



XML Prague 2016

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 11–13, 2016

XML Prague 2016 – Conference Proceedings

Copyright © 2016 Jiří Kosek

ISBN 978-80-906259-0-7 (pdf)

ISBN 978-80-906259-1-4 (ePub)

The Complete Solution for XML Authoring & Development



XML Editor

oXygen XML Editor is a complete XML editing solution for developers and content authors.



XML Author

oXygen XML Author provides a visual interface designed for user-friendly structured authoring.



XML Developer

oXygen XML Developer is an effective and easy-to-use industry-leading XML development tool.



XML Web Author

oXygen XML Web Author is the ultimate tool for editing and reviewing content in browsers on any device.



WebHelp

oXygen XML WebHelp allows you to publish DITA and DocBook in a modern, interactive web-based help system.

Table of Contents

General Information	vii
Sponsors	ix
Preface	xi
Born accessible EPUB – <i>Romain Deltour</i>	1
Extending CSS with XSL-FO, XSL-FO with CSS – <i>Tony Graham</i>	17
Virtual Document Management – <i>Ari Nordström</i>	33
Define and Conquer – <i>Dr. Patrik Stellmann</i>	49
Subjugating Data Flow Programming – <i>R. Alexander Miłowski and Norman Walsh</i>	65
Schematron QuickFix – <i>Octavian Nadolu and Nico Kutscherauer</i>	81
Validating office documents in the publishing production workflow – <i>Andrew Sales</i>	99
Data Just Wants to Be Format-Neutral – <i>Steven Pemberton</i>	109
Approaches for Leveraging XML Workflows with Linked Data – <i>Marta Borriello, Christian Dirschl, Axel Polleres, Phil Ritchie, Frank Salliau, Felix Sasaki, and Giannis Stoitsis</i>	121
Promises and Parallel XQuery Execution – <i>James Wright</i>	139
Entities and Relationships in a Document Database – <i>Charles Greer</i>	153
Transforming JSON using XSLT 3.0 – <i>Michael Kay</i>	167

General Information

Date

February 11th, 12th and 13th, 2016

Location

University of Economics, Prague (UEP)
nám. W. Churchilla 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Petr Cimprich, *Xyleme*
Vít Janota, *Xyleme*
Káťa Kabrhelová, *University of Economics, Prague*
Jirka Kosek, *xmkguru.cz & University of Economics, Prague*
Martin Svárovský, *Xyleme*
Mohamed Zergaoui, *ShareXML.com & Innovimax*

Program Committee

Robin Berjon, *science.ai*
Petr Cimprich, *Xyleme*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Ari Nordström, *SGMLGuru.org*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Andrew Sales Digital Publishing*
Felix Sasaki, *DFKI / W3C Fellow*
John Snelson, *MarkLogic*
Jeni Tennison, *Open Data Institute*
Eric van der Vlist, *Dyomedeia*
Priscilla Walmsley, *Datypic*
Norman Walsh, *MarkLogic*
Mohamed Zergaoui, *Innovimax*

Produced By

XMLPrague.cz (<http://xmlprague.cz>)
Faculty of Informatics and Statistics, UEP (<http://fis.vse.cz>)
Ubiqway, s.r.o. (<http://www.ubiqway.com>)

Sponsors

oXygen (<http://www.oxygenxml.com>)

Antenna House (<http://www.antennahouse.com/>)

le-tex publishing services (<http://www.le-tex.de/en/>)

Mercator IT Solutions Ltd (<http://www.mercatorit.com>)

OverStory Consulting Ltd (<http://www.overstory.co.uk/>)



Preface

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

In its eleventh year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup and semantic on the Web, publishing and digital books, XML technologies for Big Data and recent advances in XML technologies. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 11–13 February 2016 at the campus of University of Economics in Prague. XML Prague 2016 is jointly organized by the XML Prague Organizing Committee and by the Faculty of Informatics and Statistics, University of Economics in Prague.

The full program of the conference is broadcasted over the Internet (see <http://xmlprague.cz>)—allowing XML fans, from around the world, to participate on-line.

The Thursday and Saturday morning runs in an unconference style which provides space for various XML community meetings in parallel tracks. Friday and Saturday afternoon are devoted to classical single-track format and papers from it are published in the proceedings. Additionally, we coordinate, support and provide space for W3C XSLT and XProc working group meetings collocated with XML Prague.

We hope that you enjoy XML Prague 2016.

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

Born accessible EPUB

Let's do it!

Romain Deltour

DAISY Consortium

<rdeltour@gmail.com>

Abstract

Access to information and knowledge should be universal. That should not be controversial, but even today –in 2016– too few digital publications are “born accessible”. How can we raise the bar for inclusive publishing? This article presents best practices and guidance on how to make accessible EPUB publications.

Keywords: XML, DocBook, authoring

1. Introduction

“Access to information and knowledge is the single most powerful tool available to promote world peace.” It is with these words that the International Digital Publishing Forum (IDPF) –the organization responsible for the development of the EPUB standard– reacted to the tragic terrorist attacks that hit Paris in November 2015. In fact, the ability to access information and knowledge, regardless of disabilities, is recognized as a human right by the United Nations Convention on the Rights of Persons with Disabilities [3]. However, it is a common estimate that less than 10% of the world’s published information is accessible to people with a print disability. How can we raise the bar?

Fortunately, digital publishing is progressively bringing positive changes. The EPUB 3 set of specifications, developed by IDPF, have been designed with accessibility as a core principle. It is possible, today, to produce “born accessible” digital publications. This article presents, in a first section, an overview of the best practices for creating accessible EPUB 3 content. In a second section, we will make some suggestions on how to implement these best practices.

