XML elements from two input streams rather than from completely available and readable input documents. To identify a pair of mergeable elements the Merge Template specifies nodes which have to be identical in two considered input elements. Due to the fact, that they discuss streams as input, neither deep matching (i.e. matching via descendant nodes) nor matching of single nodes e.g. by attribute's values is supported. The approach via XPath as it is used in LeXeMe provides a lot more possibilites to define advanced matching criterias. The focus of the Merge Template and its corresponding accumulation algorithm only relies on non ordered XML data. XML languages implying ordering of nodes are thus not mergeable by Tufte and Maier's approach.

Bibliography

[LaFontaine2002]	Robin La Fontaine. Copyright © 2002. XML Europe. *Merging XML files: A new approach providing intelligent merge of xml data sets*.
[XMLMerge]	Pip Stuart. Copyright © 2004. *XML::Merge - search.cpan.org*. http://search.cpan.org/~pip/XML-Merge-1.2.565EgGd/Merge.pm *accessed 03/07/2016* .
[Brunnlieb2014]	Malte Brunnlieb and Arnd Poetzsch-Heffter. Copyright © 2014. GI. *Architecture-driven Incremental Code Generation for Increased Developer Efficiency*.
[Brunnlieb2016]	Malte Brunnlieb and Arnd Poetzsch-Heffter. Copyright © 2016. ACM. *Application of Architecture Implementation Patterns by Incremental Code Generation*.
[LaFontaine2001]	Robin La Fontaine. Copyright © 2001. XML Europe. *A Delta Format for XML: Identifying changes in XML files and representing the changes in XML*.
[Tufte2001]	Kristin Tufte and David Maier. Copyright © 2001. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. *Aggregation and Accumulation of XML Data*.
[Abiteboul2001]	Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Copyright © 1998. Stanford InfoLab. *Incremental Maintenance for Materialized Views over Semistructured Data*.

Linked Data Templates

Ontology-driven approach to read-write Linked Data

Martynas Jusevičius

AtomGraph

`<martynas@atomgraph.com>`

Abstract

In this paper we summarize the architecture of Linked Data Templates, a uniform protocol for read-write Linked Data.

We start by overviewing existing approaches as well as protocol constraints, and propose a declarative approach based on URI-representation mapping, defined as templates. We then introduce an abstract model for RDF CRUD interactions based on template matching, outline the processing model, and provide an example.

We conclude that LDT provides a new way to build applications and can be used to implement the ontology-driven Semantic Web vision.

Keywords: HTTP, REST, RDF, Linked Data, SPARQL, XSLT, SPIN, declarative, data-driven

1. Introduction

Linked Data is a vast source of information available in RDF data model. In this paper we describe Linked Data Templates: a generic method for software agents to publish and consume read-write Linked Data. By doing so, we facilitate a distributed web as well as redefine the software development process in a declarative manner.

Web applications duplicate a lot of domain-specific code across imperative programming language implementations. We abstract application logic away from source code and capture it in a machine-processable representation that enables reuse, composition, and reasoning. Our goal is to permit software to operate on itself through metaprogramming to generate higher-level applications — a feature not available in imperative languages. Having a uniform homoiconic representation, we reduce application state changes to uniform Create, Read, Update, Delete (CRUD) interactions on Linked Data representations [1].

Linked Data representations describe application resource properties. A consuming process can query and change resource state by issuing requests to resource URIs. On the producing end, representations are generated from an RDF dataset, either stored natively in a triplestore or mapped from another data model. The exact logic of how representations are generated and stored is application-specific.

By establishing a mapping between URI address space and the RDF representation space, we can model application structure in terms of RDF classes that map URIs to RDF queries and updates. We choose RDF-based ontologies as the LDT representation, as it is the standard way to define classes and satisfies both the machine-processability and the homoiconicity requirements. Ontology as a component for application structure is what distinguishes LDT from other Linked Data specifications such as Linked Data Platform.

In the following sections, we explain the motivation behind LDT and describe the LDT architecture in more detail. First we establish that applications can be driven by ontologies that can be composed. We then introduce templates, special ontology classes that map URI identifiers to request-specific SPARQL strings. We proceed to describe how a process matching templates against request URI is used by RDF CRUD interactions that generate Linked Data descriptions from an RDF dataset and change the dataset state. Lastly, we map the interactions to the HTTP protocol and show how the client can use HTTP to interact with LDT applications.

2. Distributed web as read-write Linked Data

2.1. A protocol for the web of data

Smart software agents should navigate the web of data and perform tasks for their users — that has been an early optimistic vision of the Semantic Web [2]. Ontologies and ontology-driven agents were central to this vision, to provide the means to capture domain logic in a way that sustains reason. It was largely forgotten when the high expectations were not met, and the community focus shifted to the more pragmatic Linked Data.

doi:10.14337/XMLLondon16.Jusevicius01

In order to enable smart agents, we need to provide a uniform protocol for them to communicate. Such a protocol, understood by all agents in the ecosystem, decouples software from domain- or application-specific logic and enables generic implementations. The web has thrived because HTTP is such protocol for HTML documents, and web browsers are generic agents.

REST readily provides uniform interface for a protocol, which has been successfully implemented in HTTP. The interface is defined using 4 constraints [3]:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application state

Since the Linked Data resource space is a subset of REST resource space, we can look at how these constraints apply to standard Linked Data technologies:

- URIs identify resources
- RDF is used as resource representation. Resource state can be changed using RDF CRUD interactions.
- RDF formats are used for self-describing Linked Data requests and responses

2.2. Ontology-driven Linked Data

The main point of interest for this paper is RDF CRUD interactions, the central yet underspecified component of read-write Linked Data. The only standard in this area is W3C Linked Data Platform 1.0 specification, which defines a set of rules for HTTP interactions on web resources, some based on RDF, to provide an architecture for read-write Linked Data on the web [4]. It has several shortcomings:

- It is coupled with HTTP and provides no abstract model for RDF CRUD
- In order to accommodate legacy systems, it does not mandate the use of SPARQL. SPARQL is the standard RDF query language and does provide an abstract model [5].
- It does not offer a standard way for agents to customize how CRUD interactions change resource state

The Linked Data API specification defines a vocabulary and processing model for a configurable, yet read-only API layer intended to support the creation of simple RESTful APIs over RDF triple stores [6]. Hydra Core Vocabulary is a lightweight vocabulary to create hypermedia-driven Web APIs and has similar goals to combine REST with Linked Data principles, but does not employ SPARQL and focuses on JSON-LD [7].

As we can see, currently popular Linked Data access methods are either read-only or do not use SPARQL. They use lightweight RDF vocabularies, but not formal ontologies that enable reasoning, as does the original ontology-driven Semantic Web vision. Although there has been a fair share of research and development in the area of ontology-driven applications [8] [9], it focuses mostly on domain and user interface modeling, and not representation manipulation modeling.

2.3. LDT design

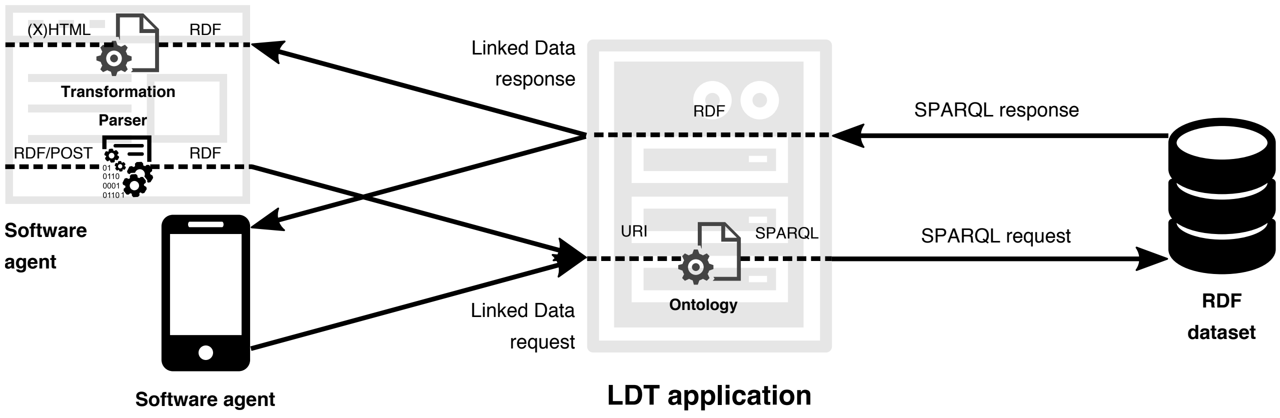
We propose Linked Data Templates, an ontology-driven approach to read-write Linked Data. It builds on the following constraints:

- there is a mapping between URI address space and the RDF representation space. It is used to determine resource's representation from its URI identifier.
- applications are read-write and backed by SPARQL 1.1 compatible services to decouple them from database implementations
- application structure is defined in an ontology to enable reasoning and composition
- application state is driven by hypermedia (HATEOAS) to satisfy REST constraints

XSLT is a homoiconic high-level language for the XML data model [10]. We wanted to follow this approach with a Linked Data specification, and as a result, XSLT heavily influenced the template-based design of LDT. We draw multiple parallels between XSLT stylesheets and LDT ontologies, XML source documents and RDF datasets, XPath patterns and URI templates etc.

AtomGraph Processor[2] is an open-source implementation of LDT. The commercial AtomGraph Platform[3] provides a multi-tenant environment and has been successfully used to build rich LDT applications for product information management and library data.

Figure 1. Main components of LDT architecture[1]



3. Application ontologies

[Definition: An LDT *application* represents a data space identified by its base URI, in which application resource URIs are relative to the base URI.] The only external interface an application provides is RESTful Linked Data: application produces RDF representations when resource URIs are dereferenced, and consumes RDF representations when requested to change resource state.

Application structure, the relationships between its resources, is communicated through representations. Representations are generated from, and stored in, an RDF dataset. Two different applications should be able to use the same dataset yet expose different structures because they produce representations and change state differently. It follows that application structure can be defined as instructions for representation processing.

An ontology is an efficient way to define such structure declaratively. We use OWL to define LDT application ontologies with RDF query and state change instructions specific to that application. We use SPARQL to encode these instructions, because it is the standard RDF query and update language and can be conveniently embedded in ontologies using SPIN RDF syntax. Using SPARQL service as the interface for the dataset, applications are independent from its implementation details.

An LDT application ontology may comprise several ontologies, contained in different RDF graphs. For a given application, one of these serves as the *principal ontology*. Ontologies are composed through the standard `owl:imports` mechanism, with the additional concept of import precedence, which makes ontologies override each other depending on the import order.

LDT does not make any assumptions about the application structure. There is however a useful one: a resource hierarchy consisting of a container/item tree. It is similar to the container/resource design in LDP, but based on the SIOC ontology instead [11]. The vast majority of Web applications can be modeled using this structure.

4. Templates

[Definition: A *template* is a declarative instruction contained in an ontology, defined using LDT vocabulary[1], and driving RDF CRUD processing.] It is a special ontology class that maps a certain part of the application URI space to a certain SPARQL string. A template can be viewed as a function with URI as the domain and SPARQL as the range, which are the two mandatory parts of the template, detailed below.

A template domain is defined using `ldt:path` property and a regex-based JAX-RS URI template syntax [12]. It is a generic way to define a class of resources based on their URI syntax: if an URI matches the template, its resource is a member of the class. Starting with a catch-all template that matches all resources in an application, we can specialize the URI pattern (e.g. by adding fixed paths) to narrow down the class of matching resources.

A template range is defined using `ldt:query` property and SPIN RDF syntax, while updates use `ldt:update`

[1] Icons used in the diagram made by Freepik http://www.flaticon.com
[2] AtomGraph Processor - https://github.com/AtomGraph/Processor
[3] AtomGraph Platform - http://atomgraph.com
[1] LDT vocabulary is planned to have http://www.w3.org/ns/ldt# namespace in the final specification

property [13]. URI that matches URI template is passed to SPARQL using a special variable binding `?this` (path variables from the URI template match, if any, are not used since URIs are opaque). Starting with the default query `DESCRIBE ?this`, we can specialize it with a graph pattern, for example to include descriptions of resources connected to `?this` resource. The query forms are limited to `DESCRIBE` and `CONSTRUCT`, as the required result is RDF graph.

An important feature of LDT templates is *annotation inheritance*, which enables code reuse but requires reasoning. It mimics object-oriented multiple inheritance: a class inherits annotation properties from its superclasses via the `rdfs:subClassOf` relation, unless it defines one or more properties of its own which override the inherited ones. SPIN takes a similar object-oriented world-view and uses subclass-based inheritance.

5. Processing model

We have established that application state is queried and changed using Linked Data requests that trigger RDF CRUD in the form of SPARQL. We can constrain the requests to 4 types of CRUD *interactions* that map to either SPARQL query or update. An interaction is triggered by a Linked Data request and results in query or change of application state by means of SPARQL execution [14].

Table 1. LDT interaction types

Interaction type	SPARQL form	Generated from
Create	`INSERT DATA`	request RDF entity
Read	`DESCRIBE/ CONSTRUCT`	`ldt:query`
Update	`DELETE; INSERT DATA`	`ldt:update;` request RDF entity
Delete	`DELETE`	`ldt:update`

`DESCRIBE` and `CONSTRUCT` forms are generated from `ldt:query` SPARQL templates; `DELETE` is generated from `ldt:update` SPARQL template. `INSERT DATA` is generated from the RDF in the request entity, either as triples or as quads. Update interaction combines two updates into one SPARQL request.

[Definition: We refer to the software that uses application templates to support the interaction as an LDT *processor*.] A processor consists of several sub-processes that are triggered by a Linked Data request and executed in the following order:

1. A validation process validates incoming RDF representations against SPIN constraints in the ontology. Invalid data is rejected as bad request. Only applies to Create and Update.
2. A skolemization process matches request RDF types against ontology classes and relabels blank nodes as URIs. Only applies to Create.
3. A matching process matches the base-relative request URI against all URI templates in the application ontology, taking import precedence and JAX-RS priority algorithm into account. If there is no match, the resource is considered not found and the process aborts.
4. A SPARQL generation process takes the SPARQL string from the matching template and applies `?this` variable binding with request URI value to produce a query or an update, depending on the interaction type. `BASE` is set to application base URI.
5. A SPARQL execution process executes the query/ update on the application's SPARQL service. If there is a query result, it becomes the response entity.
6. A response generation process serializes the response entity, if any. It uses content negotiation to select the most appropriate RDF format, sets response status code, adds ontology URI, matched template URI and inheritance rules as header metadata.

For the container/item application structure it is convenient to extend this basic model with pagination, which allows page-based access to children of a container. It requires `SELECT` subqueries and extensions to query generation and response generation processes.

Having access to application ontologies, LDT clients can infer additional metadata that helps them formulate successful requests. For example, SPIN constructors can be used to compose new resource representations from class instances, while SPIN constraints can be used to identify required resource properties. Applications with embedded clients become nodes of a distributed web, in which data flows freely between peers in either direction.

5.1. HTTP bindings

The mapping to HTTP is straightforward — each interaction has a corresponding HTTP method:

Table 2. LDT interaction mapping to HTTP

Interaction type	Request method	Success statuses	Failure statuses
Create	`POST`	`201 Created`	`400 Bad Request`
			`404 Not Found`
Read	`GET`	`200 OK`	`404 Not Found`
Update	`PUT`	`200 OK`	`400 Bad Request`
		`201 Created`	`404 Not Found`
Delete	`DELETE`	`204 No Content`	`404 Not Found`

It should be possible to use the `PATCH` method for partial modifications instead of replacing full representation with `PUT`, but that is currently unspecified.

5.1.1. Example

In the following example, an HTTP client performs an Update-Read request flow on linkeddatahub.com application, which supports LDT. Only relevant HTTP headers are included.

First, the client creates a resource representing Tim Berners-Lee by submitting its representation:

```
PUT /people/Berners-Lee HTTP/1.1
Host: linkeddatahub.com
Accept: text/turtle
Content-Type: text/turtle

@base <http://linkeddatahub.com/people/Berners-Lee> .
@prefix ldt:  <http://www.w3.org/ns/ldt#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

<> a ldt:Document ;
  foaf:primaryTopic <#this> .

<#this> a foaf:Person ;
  foaf:isPrimaryTopicOf <> ;
  owl:sameAs
  <https://www.w3.org/People/Berners-Lee/card#i> .
```

Let's assume the match for `/people/Berners-Lee` request URI is the `:PersonDocument` template in the application ontology:

```
@base         <http://linkeddatahub.com/ontology> .
@prefix :     <#> .
@prefix ldt:  <http://www.w3.org/ns/ldt#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sp:   <http://spinrdf.org/sp#> .

# ontology
: a ldt:Ontology ;
  owl:imports ldt: .

# template
:PersonDocument a rdfs:Class, ldt:Template ;
  ldt:path "/people/{familyName}" ;
  ldt:query :DescribeWithPrimaryTopic ;
  ldt:update :DeleteWithPrimaryTopic ;
  rdfs:isDefinedBy : .

# query
:DescribeWithPrimaryTopic a sp:Describe, ldt:Query ;
  sp:text
      """PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
DESCRIBE ?this ?primaryTopic
WHERE
  { ?this ?p ?o
    OPTIONAL
      { ?this foaf:primaryTopic ?primaryTopic }
  }""" .

# update
:DeleteWithPrimaryTopic a sp:DeleteWhere, ldt:Update ;
  sp:text
       """PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
DELETE {
  ?this ?p ?o .
  ?primaryTopic ?primaryTopicP ?primaryTopicO .
}
WHERE
  { ?this ?p ?o
    OPTIONAL
      { ?this foaf:primaryTopic ?primaryTopic .
        ?primaryTopic ?primaryTopicP ?primaryTopicO
      }
  }""" .
```

The variable binding `(?this, <http://linkeddatahub.com/people/Berners-Lee>)` is applied on the `DELETE` associated with the template. It is combined with `INSERT DATA` generated from the request RDF entity into a single update request. Application base URI is set on the final SPARQL string which is then executed on the SPARQL service behind the application:

```
BASE          <http://linkeddatahub.com/>
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
PREFIX  owl:  <http://www.w3.org/2002/07/owl#>
PREFIX  ldt:  <http://www.w3.org/ns/ldt#>

DELETE {
  <people/Berners-Lee> ?p ?o .
  ?primaryTopic ?primaryTopicP ?primaryTopicO .
}
WHERE
  { <people/Berners-Lee> ?p ?o
    OPTIONAL
      {
        <people/Berners-Lee>
                  foaf:primaryTopic ?primaryTopic .
        ?primaryTopic ?primaryTopicP ?primaryTopicO
      }
  } ;

INSERT DATA {
  <people/Berners-Lee>
    a ldt:Document .
  <people/Berners-Lee>
    foaf:primaryTopic <people/Berners-Lee#this> .
  <people/Berners-Lee#this>
    a foaf:Person .
  <people/Berners-Lee#this>
    foaf:isPrimaryTopicOf <people/Berners-Lee> .
  <people/Berners-Lee#this>
    owl:sameAs
    <https://www.w3.org/People/Berners-Lee/card#i> .
}
```

We assume the representation did not exist beforehand, so it is created instead of being updated (an optimized implementation might have skipped the DELETE part in this case). The application responds with:

```
HTTP/1.1 201 Created
Location: http://linkeddatahub.com/people/Berners-Lee
```

The client can choose to follow the link to the newly created resource URI, and retrieve the same representation that was included with the initial PUT request:

```
GET /people/Berners-Lee HTTP/1.1
Host: linkeddatahub.com
Accept: text/turtle
```

We omit the response, but note that the application would use the DESCRIBE query associated with the matching template to generate the representation.

6. Future work

The use of OWL and SPARQL is probably the biggest advantage and limitation of LDT at the same time. RDF ontology and query tools as well as developers are scarce for mainstream programming languages with the possible exception of Java, making implementations expensive and adoption slow. Query performance is a potential issue, albeit constantly improving and alleviated using proxy caching. On the other hand, OWL and SPARQL provide future-proof abstract models on which LDT builds.

We are working around slow adoption of Linked Data by providing a hosted LDT application platform[1]. It uses metaprogramming to implement complex data management features such as application and resource creation, autocompletion, access control, provenance tracking, faceted search — all done through a user interface, exposing as little technical RDF details as possible.

We envision an ecosystem in which applications by different developers interact with each other: ask for permissions to access or create data, send notifications to users, automate interactions etc.

7. Conclusions

In this paper we have described how read-write Linked Data applications can be modeled using standard RDF/OWL and SPARQL concepts. Linked Data Templates enable a new way to build declarative software components that can run on different processors and platforms, be imported, merged, forked, managed collaboratively, transformed, queried etc. Experience with AtomGraph software has shown that such design is also very scalable, as the implementation is stateless and functional. We expect that substantial long-term savings in software engineering and development processes can be achieved using this approach.

We have shown that SPARQL is the crucial link that reconciles ontology-driven Semantic Web and read-write Linked Data. Using SPARQL, Linked Data Templates define a protocol for distributed web of data as uniform RDF CRUD interactions. LDT already provide features

[1] AtomGraph - http://atomgraph.com

from the original Semantic Web vision, such as ontology exchange between agents, and we are confident it has the potential to implement it in full.

Bibliography

[1] *Create, read, update and delete*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[2] *The Semantic Web*. Tim Berners-Lee, James Hendler, and Ora Lassila. Scientific American. 1 May 2001.
http://www.scientificamerican.com/article/the-semantic-web/

[3] *Representational State Transfer (REST)*. Roy Thomas Fielding. 2000.
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

[4] *Linked Data Platform 1.0*. Steve Speicher, John Arwe, and Ashok Malhotra. World Wide Web Consortium (W3C). 26 February 2015.
https://www.w3.org/TR/ldp/

[5] *SPARQL 1.1 Query Language*. Steve Harris and Andy Seaborne. World Wide Web Consortium (W3C). 21 March 2013.
https://www.w3.org/TR/sparql11-query/

[6] *Linked Data API Specification*.
https://github.com/UKGovLD/linked-data-api/blob/wiki/Specification.md

[7] *Hydra Core Vocabulary*. Markus Lanthaler. 20 March 2016.
http://www.hydra-cg.com/spec/latest/core/

[8] *Agents and the Semantic Web*. James Hendler. 2001.
http://www.cs.rpi.edu/~hendler/AgentWeb.html

[9] *Ontology-Driven Apps Using Generic Applications*. Michael K. Bergman. 7 March 2011.
http://www.mkbergman.com/948/ontology-driven-apps-using-generic-applications/

[10] *XSL Transformations (XSLT) Version 2.0*. Michael Kay. World Wide Web Consortium (W3C). 23 January 2007.
https://www.w3.org/TR/xslt20/

[11] *SIOC Core Ontology Specification*. Uldis Bojārs and John G. Breslin. DERI, NUI Galway. 25 March 2010.
http://rdfs.org/sioc/spec/

[12] *JAX-RS: Java™ API for RESTful Web Services*. Marc Hadley and Paul Sandoz. Sun Microsystems, Inc.. 17 September 2009.
https://jsr311.java.net/nonav/releases/1.1/spec/spec3.html#x3-300003.4

[13] *SPIN - Modeling Vocabulary*. Holger Knublauch. 7 November 2014.
http://spinrdf.org/spin.html

[14] *Architecture of the World Wide Web, Volume One*. Ian Jacobs and Norman Walsh. World Wide Web Consortium (W3C). 15 December 2004.
https://www.w3.org/TR/webarch/#interaction

Scalability of an Open Source XML Database for Big Data

John Chelsom

City University, London

<john.chelsom.1@city.ac.uk>

Abstract

Big Data tools and techniques are starting to make significant contributions in clinical research and studies. We explore the use of XML for holding data in an electronic health record, where the primary data storage is an open source XML database of clinical documents. We evaluate the feasibility of using such a data store for Big Data and describe the techniques used to extend to the massive data sets required for meaningful clinical studies.

Using an open source Electronic Health Records system we have loaded the database with a set of patient records and measured the size of the database from 1 to 20,000 patients, together with the execution time of a typical query to retrieve and combine data across a cohort of patients.

We describe the implementation of a federated data store, whereby we can scale to millions of patient records. We then make projections for the size and search execution time at Big Data scale.

Keywords: Native XML Database, Big Data, Scalability, Electronic Health Records, XRX, HL7 CDA

1. Introduction

In healthcare, the application of Big Data tools and techniques is revolutionizing clinical research and has already contributed to breakthroughs in cancer treatments and management of long term conditions such as diabetes and coronary heart disease [1] [2].

In this paper we explore the scalability of the eXist open source, native XML database for Big Data. We have used an open source Electronic Health Records (EHR) system to load the database with a set of patient records and measured the size of the database from 1 to 20,000 patients, together with the execution time of a typical query to retrieve and combine data across a cohort of patients. Extrapolating these results, we have estimated the largest single database instance which would produce acceptable query performance.

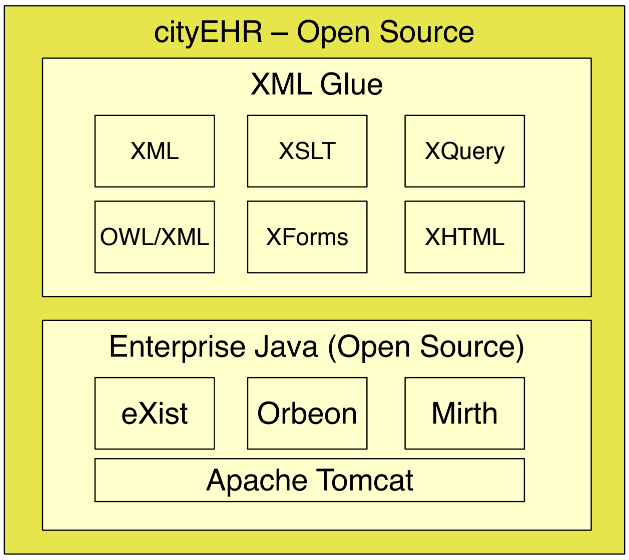
In order to reach the scale required for Big Data in clinical research, which we have assumed to be the full records for 55 million patients, we have investigated the use of federated search across multiple instances of the native XML database. By combining the results from this federated search we are able to achieve the scalability required for Big Data. Our approach to testing the scalability of the XML database follows the same principles described in the BigBench data benchmark proposal [3].

2. Electronic Health Records in XML

Health records typically contain a mix of structured and unstructured data, with some data highly structured (laboratory test results, for example), some lightly structured (clinic attendance notes) and some with no structure, other than meta data (diagnostic images). This variation makes XML an ideal candidate for the basic data representation and storage of healthcare data and has led to the development of a number of standard representations, the most commonly used being the Health Level 7 Clinical Document Architecture (HL7 CDA) [4].

cityEHR [5] is an open source EHR which is currently deployed in five hospitals in the National Health Service in England, as well as being used for research and teaching of health informatics. It is built as an XRX application (XForms – REST – XQuery) on existing open source Enterprise Java components, primarily Orbeon Forms [6], the eXist XML database [7] and the Mirth messaging engine [8] running in Apache Tomcat. The study described in this paper used Orbeon version 3.9 and eXist version 2.2. The architecture of cityEHR, shown in Figure 1, "cityEHR as an XRX Application", was inspired by a previous commercial product called Case Notes, which was implemented using a relational database [9].

Figure 1. cityEHR as an XRX Application



We have used the eXist XML database for developing XML applications in healthcare since soon after its first release in 2000; a version of eXist also ships with Orbeon as the default persistent store for XML. It was therefore a natural choice to use eXist for cityEHR, although there are several open source alternatives, most notably BaseX [10] and Berkeley DB XML Edition [11].

eXist ships with the Jetty Java Servlet container by default and for this study we have used that default installation running as a service in the MS Windows operating system. For live deployments of cityEHR in hospitals we can also run the eXist database in the same Tomcat instance as Orbeon, which is a simpler environment to deploy and maintain.

3. Scalability of the XML Database

3.1. Database Size

To measure the scalability of the XML database we used the test data generation feature of cityEHR to create databases ranging in size from 1 to 20,000 records. This data generation feature imports a sample patient record and replicates that record in the database, varying key data for each test instance created. These key data include the patient identifiers (which are anonymised), dates (which are offset by a random time period) and gender (the proportion of male/female patients can be set for the generated test data).

For the current study, we used a sample record containing 34 clinical documents, which when stored on disk as a single XML file (HL7 CDA format) was approximately 1.35Mb in size. When imported to the XML database, the database of 20k patients was 31.6Gb, including indexes. The equivalent size of the raw XML for this number of patients is 27Gb, so the database 'bloat' is about 17%, which compares well with other databases. The database size up to 20k patients (680k documents) is shown in Figure 2, “Database Size with Increasing Record Count”. Based on these results, our estimate of the size of a database of 100,000 patients is 156Gb, 500,000 patients is 780Gb and 1 million patients is 1.56Tb.

The eXist database is indexed with a range index on two XML attributes and a full text (Lucene) index on the same two attributes. These indexes are sufficient to retrieve any patient data from the HL7 CDA records, since all data follow the same pattern in terms of XML markup.

Figure 2. Database Size with Increasing Record Count

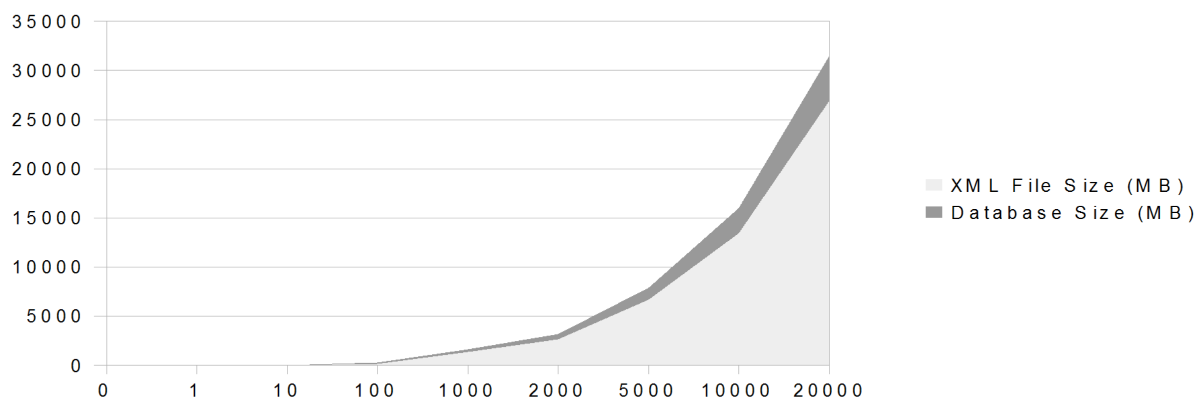


Table 1. Database size and search hits from 1 to 20,000 patients.

Patient Records	0	1	10	100	1000	2000	5000	10000	20000
Database Size (Mb)	80.3	82.1	97.3	248	1700	3210	7950	16000	31600
Hits		1	4	49	499	999	2499	5025	9775

Example 1. Definition of eXist Indexes

```
<collection
xmlns="http://exist-db.org/collection-config/1.0">
  <index>
    <!-- Disable the standard full text index -->
    <fulltext default="none" attributes="no"/>

    <!-- Full text index based on Lucene -->
    <lucene>
      <analyzer class="
org.apache.lucene.analysis.standard.StandardAnalyzer
      "/>
      <text qname="@extension"/>
      <text qname="@value"/>
    </lucene>

    <!-- New range index for eXist 2.2 -->
    <range>
      <create qname="@extension" type="xs:string"/>
      <create qname="@value" type="xs:string"/>
    </range>
  </index>
</collection>
```

3.2. XQuery Formulation and Indexing

For scalability testing we used a single test XQuery which finds all Female patients, returning the patient identifier. This query is typical of any query that finds clinical data in a set of HL7 CDA documents and uses both the range and Lucene full text indexes in eXist.

We also ran similar queries with fewer predicates, so that just the range or full text indexes were used and queries with XQuery for, let and return clauses. The purpose of these additional queries was to verify that the performance results reported here for the documented test query are representative of a wider range of queries that may be run in cityEHR. That said, all queries on clinical data in HL7 CDA are accessing the same extension and value attributes used in the test query.

Example 2. XQuery for performance testing.

```
xquery version "1.0";
declare namespace cda="urn:hl7-org:v3";

/descendant::cda:value
[ft:query(@value,'Female')]
[@extension eq '#ISO-13606:Element:Gender']
[../cda:id/@extension eq '#ISO-13606:Entry:Gender']
/ancestor::cda:ClinicalDocument
/descendant::cda:patientRole/cda:id
```

3.3. Query Execution Time

The results of loading the database with up to 20,000 patients and running the test query are shown below. Table 1, “Database size and search hits from 1 to 20,000 patients.” shows the database size and number of search hits returned; Figure 3, “Database size and search hits from 1 to 20,000 patients.” shows these as a graph.

Figure 3. Database size and search hits from 1 to 20,000 patients.

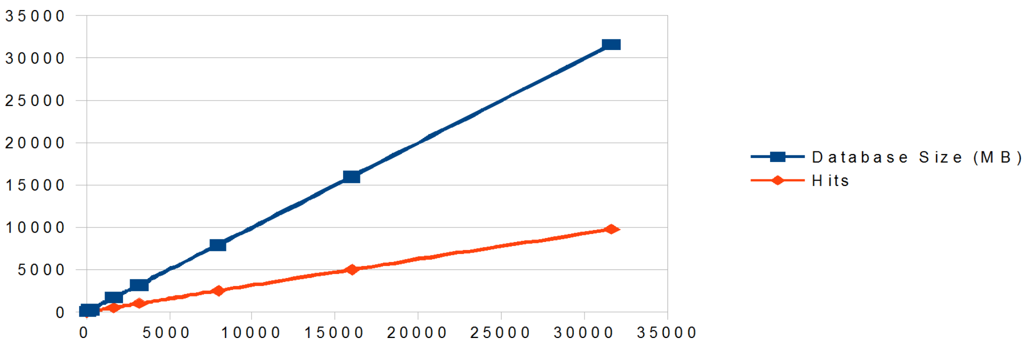


Table 2. Execution time for first and repeated queries from 1 to 20,000 patients.

Patient Records	0	1	10	100	1000	2000	5000	10000	20000
First Query Time (sec)		0.61	0.58	0.59	0.98	1.15	1.92	3.02	6.7
Repeat Query Time (sec)		0.016	0,016	0.06	0.18	0.34	0.58	0.98	2.49

Table 2, “Execution time for first and repeated queries from 1 to 20,000 patients.” shows the execution times for the first and repeated queries; Figure 4, “Execution time for first and repeated queries from 1 to 20,000 patients.” shows these as a graph. These results were obtained on a quad-core Intel i7 processor, with 16Gb RAM running Windows 8. cityEHR was running under Apache Tomcat with 4096Mb Java heap; eXist was running under Jetty with 4096Mb Java heap.

When a slightly different search is performed (e.g. to search for Male rather than Female patients) the execution time reverts to the first query time, so the cache used by eXist does not assist in this case. We have therefore based our projections and conclusions on the slower first query time, rather than the cached query, and ensured during this study that services were restarted and the cache cleared before each measurement was taken.

To check the impact of the Java heap size allocated to the eXist process, we ran the same query at heap sizes from 128Mb to 4096Mb (on a 1000 patient database running on a dual core server, with 8Gb RAM). We found that the query did not complete at 128Mb, completed in 2.9 seconds at 256Mb and then completed consistently in around 2 seconds for heap sizes from 512Mb to 4096Mb. Our conclusion is that once the Java heap is above a sufficient threshold it has no effect on query execution time.

4. Extension to Big Data Scale

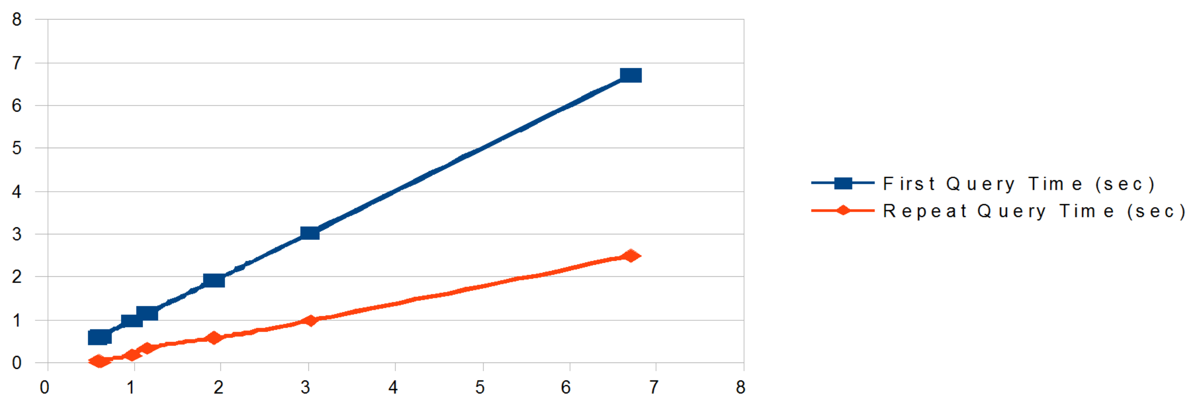
4.1. Federated Search

To reach Big Data scale requires a step change in the scale of the XML database which is unlikely to be achieved through simple expansion of a single database instance. A large acute hospital in the NHS might cover a population of 1.5 million patients and a national database for England should cover up to 55 million patients.

Fortunately, the fact that each patient record is a discrete data set allows us to scale the database as a set of separate database instances, over which a federated search can be run. Such replication or clustering could be implemented at the database level, using whatever support the database offered for this type of scaling. For example, the eXist database has some support for clustering using the Apache ActiveMQ messaging system to marshal a master and slave databases.

Our objective has been to create a federated database using the components of cityEHR (namely Orbeon and eXist) 'out of the box' with no additional software components and no database-specific implementation.

Figure 4. Execution time for first and repeated queries from 1 to 20,000 patients.



Such an approach ensures that the solution can be deployed easily on any network of servers or virtual machines.

So rather than use database-level replication or clustering facilities, we replicated the same database on five separate network servers. For the purposes of this test we used a 1000 patient database on dual core servers with 4Gb memory, using 1024Mb of Java heap for eXist.

Before testing the federated search, we ran the same test query on each database separately with the results shown in Table 3, "Execution time for database nodes of 1000 patients.". Each individual query returned 499 hits. Node 1 was running on the same server as the cityEHR (Tomcat) instance, whereas nodes 2 to 5 were running on network servers.

Table 3. Execution time for database nodes of 1000 patients.

Network Node	Node 1	Node 2	Node 3	Node 4	Node 5
Query Time (sec)	0.24	0.68	0.58	0.58	0.53

4.2. Using XML Pipelines

Our first implementation of a federated search process was implemented using an XML Pipeline in Orbeon, which iterates through the available database nodes, makes a submission to each node to run the XQuery and then aggregates the results set. The results for running on the five database nodes are shown in Table 4, "Results of federated search on 1 to 5 database nodes, using an XML pipeline.".

Table 4. Results of federated search on 1 to 5 database nodes, using an XML pipeline.

Nodes Queried	1 node	2 nodes	3 nodes	4 nodes	5 nodes
Query Time (sec)	6.2	12.54	18.65	24.25	30.22
Hits	499	998	1497	1996	2495

Were the submissions from the pipeline asynchronous, we would expect that the time taken to complete the federated query would be equivalent to the slowest query to any database instance plus the time required to aggregate the results from all queries. The results show that not only are the submissions synchronous (i.e. each waits for the previous iteration to complete before it submits) but the overhead of the aggregation in the XML pipeline is unacceptably high. We concluded that this is not a viable approach for implementation of the federated search.

4.3. Using Iteration Within XForms

Following the disappointing results using XML pipelines, we made a second implementation using iteration of the XQuery submissions from within XForms. The results of this implementation on five database nodes are shown in Table 5, "Results of federated search on 1 to 5 database nodes, using XForms.".

Table 5. Results of federated search on 1 to 5 database nodes, using XForms.

Nodes Queried	1 node	2 nodes	3 nodes	4 nodes	5 nodes
Query Time (sec)	0.28	0.49	0.82	0.94	1.15
Hits	499	998	1497	1996	2495

These results are more encouraging, showing that the overhead in the aggregation of results sets within XForms iterations is minimal; indeed the results seem to suggest performance that is better than a simple synchronous aggregation, since the combined query time for each node running as the single database instance is 2.61 seconds, whereas the federated search on all five nodes was only 1.15 seconds.

The XForms 1.1 standard does specify asynchronous submissions as the default, but in the version of Orbeon we used for these tests (3.9, community edition) only synchronous submissions are documented as being available. Assuming this to be the case, we would therefore expect a significant improvement in the federated search performance were asynchronous submissions used.

5. Conclusions

The results presented here show that the eXist database scales adequately as the number of patient records increases from 1 to 20,000. Our experiments have shown that to achieve satisfactory performance, it is vital to formulate each XQuery to ensure optimal use of the database indexes. This may explain why some other independent analyses have reached different conclusions regarding the scalability of eXist [12].

There is a considerable difference between the execution time for the first call of a query and the times for subsequent calls. This is due to caching in eXist, which would appear to be based on the pages touched by the query, rather than the full indexes. For this particular implementation in cityEHR, it is unlikely that an identical query will be run more than once in a single session and so the benefit of caching will not necessarily be seen. However, it is very likely that similar queries will be repeated in the same session, accessing the same indexes; it would therefore be more beneficial for cityEHR if the indexes themselves were cached. Therefore for our purposes we must look at the performance of the first query execution as our benchmark for performance.

Big Data queries on complex data sets can be expected to take some time to return results. It is not obvious where the threshold of acceptable performance lies for such queries, since results on larger or more complex data sets are more valuable and therefore longer execution times are likely to be more acceptable than times for queries driving a user interface (for example).

In the experiments described here, we have used the eXist database in its 'out of the box' configuration and have not attempted any database tuning beyond creating indexes, carefully formulating XQueries and setting the Java heap above the minimum required to execute the queries. Performance does seem to degrade between 10,000 and 20,000 patients and this is therefore an area for future investigation which may require more sophisticated tuning or knowledge of eXist to address. Assuming we can avoid further degradation, then we would project that a search on a 100,000 patient would complete in under 35 seconds. Limitation of time and resources have so far prevented us from implementing a database of this size, which is an obvious next step for our research.

To implement a federated search using databases of 100,000 patients (100k nodes) would require 10 nodes for one million patients and 550 nodes for 55 million. Using the benchmark of 35 seconds for a 100,000 patient database node and the results of our tests on federated search, we project a search time of just under 6 minutes for 1 million patients, one hour for 10 million and about 5.5 hours for 55 million. Total database sizes (aggregate of the federated databases) for 1, 10 and 55 million would be approximately 1.5Tb, 15Tb and 85Tb.

In conclusion, we can say that although the projected search times on an 85Tb database of 55 million records seem slow, they are based on a fairly unsophisticated approach to implementation. A comparable implementation of the Secondary Uses Service (SUS) [13], to hold data on 55 million patients in the NHS in England, was originally designed to use clustered relational database technology and hardware accelerators to hold far less structured data than we propose in this paper. Hence we conclude that the results obtained so far show enough promise to justify an extension to the next order of magnitude and we will report those results at a future date.

Bibliography

[1] *Big data and clinicians: a review on the state of the science.*. W. Wang and E. Krishnan. 2014. JMIR medical informatics, 2(1)..
http://www.ncbi.nlm.nih.gov/pmc/articles/4288113/

[2] *The meaningful use of big data: four perspectives--four challenges.*. C. Bizer, P. Boncz, M.L. Brodie, and O. Erling. 2012. ACM SIGMOD Record, 40(4), pp.56-60.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.446.6604&rep=rep1&type=pdf

[3] *BigBench: towards an industry standard benchmark for big data analytics.*. A. Ghazal, T. Rabl, M. V, F. Raab, M. Poess, A. Crolotte, and H.A. Jacobsen. 2013, June.. n Proceedings of the 2013 ACM SIGMOD international conference on Management of data (pp. 1197-1208)..
http://www.msrg.org/publications/pdf_files/2013/Ghazal13-BigBench:_Towards_an_Industry_Standa.pdf

[4] *HL7 Clinical Document Architecture, Release 2*. RH. Dolin, L. Alschuler, S. Boyer, and . 2006. J Am Med Inform Assoc. 2006;13(1):30-9. .
http://ssr-anapath.googlecode.com/files/CDAr2.pdf

[5] *Ontology-driven development of a clinical research information system.*. JJ. Chelsom, I. Pande, R. Summers, and I. Gaywood. 2011. 24th International Symposium on Computer-Based Medical Systems, Bristol. June 27-June 30.
http://openhealthinformatics.org/wp-content/uploads/2014/11/2011-06-CMBS.pdf

[6] *eXist: An open source native XML database.*. W. Meier. 2002. In Web, Web-Services, and Database Systems (pp. 169-183). Springer Berlin Heidelberg..
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.4808&rep=rep1&type=pdf

[7] *Are Server-Side Implementations the Future of XForms?*. E. Bruchez. 2005. XTech 2005.

[8] *Experiences with Mirth: an open source health care integration engine.*. G. Bortis. 2008. In Proceedings of the 30th international conference on Software engineering (pp. 649-652). ACM..
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.212.5384&rep=rep1&type=pdf

[9] *XML data warehousing for browser-based electronic health records.*. Dave Nurse and John Chelsom. 2000. Proceedings of XML Europe, 2000. IDE Alliance..
https://www.infoloom.com/media/gcaconfs/WEB/paris2000/S32-03.HTM

[10] *Visually exploring and querying XML with BaseX.*. C. Grün, A. Holupirek, and M.H. Scholl. 2007. .
https://kops.uni-konstanz.de/bitstream/handle/123456789/3007/visually_scholl.pdf?sequence=1

[11] *Berkeley DB XML: An Embedded XML Database.*. Paul Ford. 2003. O'Reilly, xml.com.
http://www.xml.com/pub/a/2003/05/07/bdb.html

[12] *eXist-db: Not Ready for High Scale.*. Robert Elwell. 2014. .
http://robertelwell.info/blog/exist-db-not-ready-for-high-scale

[13] *Electronic health records should support clinical research.*. J. Powell and I. Buchan. 2005. Journal of Medical Internet Research, 7(1), p.e4..
http://www.jmir.org/2005/1/e4/

Best Practice for DSDL-based Validation

Soroush Saadatfar

ADAPT Centre

`<Soroush.Saadatfar@ul.ie>`

David Filip

ADAPT Centre

`<David.Filip@adaptcentre.ie>`

Abstract

This paper proposes a best practice guide to apply Document Schema Definition Languages (DSDL) for validation of an arbitrary industry vocabulary. The research is based mainly on a practical case study of creating such an optimized set of DSDL validation artefacts for XLIFF 2, a complex industry vocabulary. Available schema languages have advanced functionality, enhanced expressivity and can be used in concert if needed. This advantage, on the other hand, makes the creation of a stable and robust set of validation artefacts hard, because there would usually be more than one way to describe the same Functional Dependencies or Integrity Constraints and various validation tasks can be solved by more than one schema language.

Keywords: DSDL, validation, expressivity, progressive validation, constraints, functional dependencies, XLIFF, Schematron, NVDL

1. Introduction

Validation is a key component of processing vocabularies based on XML (eXtensible Markup Language) [1]. This nontrivial task has been approached by a number of various initiatives according to the needs of specific target data models. Initially, DTD (Document Type Definition) [2] was widely used to define structure of XML documents and usually combined with programmatic approaches to validate the constraints that were not expressible in DTD. Several schema languages followed since DTD to enhance the expressivity for different XML constraints, however, the programmatic approach to tackle advanced contsraints validation had not been fully superseded. Although, the first validation technique has advantage of expressivity and trancparency being standardized.

Our focus in this paper will remain on non-programmatic, standards driven, trasparent approaches based on machine readable implementation independent artefacts. We aim to illustrate the potential DSDL methods and schema languages have to replace ad hoc programmatic validation approaches based on our experience with XLIFF (XML Localization Interchange File Format) [3], an OASIS standard which has been widely adopted in the localisation industry since its inception.

XLIFF has a multimodal structure, comrising a core namespace and several module namespaces; a complex data model designed to fulfill current and future needs of the industry. It is intended to complete a round-trip in the localisation workflow and to be dynamically *Modified* and/or *Enriched* by different *Agents* who manipulate the XLIFF data in accordance with their specialized or competing functionality. This XML vocabulary does not follow the normal XML behaviour when it comes to usage of `ID` attributes and it defines a number of scopes for NMTOKEN [4] keys (that are called IDs in XLIFF context) instead of the usual XML convention where the `ID` attributes are required to be unique throughout the document. The task of internal fragment referencing therefore cannot be implemented using native XML `IDREF` attributes either. The standard specifies several levels of date driven structures for NMTOKEN keys as well as introducing the *XLIFF Fragment Identification Mechanism* to replace the XNL `ID` and `IDREF` concepts respectively. These approaches have been chosen with needs of the industry in mind and will be discussed in detail in the following section. Also some other relational dependencies that XLIFF sets, have a complex nature and logic and some of these we will also be covered in the next section of this paper. As XLIFF files are meant to circulate in arbitrary workflows, it is vital - for purposes of lossless data exchange - that all workflow token instances conform to the XLIFF Specification completely

doi:10.14337/XMLLondon16.Saadatfar01

and do not violate any of its advanced constraints. Therefore comprehensive validation should be applied after any modification. Programmatic validation, mainly applied to its earlier version, XLIFF 1.2, led to misinterpretations of the standard in many cases and developers never implemented the full list of constraints specified by XLIFF 1.2. These issues motivated the XLIFF TC and the authors of this paper to find an exhaustive solution for validation of XLIFF 2.x (XLIFF 2.0 and its backwards compatible successors) to provide a unified and transparent platform for validating XLIFF instances against the full specification. Research revealed that the DSDL (Document Schema Definition Languages) [5] framework is capable of providing such a solution. This attempt succeeded and brought Advanced Validation Techniques for XLIFF 2 [6] to be part of the standard, starting from the 2.1 version, scheduled to release in 2016. In the following sections, we will try to generalize this work to provide mapping for XML constraints and appropriate DSDL method to use. We believe that transparent and standardized validation is a prerequisite for achieving interoperability in workflows and that the DSDL framework provides enough expressivity for producing machine readable validation artefacts for arbitrary XML industry vocabularies and data models.

2. Analysis of the XML Data Model

XML has a very simple, but at the same time universal and generalized nature as a data model: (a) it is composed of only two objects: XML nodes and their values (b) data model expressed through value and structure of nodes (c) it defines a minimum set of rules in terms of syntax. These properties enable XML to deliver its main task- extensibility. Each XML-based vocabulary represents the target data model by specifying valid scenarios of structure and values of declared nodes.

In this section, we will discuss constraints of XLIFF 2 from general XML point of view. We will first have a brief look at the structure and purpose of XLIFF. We review the literature for available definitions and notations of XML constraints in the next step and then apply them to XLIFF. This section will also include *Processing Requirements* that XLIFF specifies for different types of users (*Agents*) as they modify XLIFF instances. This type of constraints require progressive (dynamic) validation as they are based on comparison of files before and after Agents perform changes.

2.1. XLIFF Structure

The main purpose of XLIFF is to store and exchange localizable data. Typically, a localisation workflow contains processes like extraction, segmentation (process of breaking down the text into the smallest possibly translatable linguistic portions, usually sentences), metadata enrichment (like entering suggestions based on previous similar content, marking up terminology etc.), translation (editing of target content portions corresponding to source content portions) and merging the payload (the translated content) back to the original format. An XLIFF document can facilitate in its progressing instances all the aforementioned tasks. The extracted segmented content will be placed in the `source` element children of `segment`. The translated content is kep aligned in `target` siblings of the `source` elements; it can be added later in the process. In other words, the flow of original text transforms into sequence of `segment` elements within a `unit` element, the logical container of translatable data. Parts of the content which are not meant to be translated can be stored in the `ignorable` siblings of the `segment` elements. `unit` elements can be optionally structured using a `group` parent (ancestor) recursively. Finally, a `file` elements will wrap the possibly recursive structure of `group` and `unit` elements. An XLIFF instance (an XML document with the root `xliff`) can have one or more `file` elements. In order to preserve metadata and markup within text (e.g. the HTML `<b>` tag), XLIFF has 8 *inline elements* , (e.g. `pc` element in "Listing 1" for well formed paired codes) some recursive, which may appear along with text in `source/target` pairs. The native codes may be stored in `originalData` elements, which are then referenced from inline elements. To avoid confusion, we present a simplified notion of XLIFF and skip many other structural parts. "Listing 1" shows a sample XLIFF file.

Listing 1 - Sample XLIFF instance

```
<xliff version="2.0" srcLang="en" trgLang="fr">
  <file id="f1">
    <unit id="u1">
      <originalData>
        <data id="d1">&lt;b&gt;</data>
        <data id="d2">&lt;/b&gt;</data>
      </originalData>
      <segment>
        <source>
          Some <pc id="pc1" dataRefStart="d1"
          dataRefEnd="d2">Important</pc>text.
        </source>
        <target>
          Un texte <pc id="pc1" dataRefStart="d1"
          dataRefEnd="d2">important</pc> de.
        </target>
      </segment>
      <ignorable>
        <source>Non-translatable text</source>
      </ignorable>
      <segment>
        <source>Second sentence.</source>
        <target>Deuxième phrase</target>
      </segment>
    </unit>
  </file>
</xliff>
```

2.2. XML Constraints

The structural relation of XML nodes is the first step of shaping the *tree* of the targeted data model. All other constraints can be specified only afterwards. Generally, studies for defining XML constraint have proposed keys [7], foreign keys [8] and functional dependencies [9], [10]. These papers represent each type of constraint through mathematical expressions which is out of scope of this paper as our goal is to match every category with appropriate schema language in practice. Therefore we will consider general types as well as some of their special cases.

2.2.1. Keys and foreign keys

The concept of `ID` and `IDREF`, introduced by DTD, cover the category of keys and foreign keys respectively. However, these attributes implement only special cases of these types, *absolute keys/foreign keys* , setting the scope of keys to the root element, i.e. absolute path. In XLIFF, for instance, only `file` elements define absolute keys whilst keys are specified at 14 points of XLIFF core elements. Relative keys can be of various complexity depending on steps of relativity they designate (starting from zero steps, i.e. absolute keys). For example for `unit` elements, with only one step of relativity (from the root element), scope of key uniqueness is the parent `file` element. Number of steps for one key can be a variable if the corresponding element is allowed at different places of the XML tree, like `data` elements with unique keys in the scope of `originalData`, where the latter element might occur at several levels of the hierarchy. These variables, however, have a minimum value stating how close the element can be to the tree root, in the case of `data` the minimum number of steps is 3. In a more complicated scenario keys might be shared among nodes of distinct types and distinct parents. Keys for all inline elements, some of which can be recursive, are set at the `unit` level, but only for those appearing in the source text whereas elements of the target must duplicate their corresponding key in source. This type of content, where an element can have a mix of text and other elements, is usually being dropped in attempts of generalizing Keys for XML as an assumption [11] and therefore not well researched.

XLIFF, when it comes to foreign keys, has all of its referencing attributes relative (e.g. `dataRefStart` attribute in "Listing 1"). Even though cross-referencing is allowed only at the unit level, the standard defines format of IRIs pointing to XLIFF documents through Fragment Identification. This mechanism allows one to specify a unified path to any node with identifier. Therefore the notion `<pc id="1" dataRefStart="#/f=f1/u=u1/d=d1" ...` is an alternative valid value which specifies the referenced node by an absolute path instead of the shorter form used in Listing 1, where identifier is given relatively (within the enclosing unit).

2.2.2. Functional Dependencies

Attempts of generalizing this nontrivial category for XML has been limited so far and define only some variations of Functional Dependencies (*FD*) [12]. A significant progress has been made though, by applying mappings of paths in XML document to relational nodes. Basically, FDs specify functional relations of XML nodes and play an important role in data models being the most complex of XML Integrity Constraints. Co-occurrence constraints and value restrictions are some usual variations of FDs. XLIFF has various FDs, some of which we review in this paper. The `target` element has an optional `xml:lang` attribute, but when present must match `trgLang` attribute of the root element. The latter attribute, on the other hand, is initially optional (at early stages of the localization process), but must be present when the document contains `target` elements. Another interesting and complex FD in XLIFF relates to the optional `order` attribute of `target`. As was mentioned earlier, the sequence of `segment` and `ignorable` elements specifies the flow of text at the unit level, but sentences of a translated paragraph may have different order than in the source language. In this case `target` elements must specify their actual "deviating" position respective to the default sequence. This value then points to the corresponding `source` element for ID uniqueness constraints over inline elements "Listing 2" and "Listing 3" show a paragraph with three sentences in different order, represented in HTML and XLIFF respectively.

Listing 2 - Paragraph with disordered sentences after translation

```
<p lang='en'>Sentence A. Sentence B. Sentence C.</p>
<p lang='fr'>Phrase B. Phrase C. Phrase A.</p>
```

Listing 3 - Usage of `order` attribute in XLIFF

```
<unit id="1">
 <segment id="1">
  <source>Sentence A.</source>
  <target order="5">Phrase A.</target>
 </segment>
 <ignorable>
  <source> </source>
 </ignorable>
 <segment id="2">
  <source>Sentence B.</source>
  <target order="1">Phrase B.</target>
 </segment>
 <ignorable>
  <source> </source>
 </ignorable>
 <segment id="3">
  <source>Sentence C.</source>
  <target order="3">Phrase C.</target>
 </segment>
</unit>
```

The XLIFF file in Listing 3 is valid, in terms of `order` constraints, as (a) values are between 1 and 5 (segments and ignorables combined) (b) each `target` element occupying a different position than its natural explicitly declares its order.

2.2.3. Data Types

Constraints of this category apply various rules on values that can be assigned to XML nodes. The concept of Simple and Complex data types was introduced by W3C XML Schema [13] and provides a solid library that enables one to build custom data types through manipulating simple data types and specifying restrictions for them. A number of libraries have been developed after that to target specific needs. Allowed values can be defined in many different ways including set of fixed values, default values, forbidden values/characters, mathematical restrictions for numeral values, specific or user-defined format etc. We will return to this topic in the following sections.

2.2.4. Progressive Constraints

Some data models are designed to perform in different stages of their life cycle and therefore might need to be validated against different set of constraints according to the stage they are at. *Initially* optional `trgLang` attribute of XLIFF, which was mentioned earlier, serves as a good example for this case. But some advanced constraints, like XLIFF Processing Requirements, focus on the applied modifications in documents and perform validation based on comparison of data before and after changes were made. For example, if value of one attribute has been changed by the last user, value of some other nodes must be changed as well. Technically, this type can be considered as a cross-document functional dependency, but as XML vocabularies are being often used for exchange purposes, this might grow into a category on its own in the future. XLIFF classifies its users (Agents) based on the type of general task they carry out (e.g. Extracting, Enriching etc.) and assigns different sets of Processing Requirements to each group.

In the following section we will examine expressivity of popular XML schema languages against each of the aforementioned types.

3. Implementing XML Constraints

After specifying XML Integrity Constraints, we now will explore schema languages which can implement the constraint types in the previous section. W3C XML Schema is the schema language to define the structure of XML trees with the widest industry adoption. The DSDL framework, on the other hand, is a multipart ISO standard containing various languages for different validation tasks and broad domain of constraint types.

Schema languages often perform across constraint types and DSDL provides the framework for mapping and using them together. In the current section we aim to highlight *the task* each language can handle the best. Following such guidance will contribute to optimizing performance level of the validation process.

3.1. XML Schema

This schema language is useful for defining the XML tree nodes and their relations in the XML tree. XML Schema uses XML syntax and Data types [4], the second part of the language, introduces an advanced library of data types which is widely used and referenced by other schema languages. Users can apply different restrictions to values of elements or attributes. XML Schema supports only absolute Keys and foreign Keys using a limited implementation of XPath [14], a syntax for regular expressions in XML. The concept of key in XML Schema presumes that the attribute must always be present and thus cannot be applied to optional attributes. Finally, XML Schema cannot target Functional Dependencies of any level of complexity.

3.2. DSDL framework

Some parts of DSDL, like RelaxNG [15] and Schematron [16], were standalone projects initially that were subsequently standardized as part of the framework. For the goals of this paper, we only review 3 parts of the framework that together enable full expressivity for XML Integrity Constraints.

3.2.1. RelaxNG

This schema language describes structure and content of information items in an XML document through a tree grammar. The grammar-based validation RelaxNG offers is an easy and convenient approach, although it keeps the expressivity power of this language close to the level of XML Schema. RelaxNG has some basic built-in data types so other libraries (e.g. XML Schema Data types) should be used for advanced requirements in this type of constraints. Although it does not support any variations of Keys and foreign Keys, RelaxNG is able to cover some basic Functional Dependencies like co-occurrence constraints based on values or presence of XML nodes. Defining such constraints in RelaxNG, however, might wind up not pragmatic for vocabularies with a large number of nodes. such as XLIFF. For instance, the constraint on `trgLang` attribute in XLIFF, which was mentioned earlier, could, theoretically, be expressed in RelaxNG by defining two possible valid grammars, but this would unfortunately effectively double the volume of the schema. Overall, RelaxNG is a simple language to use compared to XML Schema. Its focus on targeting only one problem has made RelaxNG an efficient schema language [17] and therefore more and more industry vocabularies tend to pick this language for validation tasks. We developed an experimental RelaxNG schema for XLIFF 2, yet this was not adopted as part of the advanced validation feature for XLIFF 2.1 by the XLIFF TC due to a wide overlap with the XML Schema, which already was a normative part of the standard and needs to be kept for backwards compatibility reasons.