2. Accessibility Best Practices

An accessible publication is in essence a publication that can be usable by a wide variety of users, on a wide variety of reading devices. However, there is no single precise definition of an “accessible publication”; in fact there may be as many definitions as there are readers. It is therefore particularly interesting to avoid making too many assumptions on the target audience and reading environment, and

instead rely on data quality and universal design solutions. That is not to say that publishers or book creators have to make grandiose plans to implement accessibility fundamentals; a little common sense usually goes a long way. This section presents the principles that form the cornerstones of all accessible publications.

2.1. Structure means a lot

It may sound trite to the well-versed markup specialist, but the most fundamental accessibility guideline is to represent data with proper structure and semantics. Not only will it open you the gates of data-purity nirvana, but –more importantly– it has a direct impact on how the content will be understandable and operable by print-disabled users.

2.1.1. Structuring 101

EPUB 3 Content Documents [8] are (X)HTML5 documents; in other words, the elements and attributes used to markup EPUB content are defined in the W3C HTML5 specification [12]. It is important to have a good understanding of the HTML elements because they convey *meaning* (semantics). This meaning can be used by *Reading Systems* (the applications used to render EPUBs, the equivalent of *User Agents* in HTML terminology) to provide adequate navigability or rendering features. For instance, when a heading in a document is properly represented with one of the h1 to h6 elements, a reading system can provide navigation keys to jump to the next/previous headings, and a screen reader can announce that the user is reading a heading. If on the contrary the heading was represented with semantically-neutral `span` or `class` elements, none of this would be possible.

A particularly important reason why proper structure and semantics are significant is that they define the *logical reading order* of the content. Most publications have a primary narrative that a reader should be able to follow uninterruptedly. Of course, within this primary narrative, content should be marked up in the order in which it is expected to be read: for instance, a heading would typically appear as a first child of its containing section. What may be less commonly understood, however, is that ancillary content –e.g. sidebars, figures, footnotes, etc.– needs to be properly identified so that it will not interrupt the reading of the primary narrative. Such supplementary content can be marked up with `aside` or `figure` elements. With appropriate markup, a linear reading system like a text-to-speech-enabled reader will be able to render the primary narrative end-to-end while ignoring ancillary content; this feature is called *skippability*. An additional benefit is that, would a user decide to enable the reading of ancillary content, the reading system can offer an exit while in the midst of reading these –thereby continuing the reading right after said content–; this feature is called *escapability*.

As we said earlier, most of these structuring and semantics considerations are often common sense, especially for whoever is accustomed to designing docu-

ment-oriented data. Yet, even today, many EPUB publications are created with poor markup quality. This is especially the case of documents exported from layout-focused tools, which far too often come out as a soup of semantically neutral `div` or `p` elements. Fixing a badly structure document –for instance with an XSLT transformation– is a difficult task, usually relying on ad-hoc heuristics that cannot easily be ported to the general case. It is therefore very important to have quality data as early as possible in the production workflow; it will make accessibility easier to implement.

2.1.2. Extended semantics

We've seen already that HTML5 elements convey meaning. However, there is only so much information you can express with the limited set of available elements. While using an `aside` element as a container to footnotes or sidebar content rightly contributes to identifying the logical reading order of the document, an accessible reading system would also need to distinguish a "sidebar" `aside` from a "footnotes" `aside` in order to accurately convey the information to the user. EPUB 3 allows to extend HTML's built-in structural semantics by using an `epub:type` attribute on any element. This attribute can hold a space-separated list of values taken from well-defined vocabularies. IDPF maintains a default vocabulary, the EPUB 3 Structural Semantics Vocabulary [21]. While it is also possible to define custom vocabularies, the *semantic inflections* defined in `epub:type` attributes will only impact the accessibility on the reading systems who understand them.

2.1.2.1. Example: footnotes

One example usage of semantic inflections is to define footnotes. This can be done by setting the properties "noteref" and "footnote" on elements with the `epub:type` attribute as shown in Example 1)

Example 1. An EPUB footnote

```
<p>Attendees of XML Prague often take the opportunity
to discuss cool topics around frosty or hot beverages
<a href="#note-bev" epub:type="noteref">1</a>.</p>
...
<aside id="note-bev" epub:type="footnote">
  <p>I usually favor beer over coffee, but YMMV.</p>
</aside>
```

In this example, the "noteref" semantic inflection identifies the link as a reference to a note (be it a footnote or a rearnote). The "footnote" semantic inflection identifies the footnote content. In addition to the use of the `aside` element, which on its

own declares the footnote as being out of the primary reading flow, the semantic inflections enable an accessible reading system to announce the note reference to a print disabled user. The user can then decide to skip the note reference so that her reading is not interrupted.

Not only does this rich semantic markup enable good accessibility features, but it can also be beneficial to sighted readers too. While proper styling can already be used to give visual hints about the footnote nature (for instance by using smaller font or rendering the content at the end of the page), the additional semantic information can be used to enable dedicated rendering techniques. In the case of footnotes, some reading system will effectively use that information to render the footnote in a pop-up box, which can be read and then dismissed without losing the current reading position.

2.1.2.2. Page numbering

Another typical use case for using semantic inflection is to identify –in the markup– the location of the page breaks as they occur in the print representation of a publication. Except for a minority of digital-only publications, most of the publications are available in both digital formats and print. We have a long history of using printed material, and we often use page number references when we want to locate content in a publication. Because print disabled users do not have access to print publications, it is very important that they can follow the same references when using a digital publication. This is often a critical need, notably in education: for instance when the math teacher asks the pupils to go read the theorem #3 on page #42, her blind student must be able to access that page as directly as a sighted user would do.

In EPUB 3, the location of where the page breaks occur in the equivalent print material can be identified with elements having the semantic inflection "page-break". An empty span or div element is typically used to represent such page markers, as described in Example 2.