3.2.2. Schematron

The rule-based validation allows Schematron to catch Functional Dependencies violations. This schema language provides full support for both XPath regular expressions and functions which enables users to define comprehensive sets of XML paths to express an arbitrary Functional Dependency, key or foreign key. Each Schematron rule describes permitted relationships between document components by specifying a context, i.e. the XML node(s) where the current rule applies and then conducting the user-defined test. Because XPath integration provides a powerful mechanism for navigating through XML trees, Functional Dependencies often may be expressed in multiple ways. It is a convenient approach to first define the subject and the object of a Functional Dependency as well as whether the path, through which the subject *affects* the object, is relative or absolute. Consider the Functional Dependency for the `order` attribute, where the subject node is any `target` element whith explicitly specified order. The object then would be such a `target` element that occupies the natural position of the subject. The subject and object are related at the unit level (the common `unit` ancestor), therefore a relative regular expression needs to be applied. "Listing 4" illustrates implementation of this constraint using our convention.

Listing 4 - XLIFF Functional Dependency for the `order` attribute expressed in Schematron

```
<iso:rule context="xlf:target[@order]">
  <iso:let name="actual-pos" value="count
    (../preceding-sibling::xlf:segment|
    ../preceding-sibling::xlf:ignorable)+1"/>
  <iso:assert test="ancestor::xlf:unit//xlf:target
  [@order=$actual-pos])">
    Invalid use of order attribute.
  </iso:assert>
</iso:rule>
```

Schematron also introduces a phasing mechanism that can be used to group constraints in various phases so that rules are applied only when the relevant phase is *active* . Using this feature, alongside with `document()` function of XPath enables cross-document rules and *progressive validation* consequently. An XLIFF Processing Requirement, forbidding any changes in `skeleton` element (which stores the original data), is represented in Schematron in "Listing".

Listing 5 - XLIFF constraint on `skeleton` element expressed in Schematron

```
<iso:let name="original-xliff"
 value="document('before.xlf')"/>
  <iso:rule context="xlf:skeleton">
    <iso:assert test=
     "current()=$original-xliff//xlf:skeleton">
      Structure and content of skeleton element
      must not be changed.
  </iso:assert>
</iso:rule>
```

Schematron also offers some other useful features like variables (used in "Listing 5") and several tools to produce customized and informative error reports. The full adoption of XPath has made Schematron the most expressive schema language in the DSDL framework that is capable of handling the most complex Functional Dependencies, Keys and foreign Keys. Many of XLIFF constraints and Processing Requirements have been implemented in Schematron for the Advanced Validation feature to be avialable as of XLIFF 2.1.

3.2.3. NVDL

Namespace-based Validation Dispatching Language [18], NVDL, provides a schema language for selecting elements and attributes in specific namespaces within a document that are to be validated by a specified schema. NVDL is especially useful for muyltimodal XML vocabularies, such as XLIFF, that may contain different namespaces within a single document instance. NVDL can handle the task of mapping namespaces and assign appropriate schema artefacts for effcetive validation. The Advanced Validation feaure for XLIFF 2.1 and successors uses NVDL to compartmentalize the validation task for any potential XLIFF Document and run XML Schema and Schematron artefacts to validate static and dynamic usage of XLIFF Core and Module namespaces. Although Schematron rules can be embedded in both XML Schema and RelaxNG, it is generally advisable to use NVDL for this purpose, even in cases where the XML document declares only one namespace, as the former approach would require additional extraction and processing where NVDL is supported by various tools and libraries and provides simpler syntax for the task.

4. Conclusion

In this paper we reviewed generalized forms of XML Integrity Constraint types used to represent a data model in XML. We demonstrated, on the examples from the XLIFF industry vocabulary, that various types of Keys, foreign Keys and Functional Dependencies can be of different complexity depending on how advanced the required regular expressions are. We then explored a number of XML schema languages - focusing on the DSDL framework - in terms of their capacity to target different types of XML constraints and functional dependencies. The comparison revealed that Schematron, mainly due to its full adoption of XPath, can provide the highest expresivity for all types of constraints and fucntional dependencies among the tested schema languages. It is an industry proven best practice to validate the static structure of XML instances first, e.g. using RelaxNG or XML Schema, and to apply other advanced constraints or functional dependencies only afterwards, programmatically or by appropriate advanced schema languages, as paths towards the tested nodes must be established first and only then can be examined against any such advanced constraints. We also provided a convention to simplify and optimize defining Schematron rules through the concept of *subject/object* of Functional Dependencies.

Although the DSDL multipart standard (especially Schematron and NVDL) has resolved many issues in the XML validation domain, some of its aspects could be still improved. For instance, Schematron leaves implementation of some of its features, optional for processors, which significantly affects the functionality when using different processors or invoking Schematron rules from an NVDL schema. These features usually presented via attributes like `subject, role` would generally enhance the error reporting, if more widely adopted.

Applying the methods presented in this paper to other industry vocabularies than XLIFF is proposed as future work. Similar investigations of DSDL applications on various vocabularies would provide a valuable set of artefacts for further theoretical research and study of XML Integrity Constraints and Functional Dependencies on the basis of emerging industry needs.

Bibliography

[1] *T. Bray, J. Paoli and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C, Nov. 2008..*

[2] *T. Bray, J. Paoli and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 W3C, 1998..*

[3] *T. Comerford, D. Filip, R.M. Raya and Y. Savourel. XLIFF Version 2.0, OASIS Standard, OASIS, 2014.*

[4] *H. Thompson et al. XML Schema, Part 2: Datatypes. W3C Recommendation, Oct. 2004. .*

[5] *International Standards Organization, (2001). ISO/IEC JTC 1/SC 34, DSDL Part 0, Overview. ISO..*

[6] *S. Saadatfar and D. Filip, "Advanced Validation Techniques for XLIFF 2," Localisation Focus, vol. 14, no. 1, pp. 43–50, 2014..*

[7] *P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan, "Keys for XML," Computer Networks, vol. 39, no. 5, pp. 473–487, 2002..*

[8] *M. Arenas, W. Fan and L. Libkin, On verifying consistency of XML specifications. In PODS, 2002..*

[9] *M. Vincent, J. Liu, and C. Liu, "Strong functional dependencies and their application to normal forms in XML," TODS, vol. 29, pp. 445–462, 2004..*

[10] *A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In VLDB, 2003..*

[11] *M. Arenas and L. Libkin, "A Normal Form for XML Documents," ACM Transactions on Database Systems, vol. 29, no. 1, pp. 195–232, Mar. 2004..*

[12] *w. Fan. XML Constraints: Specificaion, Analysis and Aplications. In DEXA, 2005..*

[13] *H. Thompson et al. XML Schema. W3C Recommendation, Oct. 2004..*

[14] *J. Clark and S. DeRose. XML Path Language (XPath). W3C Recommendation, Sep. 2015..*

[15] *International Standards Organization, (2003). ISO/IEC 19757-2:2003(E), DSDL Part 2, Regular-grammar-based validation — RELAX NG. ISO..*

[16] *International Standards Organization, (2004a). ISO/IEC 19757-3, DSDL Part 3: Rule-Based Validation — Schematron. ISO..*

[17] *E. van der Vlist, RELAXNG. O'Reilly Media, Inc., 2003..*

[18] *International Standards Organization, (2004b). ISO/IEC 19757-4, DSDL Part 4: Namespace-based Validation Dispatching Language— NVDL. ISO..*

A journey from document to data

or : buy into one format, get two production-ready assets free

Andrew Sales

Andrew Sales Digital Publishing Limited

`<andrew@andrewsales.com>`

Abstract

XML is often treated as a neutral format from which to generate other outputs: HTML, JSON, other flavours of XML. In some cases, it can make sense to use it as a means to auto-generate other XML-based assets which themselves act on XML inputs, such as XSLT or Schematron schemas.

This paper will present a study of how XML and related technologies helped a publisher to streamline the production process for one of its products, enabling better consistency of data capture and an enhanced customer experience. It will describe how a legacy document-centric format was refined to allow publication processes to run more smoothly, and how an abstraction of the capturing format allowed other key assets in the workflow to be generated automatically, reducing development costs and delivering ahead of schedule.

Keywords: XSLT, Schematron, meta-programming

1. Background

This background is intended to help explain some of the design decisions made later.

The content relating to this paper was destined for a single online product, which was conceived to allow corporate lawyers to compare market activity, such as mergers and acquisitions. This valuable information is for the most part freely available in the public domain by virtue of regulatory or legal requirement, but value is added to the paid-for version by providing an editorial digest (some of the official documents can exceed 1,000 pages in length) of the key points of each transaction helping the lawyer to assess trends, for instance which countries or sectors work might be coming from and how much it might be worth.

The main idea was to enable lawyers to compare transactions of various types by differing criteria. For example, they might want to know how many companies incorporated in the UK banking and finance sector had gone public in the last six months, or the typical value of an acquisition in the mining sector.

For the first iteration of the product, the legacy editorial and publishing systems already in place had to be respected, because it was judged more important by the business to launch in some form than to delay and launch with something more tailored and functional. This meant that there had to be two main workarounds in the initial publishing workflow.

2. Legacy format

The first was the editorial format. The source material comes from HTML pages and PDFs, and analysts were used to working in a document format (in PTC Arbortext), so for speed and convenience this continued. Importantly, the editorial content included analysis as free text, and so did not neatly fall into a purely data-centric model. Some additions were made to the governing DTD, which had originally been designed as a generic digest format, and was used for other content types, in order to introduce some specific semantics for this domain: analysts could mark up e.g. countries, dates, company names, transaction type and industry sector to enable some comparisons - but the tagging of this information was not controlled (such as by Schematron) at all at source. On top of that, it was not directly embedded in the main text, but separated out in the metadata header. This loose association meant that the integrity of the information was at risk and there was scope for error.

Information could also appear "buried" in strings. For example, to express whether all resolutions had been passed at a company's AGM, there were c.50 variants on the significant value, such as:

```
<row>
<entry>All resolutions passed?</entry>
<entry>All resolutions proposed at the AGM were
passed apart from resolution xx...</entry>
</row>
```

where a yes/no value was sufficient.

3. Legacy system

The second workaround concerned the (post-editorial) publishing mechanism.

The editorial XML is converted to a common, generic publishing XML format, which is then further processed before HTML pages are generated for publication. All editorial XML follows this same route as a matter of policy. This meant that, while some front-end development was possible to enable some filtering on the added metadata mentioned above, there was limited scope for the more in-depth comparison useful to customers.

So a back-end application was then built to query the selected transaction types using XSLT to produce a spreadsheet with macros to help the user run comparisons. This was clunky, slow, extremely sensitive to any change in the editorial XML format, and hard to maintain.

4. A new hope

An opportunity to address some of the data capture and design issues presented itself when another part of the business created a similar product platform for their local market, this time in conjunction with a third-party vendor. Their platform stored the data to be compared on the back end in a (relational) database, with strict(-er) data-typing, and crucially it was free of the in-house production workflow constraints. Again as a quick win, the UK arm of the business was encouraged to adopt this platform also, and work to migrate the legacy data for upload to the new platform ensued.

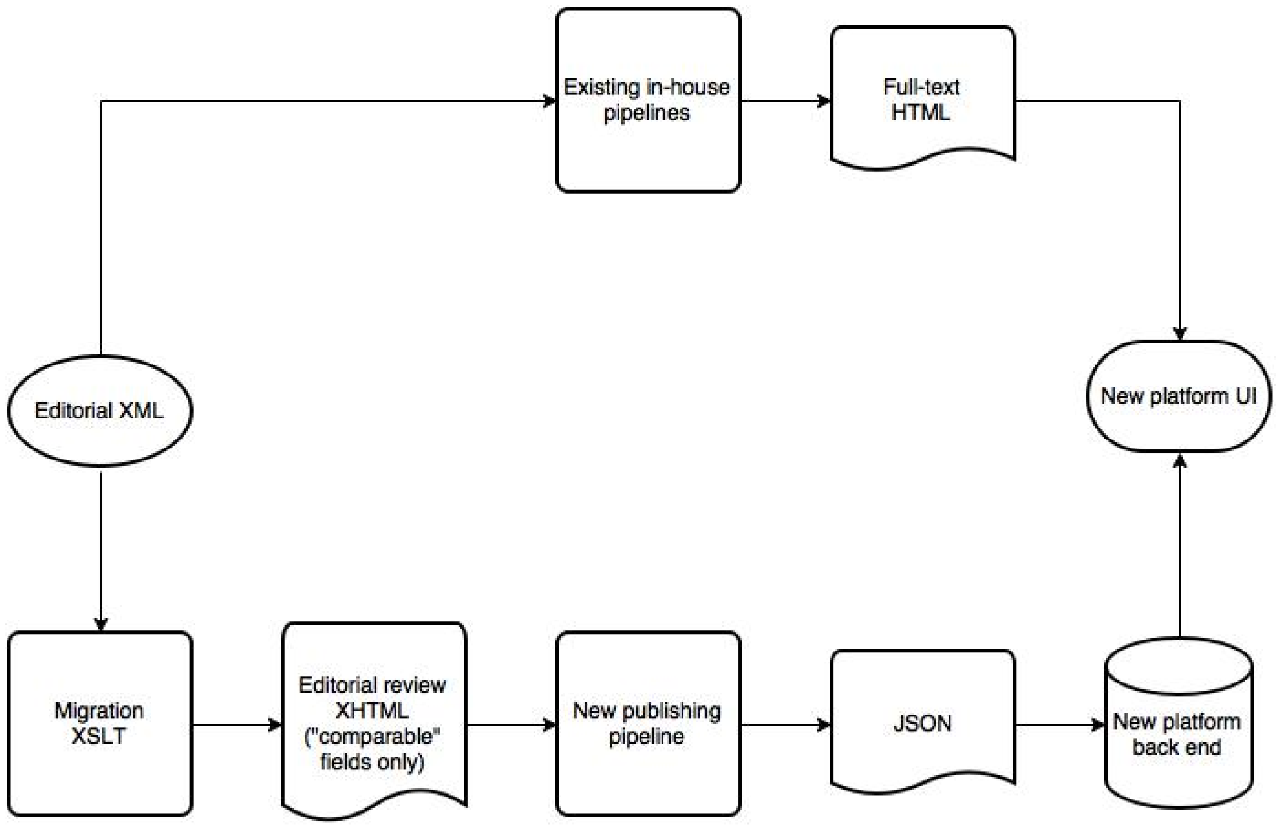
5. Migration

The rationale for migrating was threefold:

1. Although the dataset was relatively small (around 1,700 reasonably short documents), the time-and-money cost of using expert authors to re-key[1] it on the new platform as a data entry activity would be prohibitive.
2. Had it been simply transferred to the new platform, the information would have resided in the supplier's database only, raising continuity and IP issues. I therefore advocated retaining the canonical version ("editorial master") of the content in-house and generating the new platform format from it as needed.
3. There was a requirement to publish both the existing full-text view, also containing other, "non-comparable" fields, simultaneously and for this to be in sync with the "comparable" data.

[1] And re-keyed in this way it would have been: in most instances, an expert eye was needed to ensure the correct nugget of information was selected and if need be adjusted appropriately.

Figure 1. Migration workflow



So it was agreed that the migration would be semi-automatic. I wrote XSLT which extracted the relevant information, if present in the source, into two-column key-value pair XHTML tables for editorial review, with additional inline semantic elements to capture the typed data. XHTML was used because we needed a simple format that could be easily accommodated by the existing pipelines, with the secondary advantage that Arbortext was already set up to handle it. Rather than using the string value of the cells in the "key" column, each `row/@class` was used as an identifier, which editors would not normally change.

```
<tr class="announcementDate">
        <!--key-->
  <td>Date of substantive announcement</td>
        <!--value-->
  <td>
  <date day="1" month="Nov" year="2012"
    xmlns=”transaction-example”/>
  </td>
</tr>
```

Here `@class` identifies the field rather than expressing its datatype, so that the next step in the publishing pipeline can use this as a reliable hook independent of the field's name.

The datatype was constrained instead by Schematron rules applied to each "value" column. The rules could be shared across transaction types to some extent. Editors reviewed the extracted data in this format in Arbortext and only once it passed validation against the schema was it fit to publish.

It was a long haul and messy — but there were common fields across transaction types, so the lines of code to write decreased as each type was tackled (they were prioritised according to a tight release schedule, so could not be done all at once). About 40-50% of the fields could be migrated automatically, with the rest adjusted or filled in by hand. After this migration, the resulting XHTML really expressed how the data should be captured in the first place; or so it would be natural to assume.

6. De(v|ta)il

There are two wrinkles which prevent that from happening.

The first is that not all the information the analysts write fits neatly into a data-centric model: they are still authoring some discursive or other very variable text, such as a general analysis of the transaction. This content did not belong in the new platform's database on the back end, as it served no purpose to compare it across transactions. Second, it still needed to be published on the new platform, however, so the all-XML-shall-pass-through-this-pipeline still had to be used to generate the full-text version, which appeared elsewhere on the new platform, for continuity's and completeness' sake. And that pipeline consumes in-house formats most readily, so it still had to be used, again on grounds of convenience (cost, time etc).

The analysts who create the content had acknowledged independently that the way in which they captured the "comparable" data fields for the new platform would have to change[1]. With the existing editorial format having to be retained, as outlined above, the choice I made was to modify it slightly by introducing new elements inline and constraining their allowed value and occurrence using Schematron at authoring-time. To experienced XML people, this kind of validation is the *sine qua non* of data quality and integrity, but it was a significant step in this workflow, as the only validation to date had been by a (quite permissive) DTD. Furthermore, the CMS which stored the content carried out no validation on receipt. So it had been perfectly possible to publish invalid content, something less desirable when the content did not only have to look right.

So it was decided that for new content, this new editorial format would be used and enforced by a Schematron schema, making it an editorial responsibility to ensure structurally correct content.

With the XHTML to JSON pipeline already in place at the publication end of the workflow, I also needed to produce XSLT which would transform the new editorial format to XHTML. (Those for the old editorial format could have been re-worked, but new transaction types had been added in the meantime along with changes and additions to the fields to be included in the new format, so this option was less attractive.)

7. A level of indirection

The experience of the migration was like a dry run for the new editorial format. Two assets would need creating and managing over time:

1. editorial Schematron (this time applied at the content-writing stage);
2. XSLT to transform editorial format to XHTML.

Each of these would need creating for around a dozen content types, with others slated for addition.

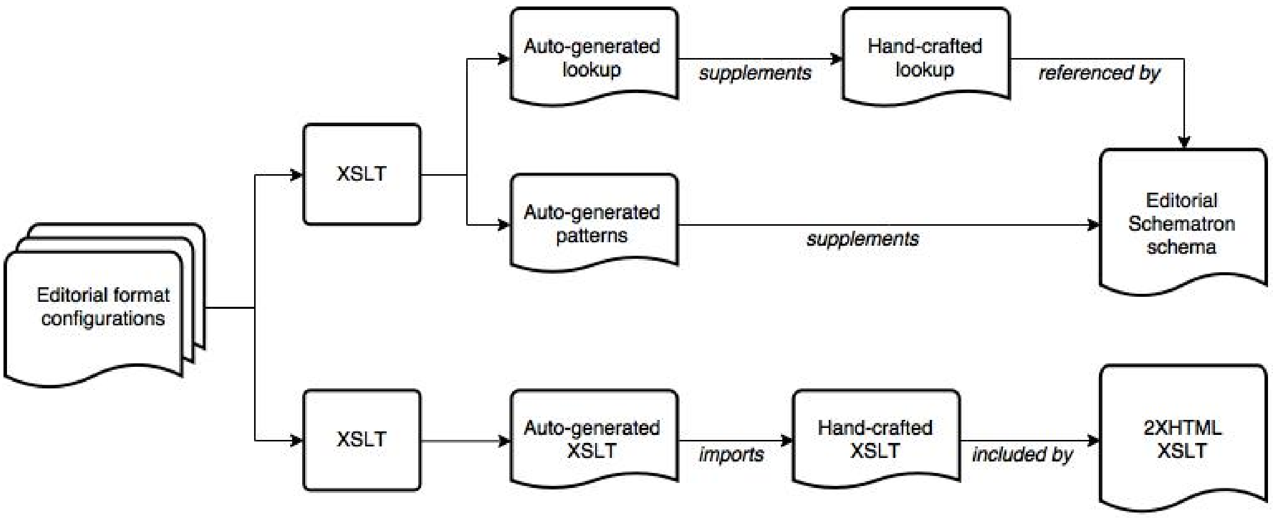
Faced with a sizeable time estimate to create these, based on the preceding migration work, I considered:

> What if the fielded data and their respective types could themselves be abstracted, and this expressed as XML, from which to generate the additional assets?

So I devised a general format that could be used for all transaction types, expressed by a RELAX NG schema plus Schematron schema. From this, it is possible to write transforms to generate both the editorial Schematron and the XSLT for creating XHTML from the editorial XML.

[1] They found that introducing consistency for fields which had a fixed set of values helped in their task.

Figure 2. Configuration format and outputs



7.1. The format

```
start =
  element transaction {
    attribute type { transaction-types },
    element group {
        attribute label { text },
        element field {
          attribute id { xsd:ID },
          attribute type { data-types },
          attribute occurs
            { occurrence-indicators }?,
          attribute controlledList{ text }?,
          attribute isCompanyName
            { string "true" | "false" }?
        }+
     }+
  }

data-types = string
"date" | "country" |
"currency" | "integer" |
"string" | "boolean" |
"percentage" | "controlled-list"

occurrence-indicators = string
"?" |
"+" |
"*" |
"1"
```

The root element is refined for each transaction by its `@type`, which is e.g. a merger and acquisition. Sections within the data are modelled by `<group>`s, which have a `@label`.

The ID of each field corresponds to the `row/@class` in the editorial format.

`field/@controlledList` refers to a controlled list contained in a separate lookup, when `field/@type='controlled-list'` is specified. `field/@type='boolean'` is strictly speaking a controlled list with two allowed values, "Yes" or "No" – but since these are very common, I used this as shorthand instead.

`@isCompanyName` is a flag to indicate that any trailing text in parentheses (optionally specifying the company's role in a transaction, such as the acquiring party in a takeover) should be removed.[1]

Note that string+ or controlled-list+ means multiple `<p>`s within the XHTML `<td>`; a percentage or currency combined with '+' implies a range of two values.

The Schematron schema contains a few sanity checks that e.g. `@type='boolean'` could only occur once for a given field.

7.2. XSLT to generate XSLT

The transform reads in the configurations for all the transaction types and outputs a single stylesheet (purely for ease of distribution), using one mode per transaction type. The skeleton structure of the XHTML output is constructed from each configuration (titled sections containing tables), with each KVP-containing table row processed by the relevant datatype-specific template.

[1] It isn't a distinct datatype of its own because it was a late addition and the exact requirement was emergent; it would perhaps more naturally be `type='companyName'` instead.

For an example, see Example 2, “Auto-generated XSLT for value(s) from a controlled list”.

These comprise the only hand-crafted portion of the editorial XML to XHTML transform.

7.3. XSLT to generate Schematron

The Schematron schema contains generic patterns which can be applied to several fields, mainly to validate their datatype and cardinality, as well as a small handful of manually-created rules of either global or very specific applicability.

For most fields, their IDs (`row/@class`) are stored in a variable according to datatype, e.g.

```
<let name="currency-classes"
value="$lookup-classes
[@type='currency-classes']/value"/>
```

and each generic pattern then references the class values:

```
<pattern id="currency-only-cell">
<rule context="row
  [@class = $currency-classes]/entry[2]">
  <assert test="*[1]
  [self::range[value] or self::value]
  and count(value|range[value]) = 1">
  "<value-of select="../entry[1]"/>"
  must contain a single currency value or range
  of values only</assert>
        </rule>
      </pattern>
```

The lookup (`$lookup-classes`) referenced by each let here is automatically generated from the configurations, therefore only the configuration files need to be maintained for the appropriate constraint to be applied to an added or amended field.

In the case of the controlled list datatype, an abstract pattern was used, with the class(es) and allowed value(s) being passed in:

```
<pattern id="controlled-list-cell-para"
  abstract="true">
<rule context="row[@class
  = $class]/entry[2]/para">
<assert test=". = $controlled-list-values">
  "<value-of select="../../entry[1]"/>"
  must contain one of:
  <value-of
    select="string-join
    ($controlled-list-values, ', ')"/>;
    got "<value-of select="."/>"</assert>
</rule>
</pattern>
```

```
<pattern id="controlled-list_ABC-XYZ"
  is-a="controlled-list-cell-para">
  <param name="class"
    value="'marketForTheIssuersShares'
    and $transaction-type = ('ABC','XYZ')"/>
  <param name="controlled-list-values"
    value="$lookup-values
    [@type='main-aim-other']/value"/>
</pattern>
```

This allowed the transaction type to be added as an optional dimension to the lookup, for instance where fields with the same ID in different transaction types used different controlled lists.[1]

Each of these concrete patterns was generated automatically also. The controlled list lookup (referred to in `$lookup-values` above) is a manually-curated XML file.

7.4. End-to-end example

To show how this works in practice, let us consider take an example of a field to be filled with one or more strings from a fixed list of values.

Example 1. Configuration for field to contain value(s) from a controlled list

```
<field id="favouriteXMLTechnology"
  type="controlled-list"
  controlledList="XML-technologies" occurs="+"/>
```

[1] I could have avoided this with IDs unique across all transaction types, but wanted to make the configurations as easy to maintain as possible.

In the editorial format this might appear as:

```
<row class='favouriteXMLTechnology'>
  <entry>Favourite XML technology/-ies</entry>
  <entry>
    <para>XSLT</para>
    <para>XQuery</para>
    <para>XForms</para>
  </entry>
</row>
```

where the hand-crafted lookup for the allowed values is:

```
<values type='XML-technologies'>
  <value>XForms</value>
  <value>XPath</value>
  <value>XProc</value>
  <value>XQuery</value>
  <value>XSLT</value>
</values>
```

The auto-generated XSLT for the field is then:

Example 2. Auto-generated XSLT for value(s) from a controlled list

```
<xsl:template
  match="row[@class='favouriteXMLTechnology']"
  mode="example">
  <xsl:call-template
    name="controlled-list-multi"/>
</xsl:template>
```

And for reference, the relevant hand-crafted templates:

```
<xsl:template name="controlled-list-multi">
  <xsl:variable name="para" as="item()+">
    <xsl:apply-templates select="entry[2]/para"
                         mode="html"/>
  </xsl:variable>

  <xsl:apply-templates select="." mode="datatype">
    <xsl:with-param name="value" select="$para"/>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="row" mode="datatype">
  <xsl:param name="value" as="item()*"/>
  <xsl:apply-templates select="entry[1]"/>
  <xsl:element name="td"
    namespace="http://www.w3.org/1999/xhtml">
    <xsl:sequence select="$value"/>
  </xsl:element>
</xsl:template>
```

Here is the accompanying Schematron:

Example 3. Auto-generated Schematron for value(s) from a controlled list

```
<pattern id="XML-technologies-controlled-list"
         is-a="controlled-list-cell-para">
  <param name="class"
         value="('favouriteXMLTechnology')"/>
  <param name="controlled-list-values"
    value="$lookup-values
            [@type='XML-technologies']/value"/>
</pattern>
```

And finally, the XHTML output:

Example 4. XHTML output for value(s) from a controlled list

```
<tr class='favouriteXMLTechnology'>
  <td>Favourite XML technology/-ies</td>
  <td>
    <p>XSLT</p>
    <p>XQuery</p>
    <p>XForms</p>
  </td>
</tr>
```

8. Conclusion

It was moving away from what had been regarded as a document format to a more data-centric one with the constraints provided by strict validation, that enabled:

1. more efficient, better quality and more consistent data capture, including rigour in the use of allowed string values through controlled lists;
2. fewer downstream processing errors;
3. the derivation of a configuration format for the Schematron schema and XSLT that were needed, which in turn allowed these assets to be managed and maintained much more easily and efficiently.

The first two result naturally enough from following best practice. Real-world circumstances often mean compromises in this respect. It was the latter point that made the difference here, and the approach arguably would not have been accepted without the initial data migration activity. In terms of lines of code written, the XSLT reduced from c.5,000 from the initial migration to c.600 for the auto-generation work. The work had been estimated to take from 8-12 weeks; the actual time taken was reduced to under four weeks.

Structure-Aware Search of UK Legislation

John Sheridan

The National Archives

Jim Mangiafico

`<jim@mangiafico.com>`

Abstract

We have created an application that enables searching the UK statute book with reference to the structure of legislative documents. Users can target individual legislative provisions and receive direct links to the matching document components. For example, users can search for chapters that contain certain phrases or for sections that have certain headings. In this paper, we describe the XML format used to represent UK legislation and the technologies used to fulfill the queries. We have developed a simple, domain-specific query language to express user requests, and we use MarkLogic to store and index the documents. We parse user requests in XQuery and translate them into native MarkLogic query directives, but because some of the searches we support cannot fully be expressed as MarkLogic queries, we sometimes construct over-inclusive queries and filter the results at the application level. We are currently preparing the application for public release at research.legislation.gov.uk.

1. Introduction

The National Archives operates legislation.gov.uk, the UK's official legislation website, underpinned by the government's legislation database. There are an estimated 50 million words in the statute book, with 100,000 words added or changed every month. There has never been a more relevant time for research into the architecture and content of law. However researchers typically lack the raw data, the tools, and the methods to undertake research across the whole statute book.

As part of the "Big Data for Law" project[1], funded by the Arts and Humanities Research Council (AHRC), The National Archives set out to develop a Legislation Data Research Infrastructure, to enable richer and deeper research into legislation. We found that many of those with interesting research questions to ask lacked the technical knowledge and skills to query the XML data that we hold. To mediate between those people with interesting questions, but who typically lack knowledge of XQuery [1], and our rich XML legislation data, we developed a powerful but easy to use application for searching legislation documents based on their structure. Of particular interest to us was the ability to frame searches to find examples of commonly occurring legal design patterns in legislation. These typically span several provisions of an act or statutory instrument. Finding such patterns proved beyond traditional full text search capabilities, leading us to develop this structure aware search application.

2. Keyword and Proximity Searches

Our application is, first of all, a complete search engine that can fulfill simple keyword searches. A user can enter any word or phrase and receive a page of results listing documents that contain the given word or phrase, including snippets of text with the search terms highlighted. Keyword searches can contain wildcards; they can be either case sensitive or case insensitive; and they can be "stemmed" or "unstemmed". Stemmed searches match all words derived from the same word stem. For example, a stemmed search for the word "is" will match not only "is" but also "am", "are", "be", "been", "being", etc. Furthermore, multiple keyword searches can be combined in a single query with the boolean operators AND, OR and NOT. The search for "`appeal AND Secretary of State`" matches documents containing both the word "appeal" and the exact phrase "Secretary of State". We also support a common alternative syntax for the boolean operators: "`&&`", "`||`" and "`!`".

[1] "Big Data for Law" project http://www.legislation.gov.uk/projects/big-data-for-law

doi:10.14337/XMLLondon16.Sheridan01

Users can also search for documents containing words or phrases occurring within a specified proximity to one another. Although we modeled our query syntax loosely on that of Apache Lucene[2], for simplicity and consistency we chose a syntax for proximity searches in which the distance in number of words precedes a parenthetical list of terms. For example, the query "`20(appeal, Secretary of State)`" matches documents in which the word "appeal" appears not more than 20 words before the phrase "Secretary of State". Proximity searches expressed with rounded brackets match only those documents in which the given terms appear in the given order. The use of square brackets specifics an unordered search, matching documents containing the given words or phrases in any order, so long as they appear within the given proximity to one another.

3. Element Searches

Because our dataset comprises highly structured documents, represented in XML and stored in a database designed for XML documents, we wanted to allow users to target their searches to the structural elements of documents. And because most elements in legal documents contain free-form text, we wanted element-specific queries to resemble simple, familiar keyword searches to the extent possible. Therefore, we allow users to limit any of the keyword searches described above to most common structural elements by enclosing the keyword parameters in parentheses placed after the name of the target element. For example, the query "`title(European Union, trade)`" matches documents whose titles contains both "European Union" and "trade". The query "`chapter(appeal, Secretary of State)`" matches all *chapters* that contain those terms. The first query, because it targets a document-level element, that is, one that appears only once in a document, returns a list of documents. The second query returns a list of chapters, grouped by document.

Element queries share other characteristics of simple keyword searches. They can contain the combination of keyword limitations with boolean operators. The query "`chapter(apple && !banana)`" matches chapters containing the word "apple" but not the word "banana". And element searches can be combined with other element searches using the same operators. For instance, the query "`chapter(apple, banana) && chapter(pear, orange)`" matches chapters containing the first two words and chapters containing the second two, in documents that contain chapters of each sort. Also, like the proximity searches, element searches can be either ordered or unordered, expressed with rounded or square brackets respectively. We currently support element searches for all of the major structural components of legislative documents: part, chapter, heading, para, subpara, and schedule. We also support queries directed at many document-level elements such as title, longtitle, intro, headnote, and subject. And we support a variety of other elements, such as footnote, annotation, signature, and department.

Not only can element searches be combined with one another; they can also be *nested* within one another. For example, to search for chapters with the word "security" in their heading, the user can submit "`chapter(heading(security))`". Element queries can be nested to an arbitrary depth, and nested queries can be combined in all of the ways described above. To search for chapters with the word "security" in their heading and the word "shareholder" anywhere in their body, a user can enter "`chapter(heading(security) && shareholder)`". To search for paragraphs with subparagraphs matching two different criteria, one can enter "`para(subpara(a,b,c) && subpara(x,y,z))`". Even proximity searches can be nested within element searches. The query "`title(5(apple,pear))`" matches documents whose titles contains the word "apple" followed within 5 words by the word "pear".

It should be noted that nesting does not always have exactly the same significance. Usually it signifies a more general ancestor/descendant relationship, but in once case it signifies a more specific parent/child relationship. For example, the query "`part(para(...) && para(...))`" matches Part elements with both types of paragraphs as descendants, regardless of whether there are intermediary levels in the document hierarchy, such as chapters or cross-headings. In contrast, the query "`part(heading(...))`" matches only those Part elements with the necessary heading as a direct child, and not those which contain matching headings belonging to some descendant chapter or paragraph. We hope we have anticipated users' expectations and that the meaning of nested queries is unambiguous in context.

Implementing queries that target direct parent/children relationships introduces an additional level of complexity to our application, for the MarkLogic database[3] we use is not optimized for them. Also, our database does not allow direct parent/child relationships

[2] Apache Lucene https://lucene.apache.org/

[3] MarkLogic https://www.marklogic.com/

to be expressed in that component of a native query which can be constructed at runtime. Because we need to build our queries at runtime, to capture the virtually limitless range of possibilities we give the user, we have had to construct approximate native queries for the database engine and then filter the results at the application level. For example, because we cannot express parent/child relationships adequately, we pass an over-inclusive ancestor/descendant query to the database, allowing it to take maximum advantage of the indexes it keeps, and then we filter out the false positives (in this case, descendants that are not direct children) in our application code. We need application-level filters in other instances as well. For instance, we allow users to specify that a given query should ignore those parts of documents that are amendments to other documents. A restriction such as this requires a "`negative ancestor`" query, a query for things that do not have a specified ancestor, and negative ancestor queries are not supported by our database. We must therefore construct the most narrowly-tailored over-inclusive query we can, to leverage any existing index available, and exclude those results with the specified ancestor at the application level. We try to avoid application-level filters except where necessary, for they are slower than what can be done at the database level. Also, they are tricky to implement, especially when related to a NOT query. If a search must be represented as an over-inclusive query combined with a limiting filter, its negative cannot be represented with the simple negation of the over-inclusive query, as that would exclude too many documents.

4. Range Queries

Although most elements in legislative documents contain natural language text, a few common query targets contain scalar values, such as numbers and dates. Some of these are metadata values, and may not typically be presented to users, but others appear within the body of the document, such as signature dates. We allow users to construct range queries targeted at many of the common scalar values by using the quantitative operators =, <, >, <=, >= and !=. For example, to search for documents from the year 2016, the user can submit "`year = 2016`". Many of the relevant scale values in legislative documents are dates, and we support expressions involving the "enacted" date of primary legislation and the "made", "laid" and "CIF" (coming into force) dates of secondary legislation. Naturally, a range restriction can be combined with other range restrictions and also with any other query component using any of the boolean operators.

We originally conceived of range queries targeted only at stored values, that is, indexable values within the documents as they are stored within the database. But we have since expanded them to support a great many *computed* values. For instance, if one wanted to limit one's search to documents with more than one schedule, one could append "`&& schedules > 1`" to one's query. It happens that we do not store the schedule count as a metadata value in our documents, although we could have. Therefore, we must count the number of schedules in each matching document ad hoc, in response to each query request. Obviously, the performance of queries based on computed values differs considerably from those based on stored (and therefore indexable) values. However, the performance of queries involving computed values differs considerably from one to the next, depending upon the extent to which the computed restriction is the limiting factor in the query. A compound query whose computed terms is not particularly limiting, that is, which happens to match most of the same documents matched by the other query terms, will not run much more slowly that would a query without the computed term. On the other hand, if the computed term is very limiting, and the value must be computed for many documents only to exclude them from the results, then the query will run quite slowly. Because we allow users to construct queries of arbitrary complexity, any computed range restriction might cause grave performance degradation when combined with certain query components, but its effect on performance might be negligible when combined with others.

Range queries based on computed values are slow to execute, but they are easy to implement, because they require no change to the underlying data. Consequently, we are able to support a large number of such queries. One can limit one's search based on the number of tables, footnotes or annotations in a document. One can even restrict one's search based on the number of days separating the dates on which an instrument was made and the date it came into force. The query "`... && days-made-cif > 7`" matches only those statutory instruments that came into force more than a week after being made. Finally, we allow range queries to target the number of "matches" to the other components in a query. For example, the query "`para(x,y,z) && matches > 2`" yields those paragraphs meeting the specified criteria only when they appear in documents containing more than two such paragraphs.

5. Counting and Grouping

Our search API, discussed above, return a list of metadata about matching documents and document elements. It is a “paged” API, returning only a batch of results at a time. Sometimes, however, one wants to know the exact number of total matches. To satisfy inquiries of this sort we provide a count API. Requests to this endpoint accept the same query strings describe above, but they return only the number of matching documents. Responses from the count API take longer to execute, but they provide complete and accurate counts.

And the count API can count many things other than documents. Users can specify the unit to be counted with the “`count`” instruction. Here we employ a syntax similar to proximity and element searches: one uses the term “`count`” followed by a parenthetical expression signifying the thing to be counted. For example, if one wants to know not the total number of documents matching one’s query but instead the total number of paragraphs in all of the matching documents, one can append to one’s query the instruction “`count(total-paragraphs)`”. Similarly, one can count the total number of tables, footnotes, annotations, or any other computed value recognized by the range queries. And one may specify any number of different counters in the same request by combining count instructions with an AND. A query containing the instruction “`count(table-rows) && count(table-cells)`” will return both the total number of table rows and the total number of table cells within the documents matching the query.

Finally, counts may be grouped by a few fields known to contain only a limited number of unique values. Suppose that one wants to know the number of documents matching one’s query for each year within a given range. Or suppose one wants to know the total number of paragraphs for each designated subject matter. Our count API provides these results in response to the queries “`... && groupby=year`” and “`... && count(total-paragraphs) && groupby=subject`”. The former provides count information for each year, and the latter provides count information for each unique value in the subject field. One can group by only a few fields, such as subject, department, and signee. And we provide limited support for a two-dimensional grouping, so long as one of the dimensions is year. Therefore, “`groupby=department&year`” will produce a table of results, showing the specified count for each department in each year. Greater support for multidimensional grouping is possible, but we imagine it would be of limited utility.

6. Conclusion

We have tried to provide a simple syntax for expressing complex queries directed at legislative documents. We hope it is intuitive enough to be learned quickly yet expressive enough to represent most of the searches expert users would construct with a general purpose query language such as XQuery. Additionally, we provide results in a variety of formats, such as HTML, CSV, JSON and RDF. We hope these tools will allow researches to identify previously undiscovered patterns in our legislation and improve the consistency and accessibility of our laws.

Bibliography

[1] *XQuery 1.0: An XML Query Language (Second Edition)*. W3C. 14 December 2010.
https://www.w3.org/TR/xquery/
Scott Boag. Don Chamberlin. F. Mary Fernández. Daniela Florescu. Jonathan Robie. Jérôme Siméon.

Interoperability of XProc pipelines

A real world publishing scenario

Achim Berndzen

<xml-project />

Gerrit Imsieke

le-tex publishing services GmbH

Abstract

Although XProc is a standard, real-life applications often use optional steps that conformant processors need not implement, or they use extensions. For some of the extensions there exists a specification, EXProc. Others are processor-specific, either bundled with the processor or written by third parties.

transpect is a comprehensive XProc framework for checking and converting XML and XML-based document formats. transpect exploits both extensions shipped with the XProc processor XML Calabash and extensions written by le-tex.

One of this paper's authors, Achim Berndzen, has ported many of transpect's XProc modules and Calabash extensions to his own processor, MorganaXProc. He also ported a conversion/checking application that is built on these modules to MorganaXProc.

This paper draws on the insights that this migration has brought forth. It presents what pipeline authors, framework developers, and XProc processor vendors need to consider in order to make their pipelines, software libraries, and processors interoperable.

Keywords: XProc

1. Introduction

XProc [1] proves to be a very efficient language when it comes to apply complex chains of operations on sequences of XML documents. In this paper we will evaluate the interoperability of XProc pipelines, i.e. the possibility to migrate a complex pipeline system developed for one XProc processor to another. We take interoperability in this sense to be an indicator for the maturity of XProc and its usability, which in turn is relevant for technology decision makers, pipeline authors, users, and the XProc community as a whole.

In order to get some assessment on the interoperability of XProc pipelines, we focus on a real world scenario: The migration of the *transpect* pipeline package[1] developed for *XML Calabash* [2] to *MorganaXProc* [3].

transpect is a framework for checking and converting XML documents and XML-based data, such as .docx, IDML, and EPUB, developed by le-tex, a premedia services and software development company based in Leipzig, Germany. *transpect* is based on open standards, in particular XProc, XSLT 2.0, Relax NG, and Schematron. It has been adopted by many publishers and standardization bodies and is considered as the largest XProc application worldwide [2].

transpect was developed using *XML Calabash*, the XProc processor created by Norman Walsh (who is also the chair of the W3C's XProc working group). For a long time *XML Calabash* was the only publicly available and actively maintained XProc implementation. It has a 100% score against the XProc test suite. *XML Calabash* can therefore be called the "gold standard" of an XProc implementation.

The target of our migration project is *MorganaXProc*, a new XProc processor developed by <xml-project />. It also is a complete implementation of XProc (including the extension steps of the EXProc.org library) which has a very high score (99.67%) against the test suite, passing all tests but three, all of which are related to optional and rarely used features of the recommendation.

Having a very complex system of XProc pipelines to be taken from a very good XProc implementation to a

[1] http://transpect.io
[2] http://xmlcalabash.com
[3] http://www.xml-project.com/morganaxproc/

doi:10.14337/XMLLondon16.Berndzen01

fairly good one, we think there is a good chance of finding some answers to the question of real world interoperability of XProc pipelines.

After giving the term "interoperability" a more precise meaning, we will set the stage for the real word scenario and give you a brief overview of *transpect* – the pipeline package to migrate. Then we will give you an insight in what kind of problems to expect for such a migration by looking at the W3C recommendation for XProc, the current state of this technology and its implementations. Based on this assessment, we will give you a report on the real problems of the *transpect* migration from one processor to another and how we solved them. Based on our findings, we will come back to the question of interoperability and suggest consequences for technology decision makers and pipeline authors as well as for the XProc community.

2. Interoperability in real world scenarios

Before we start talking about our migration project, it might be appropriate to get our notion of "interoperability in real world scenarios" a little more precise. In its origins, interoperability means the ability of two systems to work together. The sense in which we use the term "interoperability" here differs slightly from its original use. It comes from W3C's terminology in the "Technical Report Development Process", where a working group is asked to "be able to demonstrate two interoperable implementations of each feature"[1] of the technical report. This is primarily intended to make sure that the report is so precise that two independent implementers can build equivalent implementations on this basis. From the position of a technology user, having interoperable implementation means the ability to use the technology in either implementation without any or with only minor changes. This is the sense in which we use the term "interoperability" in this paper. We say that XProc pipelines are interoperable when it is possible to migrate them from one XProc implementation to another without any or with only minor changes.

Now, having cited the relevant paper from W3C, our question may seem odd: XProc is a W3C recommendation and interoperability is a requirement for becoming a recommendation, so XProc allows pipeline authors to write interoperable pipelines in our sense. Certified by W3C! No further investigation required.

However, that there *can* be interoperable pipelines does not mean, that every pipeline running successfully on one conformant processor *is* actually interoperable and will also run on another conformant processor. There are mainly two reasons to raise the question of interoperability in real world scenarios:

First, XProc in many senses is not a fixed, but an enabling technology. The most prominent feature of XProc in this respect is the open or extensible step library. Implementers are not restricted to the features defined in the recommendation, but will typically enhance their implementation in ways useful for pipeline authors. And authors develop complex systems or packages of XProc pipelines to achieve real-life aims. And to do this, they will make use of additional features that their XProc implementation offers, standard or not.

The second kind of doubt regarding interoperability might come up because as already mentioned for a long time there was only one XProc implementation publicly available and actively maintained: *XML Calabash* developed by Norman Walsh, who is also the chair of W3C's XProc working group. Therefore pipeline authors did not even have the chance to test their pipelines in different environments. Of course they may take care to use only those features covered by the recommendation, but in practice their conception of XProc will be what the implementation they use offers, not the technical standards behind the implementation. Therefore, one might argue that we do not have enough experience judging the question of interoperability beyond the test suite yet.

And this throws a slightly different light on the question of interoperability: Given that a pipeline author has successfully developed a complex package of pipelines using one XProc processor, how much effort does it take to make this package usable on another XProc processor?

Why does interoperability in real world scenarios matter? The answer to this question does obviously depend on who you are. If you are a pipeline author, developing XProc pipeline systems for a customer or your own use, our question of interoperability can be translated to the problem of write once, use everywhere. If XProc is an interoperable technology, there is a good chance to reuse a pipeline that is developed and tested for one implementation with another processor. For pipeline users interoperability means freedom of choice: If I want to use this pipeline, am I chained to a certain XProc processor or can I use every conformant XProc implementation I like?

[1] *See* [3] *, 7.4.4*

For people making technology decisions, interoperability of XProc pipelines is important, because XProc is in many respects not a technology without alternatives: You can do it with XProc, but you could also use other technologies to chain together complex operations on sequences of XML documents. XProc is best suited for this task because it was designed for it, but this is obviously not the only criterion in a decision for or against the use of XProc. Interoperability might not be the decisive criterion, but surely vendor independence and reusability will be taken into account. Finally for the XProc community a positive judgement about the interoperability would be an approval of the work done, while problems with interoperability might give hints at future tasks.

3. *transpect's* methodology and its reliance on *XML Calabash* extensions

Being a framework that is used by more than 20 publishers and standardization bodies for converting and checking a 6- to 7-digit figure of diverse manuscript and typeset pages per annum, *transpect* certainly qualifies as a real-life XProc example. What makes it a particularly good test case for migration and interoperability is its reliance on optional steps and *XML Calabash* extensions.

transpect offers functionality and methodology for converting and checking XML data.

The functionality part consists of roughly 25 modules for converting .docx to a flat, DocBook-based intermediate format[1], from this format to JATS or TEI XML, from XHTML to InDesign's IDML, etc.

The methodology part is about configuration management (we'll look at that in a minute) and error reporting – collecting errors and warnings across many conversion steps and presenting them in an HTML rendering of the input, at the error locations.

An example for a complex *transpect* conversion chain starts from docx, goes via flat and hierarchized DocBook to the publisher's preferred JATS XML vocabulary and from there to EPUB. *transpect* chooses to use an intermediate XML format because it would be too costly to implement the complex section and list hierarchizations etc. for each XML vocabulary. It is easier to do the heavy lifting within one vocabulary (for example, DocBook) and convert from there to other vocabularies such as TEI, JATS/BITS, Springer A++, WileyML, etc.

This use of a neutral intermediate format increases the number of conversion steps. After each conversion step, there may be checks that report errors for the consolidated HTML report. These checks are motivated by different quality assurance requirements, such as:

Many upconversion operations will rely on consistent use of certain styles in the Word or InDesign input files. These style names may be checked against a list of permitted styles using Schematron. After hierarchization, another Schematron check may detect whether there are appendices before the first book chapter, that all references given in the bibliography are being cited, and what else the publisher may impose as business rules. The final BITS or JATS XML will then be validated against the corresponding Relax NG schema. The resulting EPUB will be checked against IDPF's epubcheck and additional, E-book-store-specific rules that pertain to minimum image resolution[2], maximum file size, required metadata fields, etc.

In *transpect*, the first conversion step will typically insert so-called @srcpath attributes at every paragraph, formatted text span, image, etc. These @srcpath attributes will be carried along through subsequent conversion checks, including the HTML rendering of the source content. Each Schematron and Relax NG validation[3] will record the @srcpath that is closest to the error location. In a final step, the consolidated error messages will be merged into the HTML rendering, yielding the *transpect* HTML report.

The steps that unzip IDML or docx input files, determine image properties and report XPath locations for Relax NG validation errors all rely on Calabash extensions that block an easy migration path to another XProc processor.

The other main concept pertains to configuration management. Before *transpect*'s modular approach was pursued, converters frequently used case switches for handling special cases (for certain imprints, book series, individual books, …). This rendered the conversion code

[1] https://github.com/le-tex/Hub

[2] *There is a transpect module,* https://github.com/transpect/image-props-extension *, that reports pixel dimensions, color space, and other information for bitmap files. This extension has originally been written as a Calabash extension that interfaces Java libraries such as Apache Commons Imaging,* https://commons.apache.org/proper/commons-imaging/

[3] *transpect provides another Calabash extension,* https://github.com/transpect/rng-extension *, that uses a patched Jing validator that also reports the XPath location of an error. The usual line number information does not make sense in multi-step pipelines that do not serialize the intermediate XML documents.*

quickly unmaintainable. In other cases, the input files were considered too unimportant to justify case switches in the code. Therefore, they had to be converted manually or with other tools.

In *transpect*, the detailed step orchestration can be loaded dynamically or even generated for individual production lines. This occurs frequently in multi-pass XSLT conversions where certain production lines necessitate additional conversion steps.

In standard XProc, the complete pipeline that will be run must be known in advance. *transpect* uses *XML Calabash*'s cx:eval step in order to run these dynamically loaded innards of larger steps.

(In addition to the XProc orchestration, the applied XSLT, CSS, style lists, Schematron rules, etc. may be loaded dynamically from the so-called conversion cascade.)

To summarize, the two core *transpect* methodology components, error reports and cascaded configuration, rely heavily on Calabash extensions that wrap Java libraries and on cx:eval, a calabash extension step that allows dynamic evaluation of pipelines.

4. Obstacles to expect

Which obstacles are to be expected when one tries to migrate a complex pipeline system from one conformant XProc processor to another? If you have some experience with XProc, one thing or the other may cross your mind. In a more systematic perspective we can deduce five different types of obstacles for migration:

1. The distinction between required and optional steps/features
2. Implementation-defined features in the W3C recommendation
3. The proposed extension steps from the EXProc[1] community initiative
4. Processor specific steps and author defined steps in a second language such as Java, C, or whatever the processor is able to understand.
5. Problems from the underlying technologies in façade-steps

Let us shortly discuss these types to ensure a common understanding and to get those people on board, who do not work with XProc in their every day life:

4.1. Required and optional steps/features of an XProc processor

As a lot of recommendations published by W3C, the recommendation for XProc defines two levels of conformance. First there are "required features", forcing a conformant implementation to implement these features in the way defined in the recommendation. Secondly there are "optional features" to characterize those features a conformant processor is not required to implement. However if the implementation chooses to cover one or more of these features, they must conform to the recommendation. The most prominent place for this distinction is XProc step library: A conformant processor must implement all 31 required steps and it may implement one or more of the 10 optional steps. To quote from the recommendation: "The following steps are optional. If they are supported by a processor, they must conform to the semantics outlined here, but a conformant processor is not required to support all (or any) of these steps."[2]

Concerning our question of real world interoperability, the threat is obvious: A pipeline author could use an optional step in her pipeline, which is supported by one processor but not implemented in the other. The pipeline will typically fail to compile and raise a static error. And this does concern such practically important steps as running an XQuery expression over a sequence of documents (<p:xquery/>) or the validation of an XML document with Schematron, RelaxNG or XML Schema.

4.2. Implementation-defined features in the W3C Recommendation

A brief look at the W3C Recommendation for XProc shows that there are all in all 47 features of XProc listed as "implementation-defined". [3] Additionally there are 21 features marked as "implementation-dependent".[4] We will not discuss all these features here: to do so might be boring to readers and they are not all relevant to the question we discuss here. Many implementation-defined features are concerned with the connection of a top level XProc pipeline with the outer world, viz. the problem of providing documents and option values to a pipeline using the user interface of the respective implementation. So it comes up to the question, *how* to invoke a pipeline

[1] See http://exproc.org

[2] [1], 7.2

[3] *See* [1], A.1

[4] See [1], A.2

in a given implementation, not whether a pipeline is able to run on a given implementation or not.

But others may be important to the question of interoperability, for example provision (20): "The set of URI schemes actually supported is implementation-defined."[1] If my pipeline system developed for processor *A* relies on the availability of a certain URI scheme I cannot expect this scheme to be available on another processor, even though both are conformant implementations. The same does hold for provision (46), stating that processors may support different sets of methods for serializing XML documents.

In practical use cases, the most challenging provision concerning implementation-defined or implementation-dependent features may be this: "The evaluation order of steps not connected to one another is implementation-dependent."[2] This might sound scary if you come from traditional programming languages and are used to think that the sequence of operations is program-defined, not processor-defined. On the other hand: In XProc everything is about sequences of documents flowing through a sequence of steps connected by ports. And the provision says the processor has to respect the connections specified in the pipeline and is free to reorganize the evaluation order of the steps that are not connected. So everything seems fine and there is nothing to worry about.

But when you think again, you might come up with a pipeline fragment like this:[3]

```
<nasp:log-in-to-a-web-service/>
<nasp:send-data-to-this-service/>
```

Although there is no port connection between the two steps, obviously the log-in step has to be performed before any data is send. With some effort one might be able to rewrite the two steps and establish a port connection between them. But that would be totally against XProc's spirit: The order of steps is determined by data flow, but here we would construct a data flow to ensure the necessary execution order of the steps.

Other examples where the execution order is not as the pipeline auther expects it to be is when a resource stored with <p:store/> (that does not have an output port) is needed in another step, yet the store step is executed after the other step because the XProc processor may choose to do so.

So the provision that a processor is free to rearrange the execution order of steps not connected by ports surely poses a great threat on interoperability: A pipeline running perfectly well on one processor may not produce the expected results or even fail because the other processor has chosen to rearrange the steps in a different order. Now, this is obviously not only a threat to interoperability, but may also raise problems if you are working with one processor, because the provision does not state that the order of execution has to be stable concerning different runs of the same pipeline. The execution order might, for example, depend on the latency of some web service or the file system. So even using one processor, you might get wrong results from different runs of the same pipeline. Certainly there are workarounds, as one may introduce artificial connections between steps just to control the order of execution, but this may surely lead to more complex or even unreadable pipelines.

One might therefore argue that the "reorder rule" is at least a problematic aspect of the recommendation. Norman Walsh seems to agree with this, because in *XML Calabash* he provides a possibility to say that step *A* has to be executed after step *B* even if the two steps are not connected by ports. To do this, he introduced an extension attribute called "cx:depends-on" containing a whitespace separated list of step names which must be executed before the step with the attribute is executed. As Walsh stated[4], this solution does solve the problem for one processor, but is a threat to the interoperability of a pipeline with other processors, because of the very nature of extension attributes.

Extension attributes are defined in the XProc recommendation, section 3.8. Formally an extension attribute is an attribute used on an element in an XProc pipeline, usually a step, where the attribute name has a non-null namespace URI and is not in one of XProc's namespaces. Any implementer is free to introduce such attributes as long as the requirements of the recommendation are met, and any other processor "which encounters an extension attribute that it does not implement must behave as if the attribute was not present." Now the typical situation to expect when using an extension attribute is that one processor (the one who introduces the attribute) will behave differently than the other processor (not knowing the attribute and therefor ignoring it). The exact outcome of using an extension

1 [1], A.1

2 [1], A.2

3 This fragment is inspired by *[3]*.

4 [4] , *5.2*

attribute in terms of interoperability depends heavily on the semantics of the attribute. As we can suggest that the implementer introduced the attribute in order to get some effect, we can expect different behaviour of the same pipeline using different processors in all cases, but the impact might vary: One can think of extension attributes used for debugging purposes, invoking some extra effect to a step without changing its "normal" behaviour, but there might also be extension attributes completely changing the behaviour of the step by producing another kind of output when present. To sum this up: Extension attributes are a potential threat to the interoperability of pipelines as the other implementation-defined or implementation-depended features, but their impact cannot be judged in general but has to be considered from case to case.

4.3. The proposed extension steps from the EXProc community initiative

The next two possible obstacles to interoperability of pipelines result from one of the most interesting aspects of this language: the open step library. Apart from the steps defined in the recommendation (as mandatory and optional steps), there are additional steps proposed by a complementary community initiative called EXProc. The steps from the EXProc library differ from those in the recommendation in at least three aspects that are important for the question of interoperability:

- Until now, there are no procedures to decide whether a proposed step is actually a good idea or not. Therefore every implementer of XProc can decide on his own, possibly resulting in different step libraries of the XProc processors to migrate from or to.
- The definitions of the steps in the EXProc community initiative are definitely not on the same level of precision as the steps defined in the recommendation, so we might expect different interpretations of the step's behaviour.
- Thirdly there are no tests defined for these steps, so implementers cannot check their implementation by running tests ensuring the expected behaviour is actually delivered by the step's implementation.

In terms of interoperability these steps can be put in the same box as the optional steps defined in the recommendation: Each implementer is free to implement none, some or even all of them. But as these steps are not part of the recommendation, two processors may be conformant but implement different behaviour and therefore produce different output for a pipeline containing one or more of them. And this effect does not come necessarily from an error or a bug, but may result from different interpretations of the (rudimentary) description of the EXProc steps.

To give you an example: EXProc.org defines a step called <pxp:zip /> which creates a zip archive and is expected to return a description of this archive. One feature of this description is the size of a compressed file, which is not actually returned by every implementation. Some (*MorganaXProc* for example) just return "-1" because they are not able to determine the size for every type of output stream (e.g. when the zip is creates on a web service). Is this a correct implementation of <pxp:zip /> or not? You cannot tell this from the step's description. And this is certainly a possible threat to the interoperability of pipelines, because a pipeline may (for what ever reason) depend on knowing the correct size of the archive.

4.4. Processor specific steps and author defined steps in a second language

Next up on our list of obstacles to the interoperability are processor specific steps, viz. steps that are part of the processor's built-in step library, but not covered by either the recommendation or the EXProc community process. Since they come with the processor as vendor specific extensions, it is very unlikely that a pipeline containing one of these steps will run (typically not even compile) on another processor. The fact that the step did not make it to the two libraries can be taken as a hint that only a small group of people are interested in this step. So the motivation for another implementer to take the step into her processor specific library may be very low.

The second, also processor-specific threat to interoperability comes from author-defined steps in a second language: Typically, an XProc processor allows a pipeline author to define atomic steps in a second language, i.e. not in XProc but in another programming language. And normally this will be the language in which the processor itself is written, because this is obviously the easiest solution. This way of enhancing XProc's standard library is explicitly mentioned in the recommendation, but all details are declared to be "implementation-dependent"[1].

This is done with good reasons, because taking a piece of say Java or C code implementing an XProc step to a processor is only possible as a deep intervention. Any attempt to define an interface of an XProc processor to

[1] *See* [1], 5.8.1

another programming language would severely restrict the implementer. To understand this, we have to recognize we are not only facing the problem of telling the processor which part of a second language code should be used for a specific atomic step. This is in fact the easiest aspect of the problem. The more complex part is to connect the secondary language code to the processor, so the processor can call this code when necessary. Remember that an atomic step in XProc has

- a list of input ports, each holding a sequence of XML documents,
- a list of options, each holding a value, and
- a list of output ports with XML documents representing the result of the step's performance.

How to represent this information necessary for the call and how to represent the data, supplied in calling the step, is highly processor specific, because it is a part of the basic decisions an XProc implementation has to make.

Given this, one may be tempted to say, that the piece of second language code has nearly nothing to do with XProc, but has a great deal to do with the processor. It is almost impossible for an author-implemented steps used with processor A to be runnable on processor B. So here we have one aspect of the recommendation, which seems to be an insuperable barrier to interoperability of pipelines. If a pipeline needs to enhance the step library with an author-defined step in a secondary language it is impossible to take it to another processor without any changes.

4.5. Problems from the underlying technologies in façade-steps

To complete our discussion of obstacles to interoperability we would like to mention one more point: Apart from offering a good processing model for XML pipelines and having a rich step library, XProc is also designed to provide façades for other XML technologies like XSLT, XQuery, XSL-FO, to mention just a few.[1] An XProc step like <p:xslt/> acts as a standardized interface to activate an XSLT transformation on an XML document and to retrieve the transformation result. These technologies are mostly older and therefore presumably more mature than XProc, but there is no conclusive reason to see them as perfect. And consequently all the obstacles to interoperability in our sense that are connected to the XML technologies used (inaccurateness in the recommendation, implementation-defined features, idiosyncrasies of the implementation) will also directly constrain the interoperability of XProc pipelines using these technologies.