Example 2. A page marker

```
...  
<p>last paragraph of page 41</p>  
<div epub:type="pagebreak" id="42"/>  
<p>first paragraph of page 42</p>  
...
```

We will see in Section 2.2 how these page markers are referenced in a page list for easier navigation.

2.1.2.3. ARIA roles

[25], the W3C specification for Accessible Rich Internet Applications –developped within the Web Accessibility Initiative–, defines a `role` attribute that can be used to extend the semantics of HTML elements, in a similar fashion as EPUB does with the `epub:type` attribute. The default list of roles defined in ARIA 1.0 was primarily designed to represent semantics of rich internet applications, and therefore many of its members are seldom used for digital publications.

It shall be noted however that in the process of trying to align digital publishing with the Open Web Platform, the Digital Publishing Interest Group of W3C recently joined the ARIA Working Group to work on a Digital Publishing module for ARIA [4]. This module defines new ARIA roles specific to the needs of digital e-book publications. Armed with this new role ontology, the forthcoming EPUB 3.1 specification [7] (current an early editor's draft) will likely define the ARIA `role` attribute as the recommended method for inflecting semantics on HTML content, thereby superseding the `epub:type` attribute.

2.2. Navigability

The ability to easily navigate within a publication is crucial to accessibility. We've seen in the previous sections that properly structured content enables essential navigability features like the ability to escape from a block of content or the ability to jump to the next or previous semantically significant content. In addition to this in-flow navigability, it is equally important to be able to have access at once to any significant part of the publication.

In EPUB 3, direct navigation is made possible with the Navigation Document, which is a mandatory component of the publication. The Navigation Document is a specialized HTML document containing one or several `nav` elements that have a constrained content model. Each `nav` element is distinguished with a semantical inflection and represents a special navigation tree or list. An EPUB reading system will understand the navigation document and typically render it using a dedicated user interface (typically a navigation side bar or popup).

The only mandated navigation component is the table of content, colloquially referred to as the `toc nav`. In accessible publications, the table of content contains exhaustive references to all the sections that are part of the primary reading flow, regardless of their depth in the publication outline. Publishers may also want to include the Navigation Document in the primary reading order, as any other HTML content document. When they do, they can hide branches of the navigation tree from the rendering by using the `hidden` attribute. In any case, it is heavily recommended to represent the full publication hierarchy in the `toc nav` element.

Another very useful –albeit optional– component of the Navigation Document is the page list that we hinted at in Section 2.1.2.2. When the EPUB has an equiva-

lent print publication, it should contain a page list in order to be considered accessible. The page list is a `nav` element identified with the "pagelist" semantic inflection and consisting of a flat list of the publication's pages, in the reading order. Typically, if the publication content has been marked up as shown in Example 2, each link in the `pagelist nav` will point to a `pagebreak` page marker.

Finally, the Navigation Document is not limited to including a table of content and a page list. Many other navigation trees or lists can be described, such as the navigation to the "landmarks" of the publication, a list of figures, a list of tables, etc. In general, the more navigation components are provided, the more effectively a print-disabled reader will be able to navigate in the publication.

2.3. Adaptation

Print-disabled users and sighted users alike use a vast variety of systems to read digital publications. Some users read EPUBs on smartphones, some with a magnifying tool, or rendered on a desktop display, etc. An accessible EPUB is an EPUB that can adapt to all these various reading environments.

One aspect of EPUB that we haven't mentioned before is that Content Documents can be declared as being "reflowable" or "fixed layout". By default, unless explicitly declared otherwise, a content document is considered reflowable. It basically means that the rendering of the EPUB can adapt to the size of the reading viewport; reading systems often perform dynamic pagination, and the text reflows to adjust to the page dimensions. On the contrary, fixed-layout documents are intended to be rendered as a single page with a fixed size, regardless of the device's screen.

Fixed-layout EPUBs are sometimes created based on the false assumption that the digital publication should look exactly like its print equivalent. This creates obvious usability issues: because the content cannot adapt to the reading device, sighted users often have to madly scroll and zoom to visually navigate into a page. Worse, fixed-layout content is usually produced from layout-oriented visual designing authoring tools, and is often poorly semantically structured. Fixed-layout EPUBs often have rather bad accessibility properties.

For the reasons above, it is largely recommended to produce reflowable EPUBs. Although Fixed-layout EPUB make sense in some very specialized cases (e.g. for heavily design-oriented magazines, or for some children books), they are seldom a necessity. Even then, fixed-layout content producers should still strive to use semantically rich structure, which will often pay off for accessibility.

Another way that users adapt content to their needs, and which has a direct impact on accessibility, is by changing default settings like font size or background and highlight colors. Many reading systems provide built-in mechanisms to customize font- or color-related settings. These are often used for personal preferences, and sometimes for accessibility purposes. For instance, a dyslexic user may

want to force the use of a specialized font –like OpenDyslexic– that will make the text easier to read. It is therefore important to make sure that the EPUB’s design remains unobtrusive, does not rely too heavily on font or colors, and never as a mean of conveying significant information.

2.4. Alternative Content

While EPUB content is generally primarily textual, publications also often include non-text content like images and sometimes time-based content like video or audio. Accessible publications must ensure that this content –unless when used for pure decoration– is *perceivable* by the user. This is one of the top-level principle defined by the Web Content Accessibility Guidelines [26].

2.4.1. Images

The goldern rule is that images that are significant to the understanding of the publications must have a text alternative set using the `alt` attribute of the `img` element. This text alternative will for instance be read to users who rely on text-to-speech rendering. The HTML specification contains useful guidelines¹ on how to provide text alternatives.

A perhaps lesser known rule is that even images used for decorative purpose –which are not semantically significant–, should have an `alt` attribute, but with an empty value. Doing so enables assistive devices to ignore them entirely, which makes the image unobtrusive to the reading experience.

2.4.2. Beyond images

It is important to keep in mind that using images should be restricted to the representation of content that cannot be otherwise represented by semantically richer structures.

Tabular data, for instance, are always better marked-up with a `table` element than with an image. That will allow assistive devices to provide tabular navigation features, or to render the table content with text to speech.

Likewise, some graphics or diagrams can be represented with Scalable Vector Graphics [22]. SVG has built-in accessibility features, like the ability to declare a title, a longer description, focus areas, etc. Using SVG also opens the gate to some interesting –albeit experimental– advanced features like for instance the ability to sonify a line graph to make it perceivable to visually impaired users, as shown in the interactive “Sonifier” demo by Doug Schepers [23].

As for mathematical formulae, it is usually recommended to use MathML markup. Like other rich markup solutions, MathML has built-in elements and at-

¹ <https://www.w3.org/TR/html5/embedded-content-0.html#alt>

tributes to provide alternative text content; the `alttext` attribute can be used to provide simple description, and the `annotation-xml` element can be used when richer description is required. Note however that using MathML will not necessarily make a formula accessible *per se*. In his article “Is MathML Accessible?” [18], Peter Krautzberger from MathJax reminds that –apart from simply rendering the text alternatives– there is only so much an accessibility tool can do to extract meaningful information from presentational MathML. Using MathML is certainly a step forward, but it still requires paying attention to markup quality and providing text alternatives.

2.4.3. Long descriptions

Sometimes, a simple textual description is not enough to accurately describe an image and convey its information to visually impaired users. If your picture is worth a thousand words, good luck fitting these in an `alt` attribute. In this case, there are several better ways to describe at length an image, chart, diagram, or table.

In simplest cases, the textual content surrounding the image already conveys the information, as shown in Example 3. In this case, no further description is required.

Example 3. Description in the context

```
<p>The most common Czech beers are pale lagers of pilsner type
-like the one pictured below-, with characteristic transparent
golden colour, high foaminess and lighter flavour.</p>

```

When the descriptive paragraph is not directly adjacent to the image, ARIA can be used to make the association explicit, with the `aria-describedby` attribute, as shown in Example 4.

Example 4. Description associated with ARIA

```
<p id="czech-beer">The most common Czech beers are pale lagers
of pilsner type with characteristic transparent golden colour,
high foaminess and lighter flavour.</p>
<p>.... more content ...</p>

```

Another useful pattern is to embed the image in the HTML `figure` element, and add the description to a `caption` element. As shown in Example 5.

Example 5. Image and description in a figure

```
<figure>
  
  <figcaption>The most common Czech beers are pale lagers of
    pilsner type with characteristic transparent golden colour,
    high foaminess and lighter flavour.
  </figcaption>
</figure>
```

Finally, sometimes a long description is simply not available –or cannot be added– in the publication text. This case requires that the long description is placed in an external document. Unfortunately, at the time of writing there is no consensus on what is the best approach to provide external long descriptions. The `longdesc` attribute, for instance, which had been removed from HTML5 and has been recently reinstated as an HTML5 extension [13], suffers from being only usable for images (and not tables, for instance) and some browser vendors stated they will not implement it. Work is ongoing within W3C’s Digital Publishing Interest Group and ARIA Working Group to better define digital publishing’s requirements for extended description and provide an appropriate technical solution. In the meantime, the avid reader can follow the guidelines established by the DIAGRAM Center [6] or look at the analysis of alternative solutions as described by the Digital Publishing IG [5] (keeping in mind that this is a working document).

2.4.4. Timed media

Another type of media that requires alternative textual description is *timed media*, like video or audio content. A video may contain visual information that cannot be perceived by a visually impaired user. An audio track cannot be perceived by people with hearing impairment. In both cases, HTML’s `track` element can be used to include external timed text tracks. A detailed transcript can also be provided, using the techniques described in the preceding section.

2.5. Another way to read

Many reading systems are able to render EPUB content as auditory information with text-to-speech technologies. In EPUB 3, it is also possible to provide a built-in audio representation, synchronized with the HTML content documents. The specification describing this mechanism is called EPUB Media Overlays [11]. It relies on the Synchronized Multimedia Integration Language [20] to describe the timing of a pre-recorded audio representation, linked with the related fragments of EPUB content documents. A simple Media Overlay document is shown in Example 6.

Example 6. A simple Media Overlay document

```
<smil xmlns="http://www.w3.org/ns/SMIL" version="3.0">
  <body>
    <par id="par1">
      <text src="chapter1.xhtml#sentence1"/>
      <audio src="chapter1_audio.mp3" clipBegin="0s" clipEnd="10s"/>
    </par>
    <par id="par2">
      <text src="chapter1.xhtml#sentence2"/>
      <audio src="chapter1_audio.mp3" clipBegin="10s" clipEnd="20s"/>
    </par>
    <par id="par3">
      <text src="chapter1.xhtml#sentence3"/>
      <audio src="chapter1_audio.mp3" clipBegin="20s" clipEnd="30s"/>
    </par>
  </body>
</smil>
```

Publications with Media Overlays are particularly useful for people with print disability; they are also popular for educational content or children literature. One of the benefit of providing pre-recorded audio is simply is that the publisher has a fine control over the quality of the audio. Whether it uses narration by professional voice actors, or text-to-speech output from heavily customized system, pre-recorded audio usually surpasses by far the quality of an on-the-fly text-to-speech rendering. Additionally, Media Overlays-aware reading systems can provide usability features like being able to navigate in the audio structure ("jump to the previous/next audio phrase"), or to synchronize the highlighting of text content, or define customized highlighting color.