To give a concrete example of this type of problem, one might refer to Saxon's implementation of the collection function[2] where a file URI invokes a catalog-resolving process, while a directory URI might have query elements to select specific files from the folder. These very useful mechanisms are used quite a lot, but they are not standardized. The threat to interoperability here does not rise directly from anything in the XProc recommendation, but from the fact that there is a <p:xslt/>-step defined for which different XProc processors may use different third party implementations.

5. Back to our real world example: What obstacles to expect?

Now having looked at possible obstacles of migration to expect from the knowledge of XProc and its specification, which of them did we actually expect to matter for our migration project?

As we said, both XProc implementations, *XML Calabash* and *MorganaXProc*, implement all required members of the step library as well as the optional library and the step libraries proposed by EXProg.org. *XML Calabash* has a perfect score of 100% conformance with the test suite, *MorganaXProc* is almost equivalent with 99.67%. The three tests where *MorganaXProc* fails cover optional features (PSVI and XML 1.1.), which are not relevant for our project. So there seems to be a fairly good chance to prove interoperability of XProc by successfully migrating *transpect* from *XML Calabash* to *MorganaXProc*.

On the other hand it was clear from our very start that we had to face problems concerning our migration project in at least four points:

The first point has nothing or very little to do with XProc, but with the practical requirements of complex XProc pipeline systems to be deployed to different users: *resource management*. Real live XProc pipelines are typically no self-containing files, but have links to some sources outside the pipeline. For example a pipeline may import a library with XProc steps declarations, call an XSLT stylesheet stored in another file or use an

[1] *See [5], p. 133*

[2] See http://www.saxonica.com/html/documentation9.6/sourcedocs/collections.html

Schematron schema stored in yet another file. This is a typical situation for a complex pipeline system because one may put the resource into the XProc pipeline document itself, but in production contexts this is not an option for reasons related to readability, maintenance and storage size.

XProc as a language has only rudimentary support for this kind of situation. One may do all the references with relative URIs because XProc will resolve all relative URIs by using the pipeline's base URI. This might work for some relatively small systems of pipelines but is of course difficult to maintain in complex systems.

To cope with the problem of resource or dependency management, at least two different additions to XProc are in the field: XML Catalog[1] and the EXPath packaging system [6]. The common basic idea is to use some kind of virtual or canonical URI in pipelines to point to external resources and then to have a system for resolving this virtual URI to the "real" URI of the requested resource. Now unfortunately *transpect* uses XML Catalog, which is not supported by *MorganaXProc*, which in turn uses the EXPath packaging system. *XML Calabash* also supports the latter via an extension developed by Florent Georges [7]. So the problem does not seem insuperable, but there is definitely some work to be done to get the pipeline system from *XML Calabash* to *MorganaXProc*.

The second source of problems, which was clear from the start, are user-written steps in a secondary language, here in Java. *transpect* comes with four additional steps, which are quite necessary for the tasks to perform:

1. <tr:unzip/> has the task to extract a complete zip file or a single contained file to a specified destination folder. This task cannot be performed by <pxp:unzip/>, because the latter is designed to produce an XML document on the result port. The task of <tr:unzip/> on the other hand is to unzip the complete archive with XML documents, graphic files and so on to a specific location in the file system.
2. <tr:validate-with-rng/> uses a patched version of "jing.jar" to generate a report of the validation, where each error has its location in the document expressed as XPath instead of line numbers. So here we get a detailed report of the validation result, while the optional step <p:validate-with-relax-ng/> in the XProc recommendation is just designed to make sure that the document passes the validation.
3. <tr:image-identify/> reads image properties from raster images and
4. <tr:image-transform/> transforms raster images.

It might be unusual for a typical XProc pipeline system to depend on user-written extension steps in a secondary language, but XProc does offer this mechanism and so any migration project like ours has to be aware of it. As both *XML Calabash* and *MorganaXProc* are written in Java and the user-written steps are in Java as well, our concrete project does not have to face the full range of problems. But there is still the problem of connecting the user-written code to two implementations with a very different inner life.

The third problem rises from the above-mentioned differences in the implementation-specific step library. *XML Calabash* supports all in all more than 25 steps neither mentioned in the recommendation nor being part of the EXProc community process. In contrast *MorganaXProc* supports only two special or processor specific extension steps. Surprisingly only two steps proved to be relevant: *transpect* uses *XML Calabash*'s extension steps <cx:eval/> and <cx:message/> quite a lot. And *MorganaXProc* does not support either of the steps, if we are to speak strictly.

Just a quick description for those of you not perfectly familiar with *XML Calabash*'s extension library: <cx:eval/> evaluates a dynamically constructed pipeline. This clearly adds a new level of possibilities to XProc: You can construct a pipeline within XProc, using XProc's steps or XSLT or XQuery and then you can execute this pipeline within the XProc environment. The second step <cx:message/> is much more boring: It simply writes a message out to the console. Two quite useful steps.

What does it mean when we say that *MorganaXProc* does not support these two steps if one is speaking strictly? The obvious problem is that the names of the two steps are in a namespace and that this namespace is connected or represented by an URI starting with "http://xmlcalabash.com/". One might argue that this namespace belongs to *XML Calabash* and that therefore no other processor should use it. This is because it is up to *XML Calabash* and its developer to define the semantics of any step in this namespace. And it is also the exclusive right of the namespace's owner to redefine the semantics without consent of any other person and without prior announcement. So it is presumably a wise decision of any other implementer, not to use this namespace, because the semantics of the steps implemented might change and his implementation is not keeping up with this change. Given this line of argumentation, which is surely disputable, *XML*

[1] http://xmlcatalogs.org

Calabash and *MorganaXProc* cannot have common steps beyond the scope of the recommendation and the EXProc.org process. Even if both implementers choose to implement a step with exactly the same behaviour, these never ever will be the *same step*, because they have different names in different namespaces.

This argumentation might be too strongly rooted in linguistic theories for our practical purposes. But it also gives us a hint of how to solve the problem: That an identical concept (here: step) is expressed by different names in different languages is a known situation in linguistics. In fact it is the authority for translating one word from one language into another word in another language.

The problem with <cx:message/> in the context of migration does not seem too big, since these messages do not contribute to the results of any pipeline. One may simply ignore the step when the pipeline runs on a processor that does not know how to execute it. To raise the excitement: We found another solution.

Now while <cx:message/> might be ignored, <cx:eval/> surely is a foreseeable burden for the migration because *transpect* deeply depends on it and there is no workaround in sight using "conventional XProc".

The last problem we expected comes from the extension attribute "depends-on" introduced by *XML Calabash*, which is heavily used in *transpect*. As mentioned, this attribute allows pipeline authors to state that step *B* must be executed after step *A*, even when no input port of *B* is connected to any output port of *A*. At the first sight this does not seem to be a problem at all as *MorganaXProc* does not do any reordering or parallel processing of steps. It simply executes the steps in the physical order written in the pipeline, i.e., in document order. But on second thought one might imagine a pipeline author relying on the attribute and its semantics while writing a pipeline that should not be executed in the physical order of the steps. Luckily *MorganaXProc* also supports an extension attribute "depends-on". Here we have the problem with the different namespaces and therefore with the different attributes again. But the recommendation holds the key to an easy solution: "A processor which encounters an extension attribute that it does not implement must behave as if the attribute was not present." So *XML Calabash* is advised to ignore an attribute in *MorganaXProc*'s namespace and *MorganaXProc* must ignore the attribute belonging to *XML Calabash*'s namespace. Consequence: We can have a step with both attributes to enforce execution order, each processor just reacting to the attribute that it recognizes.

6. Off to the lab: Found problems

Expectations are one thing; reality in most cases is another. So: What types of additional problems did we actually face when trying to migrate *transpect*?

The first thing that struck us when starting the actual migration process was that there is no defined way in the recommendation to import extension steps. Although *XML Calabash* and *MorganaXProc* both support the complete range of atomic extension steps of EXProc.org, you cannot compile a pipeline created for *XML Calabash* with *MorganaXProc*. The reason for this is that, according to the recommendation, the "processor must know how to perform."[1] Neither *XML Calabash* nor *MorganaXProc* know how to perform say <pxf:mkdir/> without any import. They fail with XS0044: "… the presence of atomic steps for which there is no visible declaration may raise this error".

How can we make the declaration visible in *XML Calabash* and *MorganaXProc*? Surprisingly neither EXProc.org nor the recommendation defines a strategy. *XML Calabash* chooses to incorporate all extension steps into a global library to be imported by "http://xmlcalabash.com/extension/steps/library-1.0.xpl".[2] *MorganaXProc* chooses to use the namespace URI of the extension library also for the import. And both processors do not actually read the step declarations from the URI, but have an internal representation of the steps to be imported when the URI is found on a <p:import/>. Now this obviously makes the migration of a pipeline importing one of these steps very difficult as the URI used in the import is a "special trigger" for one processor, but seems to be a regular URI pointing to a library for the other. Currently the URI used by *XML Calabash* actually points to a textual representation of the pipeline to be imported. But that may change over time. The URI used by *MorganaXProc* does not point to a library and since the URI starts with "http://exproc.org", there is no way to create such a textual representation for the processor's implementer.

One consequence of this situation is that you cannot have a pipeline using both ways of importing the step running on both processors. *MorganaXProc* accepts the "special trigger" of *XML Calabash* (as there currently is a file at the designated position), but *XML Calabash* will

[1] [1], 4.8

[2] *See [4], 5.6*

not accept a pipeline using "http://exproc.org/proposed/steps/file" to import the step library. It will correctly raise a static error (err:XS0052), because there is no library at this URI.

The second type of problem we found trying to run *transpect* on *MorganaXProc* was a different behaviour of steps from the standard library in the two processors. This came as a big surprise, because the steps in the standard library seem to be well defined in the recommendation and the conformity of an implementation seems to be guaranteed by passing the respective test in the test suite. However this is not true for at least three steps, where *XML Calabash* and *MorganaXProc* have different interpretations and therefore implement different behaviour.

The first step is <p:store/> which is used to write a serialized version of the document on the step's input port to a URI named by the step's option "href". According to the recommendation, a dynamic error (err:XC0050) must be raised, "if the URI scheme is not supported or the step cannot store to the specified location." The differences in interpreting the step's definition apply to the error conditions in which the processor has to raise "err:XC0050". Supposing the URI scheme used is "file", what to do, if the document is to be stored into a folder that does not exist? *XML Calabash* creates the folder hierarchy to store the file while *MorganaXProc* sees itself unable to store the file at the specified position and therefore raises "err:XS0050".

There are good arguments for both interpretations: *MorganaXProc* takes a very literal approach to the provisions in the recommendation reading: The serialized version of the document must be storable at the specified location or an error must be raised. This seems to be a legitimate reading of the recommendation. *XML Calabash* obviously has a broader conception of "storing at a specified location" which includes creating the folders necessary to do so. One basic argument for this interpretation may come from the fact that <p:store/> does not only support URI scheme "file" but may also support "http" and that there is no concept of folders associated with "http". As URIs are opaque here, an XProc processor cannot ask for a parent folder to exist. So creating the folder hierarchy with "file" is perfectly legitimate. Another argument in support for the interpretation put forward by *XML Calabash* comes from the fact that there is no way to create a folder in the XProc standard library. Therefore pipelines could never put their results into a specific folder unless the user created this folder before running the pipeline. And even when we take into account that there is a step <pxf:mkdir/> defined in the EXProc.org extension library, the solution *XML Calabash* found would comprise only one step, while with *MorganaXProc* one has to call at least two steps.

Looking deeper into this problem, we found that the different interpretations do not only apply to <p:store/> but also to <pxf:copy/> and <pxf:move/>. *XML Calabash* will copy or move a resource to the designated URI and will at the same time make sure that the respective parent resources will be created. Now *MorganaXProc* is also consistent with its interpretation, so it will raise "err:FU01" if the resource cannot be copied or moved to the position in the file system because the folder hierarchy necessary to do so does not exist.

To find different possibilities of interpretation with a step in the standard step library was quite astonishing, but as hinted above, the provisions concerning <p:store/> are not the only source of argumentation. The second step where *XML Calabash* and *MorganaXProc* differ in the interpretation of the recommendation is <p:xslt/> in very special cases making use of XSLT 2.0 (or higher). According to the recommendation, for <p:xslt/>, the "primary result document of the transformation appears on the result port. All other result documents appear on the secondary port."[1] But what is supposed to happen if a stylesheet only produces result documents on the secondary port? What is to be found on the result port? Does my stylesheet have to produce a result for the primary port?

As the primary output port "result" is a non-sequence port, the answer is clearly "yes", but *XML Calabash* and *MorganaXProc* disagree on the position, where the required error "XD007" appears. *XML Calabash* will only complain if one tries to connect the primary output port to a non sequence input port, but does accept an xslt stylesheet producing no primary result. So a <p:sink/> after <p:xslt/> solves everything. This can be called a consumer oriented approach, because *XML Calabash* will raise the error only if another step tries to consume or read the non-existing result of <p:xslt/>. *MorganaXProc* on the other hand implements a producer-oriented approach and will raise the necessary error at the moment an XSLT stylesheet completes without a result for the primary port. One argument for this strategy is that it makes the resulting error message more readable, especially in cases, when the xslt-stylesheet is not literally part of the pipeline but imported via <p:document/>. But this is surely a case to argue about because the recommendation does not say a

[1] [1], 7.1.31

processor has to enforce an atomic step to produce a result document. So here *MorganaXProc* is stricter in the interpretation of the recommendation than *XML Calabash* (or may be even wrong taking into account Norm Walsh's double role).

And we found a third step in the standard library where *XML Calabash* and *MorganaXProc* have different interpretations of the recommendation and therefore implement a different behaviour. As in the other two cases, it is a very special situation where this difference appears: There is a note in the section about <p:http-request/> asking implementers "to support as many protocols as practical" for method "get". Especially protocol "file" should be supported to allow pipelines to use computed URIs. Consequently both *XML Calabash* and *MorganaXProc* support this feature. But what is expected to happen when the file resource supplied for "get" does not exist?

Here *XML Calabash* and *MorganaXProc* take different approaches as the first throws an error (as it is required for <p:load/> or <p:document/>) while *MorganaXProc* returns a <c:response/> document with status "404", as required when <p:http-request/> is used with another protocol. As in the other cases, you can argue for both solutions and you will not find decisive information in the recommendation. Surely it is a minor or exotic situation, but concerning our question of interoperability we have to state that a pipeline relying on the behaviour of one processor will not produce the same (expected) result on the other processor.

We found a fourth difference in interpretation and therefore in implementation with <pxf:mkdir/> from the EXProc.org library. This step is supposed to create a directory, and should fail or return a <c:error/> if the directory cannot be created. Now what is to happen when the directory already exists? *MorganaXProc* will fail, while *XML Calabash* will not. You can argue for both solutions, so here we have one case where the description of EXProc.org definitely needs to be enhanced or reference tests will be useful.

Another unexpected problem rises from the fact that XOM, the object model for XML used in *MorganaXProc* is even stricter than the S9Apis used by *XML Calabash*. When you try to load a document with a namespace declaration like "xmlns:d='dummy'", *MorganaXProc* will fail with err:XD0011. This is because XOM does not regard a document as a well-formed XML document when there is a missing scheme in the URI of a namespace declaration. The URI has to be in conformance with RFC 3986, which states: "Each URI begins with a scheme name that refers to a specification for assigning identifiers within that scheme."[1] The only way to get XOM and consequently *MorganaXProc* to accept this document is to change the namespace declaration to something like "http://dummy".

7. Problems solved: Lifting the burdens of migration

Having listed the problems we had to face when trying to migrate *transpect* from *XML Calabash* to *MorganaXProc*, let us now look at the solutions we found. No fear! We will not discuss every detail of our migration here. We will focus on those problems we take to be typical for migration projects and which therefore may throw a light on the question of interoperable XProc pipelines. We also believe we have found some recipes which might be interesting for other XProc projects.

7.1. Resource management

The first problem we had to address in order to get *transpect* running on *MorganaXProc* relates to resource management. As *transpect* heavily relies on XML Catalog to resolve imports and other resources (e.g. loading stylesheets for XSLT or schema documents for validation), it was impossible to run it out of box with *MorganaXProc*, which does not support XML Catalog. We had four options to solve this problem:

The first option was to rewrite all references to external resources in the pipelines to relative URIs resolved by the used XProc processor. This would have taken quite a while, but is obviously the easiest alternative. We do no longer rely on XML Catalog as an additional tool to XProc but only use the tools built into the language. But there are good reasons why le-tex chose not to go this way but made use of XML Catalog when *transpect* was originally developed: Using a resource management system is a reasonable solution when you deal with large pipeline systems to be maintained and deployed into different user contexts.

Knowing that we needed some kind of resource management for *transpect* the second option was to use the EXPath packaging system (supported by *MorganaXProc*) and XML Catalog (supported by *XML Calabash*) side by side. This was actually the first step taken to make *transpect* run with *MorganaXProc*: We rewrote the dispatching rules in the catalog files of XML Catalog for the EXPath system, which took about half an

[1] [9], p. 16

hour's time. This seems to be a reasonable solution for some cases but makes it obviously more difficult to maintain the *transpect* libraries in the future: Every change in the dependency must be marked in two files, which is an error prone process. Since XML Catalog and the EXPath packaging system both use XML as their base format, one might think about an XSLT stylesheet doing this work automatically, but even then one might forget to invoke the stylesheet resulting in a broken delivery for one of the supported processors.

Having taken the decision to rely on just one system for resource management, there are some arguments to use the EXPath packaging system for *transpect*. The most obvious one is that there is an implementation for both processors. This option would have taken a rather short time to fix the problems once and for all. We would just have to rewrite the resource management for *transpect* using EXPath and we are done. However this option was not taken. One argument against this solution was that *XML Calabash* does not support the EXPath packaging system out of the box, but only via an extension. As a consequence the process of delivering *transpect* (which includes *XML Calabash*) would become somewhat more difficult because the additional software would have to be packed and delivered, too. The other argument against this solution in the broader perspective of interoperable XProc pipeline is that it creates an isolated application. There might be use cases where the XProc project for some reason or another has to use XML Catalog and cannot move to the EXPath packaging system. So we would not provide a solution for these situations.

Having this broader perspective in mind, we finally convinced ourselves to add native support for XML Catalog to *MorganaXProc*. This was obviously the most expensive option because providing different kinds of resources to different types of steps is one of the basic tasks an XProc processor has to fulfil. Therefore a lot of code has to be written in different classes in order to get the task done, always trying not to interfere with the existing support for the EXPath packaging systems. As *MorganaXProc* provides a pluggable file system, allowing users to add support for special kinds of external resources (i.e. databases etc.), implementing XML Catalog for *MorganaXProc* was a non-trivial task.

Taking on this task might not be justified for just the special case of migrating *transpect* from *XML Calabash* to *MorganaXProc*. But as discussed above, resource management is a general requirement for complex XProc pipeline systems and so we can expect other projects to face a similar problem. As we already said: Resource management does not have anything to do with XProc as a language, but when you look at large, real-life projects, there will always be some kind of resource management involved. So we think our solution for the problem is also a contribution to enhance interoperability of real-life XProc projects. And of course it introduces an additional feature for users of *MorganaXProc*, so it is a win-win situation.

7.2. Divergent interpretations of the recommendation

While we are at it, we decided to make some further changes to *MorganaXProc* in order to enhance interoperability. As discussed above, *XML Calabash* and *MorganaXProc* take different approaches when it comes to store, copy or move a resource to a location that does not exist in the file system. *XML Calabash* will create the necessary folders while *MorganaXProc* will raise an error because it sees itself unable to store, copy, or move the resource. As the recommendation does only say a processor has to raise an error if the resource cannot be created, but does not explicate these conditions, both interpretations seem to be fully justified. However they lead to non-interoperable pipelines, allowing a pipeline executable with *MorganaXProc* to be executed with *XML Calabash* but not vice versa.

It would be possible to rewrite the pipelines to make them interoperable as both processors implement <pxf:mkdir/> which is used to create folders. But as we have shown above, there is also an interpretation problem with this step, as *XML Calabash* will not raise an error when the folder to be created already exists while *MorganaXProc* will complain. So the rewrite would be a little bit more complicated:

1. Check whether the parent folder exists, if not create it.
2. Store/copy/move the resource.

Anyone familiar with XProc sees a lot of lines to be written to make this happen. Of course one could declare three new steps in a <p:library/> doing this task until the divergent interpretations are ruled out, but this would not be very handy and at the time of writing there is no time horizon in sight for an authorized judgement. As it is also no principle question of the understanding of the steps and as the interpretation put forward by *XML Calabash* leads to shorter pipelines, *MorganaXProc*'s behaviour was adapted to *XML Calabash*. From release 0.9.5.10 on *MorganaXProc* will create all necessary paths for store, copy and move and thereby reduce the burdens of migration.

7.3. User-defined steps in second language

This leads us to the first point where pipeline authors have to do some work. In our discussion of *transpect* we mentioned the fact that it relies on four additional XProc steps for which a Java implementation is supplied. As XProc comes with a quite large library of built-in steps and there are additional libraries defined by EXProc.org, it is surely not typical for every XProc project to introduce new steps written in a second language. But it might be necessary to do this in some cases, as we saw with the four steps in *transpect* that cannot be expressed using XProc.

As explained above, it is quite impossible for a user-defined step written in Java etc. to run on more than one processor. Therefore if you need this kind of extension to XProc, you will always face the task of providing an implementation for every processor that the step is expected to run on. So in our special case, having implementations of the four steps for *XML Calabash*, we had to develop four additional classes for *MorganaXProc* so they could be called from the *transpect* pipelines.

Can one think of an easier solution? Maybe, but that would surely be a topic of another paper. What might be interesting for the themes discussed here is that we can distinguish three different tasks to fulfil when providing a piece of secondary language as implementation for an XProc step: Firstly, you have to tell the processor that a declared step with a given signature is to be performed using the secondary language code. Secondly, when the step is called, you have to provide a mechanism of connecting the actual documents on the input ports and the option values from the XProc processor to the user written code, so the code can access the documents and options and create the result documents, which in turn must be sent back, so the processor knows what documents are on the output port. And thirdly you have to implement the step's logic itself, i.e. provide a piece of code taking the input values (ports and options) and produce the result values.

Now while the first and the second task are inherently processor dependent, the third is not. Actually it might be quite useful to separate the concerns for maintenance reasons. Our proposed strategy to have different implementation of the same step for each processor will face problems when bugs have to be removed or additional functionality is to be provided. The implementer always has to make the changes twice and has to provide different tests for her implementation. To solve these kinds of problems we developed an alternative solution by separating the "connect task" (steps 1 and 2) from the "execution task" (step 3). Our implementation of each step actually does not consist of two Java classes (one for *XML Calabash* and one for *MorganaXProc*), but of three. The third class implements the step's logic while the other two serve the task of connecting this logic to the respective processor. So we just have one class to maintain as long as the step's signature is not changed.

While having some benefits, our solution is obviously only possible because both processors, *XML Calabash* and *MorganaXProc*, are written in Java and therefore any implementation of extension steps will most likely also consist of Java code. It is very difficult to hold on to our type of solution when it comes to implement an XProc step for two processors written in different programming languages. And there is another drawback of our solution, which concerns efficiency: As we chose to implement the step's functionally in a separate class used by the processor specific class which makes the connection to the processors we had to find a common representation for XML documents in Java. As we said above, *XML Calabash* uses the object model introduced by Saxon while *MorganaXProc* relies on XOM. So the document inputs coming from the two processors to our two connection-classes are in different object models, but we obviously need a common object model for the implementing-class to make sense. At the current level of development we used the most common object model possible: the Java string class. So currently we serialize the document to strings in the connection-classes, perform string operations in the implementing-class and then build the needed object model from the resulting string in the connection-classes.

This process of serialization and building is surely not the most efficient way, to say the least. We shall think of better ways to do this, probably using the class "ContentHandler" from the saxproject[1]. But we think that our solution even in its current state may serve as a blueprint for other projects, which have to enhance XProc with secondary code on one hand and try to avoid a vendor lock-in on the other hand.

7.4. A short look at the remaining agenda

Let us just see, which problems are still on the agenda now:

1. Namespace declarations without scheme names
2. Missing primary results for <p:xslt/>

[1] http://www.saxproject.org

3. Divergent step libraries for messaging and executing pipelines
4. Different import mechanisms for EXProc.org libraries
5. Implementation specific extension attributes as "depends-on"
6. Different interpretations of the expected behaviour of <pxf:mkdir/>
7. Different error behaviour for non-existing files in <p:http-request/>

Now the first two problems have to be solved by hand because there is obviously no workaround. The XOM object model used in *MorganaXProc* will not build a document having a namespace declaration with an URI not conforming to RFC 3986, notably not having a scheme name. So there is no other way than go to the source files and correct the namespace declaration – or avoid these kinds of declarations right from the start.

Handwork also seems to be required when it comes to make sure every stylesheet in <p:xslt/> produces a document to appear on the primary output port. This is due to the above-mentioned mechanism in *MorganaXProc* to raise an error about the missing result immediately after finishing stylesheet execution. We could invent a work-around by putting every <p:xslt/> into a <p:try/> and then writing a <p:catch/> to check out whether the error results from a missing document and if so to provide this document. And we could avoid manual adaptations by writing a pipeline or an XSLT stylesheet wrapping every occurrence of <p:xslt/> into the descripted code. This would be a viable solution, but the price for the automation would be very high: There would be 20+ lines of code around every <p:xslt/> dramatically affecting the readability of our pipelines. And: The <p:try/> will slow down the performance of our pipelines if it is not actually needed, i.e. when the pipeline runs with *XML Calabash* or the <p:xslt/> does produce a proper result for the primary output port. So we should use the workaround only for those cases where no proper result is produced. And as those cases can only be inspected by hand, it seems to be much easier to change the stylesheets found directly than to write a wrapper around the <p:xslt/> using these stylesheets.

And what about the other problems? While we rejected the idea of using automation to make our pipelines interoperable for the special case of supplying missing primary results for <p:xslt/> it seems to be a good idea for the other cases. And it works. So let us see how to use XProc to make XProc pipelines interoperable!

7.5. XProc to the rescue

In order to make XProc pipelines that are running successfully on *XML Calabash* also usable on *MorganaXProc* we actually need two XProc pipelines:

The first pipeline, actually a library, is called "xproc-iop.xpl". The basic task of this library is to bridge the "namespace gap" between steps in the implementation specific libraries of two (or more) XProc processors. So the use case here is the mentioned fact that *XML Calabash* and *MorganaXProc* do have steps performing (nearly) the same operations, but having different names or being in different namespaces. The library establishes a common ground by declaring steps in a publicly available namespace, which can be used in either processor. The body (or subpipeline) of the step is then just a call of the respective step in the step library of the XProc processor actually running. Here is an excerpt from the library for one step, just to illustrate the idea:

```
<p:declare-step type="iop:message"
                name="message">
  <p:input port="source" sequence="true"/>
  <p:output port="result" sequence="true"/>
  <p:option name="message" required="true"/>
  <!-- if XML Calabash is used -->
  <cx:message p:use-when=
    "p:system-property('p:product-name')
       = 'XML Calabash'">
    <p:with-option name="message"
      select="$message"/>
  </cx:message>
  <!-- if MorganaXProc is used -->
  <mod:report p:use-when=
    "p:system-property('p:product-name')
       = 'MorganaXProc'">
    <p:with-option name="message"
      select="$message"/>
  </mod:report>
</p:declare-step>
```

We declare a step named <iop:message/> which can be used in pipelines running on either processor. The rest relies on the attribute "p:use-when" which triggers a conditional element exclusion if it's value is false. To quote from the recommendation: "If the attribute is present and the effective boolean value of the expression is false, then the element and all of its descendants are effectively excluded from the pipeline document."[1] As the exclusion has to be done before any static analysis of the pipeline, *XML Calabash* actually will just see its known

[1] [1], 3.9

<cx:message/> while *MorganaXProc* will just see <mod:report/>. Because this is done even before compiling the library, we do not have to fear any loss in the pipeline's performance. The other step currently implemented is <iop:eval/> which serves as a bridge between <cx:eval/> (*XML Calabash*) and <mocc:eval/> (*MorganaXProc*).

Additionally this library could be used to force a common behaviour of the two processors for <pfx:mkdir/> (if the folder to be created does already exist) and/or <p:http-request/> (when trying to get a non-existing file). As it is not totally clear which behaviour is expected, we have not incorporated a solution in our prototype of the library, but you can easily think of a private or in-house-version of this library declaring a step to enforce the behaviour your pipelines rely on.

Another important aspect of our solution is that it copes with the (hopefully) transient need for this kind of library. Suppose the working group decides to make something like <cx:message/> or <mod:report/> part of the standard library as a step in the XProc namespace, then all you have to do is to replace the subpipeline with a call of the newly introduced step. Or: You might run a pipeline over all of your pipelines to replace every call to <iop:message/> with a call to the new step. So our solution does not only work for the moment, but can also be adapted to future developments.

Of course you have to change your pipeline currently running successfully on one processor in order to use our solution. You have to insert a <p:import/> to make our library visible and you have to change the respective step names to the library-declared steps. Here our second pipeline, called "interoperator.xpl" comes into play. If you develop a new pipeline from scratch you will probably not need "interoperator.xpl", but if you want an existing pipeline to be interoperable, you can use it to solve your problems. So what does Interoperator do?

Interoperator relies on the fact that every XProc pipeline is an XML document. Therefore we can use XProc's technology to make XProc pipelines interoperable. Our pipeline will request a pipeline's URI as an option and do all the necessary changes in this pipeline to make it run with *XML Calabash* and *MorganaXProc*. And of course it will not only change the pipeline itself but also all pipelines and libraries imported. So you just have to call Interoperator once with the top most pipeline of your project and what you get as a result is an interoperable pipeline system running on both processors.

We will not bore you discussing the pipeline step by step so let us just sum up the tasks to do:

- Make import of *XML Calabash*'s extension library conditional, so it is only used when the processor is actually running.
- Add attribute "mox:depends-on" to every step that has an attribute "cx:depends-on" and vice versa.
- Rename every step <cx:message/> to <iop:message/>
- Rename every step <cx:eval/> to <iop:eval/>
- Add <p:import/> for "xproc-iop.xpl" if its needed.
- Add <p:import/> for "http://exproc.org/proposed/steps/os" (conditional import when running *MorganaXProc*) provided a step from the library is used.
- Add <p:import/> for "http://exproc.org/proposed/steps/file" (conditional for *MorganaXProc* only), provided a step from this library is used.
- Add <p:import/> for "http://exproc.org/proposed/steps" (conditional for *MorganaXProc* only), provided a step from this library is used.
- Make sure all EXProc.org-steps are used with their "exproc.org"-namespace. This step is necessary because *XML Calabash* offers proprietary namespaces for the File and the OS libraries.[1] Since the "xmlcalabash.com/xxx" and the "exproc.org/…/xxx" namespaces contain the same step declaration, one can either rename the steps with a <p:rename/> or rename the whole namespace (<p:namespace-rename/>).
- Rename steps <cx:zip/> to <pxp:zip/>, <cx:unzip/> to <pxp:unzip/> and <cx:nvdl/> to <pxp:nvdl/>. This is necessary because for convenience reasons *XML Calabash* allows these steps also to be used with the "Calabash extension namespace". This is handy when you actually use *XML Calabash* but restrains interoperability of the respective pipelines.