Beyond the traditional full-text, full-audio book, Media Overlays can also be used to implement other kind of publications, like audio-only EPUB books. The DAISY Consortium notably provides a set of guidelines to represent navigable audio-only EPUB 3 [2].

2.6. Metadata

Although until now we've primarily discussed the *representation* of accessible content, another crucial aspect of accessible publishing is to provide quality *metadata* about the publication. In particular, metadata can be used to describe the accessibility features of the publication. With accurate metadata, print disabled users will be able to know if their particular accessibility needs are covered; metadata can also be used by search engines to improve the discoverability of accessible material. Picture yourself going to a library, buying the latest novel of your favorite author, and –once back at home and sitting comfortably in your cozy armchair– finding out that the novel is actually represented in morse code and absolutely

unreadable to you. Well, this is the kind of problems that print disabled users have to face constantly, typically when using mainstream distribution channels.

Two metadata vocabularies are especially relevant to describe the accessibility features of a publication. One is ONIX Code List 196 [17], part of the ONIX metadata standard developed by EDItEUR. Another is the properties defined by the Accessibility Metadata Project and contributed to the schema.org vocabulary [19].

3. Getting it done

The previous sections presented some best practices that should be followed to represent accessible digital publications with EPUB 3. This section intends to give some guidance and pointers on how to implement it effectively.

3.1. Plan for it

The most fundamental advice is maybe that implementing accessibility needs to be planned upfront. Adding accessibility features to an existing inaccessible EPUB can be very costly, and is never a sensible long term approach. In an ideal world, digital publications would be born accessible. Unfortunately, far too often, mainstream publications need to be heavily reworked –sometimes from scratch– by specialized organizations.

Establishing a publication process with accessibility in mind is a first step. Accessibility needs to be taken into account in every step of the process. The tools used to create publications need to be accessible themselves, the in-house documentation system needs to be accessible, the retailing system needs to be accessible.

The importance of taking accessibility into account in the early steps of the process is perhaps best exemplified by looking at structural and semantical quality, as described in Section 2.1. Consider on the one hand a publication workflow putting the initial focus on print layout, with a visual design tool exporting an EPUB with low markup quality. Enriching or fixing this markup will be very time consuming, resulting in half-baked accessibility. On the other hand, a publication workflow designed with accessibility in mind will often focus on data quality very early in the process, which will allow to more easily and effectively produce key EPUB components like table of content, page lists, package documents, etc. At the end of the day, the result will be more accessible, and the overall cost reduced.

3.2. Follow the guide

Accessibility is related to many transversal aspects of the Web and Digital Publishing; knowing all the pitfalls and techniques can be challenging. Fortunately,

several guideline documents are available and can prove to be invaluable resources.

The first set of guidelines are the Web Content Accessibility Guidelines [26], developed at W3C. Because EPUB 3 is largely based on Web technologies, most of the WCAG recommendations apply. WCAG consist in a rather concise set of guidelines organized under 4 principles: *perceivable*, *operable*, *understandable*, and *robust*. In addition, it is complemented by more extensive supporting material which provides advanced guidance and specific details on how to develop accessible content.

The second set of guidelines are the EPUB 3 Accessibility Guidelines [10], and have been developed by Matt Garrish at IDPF. Rather than a normative document, it describes various concrete techniques that will help in the creation of accessible EPUB 3 content. It is a good companion to the “Accessible EPUB 3” book [9] by the same author.

3.3. Use the right tools

Tooling can play a significant role in getting accessibility right –or wrong, for that matter, as we’ve seen in earlier sections.

While giving a complete list of tools would be out of the scope of this article, the following list is an example of a few approaches or solutions that can be used as part of an EPUB production system:

- **XML processing languages** (e.g. XSLT and XProc): automated processing of XML or HTML –as done for instance by le-tex transpect or DAISY Pipeline– is an effective approach to EPUB production. The processing steps can gradually work to enrich markup, extract semantically significant data and automatically generate key EPUB components like the table of content, the page list, the package document, etc.
- **DAISY Tobi** [24] and **Obi** [16] are authoring tools for DAISY talking books and accessible EPUB 3 with Media Overlays. Tobi is designed to ease the process of synchronizing a text document with human narration. Obi is designed to produce audio-only books. Both support live recording or importing pre-recorded audio files.
- **MathML cloud** [15] is a tool to convert math expressions written in a variety of formats (LaTeX, asciimath, MathML) into text descriptions, PNG images, SVGs, MathML. Math Cloud adopts a SaaS model intended to be integrated in production workflows.
- **aeneas** [1] is a library and set of tools to automagically synchronize audio and text. With a little further processing, the synchronization map produced by aeneas can easily be used to produce EPUB Media Overlays. Aeneas is also deployed as a web application that can be used in a SaaS approach.

All the tools described above are open source.

3.4. Test it!

The final advice is to test the publications in various environments, with various reading systems, and with various assistive devices. There is only so much an automated tool –like EpubCheck or specialized accessibility checkers– can report. At the end of the day, regardless of how diligent you were in following the guidelines, evaluating accessibility will require the intervention of human judgement.

4. Conclusion

Providing equal access to knowledge and information, regardless of print disabilities, is a fundamental human right. By making accessibility one of its core design principle, the EPUB 3 set of specifications is bringing positive change to inclusive publishing, which keeps improving with the successive standard revisions. The best practices described in this article provide an overview of the techniques and approaches to designing accessible EPUB 3 publications. The set of tips provided in the last section hint that producing accessible EPUB 3 is far from being an insurmountable endeavor. In fact, various tools and guidelines exist, and committing to accessibility will result in generally higher quality EPUB 3 content, with improved usability for all.