This is a pretty long list, but remember: You have to call *Interoperator* only once for every pipeline (or pipeline system since imports are respected) and you get a pipeline runnable on both processors. So you do not have to worry about loss of performance. The pipeline will only be increased by a few lines for the additional import statements that are only used when the respective XProc processor is in operation.

We will release both pipelines on github[2], so anyone can use it for her/his XProc projects. It will also serve as a

[1] *See* the declarations in http://xmlcalabash.com/extension/steps/library-1.0.xpl

[2] https://github.com/xml-project

basis to enhance interoperability of XProc pipelines. So anyone who finds other obstacles to interoperability may contribute by creating an issue or better by sending a pull request with an improved version of the pipelines. This is of course not only restricted to the task of making pipelines from *XML Calabash* runnable on *MorganaXProc* and vice versa, but does also apply to other XProc processors. And the two pipelines also may serve an additional purpose: In some aspects the particular state of the two pipelines at any given time might be taken as an indicator for the state of interoperability, so they may also be seen as a tool to document the obstacles to interoperability of XProc pipelines.

8. Conclusions from our projects

We dug deep into the inner life of XProc as a technology, examined aspects of the anatomy of two XProc processors and we got two XProc pipelines helping us to make other XProc pipelines interoperable. What lessons are to be learned for migration projects in particular and what about our starting point, the question of interoperability of XProc pipelines?

If you are a pipeline author who wants or has to develop interoperable XProc pipelines, there are a lot of conclusions to be drawn: First, probably as always, there is the KISS-principle, in our case spelled out as: keep it standard, stupid. If you can solve your problems by writing XProc pipelines which only rely on the standard library and which only use those features marked in the recommendation as “required”, you are pretty safe. If these restrictions do not work for you, we showed obstacles you might run into. The two pipelines we developed for our project, both document the difficulties to be expected and serve as a tool to cope with the problems of interoperability. So if we translate our opening question into the question, whether it is possible to develop a complex and interoperable XProc pipeline system, the answer is obviously “yes”. We proved it by successfully migrating *transpect* from *XML Calabash* to *MorganaXProc*.

From this, we can obviously also proclaim good news for pipeline users and technology decision makers: Our two pipelines should in most cases solve the problem of taking pipelines from one processor to the other without deeper skills in XProc or actually, without any knowledge at all. So as a pipeline-user you have the freedom of choice: If you invest a little work, you can use your XProc pipeline with any processor you like. And finally: If you are a technology decision maker taking into consideration using XProc, you do not need to worry about vendor independence and reusability of pipelines. There are minor obstacles, but they are easy to overcome.

And what lessons are to be learned for the XProc community? As we have shown, even very complex pipeline system can be transformed to be interoperable. The Working Group and everybody involved in the process of developing XProc did a great job. But as we saw also, there are some things left to be done. The sheer necessity of the two XProc pipelines to make *transpect* work on the two XProc processors shows we are not completely finished with making XProc a fully useful *and* interoperable language.

Bibliography

[1] *XProc. An XML Pipeline Language*. 11th May 2010. World Wide Web Consortium (W3C).
http://www.w3.org/TR/xproc/

[2] Private communication by XML Prague 2016 participants involved in XProc development and specification.

[3] *W3C Technical Report Development Process*. 14 October 2005. World Wide Web Consortium (W3C).
http://www.w3.org/2005/10/Process-20051014/tr.html

[4] Norman Walsh. *Wiki editing with XProc*. 07th March 2010.
http://norman.walsh.name/2010/03/07/wikiEdit

[5] Norman Walsh. *XML Calabash Reference*. 09 June 2015.
http://xmlcalabash.com/docs/reference/

[6] James Fuller. *Diaries of a desperate XProc Hacker*. Managing XProc dependencies with depify.
XML London 2015.
doi:10.14337/XMLLondon15.Fuller01

[7] *Packaging System. EXPath Candidate Module 9 May 2012*.
http://expath.org/spec/pkg

[8] Florent Georges. *EXPath Packaging System: the on-disk repository layout*. 15 November 2009.
http://fgeorges.blogspot.de/2009/11/expath-packaging-system-on-disk.html

[9] Tim Berners-Lee, Roy Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. The Internet Society. January 2005.
https://www.ietf.org/rfc/rfc3986.txt

Using XForms to Create, Publish, and Manage Linked Open Data

Ethan Gruber

American Numismatic Society

<gruber@numismatics.org>

Abstract

This paper details the numismatic thesaurus, Nomisma.org, and its associated front-end and back-end features. The application's architecture is grounded in XML technologies and SPARQL, with XForms underpinning the creation, editing, and publication of RDF. Its public user interface is driven by the XML Pipeline Language in Orbeon, enabling transformation of RDF/XML and SPARQL XML responses into a wide array of alternative serializations, driving geographic visualizations and quantitative analyses in other digital numismatic projects.

Keywords: Semantic Web, XForms, Numismatics

1. Introduction

Nomisma.org is a collaborative project to define the intellectual concepts of numismatics following the principles of Linked Open Data: URIs for each concept with machine-readable serializations (RDF/XML, Turtle, JSON-LD, etc.) conforming to a variety of relevant ontologies (like SKOS: the Simple Knowledge Organization System) [SKOS]. What began as a prototype created in 2010 by Sebastian Heath and Andrew Meadows (then of New York University and the American Numismatic Society, respectively) to demonstrate the potential of applying semantic web technologies to numismatic research has evolved into the standard thesaurus for the discipline, driven by a scientific committee of scholars and information technologists, and adopted by a growing number of cultural heritage institutions. These Nomisma-defined concepts, and the software architecture built upon them, are the backbone for projects such as Coinage of the Roman Republic Online (CRRO) and Online Coins of the Roman Empire (OCRE), which seek to define all typologies of the Roman Republic and Empire, facilitating the aggregation of coins from museum and archaeological databases that are related to these typologies.

2. Numismatic Concepts as Linked Open Data: A Brief Introduction

Numismatics as a discipline emerged during the Medieval period and gradually became more scientific over the centuries. By the late 18th century, the classification methodology had evolved into system still used today [GRUBER]. Coins have historically been categorized by a variety of individual attributes: the manufacture process, material, monetary denomination, production place (or mint), date, entities responsible for issuing the coin (whether individual rulers or corporate organizations), and the iconography and inscriptions (or "legend" in numismatic terminology) on the front and back (obverse and reverse) of the coin. The combination of each of these individual attributes comprised a coin "type," and types were often uniquely numbered, thematically organized, and published in volumes of printed books. For example, Roman Republican coins have been published in numerous volumes over the last century, but the standard reference work for the period remains Michael Crawford's 1974 publication, Roman Republican Coinage (RRC). Collections of Republican coins therefore refer to standard type numbers from RRC, e. g., 244/1, a silver denarius minted in Rome in 134 B.C. These numbers were once printed in collection inventories or cards associated with each coin, but are now inserted into bibliographic fields in museum databases.

These databases, however, are authored in the native language of the collection. The Roman emperor, Augustus, is the same entity as Auguste in French or アウグストゥス in Japanese. In order to perform large-scale analyses of related coins across many different databases, each with its own terminology, the discipline needed to rethink authority control, and so the Linked Open Data approach to taxonomies was adopted.

Augustus could be represented by a URI, http://nomisma.org/id/augustus (with a CURIE of nm:augustus), defined as a foaf:Person in the Friend of a Friend ontology [FOAF]. This URI serves both as a unique, language-agnostic identifier for the entity, but also a web page where both human- and machine-readable information can be extracted. Below we discuss the RDF data models that comprise the Nomisma information system, organized into three broad divisions: concepts in the Nomisma.org thesaurus, coin types published in OCRE, CRRO, and other projects, and the model that defines physical coins.

2.1. The Data Models

2.1.1. Thesaurus

We implement a variety of data models for different types of data objects, mixing and matching classes and properties from numerous ontologies. The SKOS ontology was implemented for modeling the intellectual concepts of numismatics, which include not only the rulers responsible for issuing coinage, but each of the aforementioned categories: manufacture method, material, mint, denomination, field of numismatics (broad cultural areas, like Greek or Roman), and many others. Many of these categories are specific to the discipline, and are therefore defined by classes in a numismatic ontology (http://nomisma.org/ontology#, prefix: nmo) developed and published by the Nomisma.org scientific committee.

Other more generalizable types of data objects are bound to classes from other common ontologies. People and organizations carry foaf:Person and foaf:Organization classes, respectively, historical periods are defined by CIDOC-CRM, a conceptual reference model from the cultural heritage domain [CRM], the W3C Org ontology [ORG] has been implemented for defining the role a person plays with respect to the issuing of coinage (e.g., Roman emperor, king, or magistrates of various titles).

With respect to RDF properties, preferred labels and definitions may be inserted in as many languages as necessary to facilitate multilingual interfaces, and concepts may be linked hierarchically via skos:broader. And, importantly, Nomisma uses SKOS properties like exactMatch to link to identical concepts in other linked data systems (such as Geonames, OCLC's Virtual International Authority File (VIAF), Wikidata, and the Getty vocabularies), which enable the integration of coins into a wider array of cultural heritage projects, such as Pelagios Commons. While typical SKOS properties are implemented within instances within Nomisma, properties from other ontologies are implemented conditionally upon the class of object. Mints and regions may bear coordinates from the W3C basic geo (WGS84) vocabulary [WGS84] or from the UK's Ordnance Survey ontology in the form as geoJSON [OSGEO].

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix nm: <http://nomisma.org/id/> .
@prefix nmo: <http://nomisma.org/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

nm:byblus a nmo:Mint;
        rdf:type <http://www.w3.org/2004/02/skos/core#Concept>;
        dcterms:isPartOf <http://nomisma.org/id/greek_numismatics>;
        skos:prefLabel "Byblus"@en,
                "Byblos"@fr;
        skos:definition "The mint at the ancient site of Byblus in Phoenicia."@en;
        skos:closeMatch <http://dbpedia.org/resource/Byblos>,
                <http://pleiades.stoa.org/places/668216>,
                <http://www.geonames.org/273203>,
                <http://collection.britishmuseum.org/id/place/x30547>,
                <http://vocab.getty.edu/tgn/7016516>,
                <http://www.wikidata.org/entity/Q173532>;
        geo:location <http://nomisma.org/id/byblus#this>;
        skos:broader <http://nomisma.org/id/phoenicia>;
        skos:altLabel "Byblos"@en.

nm:byblus#this a geo:SpatialThing;
        geo:lat "34.119501"^^xsd:decimal;
        geo:long "35.646846"^^xsd:decimal;
        dcterms:isPartOf <http://nomisma.org/id/phoenicia#this>
```

The Org ontology has been applied to connect people with dynasties and corporate entities, including their roles within these organizations and dates these offices have been held. Dublin Core Terms such as dcterms:isPartOf and dcterms:source have been applied for hierarchical linking and bibliographic references, respectively. The thesaurus models are stable, but do evolve to meet increased demands by our users. At the moment, the system is deficient in tracking data provenance, and we do plan to implement PROV-O soon [PROVO].

2.1.2. Coin Types

Just as individual concepts have been defined by URIs, so too are more complex coin types. RRC 244/1 is represented by http://numismatics.org/crro/id/rrc-244.1, an instance of an nmo:TypeSeriesItem in the Nomisma ontology, which contains a variety of properties connecting the type to individual categorical attributes (concepts in the Nomisma thesaurus) and literals for the obverse and reverse legends, symbols, and iconographic descriptions. Below is the RDF/Turtle representing RRC 244/1:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix nmo: <http://nomisma.org/ontology#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix nm: <http://nomisma.org/id/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

<http://numismatics.org/crro/id/rrc-244.1>
        rdf:type <http://www.w3.org/2004/02/skos/core#Concept>;
        skos:prefLabel "RRC 244/1"@en;
        skos:definition "RRC 244/1"@en;
        dcterms:source nm:rrc;
        nmo:representsObjectType nm:coin;
        nmo:hasManufacture nm:struck>;
        nmo:hasDenomination nm:denarius;
        nmo:hasMaterial nm:ar;
        nmo:hasIssuer nm:c_abvri_gem_rrc;
        nmo:hasMint nm:rome;
        nmo:hasStartDate "-0134"^^xsd:gYear;
        nmo:hasEndDate "-0134"^^xsd:gYear;
        nmo:hasObverse <http://numismatics.org/crro/id/rrc-244.1#obverse>;
        nmo:hasReverse <http://numismatics.org/crro/id/rrc-244.1#reverse>.

<http://numismatics.org/crro/id/rrc-244.1#obverse>
        nmo:hasLegend "GEM X (crossed)";
        dcterms:description "Helmeted head of Roma, right. Border of dots."@en;
        nmo:hasPortrait <http://collection.britishmuseum.org/id/person-institution/60208>.

<http://numismatics.org/crro/id/rrc-244.1#reverse>
        nmo:hasLegend "C·ABVRI";
        dcterms:description "Mars in quadriga, right, holding spear, shield and reins in left hand
          and trophy in right hand . Border of dots."@en;
        nmo:hasPortrait <http://collection.britishmuseum.org/id/person-institution/59284>.
```

2.1.3. Physical Coins

The RDF model for physical coins implements properties from several ontologies, but the Nomisma ontology is the most prominent. The complexity of the model is variable depending on the condition of the coin ands the certainty by which it may be linked to a URI for a coin type. If the type RRC 244/1, http://numismatics.org/crro/id/rrc-244.1, is an nmo:TypeSeriesItem, then physical specimens that represent this typology are linked with the nmo:hasTypeSeriesItem property. By semantic reasoning, a physical coin of the type RRC 244/1 is a denarius minted in Rome and issued by the magistrate, C. Aburius Geminus, even if these URIs are not explicit within the RDF of http://numismatics.org/collection/1978.64.316. On the other hand, many archaeologically excavated coins are worn beyond certain attribution. In these cases, what information that may be ascertained are made explicit in the triples for a coin. The portrait may be identifiable as the emperor, Vespasian, but an illegible legend prevents linking to a specific coin type. This coin is still available for query alongside other positively identified coins of Vespasian within the Nomisma.org SPARQL endpoint.

In addition to recording the type URI or typological characteristics of a coin, metrological data, like weight and diameter, may also be included. These measurements are vital for large-scale (both in terms of time and space) economic analyses. If the coin has been photographed, foaf:thumbnail and foaf:depiction may be used to link to images. Finally, geodata (whether coordinates and/or a gazetteer URI) related to the find spot may be included, if known. In the event that the coin was found within a hoard (`dcterms:isPartOf ?hoard`), the find coordinates may be extracted from the triples for the hoard via the graph [NMDOCS].

2.2. Gradual Acceptance and Implementation by the Community

A growing body of institutions are adopting Nomisma URIs as standard identifiers for numismatics. The four of the five largest collections of ancient coins in world (American Numismatic Society, Bibliothèque nationale de France, British Museum, and Berlin Münzkabinett) are in various stages of implementation, as are a handful of European coin find databases (such as the UK's Portable Antiquities Scheme and the German Antike Fundmünzen Europa), and smaller museums such as the University of Virginia and University College Dublin. For a full list of contributors, see the Nomisma.org datasets list.

The landscape of digital numismatics has progressed significantly in the six years since the launch of Nomisma, and we are hopeful that, over the next decades, this information system will include millions of coins, enabling scholars to perform large-scale economic analyses of 800 years of Roman coinage distributed from Scotland to India. We hope (or expect) these technical methodologies will open pathways for similar research questions in other fields of numismatics. The foundation for these sophisticated query interfaces is a combination of XML and Semantic Web technologies, bound together in a collection of XML Pipelines, XSLT stylesheets, and Javascript/CSS for the public user interface, all of which are open source (on Github) and built on other open source server applications and open web standards.

3. Architecture

Nomisma.org's architecture is based on an adaptation of XRX (XForms, REST, XQuery), with SPARQL substituted for XQuery. It utilizes a variety of well-supported and documented open source Java-based server applications. Apache Solr facilitates faceted browsing, search-oriented web services, and a geospatial extension to the SPARQL endpoint. Apache Fuseki (part of the Jena project) is the triplestore and SPARQL endpoint. Orbeon XForm s is the core middleware that connects Solr, Fuseki, and external REST services—in both the front-end with XML pipelines and the back-end XForms engine. The SPARQL endpoint was deployed in production in early 2013, enabling us to aggregate Roman Imperial coins from the American Numismatic Society and Berlin Münzkabinett in an early version of OCRE (which contributed to a successful three-year, $300,000 bid with the National Endowment for the Humanities the following year). Orbeon was implemented before this–in early 2012—to deal with challenges surrounding the creation and maintenance of Nomisma IDs.

The architecture is modular. The greatest advantage of XForms, a W3C standard, is that the author may focus solely on the MVC functionality of the web form; the client-side Javascript and CSS and server-side code are inherent to the XForms processor, and so migration from one platform to another should, at least in theory, require little effort. Apache Fuseki was chosen for its ease of deployment, but it may be swapped with any SPARQL 1.1-compliant endpoint. Replacing Solr with another search index would be more difficult, but any application that supports REST interactions via the XForms engine is suitable.

3.1. XForms for CRUD operations of RDF

When Nomisma.org launched in 2010, it was published entirely in Docuwiki, an open source wiki framework. The URI for the mint of Rome, http://nomisma.org/id/rome, was created by hand-editing XHTML embedded with RDFa properties in a single textarea element in an HTML web form. This presented some problems: there was no check for well-formedness of XHTML fragments, so malformed XML could break a web page, and the particular XHTML document model might be inconsistent and not translate into the the appropriate RDF model using the W3C's RDFa 1.1 distiller.

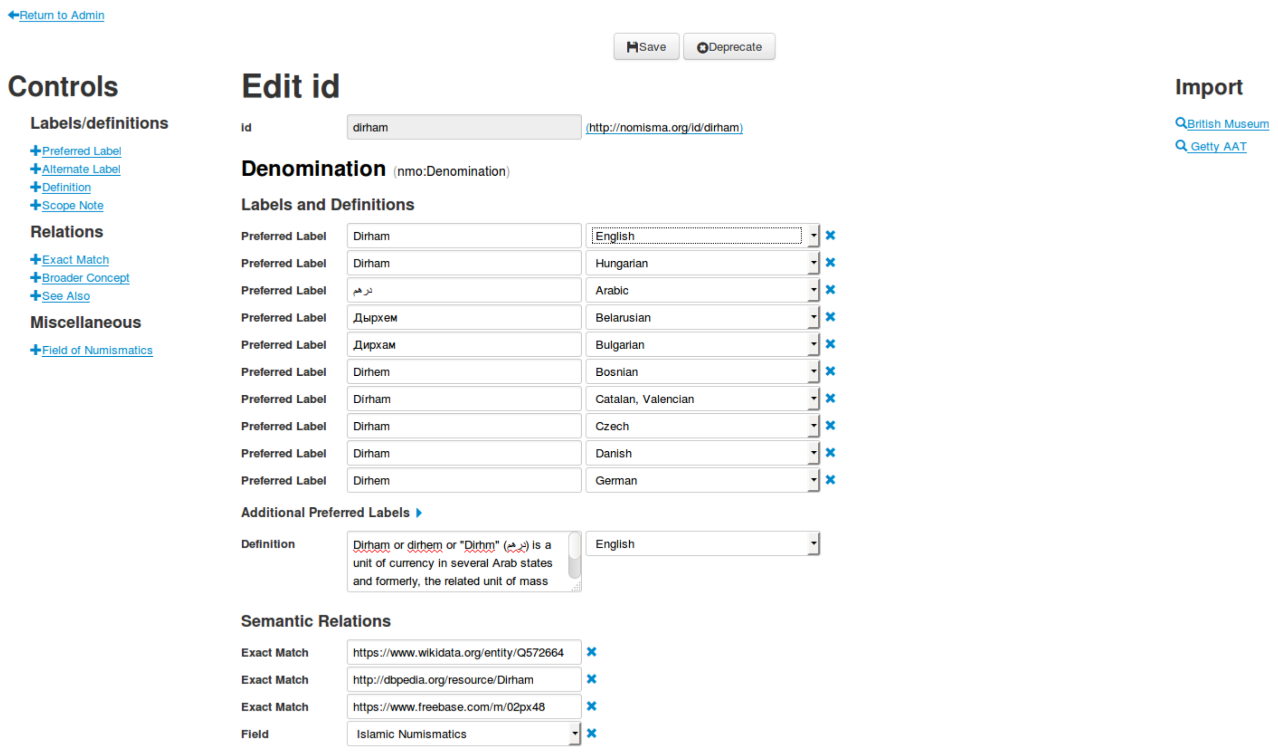
Ensuring data consistency was a primary concern, and so we developed an XForms application to handle the editing of these XHTML+RDFa fragments. By curbing direct human editing of XTHML, we eliminated malformed XML problems outright. XForms bindings enabled stricter validation based on XPath, for example:

- To require one English SKOS Preferred Label and Definition
- Restrict labels and definitions in other languages to a maximum of one
- Latitudes and longitudes must be decimal numbers between -180 and 180
- A variety of conditionals that restrict certain RDF properties to particular classes of data object; e. g., that geographic coordinates apply only to skos:Concepts that represent regions or mints

Since linking Nomisma URIs to concepts in other systems is a vital feature of Five-Star Linked Open Data, we implemented a variety of simple lookup mechanisms that pass search keywords from the web form to XForms submissions in order to query external APIs (for example, of the Getty Vocabularies and British Museum SPARQL endpoints, Wikidata's XML response, or VIAF's RSS feed). These widgets simplified the process by which skos:exactMatch or skos:closeMatch URIs might be integrated into Nomisma data, reducing errors in manually transcribing URIs into the web form. Wikidata's REST API is especially useful, as we are able to extract article titles in many languages to rapidly assemble a list of skos:prefLabels (Figure 1, "Creating nm:dirham, after importing from Wikidata"). These lookup mechanisms are authored in the form of XBL components in Orbeon, and can be (and have been) easily reimplemented in other XForms applications, such as xEAC and EADitor , which are archival publishing frameworks.

In addition to external lookups, the editing interface includes internal lookup mechanisms that query

Figure 1. Creating nm:dirham, after importing from Wikidata



Edit id

id	dirham	(http://nomisma.org/id/dirham)

Denomination (nmo:Denomination)

Labels and Definitions

Preferred Label	Dirham	English
Preferred Label	Dirham	Hungarian
Preferred Label	درهم	Arabic
Preferred Label	Дырхем	Belarusian
Preferred Label	Дирхам	Bulgarian
Preferred Label	Dirhem	Bosnian
Preferred Label	Dirham	Catalan, Valencian
Preferred Label	Dirham	Czech
Preferred Label	Dirham	Danish
Preferred Label	Dirhem	German

Additional Preferred Labels

Definition	Dirham or dirhem or "Dirhm" (درهم) is a unit of currency in several Arab states and formerly, the related unit of mass	English

Semantic Relations

Exact Match	https://www.wikidata.org/entity/Q572664
Exact Match	http://dbpedia.org/resource/Dirham
Exact Match	https://www.freebase.com/m/02px48
Field	Islamic Numismatics

Nomisma's own SPARQL endpoint or Atom feed to link to broader concepts, to associate a concept with a field of numismatics (e.g., to say that a denarius is part of Roman numismatics; `nm:denarius dcterms:isPartOf nm:roman_numismatics`), or to associate a person with a dynasty or organization.

Upon clicking the 'Save' button, the data model was serialized into an XML file and written to the filesystem (as opposed to an XML or NoSQL database), in order that commits could be made nightly to a Github repository for our data [NMDATA]. The Github backups remain an integral part of the data publication workflow today.

The framework eventually grew to incorporate Solr and a SPARQL endpoint. The 'Save' button then hooked into several additional XForms actions and submissions. The source model (XForms instance) is transformed by XSLT into an XML document conforming to Solr's ingestion model, and posted into Solr's REST interface. Next, the XForms engine constructs a SPARQL query to purge triples associated with the URI from the endpoint, which is posted to the endpoint via the SPARQL/Update protocol. Finally, the updated RDF is posted into the endpoint.

In 2014, we begin the transition of migrating from Heath and Meadows' original XHTML+RDFa document model into RDF/XML that conforms proper RDF ontologies, including the introduction of the formal Nomisma ontology developed by Karsten Tolle, a computer scientist at the University of Frankfurt. The model was separated from the view, making it easier to extend the functionality of the public user interface without interfering with the RDFa distillation or exports into other serializations of linked data, like Turtle or JSON-LD. The APIs were rewritten and made dramatically more efficient, having eliminated the need to preprocess XHTML+RDFa into RDF/XML before delivering content to OCRE or other projects.

The XForms editing interface was adapted to RDF/XML, which led to improved consistency of the data, since our export APIs (XSLT transformations of XHTML into other formats) did not always account for every imaginable permutation of RDFa properties jammed into a document model. Today, Nomisma.org's APIs serve out large quantities of RDF data to other American Numismatic Society or partner projects for web maps, multilingual interfaces, etc., as efficiently as possible.

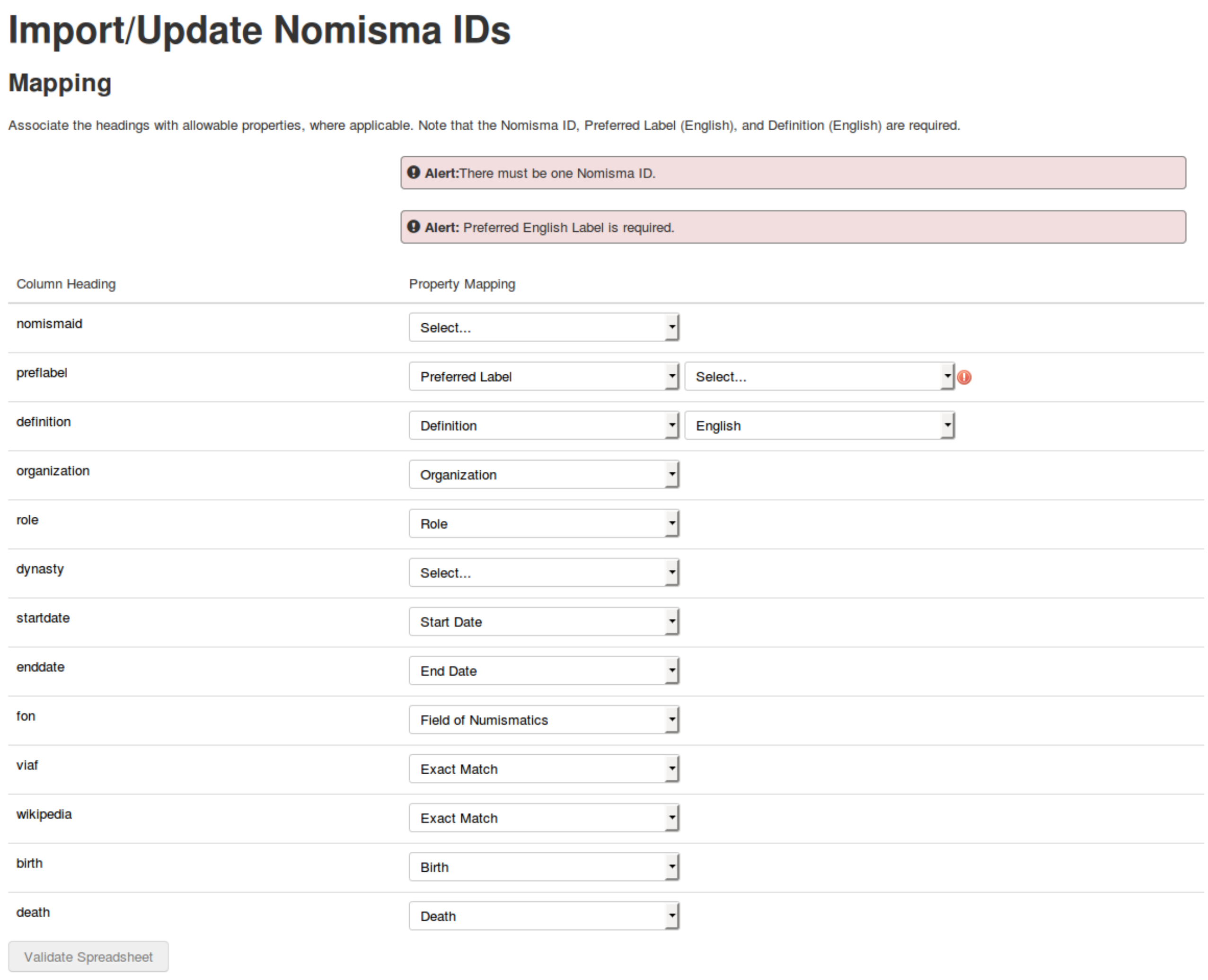
3.2. Batch Concept Creation

Nomisma's XForms back-end functions well for managing individual skos:Concepts. It is an intuitive system used primarily by academics to create or update identifiers required for existing digital numismatics projects. When creating new projects, however, we had a need to create potentially hundreds of new identifiers for mints, rulers, denominations, etc. Academics are accustomed to working in spreadsheets, and the author was often tasked with writing one-off PHP scripts to read spreadsheets as CSV and process data into RDF.

In summer 2015, this functionality was ported into a new XForms application [NMBATCH]. This application requires a Google spreadsheet that conforms to some basic requirements, which must be published to the web so that it can be made available as an Atom feed through the Google Sheets API. All IDs in the spreadsheet must be the same class of data object (for example, a mint or denomination). Each column heading will be read from Atom in the XForms engine, and the user may choose to map a column to a permitted list of RDF properties (Figure 2, “Mapping spreadsheet columns to RDF properties”). There are some XPath controls on the properties that are available for selection—there must be an English preferred label and definition, latitude and longitude are only available for mints, and other conditions described in the Github wiki. If the mapping itself is valid, the user may click a button to validate each row of the spreadsheet. The XForms engine ensures that latitudes and longitudes are valid decimal numbers, that each row has a preferred label and definition, that a URI under a skos:broader mapping does conform to the appropriate RDF Class. After this phase of validation, the application will display a list of validation errors, or if there are none, present the user with a button to publish the data into Nomisma. The publication workflow transforms each Atom entry element into RDF/XML, writes and updates the file on the disk, publishes the data to Solr, and creates or updates the triples in the SPARQL endpoint.

The publication process includes an additional feature where any SKOS matching property with a Wikipedia URL or DBpedia URI is parsed, and a series of API calls are executed to extract preferred labels and alternative authority URIs from Wikidata. This feature has enabled us to enhance concepts that have already been published; we can execute a SPARQL query of all Nomisma IDs with a DBpedia URI to download CSV, upload the CSV to Google Sheets, and then re-run the spreadsheet through the XForms import mechanism to pull labels and other URIs from Wikidata into Nomisma. This is

Figure 2. Mapping spreadsheet columns to RDF properties



Import/Update Nomisma IDs

Mapping

Associate the headings with allowable properties, where applicable. Note that the Nomisma ID, Preferred Label (English), and Definition (English) are required.