In the coming months, the DAISY Consortium, together with its members, supporters, and partners, will work on establishing a “Baseline for Born Accessible EPUB”. This project will provide minimum requirements, clear guidance for publishers, and assurance to consumers on the minimum accessibility they can expect. By collaborating and pooling resources, we can change the world!

Bibliography

- [1] Alberto Pettarin. *Aeneas, a library to automagically synchronize audio and text*. ReadBeyond. 2015. <https://github.com/readbeyond/aeneas> .
- [2] Avneesh Singh. *Navigable audio-only EPUB3 Guidelines*. DAISY Consortium. 2015. <http://www.daisy.org/ties/navigable-audio-only-epub3-guidelines> .
- [3] Committee on the Rights of Persons with Disabilities. *Convention on the Rights of Persons with Disabilities*. United Nations Office for the High Commissioner for Human Rights. 30 March 2007. <http://www2.ohchr.org/english/law/disabilities-convention.htm#21> .
- [4] Matt Garrish, Tzviya Siegman, Markus Gylling, and Shane McCarron. *Digital Publishing WAI-ARIA Module 1.0*. W3C Working Draft. W3C. <https://www.w3.org/TR/dpub-aria-1.0/> .

- [5] Digital Publishing Interest Group. *Extended Description Analysis*. Working Document. W3C. 2015. <https://w3c.github.io/dpub-accessibility/extended-description-analysis.html> .
- [6] DIAGRAM Center. *Image Guidelines for EPUB 3*. DIAGRAM Center. <http://diagramcenter.org/59-image-guidelines-for-epub-3.html> .
- [7] Markus Gylling, Tzviya Siegman, and Matt Garrish. *EPUB 3.1*. Editor's Draft. IDPF. 30 January 2016. <http://www.idpf.org/epub/31/spec/epub-spec-20160130.html> .
- [8] Markus Gylling, William McCoy, Erika J. Etemad, and Matt Garrish. *EPUB Content Documents 3.0.1*. Recommended Specification. IDPF. 26 June 2014. <http://www.idpf.org/epub/301/spec/epub-contentdocs-20140626.html> .
- [9] Matt Garrish. *Accessible EPUB 3*. O'Reilly. 2012. <http://shop.oreilly.com/product/0636920025283.do> .
- [10] Matt Garrish. *EPUB 3 Accessibility Guidelines*. IDPF. <http://www.idpf.org/accessibility/guidelines/> .
- [11] Marisa DeMeglio and Daniel Weck. *EPUB Media Overlays 3.0.1*. Recommended Specification. IDPF. 26 June 2014. <http://www.idpf.org/epub/301/spec/epub-mediaoverlays-20140626.html> .
- [12] Ian Hickson et al.. *HTML5*. W3C Recommendation. W3C. 28 October 2014. <http://www.w3.org/TR/html5/> .
- [13] Charles McCathieNevile and Mark Sadecki. *HTML5 Image Description Extension (longdesc)*,. W3C Recommendation. W3C. 26 February 2015. <https://www.w3.org/TR/html-longdesc/> .
- [14] MathJax. *MathJax Semantic Enrichment project*. MathJax. <https://github.com/mathjax/MathJax/wiki/Semantic-Enrichment-project> .
- [15] Benetech. *MathML Cloud*. Benetech. <https://www.mathmlcloud.org/> .
- [16] DAISY Consortium. *Obi*. DAISY Consortium. <http://www.daisy.org/project/obi> .
- [17] EDItEUR. *ONIX for Books*. EDItEUR. 24 January 2016. <http://doi.org/10.4400/akjh> .
- [18] Peter Krautzberger. *Is MathML Accessible?*. 15 Nov 2015. <http://www.idpf.org/accessibility/guidelines/content/about.php> .
- [19] Accessibility Metadata Project. *Schema.org properties, Accessibility Metadata Project*,. Accessibility Metadata Project. 2013. <http://www.a11ymetadata.org/the-specification/> .

- [20] Dick Bulterman. *Synchronized Multimedia Integration Language (SMIL 3.0)*. W3C Recommendation. W3C. <http://www.w3.org/TR/SMIL3/> .
- [21] *EPUB Structural Semantics Vocabulary*. IDPF. <http://www.idpf.org/epub/vocab/structure/#> .
- [22] Erik Dahlström et al.. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. W3C. <http://www.w3.org/TR/SVG11/> .
- [23] Doug Schepers. *Invisible Visualization*. 22 april 2014. <http://schepers.cc/invisible-visualization> .
- [24] *Tobi, DAISY Consortium,*. <http://www.daisy.org/project/tobi> .
- [25] James Craig et al.. *Accessible Rich Internet Applications (WAI-ARIA) 1.0*. W3C Recommendation. W3C. <http://www.w3.org/TR/wai-aria/> .
- [26] Ben Caldwell et al.. *Web Content Accessibility Guidelines (WCAG) 2.0*. W3C Recommendation. W3C. 11 December 2008. W3C. <https://www.w3.org/TR/WCAG20/> .

Extending CSS with XSL-FO, XSL-FO with CSS

Tony Graham
Antenna House, Inc.
<tony@antennahouse.com>

Abstract

Discusses the Antenna House approach to merging the features of XSL-FO and CSS in AH Formatter. This showcases some of the features of one stylesheet language that have crossed between the two flavours of AH Formatter to become an extension in the other; for example, CSS numbering styles in XSL-FO and XSL-FO table header and footer control in CSS. AH Formatter uses a common layout engine when formatting either XSL-FO or CSS, plus there are a lot of common properties in XSL 1.1 and CSS 2.0. Allowing properties to cross over to the other stylesheet language can be as simple as adding a prefixed property in CSS or adding a namespaced property in XSL-FO. In some cases, however, only part of the implementation in one language can or should be reimplemented for the other, and there are some parts for which it is not practical to reimplement for the other stylesheet language.

1. Introduction

AH Formatter [1] from Antenna House [2] is unique in offering formatting of XML and HTML using either XSL-FO or CSS to produce PDF, PostScript, SVG, and a range of other output formats.

There are many similarities between the properties supported by XSL-FO and those of CSS. It was something of a shotgun wedding, and it was followed by an acrimonious separation, but there was a time during the development of XSL 1.0 and CSS 2 where the emphasis was on aligning XSL-FO and CSS. Their processing models and syntaxes are very different, but at the time of XSL 1.0 and CSS 2, they shared the same models for borders, margins, padding, font properties, and many other aspects of styling markup. And, thanks to backwards compatibility, that's still largely true.

AH Formatter was originally developed as an XSL-FO-only formatter. CSS support was added by using the CSS to drive the underlying layout engine, rather than somehow translating the CSS into XSL-FO and formatting that XSL-FO. A side-effect, if you like, of implementing CSS features in a layout engine that already implements XSL-FO is that features of CSS that fit the syntax and process-

ing model of XSL-FO can be made available to XSL-FO, and, obviously, features of XSL-FO that can fit the syntax and processing model of CSS can also be made available to CSS stylesheets.

2. Reimplementation strategies

There are several ways to approach reimplementing a CSS feature in XSL-FO or an XSL-FO feature in CSS. These include:

- **Full implementation** – There is a direct correspondence between every aspect of the standard feature of one stylesheet language and its reimplementation as an extension to the other stylesheet language. For example, the CSS `@counter-style` ‘at-rule’ and associated properties reimplemented as an `axf:counter-style` extension element and associated properties.
- **Partial implementation** – Some parts of a feature of one stylesheet language either doesn’t map well to the other or overlaps with a standard feature of the other, so only part of the feature can usefully be reimplemented as an extension to the other stylesheet language. For example, many but not all of the features of the support for XSL-FO footnotes could be reused when implementing support for CSS footnotes, plus some of the extra features required for CSS footnotes were reimplemented for XSL-FO as extension properties and extension elements.
- **Different surface syntax** – Sometimes the same or quite similar constructs have quite different expressions in CSS and XSL-FO but boil down to the same sorts of areas on the formatted page. For example, both the CSS 3 named strings and running elements constructs can be and were implemented using the machinery underlining the implementation of XSL-FO `fo:marker` and `fo:retrieve-marker` formatting objects.
- **Don't implement** – Either the second stylesheet language already has an equivalent feature or the different approaches of the two languages make it impractical to reimplement a feature in the other language. For example, the CSS `list-style`, `list-style-image`, and `list-style-position` properties are not going to be reimplemented for XSL-FO because XSL-FO already allows more control over the layout and content of list item labels than CSS does, and the XSL-FO `fo:page-sequence-master` [3] is not reimplemented for CSS because CSS has its own page selection mechanism.

2.1. Identifying reimplemented features

Some CSS or XSL-FO features of AH Formatter are easy to identify as reimplementations of features of the other stylesheet language, but others aren’t. Identifying reimplemented properties is mostly easy, since the AH Formatter Online

Manual [4] includes a chapter [5] listing all of the XSL-FO and CSS properties that AH Formatter implements (see Figure 1).

XSL/CSS Properties List

The following table shows XSL-FO properties and their corresponding CSS properties. A blank cell indicates that no corresponding CSS property is implemented for the XSL-FO property, or vice-versa. See also [XSL-FO Conformance](#) and [CSS Conformance](#) to learn the current implementation status for each property. [CSS Conformance](#) also defines the CSS module abbreviations, such as [CSS3-GCPM] etc., that are used in the table. Showing properties as corresponding does not always mean that their specifications in XSL-FO and CSS are completely aligned. Some correspondences just indicate that the properties are functionally equivalent or mostly similar.

XSL	CSS	Description
axf:abbreviation-character-count	-ah-abbreviation-character-count	Specifies the minimum number of characters considered to be an abbreviation.
7.6.1 absolute-position	[CSS2.1] position	
axf:action-type	-ah-action-type	Specifies the action of External Link or Form Actions .

Figure 1. AH Formatter XSL/CSS Properties List (detail)

In general:

- XSL-FO properties with an `axf:` prefix are Antenna House extensions to XSL-FO.
- CSS properties with an `-ah-` prefix may be Antenna House extensions or may be still under development by the W3C CSS Working Group.
- CSS properties with an `(-ah-)` prefix can be used with or without an `-ah-` prefix: they can be used unprefixed because they have been standardised by the CSS WG, but the prefixed form is still supported for backwards compatibility with earlier AH Formatter versions.
- Unprefixed CSS properties are implemented by AH Formatter without previously being implemented in a prefixed form.
- Numbers before an XSL-FO property name indicate the section of the XSL 1.1 Recommendation in which the property is defined.
- An abbreviation in square brackets before a CSS property name indicates the CSS module defining the property.

so:

- A CSS property without a CSS module abbreviation that corresponds to an XSL-FO property with an XSL 1.1 section reference is likely to be an XSL-FO property reimplemented for CSS.
- An XSL-FO property with an `axf:` prefix that corresponds to a CSS property with a CSS module abbreviation is likely to be a CSS property reimplemented for XSL-FO.

Some of the CSS “at-rule” [6] supported by AH Formatter have been reimplemented for XSL-FO as extension elements. The “Extended Elements” [7] section of the Online Manual shows both the XSL-FO and CSS forms for these.