Alert: There must be one Nomisma ID.

Alert: Preferred English Label is required.

Column Heading	Property Mapping	
nomismaid	Select...	
preflabel	Preferred Label	Select...
definition	Definition	English
organization	Organization	
role	Role	
dynasty	Select...	
startdate	Start Date	
enddate	End Date	
fon	Field of Numismatics	
viaf	Exact Match	
wikipedia	Exact Match	
birth	Birth	
death	Death	

one reason projects like OCRE and CRRO are available in Arabic.

3.3. Public User Interface

The front-end of Nomisma.org is delivered through Orbeon's Page Flow Controller and XML Pipeline Language (XPL). URIs for concepts are constructed by a pipeline that aggregates the source RDF file from the filesystem with, and, depending on whether the concept is mappable, two SPARQL query responses to ascertain whether there are mints or findspots connected with the skos:Concept. This aggregate XML model is passed through XSLT in order to generate an HTML5 web page. The URI for Julius Caesar, http://nomisma.org/id/julius_caesar, therefore includes the RDF transformed into HTML5+RDFa, plus a map rendered with the Javascript library, Leaflet, which shows a layer for mints that struck coins issued by Casesar, a heatmap showing the geographic distribution of all locations where coins of Caesar have been found, and two additional layers (off by default) that show points for hoards (three or more coins found in the same archaeological context) or individual coin finds. These layers are generated by Nomisma APIs that interface with the SPARQL endpoint, passing the XML response from the endpoint through a pipeline to transform it into geoJSON. Additionally, when available, the URI web page will display a list of coin types associated with the skos:Concept, including thumbnails of coins from partner institutions.

The browse page and Atom feed are both generated by XPL which send request parameters to Solr's REST interface, and pipe the XML response through XSLT into the appropriate view (HTML or Atom). There are additional pages for the current and previous versions of the formal ontology, APIs, documentation, and the SPARQL endpoint. We strive to make data available in as many formats through as many protocols as possible. A user may request alternative serializations by appending an extension on a concept URI, e. g., http://nomisma.org/id/augustus.jsonld, to receive JSON-LD by REST, but concept URIs, the browse page, ontology URI, and SPARQL endpoint offer interactivity by Content Negotiation. Content Negotiation is vital for conforming to linked data framework standards, and these advanced HTTP features are possible with Orbeon's XPL.

4. Results

According to a 2014 survey by OCLC, Nomisma.org is one of the most heavily-used Linked Open Data systems in the Library, Archive, and Museum sector, behind OCLC's own VIAF and Worldcat.org services (100,000+ requests per day), and in the same range as the British Library's British National Bibliography service, at between 10,000-50,000 requests per day [OCLC]. Nomisma's load has doubled since then, serving between 40,000-50,000 API calls per day (including about as many SPARQL queries, though there is some overlap between API and SPARQL requests), nearly all of which are from non-search robot machines that facilitate dynamic mapping and multilingual interfaces in a wide variety of numismatic projects. The architecture uses both SPARQL and Solr to their natural advantages while minimizing SPARQL's known scalability limitations. Nomisma.org has suffered approximately three minutes of downtime in three years, which is remarkable considering it runs on a Rackspace cloud server with only 4 GB of RAM and an annual budget of little more than $1,000.

We are aware that we will likely need to upgrade our infrastructure eventually, as our triplestore will grow exponentially in the coming years as more institutions become involved in the numismatic linked data cloud. We strive to continually build more sophisticated query and visualization interfaces that, in turn, require greater server resources. Numismatics has technologically evolved dramatically over the last half-decade, and its march toward Big Data is inevitable, finally making it possible to conduct research in ways that have been dreamed about for centuries.

Bibliography

[WGS84] Dan Brickley, ed. W3C Semantic Web Interest Group: Basic Geo (WGS84 lat/long) Vocabulary. 2003. World Wide Web Consortium (W3C).
https://www.w3.org/2003/01/geo/

[FOAF] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.99. 14 January 2014.
http://xmlns.com/foaf/spec/

[CRM] Nick Crofts, Martin Doerr, Tony Gill, Stephen Stead, and Matthew Stiff, eds. Defintion of the CIDOC Conceptual Reference Model. November 2011. ICOM/CIDOC Documentation Standards Group.
http://www.cidoc-crm.org/html/5.0.4/cidoc-crm.html

[GRUBER] Ethan Gruber. Recent Advances in Roman Numismatics. 15 May 2013. MA Thesis, University of Virginia.
doi:10.5281/zenodo.45328

[NMDOCS] Ethan Gruber. How to Contribute Data.
http://nomisma.org/documentation/contribute

[NMBATCH] Ethan Gruber. Import/Update IDs. 28 July 2015.
https://github.com/nomisma/framework/wiki/Import-Update-IDs

[PROVO] Timothy Lebo, Satya Sahoo, and Deborah McGuinness, eds. PROV-O: The PROV Ontology. 30 April 2013. World Wide Web Consortium (W3C).
https://www.w3.org/TR/prov-o/

[SKOS] Alistair Miles and Sean Bechhofer, eds. SKOS Simple Knowledge Organization System. 18 August 2009. World Wide Web Consortium (W3C).
https://www.w3.org/2004/02/skos/

[NMDATA] Nomisma.org. Nomisma Data. Created 2012. Github.org repository.

https://github.com/nomisma/data

[OSGEO] Geometry Ontology. Ordnance Survey.
http://data.ordnancesurvey.co.uk/ontology/geometry/

[ORG] Dave Reynolds, ed. The Organization Ontology. 16 January 2014. World Wide Web Consortium (W3C).
https://www.w3.org/TR/vocab-org/

[OCLC] Karen Smith-Yoshimuri. Linked Data Survey results 1 – Who's doing it (Updated). 4 September 2014. OCLC.
http://hangingtogether.org/?p=4137

Dynamic Translation of Modular XML Documentation Using Linked Data

Simon Dew

STANLEY Black and Decker Innovations Limited

<simonjabadaw@gmail.com>

Abstract

STANLEY Black and Decker Innovations had a requirement to produce and maintain DocBook-based documentation, which is translated into up to 10 languages. Documents are built by transclusion from several source files, some of which may be stored remotely. Each document may contain SVG illustrations which also needed translation.

We selected XLIFF as a translation file format. To keep maintenance effort to a minimum, we needed tools that enabled dynamic translations, i.e. translations made at publication time without skeleton files. We also needed tools that could find the correct translations for each translatable element after all the source files (including remote source files) had been transcluded into a single document.

This article describes the solutions we developed. These included markup (mostly ITS 2.0) in the source documentation, linked data (using RDF/XML) to identify the translation resources, and a set of XSLT stylesheets to handle the transformations.

Keywords: XML, Translation, DocBook, SVG, ITS, XLIFF, Linked Data

1. Introduction

XML-based documentation often requires translation and localisation. The QRT team at STANLEY Black and Decker Innovations had a requirement to produce and maintain modular documentation in up to 10 languages. Any translated document might be published in up to 4 different brandings, for a number of different varieties of a product, in several different output formats, e.g. PDF, CHM, web help; and each document might contain illustrations which also needed translation. This article describes the solutions we developed to minimise the effort and cost required to do this.

2. Background

2.1. Translation File Format

We required an open source format to store translations. We considered two main standards: the *GNU gettext* standard and the *XLIFF* file format.

The GNU gettext standard is an internationalisation and localisation system, released by the GNU project and based on earlier work by Sun Micrososystems [gettext]. It uses text files known as Portable Objects to hold translation strings. Commonly used for user interfaces in UNIX-like operating systems.

XLIFF (XML Localisation Interchange File Format) is a data exchange standard for localisation, originally designed for passing localisation data between translation tools [XLIFF]. It's an XML-based file format, standardised by OASIS (the Organization for the Advancement of Structured Information Standards) in 2002.

We chose the XLIFF file format because:

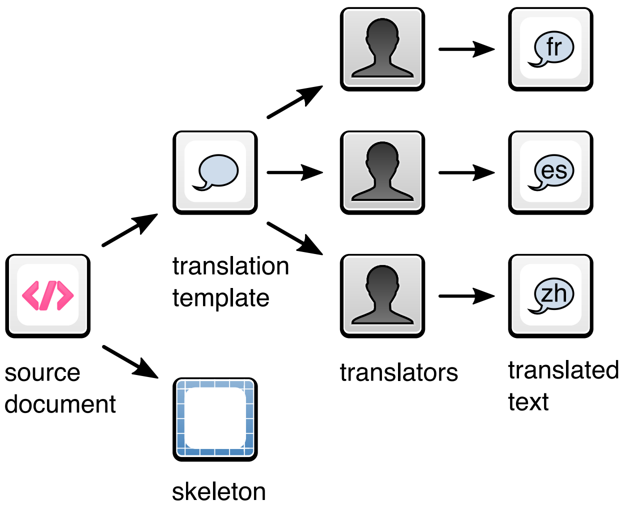
1. Commercial translation houses can use XLIFF comfortably with their proprietary translation tools;
2. XLIFF being XML-based can be transformed using XSLT [XSLT], and it made sense to use this with our DocBook XML documentation [DocBook] and SVG images [SVG].

2.2. Traditional XLIFF Workflow

The traditional XLIFF workflow demands that the source document is converted into a *skeleton file* which contains the structure of the document, and an XLIFF file which contains the translation units themselves, as shown in Figure 1, "Translation with skeleton file".

doi:10.14337/XMLLondon16.Dew01

Figure 1. Translation with skeleton file

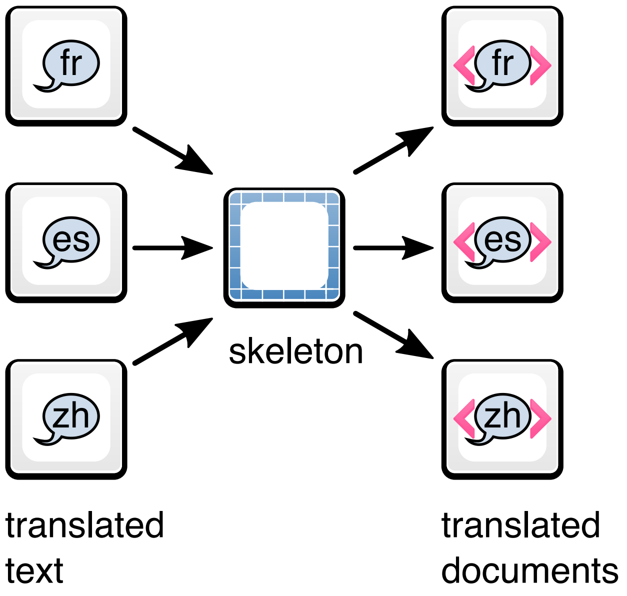


Filters store the non-translatable portions in special files called skeletons.

--[IBM]

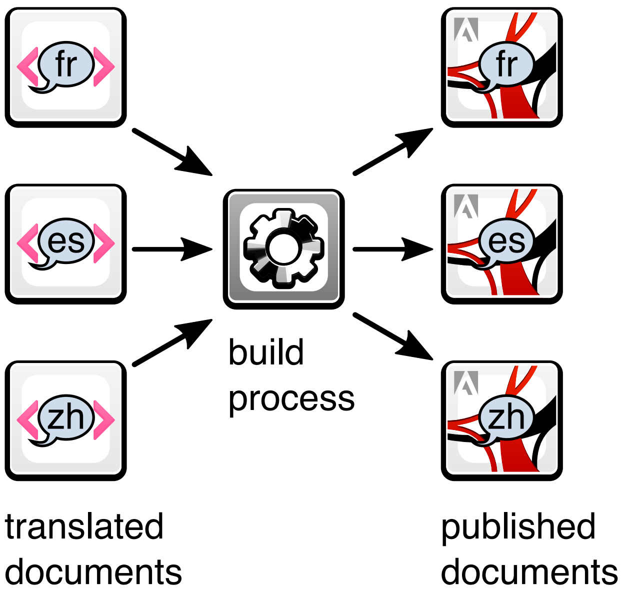
The XLIFF file is sent to the translators. After translation, the translated XLIFF files are combined with the skeleton file to create translated documents in each of the target languages, as shown in Figure 2, "Merge with skeleton file".

Figure 2. Merge with skeleton file



The translated documents are then processed to create the desired output formats, as shown in Figure 3, "Publishing translated documents".

Figure 3. Publishing translated documents



The translated XLIFF must now be merged with the skeleton file to produce a translated document in the desired output format.

--[IBM]

This means, however, that the skeleton file and the translated documents become resources which must themselves be maintained:

Translated file is checked into XML repository

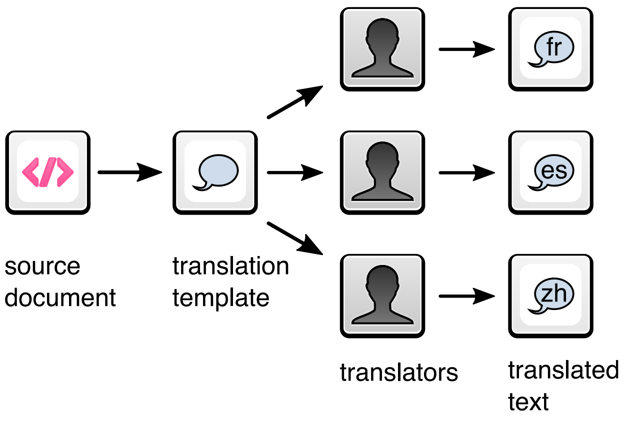
--[Tektronix]

Maintaining skeleton files and a large number of translated files can become a problem, especially when authors have to produce several different versions of the documentation, e.g. for different brandings or product variants.

2.3. Dynamic Translations

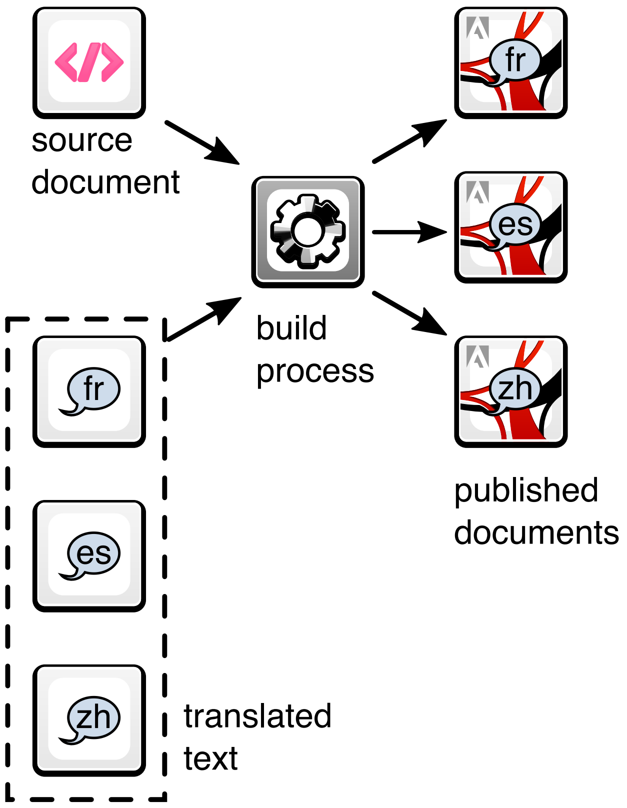
One solution to the problem of maintenance of translated documentation is to use *dynamic translations*. The translation strings are extracted from the source file, but no skeleton file is produced, as shown in Figure 4, "Translation without skeleton file".

Figure 4. Translation without skeleton file



At publication time, the build tool takes the structure of the source document and replaces the source text with the corresponding translated text, including any inline markup, as shown in Figure 5, “Publishing translated documents dynamically”.

Figure 5. Publishing translated documents dynamically



The advantage of fully dynamic translation is that elements can be moved around or deleted in the source document and these structural changes are reflected automatically in the translated publications, without the need to regenerate the skeleton file or update intermediate translated source files.

Dynamic translation is used with some GNU gettext workflows. For example, authors who write documentation in the Mallard XML format [Mallard] can use this approach to publish documentation to yelp, the GNOME help viewer [Yelp]. Authors can place translated PO files into a specific subdirectory and add instructions to a configuration file so the build process can translate the document into that language:

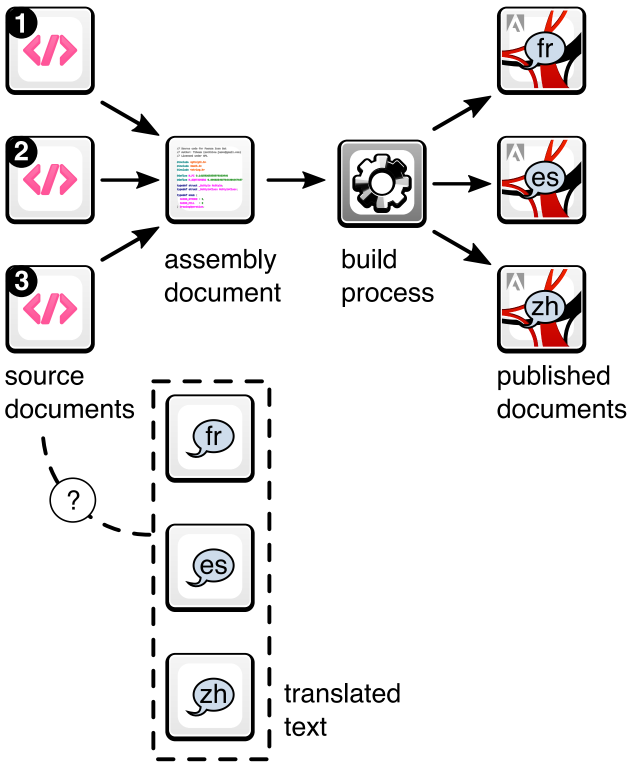
> This integrates the British English language translation with the yelp.m4 build process, so that the translations from the PO file are used to create translated Mallard pages during the make process. As with the basic integration, the translated pages are validated during make check and installed in the install directory during make install.
>
> --[Yelp]

2.4. Problem

However, we faced two problems before we could adopt this approach:

1. We weren’t aware of any solution that would enable dynamic translation using XLIFF files.
2. A further problem arises with modular documentation, i.e. documents that are built at publication time from several source files, all transcluded into the main assembly document. Some of these source files may be stored remotely. The publication process thus needs to know where to find the correct translation for every translatable element in the main document after all the document source files have been transcluded into a single document. See Figure 6, “Problem with translating modular documentation”.

Figure 6. Problem with translating modular documentation



3. Design

We designed a solution based around linked data to identify the translation files for each document, document part or image; markup in the source documentation to identify the correct translation for each translatable element; and a set of tools to handle the translation process. See Figure 7, "Dynamic translation using linked data".

3.1. Design: Linked Data

Each source document can use linked data to identify the XLIFF files that contain translations of the elements within the document. We wanted linked data to reuse established vocabularies wherever possible.

If necessary, specific parts of a document (e.g. document chunks which have been transcluded from a remote location) can use linked data to identify the XLIFF files that contain translations of the elements within that part.

Linked translation data takes the form of RDF/XML statement(s) [RDF/XML]. We chose RDF/XML because of the flexibility that the markup offers, and in the hope that we may find further ways to analyse our translations using linked data tools in future, e.g. to analyse translation coverage.

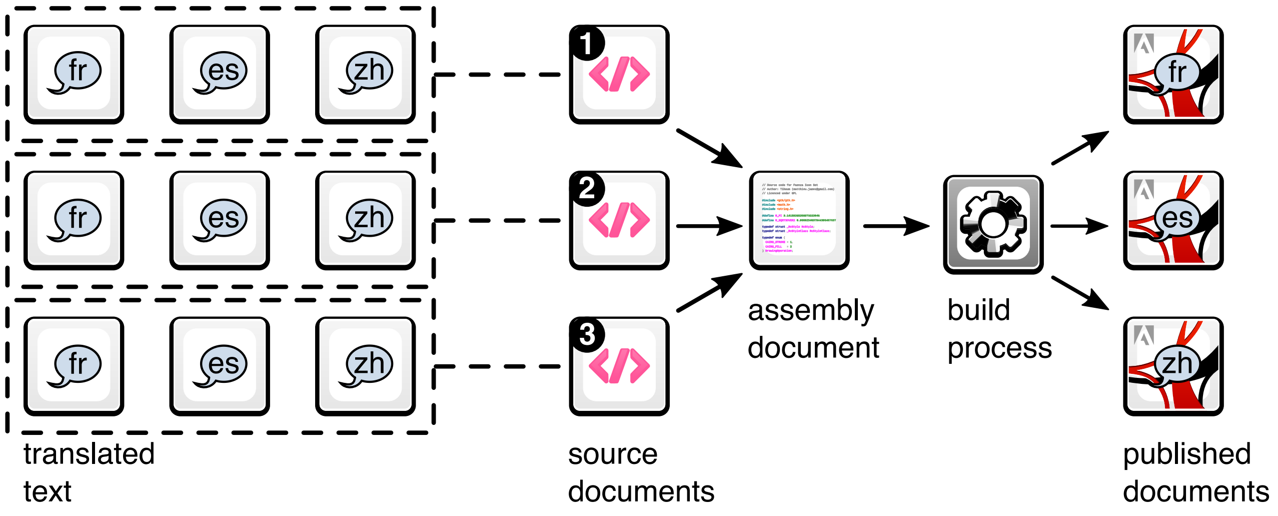
3.2. Design: Markup

Each translatable element must contain markup that identifies the corresponding translation unit in the XLIFF file.

3.3. Design: Tools

Translation will take place as part of the document build process, after transcluding the document modules into a single document, and before transforming the document into its publication format. The solution must therefore

Figure 7. Dynamic translation using linked data



include an automatic translation tool which can be used as part of the document publication build process.

For each translatable element in the document, the automatic translation tool must find the correct XLIFF file for the correct language, and replace the inline content of the element with the translated content from the XLIFF file.

The solution must include tools to prepare the source documentation for translation, i.e. by adding translation markup automatically; and to extract the translatable strings to an XLIFF file so that it can be sent to translators.

The solution must also include tools for the management of XLIFF files: for example, comparing the source file to the translation file when the source file has changed; merging completed translations into the existing translation files; and purging duplicate translation units from existing translation files.

3.4. Related Work

As stated previously, we were aware of gettext for handling dynamic translations. However, gettext is not aimed at XLIFF files, and does not address the issue of translating modular documentation.

We were also aware of the ITS Tool[1] package for maintaining translation files. This package provided useful inspiration, particularly the use of ITS (Internationalisation Tag Set) rules files to identify translatable elements [ITS]. However, this package is also aimed at gettext Portable Objects rather than XLIFF files, and does not address the issue of translating modular documentation.

The Okapi Framework[2] is a suite of tools for managing XLIFF files, particularly useful for extracting XLIFF translatable units from a wide variety of file formats. It does not handle dynamic translation, or provide any solution to the issue of translating modular documentation.

3.5. Design Decision: Mapping Elements to Translation Units

When converting from the source documentation to XLIFF, the preparation tools need to be able to determine what a translatable element is, and how the translatable elements map to XLIFF translation units.

One approach would be to convert all elements containing text nodes into translation units. However this might lead to highly segmented text, which would create difficulties for translators.

To aid translators, and to keep the implementation simple, we decided to define a reasonable set of elements for each supported file type as “block” elements. Each block element maps to a single translation unit. So for example, in DocBook XML, a `para` or a `title` element would be a block element. Child elements of block element are “inline” elements, contained within the XLIFF translation unit.

The ITS 2.0 standard provides the Elements Within Text data category, which can be used to determine which elements in a file are block elements and which are inline. We decided to define a standard ITS rules file for the file formats we use. The XLIFF preparation tools must use this ITS rules file to map the elements in the source document to translation units in an XLIFF file. The XLIFF preparation tools must be able to use different ITS rules files if necessary.

3.6. Design Decision: Inline Markup

Elements from foreign namespaces are not permitted within translated text in an XLIFF file. XLIFF supports two strategies for marking up inline elements. These may be called “raw” (escaped) markup and “cooked” (transformed) markup.

With raw markup, inline XML elements in the source document are marked up in the XLIFF file as escaped text. Escaped start and end tags can be wrapped within XLIFF `bpt` and `ept` elements; escaped standalone tags can be surrounded by XLIFF `ph` elements. The escaped markup has to be unescaped when inserting text into the translated document.

With cooked markup, inline XML elements in the source document are transformed into XLIFF elements. Inline elements with content can be transformed into XLIFF `g` elements; inline elements with no content can be transformed into XLIFF `x` elements. Since we want to do without a skeleton file, the XLIFF maintenance tools would have to store the `g` and `x` elements with enough information to be able to transform them losslessly back into the original inline elements, with the original namespace, local name and attributes. We decided that when using cooked markup, we would use the `ctype` attribute to record the original namespace and local

[1] http://itstool.org

[2] http://okapi.opentag.com

name of inline elements. XLIFF permits attributes from foreign namespaces.

We prefer using raw inline tags in our XLIFF files in-house. When using an XLIFF-capable editor such as Virtaal[3], we found it useful to see the full content of any inline markup. We decided to use raw (escaped) inline markup as the primary format for XLIFF files. However our translation houses preferred to work with cooked (transformed) inline markup. We decided that our maintenance tools must therefore be able to round-trip XLIFF files, i.e. convert an XLIFF file from one format to the other, preserving the structure of the inline elements.

3.7. Design Decision: Nested Block Elements

Our primary documentation format, DocBook XML, allows nested block elements. For example, a DocBook `para` element can contain a `footnote` element, which in turn can contain another `para` element. This could be handled by XLIFF, e.g. by representing the inner block element as an inline element, and using the `xid` attribute to refer to another translation unit. We decided however, for the sake of simplicity, not to implement this at first. We decided to enforce a local writing convention that authors could not nest `para` elements. If necessary, we would use a `simpara` element within a `para` element. The XLIFF preparation tool must regard `simpara` as an inline element.

Similarly, we decided to enforce a local writing convention that no attributes could contain translatable text.

[3] http://virtaal.translatehouse.org

4. Solution

We implemented this design as part of a broader document localisation and publication project, which was known within STANLEY Black and Decker Innovations as *PACBook*.

PACBook is released under version 3.0 of the GNU Lesser General Public License [LGPL]. It is available from the PACBook repository on GitHub at https://github.com/STANLEYSecurity/PACBook.

The parts of PACBook which implement dynamic translation for XML documents using linked data are described here. Full documentation for the entire project is available from the PACBook repository on GitHub.

4.1. Implementation: Linked Data

To be able to translate an XML document using this solution, authors must add a *translation statement* to the metadata section of the XML document. So, for a DocBook 5.0 document, you would add it to the `info` element at the start of the document. For an SVG image, you would add it to the `metadata` element at the start of the image.

We use terms from the Bibliographic Ontology (BIBO) to identify the document, document part or image that needs translation. [BIBO]

We use terms from the VIVO Integrated Semantic Framework vocabulary to specify the location of the XLIFF files which contain the translations for this document, document part or image. [VIVO]

4.1.1. Linked Data for a Document

A typical translation statement for a DocBook XML document is shown in Example 1, “Translation statement for a DocBook document”.

- The `bibo:Document` property declares that this file is a document and that it is the subject of this statement.
- The `vivo:hasTranslation` property declares that this document has translations associated with it. The optional `xml:base` attribute can be used to indicate where all the translations are.
- The translation resources are indicated by the `rdf:li` elements. For each translation, the `xml:lang` attribute represents the language; the `rdf:resource` attribute shows the URI of the XLIFF file which contains the translation in that language. The `rdf:resource` attribute may contain an absolute or relative URI; if

Example 1. Translation statement for a DocBook document

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:bibo="http://purl.org/ontology/bibo/"
         xmlns:vivo="http://vivoweb.org/ontology/core#">
  <bibo:Document rdf:about="">
    <vivo:hasTranslation xml:base="http://DBK/Topics/512_Series/xlate/">
      <rdf:Alt>
        <rdf:li xml:lang="de" rdf:resource="ac_de.xliff"/>
        <rdf:li xml:lang="es" rdf:resource="ac_es.xliff"/>
        <rdf:li xml:lang="fr" rdf:resource="ac_fr.xliff"/>
        <rdf:li xml:lang="nb" rdf:resource="ac_nb.xliff"/>
        <rdf:li xml:lang="nl" rdf:resource="ac_nl.xliff"/>
        <rdf:li xml:lang="sv" rdf:resource="ac_sv.xliff"/>
        <rdf:li xml:lang="zh" rdf:resource="ac_zh.xliff"/>
      </rdf:Alt>
    </vivo:hasTranslation>
    <!-- Other RDF properties ... -->
  </bibo:Document>
</rdf:RDF>
```

relative, it is combined with the `xml:base` attribute of the `vivo:hasTranslation` property to define the full URI.

- Note that you need to declare the `rdf`, `bibo` and `vivo` namespaces. The namespace URIs are shown in Example 1, "Translation statement for a DocBook document".

The DocBook 5.0 schema allows foreign namespaces within the `info` element, so there is no need to extend the DocBook 5.0 schema to add the translation statement.

4.1.2. Linked Data for an Image

You would use `bibo:Image` instead of `bibo:Document` in a translation statement that applies to an SVG file.

SVG files allow elements in foreign namespaces anywhere, so similarly there is no need to extend the SVG schema to add the translation statement.

4.1.3. Linked Data for Document Parts

You can also specify a translation statement for part of a file. The easiest way to do this is to add it to the metadata section for that part. So, in a DocBook file, you would add a translation statement that only applies to a particular chapter to the `info` element at the start of the chapter. This would override any translation statement at the start of the book. If the chapter is transcluded from a different location, its translation statement can be transcluded along with it.

You would use `bibo:DocumentPart` instead of `bibo:Document` in a translation statement that only applies to part of a document.

4.2. Implementation: Markup

In the source documents, authors mark up translatable elements using an attribute from the XLIFF namespace, namely `xlf:id`. The `xlf:id` attribute corresponds to the ID of a single, unique translation unit in an XLIFF file. There's no requirement that the value of `xlf:id` should be unique in the source document.

We also allow all the local attributes from the ITS namespace in the source documents. ITS local attributes are used as specified by the ITS 2.0 recommendation, e.g. to mark up localisation notes and terminology, or to indicate whether a translatable element should be translated.

Note

Why do we use `xlf:id`? The ITS 2.0 recommendation stipulates that:

> The recommended way to specify a unique identifier is to use `xml:id` or `id` in HTML.
>
> --[ITS]

However, after transclusion, the same translatable element may appear more than once in the document. Obviously two elements cannot have the same `xml:id`.

Further, if the `xml:id` of an element were changed or fixed up after transclusion, it would be more difficult to use the `xml:id` to find the correct translation unit for that element.

Typical markup of translatable elements in a DocBook XML document is shown in Example 2, "Markup of translatable elements".

Example 2. Markup of translatable elements

```
<important xmlns="http://docbook.org/ns/docbook"
  xmlns:its="http://www.w3.org/2005/11/its"
  xmlns:xlf="urn:oasis:names:tc:xliff:document:1.2"
  version="5.0-variant PACBook"
  its:version="2.0">
  <title its:translate="no"/>
  <itemizedlist>
    <listitem>
      <para xlf:id="u00182">Risk of explosion if
      battery is replaced by an incorrect type.
      Use only 12V sealed lead acid battery.</para>
    </listitem>
    <listitem>
      <para xlf:id="u00183">Dispose of used
      batteries in accordance with local and
      national regulations.</para>
    </listitem>
  </itemizedlist>
</important>
```

Note that you need to declare the `its` and `xliff` namespaces. The namespace URIs are shown in Example 2, "Markup of translatable elements".

Note also that documents which use the ITS 2.0 local attributes must have an `its:version` attribute set to `2.0`.

We've created a custom extension to the DocBook 5.0 schema which adds these attributes. The custom extension is implemented in a Relax NG schema called `pacbook.rng` [RELAX NG]. This is available at the PACBook repository on GitHub. Documents using this extended schema must have a `version` attribute set to `5.0-variant PACBook`.

SVG files allow attributes in different namespaces, so there is no need to create a custom schema to allow these attributes in SVG files.

4.3. Implementation: Tools

We implemented the translation tools using XSLT stylesheets. These XSLT stylesheets are available from the PACBook repository on GitHub. The stylesheets can be divided into three groups:

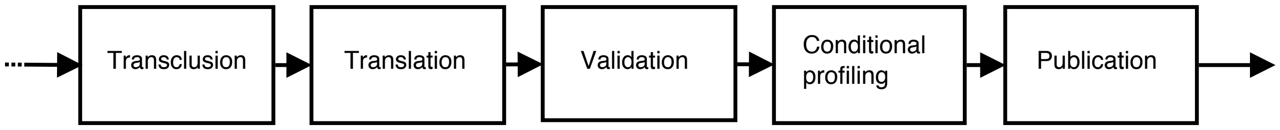
1. Stylesheets which enable authors to prepare a document for translation.
2. Stylesheets which translate the source document as part of the build process.
3. Stylesheets for the management of XLIFF files.

A full description of each XSLT stylesheet is beyond the scope of this article. A brief overview of each stylesheet is given here. Full documentation on each of them can be found in the PACBook repository on GitHub.

4.3.1. Preparation Stylesheets

1. **XlateMarkup.xsl.** Adds `xlf:id` attributes to a file. This stylesheet uses an ITS rules file to identify the translatable elements. The location of the ITS rules file is specified by stylesheet parameter. Each `xlf:id` attribute is given a unique consecutive numerical value. The latest numerical value is stored in an XML file whose location is also specified by stylesheet parameter.
2. **XlateExtract.xsl.** Extracts all translatable elements to an XLIFF file. The XLIFF file can then be sent to translators.
3. **XlateDiff.xsl.** Compares a source document to an existing XLIFF file and creates a new XLIFF file containing only the new and changed translation units. This is useful when a source document has changed.
4. **XlateMerge.xsl.** Merges complete translations from a translated XLIFF file into an existing XLIFF file. This is useful when a XLIFF file comes back from the translators.

Figure 8. Build process



4.3.2. Build Stylesheets

1. **XlateConvert.xsl.** Translates the document to a single language, specified by stylesheet parameter. This stylesheet uses the linked translation data in the document to work out which XLIFF file to use when translating each element.
2. **XlateCombine.xsl.** Creates inline multilingual translations. Similar to `XlateConvert.xsl`; however, this stylesheet translates the document to each of the languages specified by the stylesheet parameter and combines each translation inline.
3. **XlateID.xsl.** Fixes up `xml:id` attributes in multilingual translations. It simply adds the current language code to the end of all `xml:id` and link attributes. This is useful if you use a two-step build process to create multilingual translations, first translating the source document into several languages and then combining the translations into a larger document.

4.3.3. XLIFF Management Stylesheets

1. **XliffDupe.xsl.** Removes duplicate translation units from an XLIFF file. Duplicate translation units can occur in an XLIFF file that has been extracted from a source document with repeated identical translatable elements, e.g. translatable elements that have been transcluded into more than one location. This stylesheet performs a deterministic deduplication; if more than one translation unit has the same ID, the first is kept and the rest are discarded.
2. **XliffPurge.xsl.** Removes completed translation units from an XLIFF file. This is useful when an XLIFF file has been partially translated, e.g. by an in-house translation memory, and you want to remove the translated strings before sending the XLIFF file for translation.
3. **XliffRaw.xsl.** Escapes inline markup in an XLIFF file — preferred by some translation houses.
4. **XliffTemp.xsl + XliffTag.xsl.** Unescapes inline markup in an XLIFF file — if required.
5. **Xliff2CSV.xsl.** Exports an XLIFF file to CSV.
6. **XL2Xliff.xsl.** Imports from a Microsoft Excel 2003 XML file to XLIFF.

4.4. Solution: Build Process

We use Apache Ant[1] build scripts to automate the transclusion, profiling, translation and publication process.

We maintained a library of build scripts to handle the build process. Each major documentation project then had its own master build script, that called on the build scripts library to publish the required outputs for each document in the required languages.

These build scripts have not been released as part of the PACBook project. A full description of the build scripts is outside the scope of this article. However, in most cases the build process carried out the following steps, as illustrated in Figure 8, “Build process”.

1. Transclude document modules into the assembly document.
2. If required, translate the complete document to a single specified language.
3. Validate the translated document.
4. Perform conditional profiling.
5. Perform any final steps and convert to the output format.

Note

It’s necessary to validate the document after translation, as translated elements include inline markup. Any errors in the inline markup will be copied into the translated document.

We also use separate build scripts to handle the translation and conversion of images. These are called from the master build script for a particular documentation project.

[1] http://ant.apache.org

Within the image build scripts, we use the Apache Batik SVG Rasterizer[2] to convert translated SVG files into the formats required for publication, e.g. PNG.

There's no intrinsic reason why the build process has to be written in Apache Ant. Since each step involves an XML document, and the output of each step becomes the input of the next step, the ideal solution is probably an XProc pipeline [XProc].

5. Conclusion

This article outlined a solution that STANLEY Black and Decker Innovations developed to enable dynamic translation of XML documentation using XLIFF files, and to find the correct translations for each translatable element when the XML documentation consists of modular document source files that are transcluded into a single document.

5.1. Benefits

The solution outlined in this article was used successfully within STANLEY Black and Decker Innovations over several years to handle document translation and to manage XLIFF files. However, as an in-house project, the XSLT stylesheets have not had the benefit of scrutiny from outside users.

The solution gives a great deal of freedom to authors to create their own strategy for associating XLIFF files with source documents. You could use single monolithic XLIFF files for all translations across all projects, or separate XLIFF files for every topic, or anything in between. Like any solution which allows linking or transclusion, it's best for each author to find the right balance between content reuse and maintainability.

The XLIFF files that were produced by this solution could be passed directly on to our translation house. More importantly, completed translations could be dropped into our documentation source control with very little intervention. This vastly minimised author effort, translation errors and the number of files to maintain. Translation costs were also reduced as common content and its associated translations could be shared more easily between several documents.

5.2. Issues

The translation stylesheets currently find the correct XLIFF file for each translatable element by searching up through the XML element hierarchy to find the nearest ancestor element that contains a translation statement. We'd also like to be able to associate translation statements with a particular part of the document by using the `rdf:about` attribute to refer to an element's `xml:id` attribute. This is not yet implemented.

The markup was originally designed for DocBook XML and SVG. Some work would be required to adapt it for different documentation standards, such as HTML [HTML]. The markup could easily be adapted to work with HTML 5, perhaps using `data` attributes. We haven't investigated how the linked translation data could be adapted to work with HTML 5.

The stylesheets were written in XSLT 1.0 and EXSLT [EXSLT]. This is because at first we performed the transformations using xsltproc[3], which doesn't support later versions of XSLT. We later migrated to Saxon[4] to perform XSLT transformations, so it should be possible to migrate the stylesheets to XSLT 2.0 or XLST 3.0, which may make some of the XSLT code simpler.

The solution as described is designed to work with large scale, context-free transclusion, i.e. transcluding chapters or sections, which are then translated after transclusion. Small-scale, parametrised transclusion, i.e. transcluding words or phrases, raises an entirely different set of linguistic problems, which this article does not attempt to address. However, the PACBook project includes further markup and stylesheets which attempt to solve the linguistic problems associated with parametrised transclusion. For more information, see the PACBook repository on GitHub.

5.3. Future Development

To finish on a personal note: Stanley Black and Decker Innovations was wound down at the end of July 2015, although I've continued to do documentation work with Stanley Black and Decker. I've cloned the PACBook GitHub repository at https://github.com/janiveer/PACBook, so that development on the project can continue. I'd be very interested to make contact with people who are working in similar areas and would be interested in collaboration. If anyone would like to contribute, please contact the author.

[2] http://xmlgraphics.apache.org/batik/tools/rasterizer.html

[3] http://xmlsoft.org/XSLT/xsltproc2.html

[4] http://www.saxonica.com

References

[XLIFF] *XLIFF Version 1.2.*
http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html
OASIS XLIFF Technical Committee. OASIS Standard. 1 February 2008. Accessed 2016-03-04.

[gettext] *GNU gettext 0.19.7.*
https://www.gnu.org/software/gettext/manual/index.html
Free Software Foundation. 28 December 2015. Accessed 2016-04-26.

[DocBook] *The DocBook Schema Version 5.0.*
http://docs.oasis-open.org/docbook/specs/docbook-5.0-spec-os.html
DocBook Technical Committee. OASIS Standard. 1 November 2009. Accessed 2014-03-09.

[RELAX NG] *RELAX NG Specification.*
https://www.oasis-open.org/committees/relax-ng/spec-20011203.html
James Clark and Murata Makoto. OASIS Committee Specification. 3 December 2001. Accessed 2016-05-03.

[Mallard] *Mallard 1.0.*
http://projectmallard.org/1.0/index
Shaun McCance and Jim Campbell. Project Mallard. 23 July 2013. Accessed 2014-03-09.

[SVG] *Scalable Vector Graphics (SVG) 1.1 (Second Edition).*
https://www.w3.org/TR/SVG/
Erik Dahlström, et al. W3C Recommendation. 16 August 2011. Accessed 2016-03-5.

[IBM] *XML in localisation: Use XLIFF to translate documents.*
http://www.ibm.com/developerworks/library/x-localis2
Rodolfo Raya. IBM developerWorks. 22 October 2004. Accessed 2016-02-06.

[Tektronix] *Improving Localization Process at Tektronix Using XML and XLIFF: A Case Study.*
http://www.moravia.com/files/download/Improving_Localization_Process_at_Tektronix_Using_XML_and_XLIFF.pdf
Anette Hauswirth (Moravia Worldwide) and Bryan Schnabel (Tektronix). 11 January 2008. Accessed 2016-02-06.

[Yelp] *Introduction to Mallard: Build System Integration.*
http://en.flossmanuals.net/introduction-to-mallard/build-system-integration
FLOSS Manuals. Accessed 2016-02-06.

[XProc] *XProc: An XML Pipeline Language.*
http://www.w3.org/TR/xproc
Norman Walsh, Alex Milowski, and Henry S. Thompson. W3C Recommendation. 11 May 2010. Accessed 2014-03-09.

[XSLT] *XSL Transformations (XSLT) Version 1.0.*
http://www.w3.org/TR/xslt
James Clark. W3C Recommendation. 16 November 1999. Accessed 2014-03-09.

[EXSLT] *EXSLT.*
http://exslt.org
Jeni Tennison, Uche Ogbuji, Jim Fuller, and Dave Pawson, et al. 14 October 2003. Accessed 2016-05-03.

[ITS] *Internationalization Tag Set (ITS) Version 2.0.*
https://www.w3.org/TR/its20
David Filip, et al. W3C Recommendation. 29 October 2013. Accessed 2016-03-04.

[BIBO] *Bibliographic Ontology Specification Revision 1.3.*
http://bibliontology.com

Bruce D'Arcus and Frédérick Giasson. Structured Dynamics. 4 November 2009. Accessed 2016-03-05.

[VIVO] *VIVO-ISF Ontology version 1.6.*
http://vivoweb.org
Open Research Information Framework. 13 December 2013. Accessed 2016-03-05.

[RDF/XML] *RDF 1.1 XML Syntax.*
http://www.w3.org/TR/rdf-syntax-grammar
Fabien Gandon and Guus Schreiber. W3C Recommendation. 25 February 2014. Accessed 2016-03-05.

[HTML] *HTML5.*
http://www.w3.org/TR/html
Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. World Wide Web Consortium (W3C). 28 October 2014. Accessed 2016-03-05.

[LGPL] *GNU Lesser General Public License Version 3.*
http://www.gnu.org/licenses/lgpl.html
Free Software Foundation, Inc. 29 June 2007. Accessed 2016-03-05.

Faenza icons by Matthieu James[1].

[1] https://launchpad.net/~tiheum

Parse Earley, Parse Often

How to Parse Anything to XML

Steven Pemberton

CWI, Amsterdam

Abstract

Invisible XML, ixml for short, is a generic technique for treating any parsable format as if it were XML, and thus allowing any parsable object to be injected into an XML pipeline. Based on the observation that XML can just be seen as the description of a parse-tree, any document can be parsed, and then serialised as XML. The parsing can also be undone, thus allowing roundtripping.

This paper discusses issues around grammar design, and in particular parsing algorithms used to recognise any document, and converting the resultant parse-tree into XML, and gives a new perspective on a classic algorithm.

1. Introduction

What if you could see everything as XML? XML has many strengths for data exchange, strengths both inherent in the nature of XML markup and strengths that derive from the ubiquity of tools that can process XML. For authoring, however, other forms are preferred: no one writes CSS or Javascript in XML.

It does not follow, however, that there is no value in representing such information in XML. Invisible XML [1] [2], is a generic technique for treating any (context-free) parsable document as if it were XML, enabling authors to write in a format they prefer while providing XML for processes that are more effective with XML content. There is really no reason why XML cannot be more ubiquitous than it is.

Ixml is based on the observation that XML can just be seen as the description of a parse-tree, and so any document can be parsed, and serialised as XML.

Thus can a piece of CSS such as

```
body {
  color: blue;
  font-weight: bold;
}
```

with ixml be read by an XML system as

```
<css>
  <rule>
    <selector>body</selector>
    <block>
      <property>
        <name>color</name>
        <value>blue</value>
      </property>
      <property>
        <name>font-weight</name>
        <value>bold</value>
      </property>
    </block>
  </rule>
</css>
```

or as

```
<css>
  <rule>
    <selector>body</selector>
    <block>
      <property name="color" value="blue"/>
      <property name="font-weight" value="bold"/>
    </block>
  </rule>
</css>
```

The choice is yours. Similarly an arithmetic expression such as

```
a×(3+b)
```

can be read as

```
<expr>
  <prod>
    <letter>a</letter>
    <sum>
      <digit>3</digit>
      <letter>b</letter>
    </sum>
  </prod>
</expr>
```

doi:10.14337/XMLLondon16.Pemberton01

and a URL such as

```
http://www.w3.org/TR/1999/xhtml.html
```

could be read as

```
<uri>
  <scheme>http</scheme>
  <authority>
    <host>
      <sub>www</sub>
      <sub>w3</sub>
      <sub>org</sub>
    </host>
  </authority>
  <path>
    <seg>TR</seg>
    <seg>1999</seg>
    <seg>xhtml.html</seg>
  </path>
</uri>
```

Previous presentations on ixml have centred on the basics, the design of grammars, and round-tripping back to the original form. This paper discusses a suitable parsing algorithm, and gives a new perspective on a classic algorithm.

2. Parsing

Parsing is a process of taking an essentially linear form, recognising the underlying structure, and transforming the input to a form that expresses that structure. The fact that the resulting structure is represented by a tree means that converting it to XML is a relatively simple matter.

There are many parsing algorithms with different properties such as speed, run-time complexity, and space usage [3]. One of the most popular is LL1 parsing; in particular, the "1" in this name refers to the fact that you can parse based only on the knowledge of the next symbol in the input, thus simplifying parsing.

Consider the following example of a grammar describing a simple programming language.

A grammar consists of a number of 'rules', each rule consisting, in this notation, of a name of the rule, a colon, and a definition describing the structure of the thing so named, followed by a full stop. The structure can consist of one or more 'alternatives', in this notation separated by semicolons. Each alternative consists of a sequence of 'nonterminals' and 'terminals' separated by commas. Nonterminals are defined by subsequent rules; terminals, enclosed in quotes, are literal characters that have to be matched in the input.

```
program: block.
block: "{", statements, "}".
statements: statement, ";", statements; empty.
statement: if statement; while statement;
           assignment; call; block.
if statement: "if", condition, "then", statement,
               else-option.
else-option: "else", statement; empty.
empty: .
while statement: "while", condition, "do",
                 statement.
assignment: variable, "=", expression.
call: identifier, "(", parameters, ")".
parameters: expression, parameter-tail; empty.
parameter-tail: ",", expression, parameter-tail;
                empty.
```

This grammar is almost but not quite LL1. The problem is that the rules 'assignment' and 'call' both start with the same symbol (an identifier), and so you can't tell which rule to process just by looking at the next symbol.

However, the language is LL1. This can be shown by combining and rewriting the rules for assignment and call:

```
statement: if statement; while statement;
           assignment-or-call; block.
assignment-or-call: identifier, tail.
tail: assignment-tail; call-tail.
assignment-tail: "=", expression.
call-tail: "(", parameters, ")".
```

Now the decision on which rule to use can be taken purely based on the next symbol.

One of the reasons that LL1 parsing is popular, is that it is easy to translate it directly to a program. For instance:

```
procedure program = { block; }
procedure block = { expect("{"); statements;
                    expect("}")}
procedure statements = {
  if nextsym in statement-starters
  then {
    statement;
    expect(";");
    statements;
  }
}
procedure statement = {
  if nextsym="if" then ifstatement;
  else if nextsym="while" then whilestatement;
  else if nextsym=identifier
    then assignment-or-call;
  else if nextsym="{" then block;
  else error("syntax error");
}
```

etc. (This example is much simplified from what an industrial-strength parser would do, but demonstrates the principles).

However, rather than writing the parser as code, you can just as easily write what could be seen as an interpreter for a grammar. For instance:

```
procedure parserule(alts) = {
  if (some alt in alts has nextsym in starters(alt))
    then parsealt(alt);
  else if (some alt in alts has empty(alt)
    then do-nothing;
  else error("syntax error");
}
procedure parsealt(alt) = {
  for term in alt do {
    if nonterm(term) then parserule(def(term));
    else expectsym(term);
  }
}
```

One disadvantage of LL1 parsing is that no rule may be left-recursive. For instance, if the rule for 'statements' above were rewritten as

```
statements: statements, statement, ";"; empty.
```

this could not be parsed using LL1. It is easy to see why if you convert this to code, since the procedure would be:

```
procedure statements = {
  if nextsym in statement-starts {
    statements;
    statement;
    expect (";");
  }
}
```

in other words, it would go into an infinite recursive loop. In the case of a statement list, there is no problem with expressing it as a right-recursive rule. However, there are cases where it matters. For example, with subtraction:

```
subtraction: number; subtraction, "-", number.
```

If we rewrite this as

```
subtraction: number; number, "-", subtraction.
```

then an expression such as

```
3-2-1
```

would mean in the first case

```
((3-2)-1)
```

and in the second

```
(3-(2-1))
```

which has a different meaning.

To overcome this problem, grammars that are to be parsed with LL1 methods must have a notation to express repetition. For instance:

```
statements: (statement, ";")*.
subtraction: number, ("-", number)*.
```

which can be translated to procedures like this:

```
procedure subtraction = {
  number;
  while (nextsym="-") do {
    skipsym;
    number;
  }
}
```

Another disadvantage of LL1 and related techniques is that there has to be an initial lexical analysis phase, where 'symbols' are first recognised and classified. If not, then the level of terminal symbols that are available to the parser are the base characters, such as letters and digits, etc., meaning that for instance if the next character is an "i" you can't tell if that starts an identifier, or the word "if".

Finally, a problem with these techniques is the need to understand the LL1 conditions, express a grammar in such a way that they are satisfied, and the need to have a checker that determines if the grammar indeed satisfies the conditions before using it [4] [5].

3. General Parsing

To summarise the advantages of LL1 parsing: it is fast, its run-time complexity is low, proportional only to the length of the input, and it is easy to express as a program.

However the disadvantages are that it can only handle a certain class of restricted languages, it requires the author of a grammar to understand the restrictions and rewrite grammars so that they match, and it requires a lexical pre-processing stage.

To overcome these disadvantages we have to consider more general parsing techniques.

One classic example is that of Earley [6], which can parse any grammar, for any language. (There is actually one restriction, that the language is context-free, but since we are only using grammars that express context-free languages, the issue doesn't apply here).

Although the Earley algorithm is well-known, it is apparently less-often used. One reason this may be so is because the run-time complexity of Earley is rather poor in the worst case, namely $O(n^3)$. However, what potential users who are put off by this number do not seem to realise is that it is a function of the language being parsed, and not the method itself. For LL1 grammars, Earley is also O(n), just like pure LL1 parsers.

So what does Earley do?

4. Pseudo-parallelism

Modern operating systems run programs by having many simultaneously in store simultaneously (242 on the computer this text is being written on), and allocating brief periods of time (a fraction of a second) to each in turn. A program once allocated a slot is allowed to run until it reaches the end of its execution, its time allocation runs out, or the program asks to do an operation that can't be immediately satisfied (such as accessing the disk). Programs that are ready to run are added to one or more queues of jobs, and at the end of the next time slot a program is selected from the queues, based on priority, and then allowed to run for the next slot. Because the time slots are so short by human measure, this approach gives the impression of the programs running simultaneously in parallel.

Earley operates similarly. Just as the example earlier, it is an interpreter for grammars, with the exception that when a rule is selected to be run, all of its alternatives are queued to run 'in parallel'. When an alternative is given a slot, it is allowed to do exactly one task, one of:

- note that it has nothing more to do and restart its parent (that is, terminating successfully)
- start a new nonterminal process (and wait until that completes successfully)
- match a single terminal (and requeue itself)
- note that it cannot match the next symbol, and effectively terminate unsuccessfully.

The queue contains each task with the position in the input that it is at. The queue is ordered on input position, so that earlier input positions get priority. In this way only a small part of the input stream needs to be present in memory.

There is one other essential feature: when a rule starts up, its name and position is recorded in a trace before being queued. If the same rule is later started at the same position, it is not queued, since it is either already being processed, or has already been processed: we already know or will know the result of running that task at that position. This has two advantages: one is pure optimisation, since the same identical process will never be run twice; but more importantly, this overcomes the problem with infinite recursion that we saw with LL1 parsers.

To take an example, suppose the rule `statement` is being processed at the point in the input where we have the text

```
a=0;
```

Processing `statement` mean that its alternatives get queued: namely `if statement`, `while statement`, `assignment`, `call`, and `block`.

With the exception of `assignment` and `call`, all of these fail immediately because the first symbol in the input fails to match the initial item in the alternative.

Assignment and call both have as first item `'identifier'`. `Identifier` gets queued (once) and succeeds with the input 'a', so both alternatives get requeued. Since the next symbol is '=', `call` fails, and `assignment` gets requeued (and eventually succeeds).

5. The structure of tasks

Each task that gets queued has the following structure:

- The name of the rule that this alternative is a part of (e.g. statement);
- The position in the input that it started;
- The position that it is currently at;
- The list of items in the alternative that have so far been successfully parsed, with their start position;
- The list of items still to be processed.

When a task is requeued, its important parts are its current position, and the list of items still to be processed. If the list of items still to be processed is empty, then the task has completed, successfully.

6. Earley

So now with these preliminaries behind us, let us look at the Earley parser (here expressed in ABC [7]):

```
HOW TO PARSE input WITH grammar:
INITIALISE
START grammar FOR start.symbol grammar AT start.pos
WHILE more.tasks:
 TAKE task
 SELECT:
  finished task:
   CONTINUE PARENTS task
  ELSE:
   PUT next.symbol task, position task IN sym, pos
   SELECT:
    grammar nonterminal sym:
     START grammar FOR sym AT pos
    sym starts (input, pos): \Terminal, matches
     RECORD TERMINAL input FOR task
     CONTINUE task AT (pos incremented (input, sym))
    ELSE:
     PASS \Terminal, doesn't match
```

So let us analyse this code line-by-line:

```
START grammar FOR start.symbol grammar AT start.pos
```

All grammars have a top-level start symbol, such as `program` for the example programming language grammar. This line adds all the alternatives of the rule for the start symbol to the queue.

```
WHILE more.tasks:
```

While the queue of tasks is not empty, we loop through the following steps.

```
TAKE task
```

Take the first task from the queue.

```
SELECT:
```

There are two possibilities: the rule has nothing more to do, and so terminates successfully, or there are still symbols to process.

```
finished task:
  CONTINUE PARENTS task
```

The task has nothing more to do, so all the parents of this task (the rules that initiated it) are requeued.

```
ELSE:
```

Otherwise this task still has to be processed.

```
PUT next.symbol task, position task IN sym, pos
```

We take the next symbol (terminal or nonterminal) that still has to be processed, and the current position in the input we are at.

```
SELECT:
```

The next symbol is either a nonterminal or a terminal.

```
grammar nonterminal sym:
  START grammar FOR sym AT pos
```

The symbol is a nonterminal, so we queue all its alternatives to be processed at the current position.

Otherwise it is a terminal:

```
sym starts (input, pos): \Terminal, matches
  RECORD TERMINAL input FOR task
  CONTINUE task AT (pos incremented (input, sym))
```

If the symbol matched the input at the current position, then we record the match in the trace, and requeue the current task after incrementing its position past the matched characters.

```
ELSE:
  PASS
```

Finally, it was a terminal, but didn't match the next symbol in the input, and so the task doesn't get requeued, and so effectively terminates (unsuccessfully).

7. The Trace

The result of a parse is the trace, which is the list, for all positions in the input where a task was (re-)started, of the tasks that were (re-)started there. This is the list that is used to prevent duplicate tasks being started at the same position, but also effectively records the results of the parse.

So for instance, here is an example trace for the very first position in the input (before dealing with any input characters), for the example grammar:

```
program[1.1:1.1]: | block
block[1.1:1.1]: | "{" statements "}"
```

Two rules have been started, one for program, and consequently one for block.

The positions have been recorded as `line-number.character-number`, and here represent the position where we started from, and the position up to which we have processed for this rule, in this case, both of them `1.1`.

After the colon are two lists, the items in the respective rule that have already been processed (none yet in this case), and the items still to be processed, the two separated in this output by a vertical bar.

In parsing a very simple program, namely `{a=0;}`, after parsing the first opening brace, this will be in the trace:

```
block[1.1:1.2]: "{"[:1.2] | statements "}"
```

signifying that we have parsed up to position 1.2, and that we have parsed the open brace, which ends at 1.2.

Later still, after we have parsed the semicolon we will find in the trace

```
block[1.1:1.6]: "{"[:1.2] statements[:1.6] | "}"
```

which signifies we have matched the opening brace up to position 1.2, something that matches 'statements' up to 1.6, and there only remains a closing brace to match.

And finally at the end, we will find

```
block[1.1:2.1]: "{[:1.2] statements[:1.6] "}[:2.1]
```

which since the 'to do' list is empty signifies a successful parse from position 1.1 to 2.1.

Since it is only completed (sub-)parses that we are interested in, here is the complete trace of all successful sub-parses for the program `{a=0;}`:

```
1.1
1.2
  empty[1.2:1.2]: |
  statements[1.2:1.2]: empty[:1.2] |
1.3
  identifier[1.2:1.3]: "a"[:1.3] |
  variable[1.2:1.3]: identifier[:1.3] |
1.4
1.5
  assignment[1.2:1.5]: variable[:1.3] "="[:1.4]
 expression[:1.5] |
  statement[1.2:1.5]: assignment[:1.5] |
  expression[1.4:1.5]: number[:1.5] |
  number[1.4:1.5]: "0"[:1.5] |
1.6
  statements[1.2:1.6]: statement[:1.5] ";"[:1.6
] statements[:1.6] |
  empty[1.6:1.6]: |
  statements[1.6:1.6]: empty[:1.6] |
2.1
  block[1.1:2.1]: "{"[:1.2] statements[:1.6] "}
"[:2.1] |
  program[1.1:2.1]: block[:2.1] |
```

8. Serialising the Parse Tree

So the task of serialising the trace, is one of looking at the list in the trace for the last position in the input for a successful parse for the top level symbol of the grammar, and working from there downwards:

```
SERIALISE start.symbol grammar
          FROM <start position> TO <end position>
```

where the procedure SERIALISE looks like this:

```
HOW TO SERIALISE name FROM start TO end:
  IF SOME task IN trace[end] HAS
    (symbol task = name AND finished task AND
     start.position task = start):
    WRITE "<", name, ">"
    CHILDREN
    WRITE "</", name, ">"
CHILDREN:
  PUT start IN newstart
  FOR (sym, pos) IN done task:
    SELECT:
      terminal sym: WRITE sym
      ELSE:
        SERIALISE sym FROM newstart TO pos
    PUT pos IN newstart
```

For our example program, this will produce:

```
<program>
  <block>{<statements>
      <statement>
        <assignment>
          <variable>
            <identifier>a</identifier>
          </variable>=<expression>
            <number>0</number>
          </expression>
        </assignment>
      </statement>;<statements>
        <empty/>
      </statements>
    </statements>}</block>
</program>
```

The simple task of adapting this to the exact needs of ixml as described in earlier papers is left as an exercise for the reader.

9. Conclusion

The power of XML has been the simple and consistent representation of data, allowing widespread interoperability.

What ixml shows is that with a simple front-end, XML can be made even more powerful by making all parsable documents accessible to the XML pipeline.

References

[1] *Invisible XML*.
doi:10.4242/BalisageVol10.Pemberton01
Steven Pemberton. Presented at Balisage: The Markup Conference. Balisage. Montréal, Canada. August 6 - 9, 2013.

[2] *Data just wants to be (format) neutral*.
http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf
109-120. Steven Pemberton. XML Prague. Prague, Czech Republic. 2016.

[3] *The Theory of Parsing, Translation, and Compiling*.
ISBN 0139145567.
V. Alfred Aho. D. Jeffrey Ullman. Prentice-Hall. 1972.

[4] *Grammar Tools*.
http://www.cwi.nl/~steven/abc/examples/grammar.html
Steven Pemberton. 1991.

[5] *A Syntax Improving Program*.
doi:10.1093/comjnl/11.1.31
M. J Foster. Computer Journal. 11. 1. 31-34. 1967.

[6] *An Efficient Context-Free Parsing Algorithm*.
doi:10.1145/362007.362035
Jay Earley.

[7] *The ABC Programmer's Handbook*.
ISBN 0-13-000027-2.
Leo Geurts. Lambert Meertens. Steven Pemberton. Prentice-Hall. 1990.

Charles Foster (ed.)

XML London 2016
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 2016

ISBN 978-0-9926471-3-1