There are currently no XSL-FO formatting objects that are reimplemented for CSS as custom values of the CSS `display` property, but the areas generated by some formatting objects can, obviously, also be generated by some CSS features.

2.2. Full implementation

2.2.1. CSS to XSL-FO: Counter styles

A “counter style” is the definition and/or implementation of the sequence of numbers, letters, and/or symbols to use to represent a numbering sequence. CSS 1 defined a handful of counter styles [8] based on what HTML traditionally allowed on lists. CSS Counter Styles Level 3 [9] defines the ‘@counter-style’ rule, which provides a mechanism for defining custom counter styles, plus it defines a number of counter styles that should all (eventually) be expected to be built into browsers.

The core of a CSS 3 counter style is that it attaches a name to an algorithm for generating string representations of integer counter values. A counter style may also include properties indicating a prefix and/or suffix to add to the generated values, additional strings to indicate negative numbers, etc. The counter style can be used in the ‘list-style-type’ and in the CSS ‘counter()’ and ‘counters()’ functions. Example 1 shows a ‘my-cjk-decimal’ counter style that is a copy of the ‘cjk-decimal’ counter style from CSS Counter Styles Level 3. As the name suggests, the counter style uses the ideographs for zero to nine to represent decimal numbers, and the numbers are followed by an ideographic comma suffix. The counter style is used when numbering the items in an `ol`.

Example 1. CSS3 @counter-style

```
<style type='text/css'>
@counter-style my-cjk-decimal {
  system: numeric;
  range: 0 infinite;
  symbols: \3007 \4E00 \4E8C \4E09 \56DB \4E94 \516D \4E03 \516B ►
\4E5D;
/* 〇 一 二 三 四 五 六 七 八 九 */
  suffix: "\3001";
/* ", " */
}
ol li { list-style-type: my-cjk-decimal; }
/* the following CSS is not part of the test */
.test { font-size: 25px; }
ol { margin: 0; padding-left: 8em; }
</style>
...
```

```
<ol>
  <li title="1">一</li>
  <li title="2">二</li>
</ol>
```

Example 2 shows the equivalent XSL-FO markup. The counter style is declared using an `axf:counter-style` element. The element has attributes corresponding to the CSS properties. The `axf:number-transform` property on each `fo:list-item-label/fo:block` refers to the counter style, so the contents of the `fo:block` is formatted using the counter style. The XSL-FO markup is more verbose than the HTML/CSS partly because it is meant to be generated, not authored directly, and partly as a consequence of XSL-FO allowing you to put practically anything – even a table, if that’s what you want – as the list item label.

Example 2. counter-style in XSL-FO

```
<fo:declarations>
  <axf:counter-style name="my-cjk-decimal" system="numeric" range="0 ►
infinite"
symbols="'〇' '一' '二' '三' '四' '五' '六' '七' '八' '九'" suffix="、'"/>
</fo:declarations>
...
<fo:list-block provisional-distance-between-starts="60mm"
provisional-label-separation="5mm">
  <fo:list-item>
    <fo:list-item-label start-indent="5mm" end-indent="label-end()">
      <fo:block text-align="right" color="red"
axf:number-transform="my-cjk-decimal">1</fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()"><fo:block>一</fo:block>
    </fo:list-item-body>
  </fo:list-item>
  <fo:list-item>
    <fo:list-item-label start-indent="5mm" end-indent="label-end()">
      <fo:block text-align="right" color="red"
axf:number-transform="my-cjk-decimal">2</fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()"><fo:block>二</fo:block>
    </fo:list-item-body>
  </fo:list-item>
</fo:list-block>
```

The CSS example is based on an `@counter-styles` test from the W3C I10n WG, and the corresponding XSL-FO example includes literal numbers to be formatted with the counter style, but `axf:number-transform` can be used with formatting objects that generate numbers – such as `fo:page-number` – as well.

There are two more usual ways to generate numbers to be formatted using XSL-FO:

- Generate literal numbers in the XSL-FO using `xsl:number` [10] in the XSLT transformation stage.
- Format literal numbers using the properties for number-to-string conversion: `format`, `grouping-separator`, `grouping-size`, and `letter-value` [11].

In both cases, however:

- Only a few formats are defined for XSLT or XSL-FO, and anything else is implementation-defined, whereas CSS-style counter styles both have more predefined formats and allow definition of custom styles within the current document.
- Formats for large numbers – e.g., numbers up to 9,999 for several CJK numbering styles – is better defined for the predefined CSS-style counter styles than for the predefined XSLT and XSL-FO formats.

2.2.2. XSL-FO to CSS: Omitting table header/footer at break

The styling of tables is one of the few areas of CSS 3 that has not advanced from CSS 2.1. The CSS3 Tables Module [12] is not actively maintained and table styling “consists of the text of the CSS2 chapter on tables, with almost no changes yet.”

The XSL-FO table model is closely aligned with the CSS 2 table model. For example, automatic table layout and fixed table layout are defined simply by referring to the definitions in section 17.52 of the CSS 2 specification [13], plus many of the table-related properties are identical to CSS properties. XSL 1.1 defines some additional properties related to tables which aren’t in CSS 2 and, obviously, haven’t been added since. Two of these are `table-omit-header-at-break` [14] and `table-omit-footer-at-break` [15], which control whether or not the table header (or footer) is repeated where the table breaks across a page.¹ They are available in CSS stylesheets as `-ah-table-omit-header-at-break` and `-ah-table-omit-footer-at-break`. Their effect on the document in Example 3 is shown in Figure 2.

Example 3. Table samples

```
<style type='text/css'>
@page {
  size: 50mm 50mm;
}
table {
  page-break-before: always;
}
```

¹AH Formatter extends the `true` and `false` values with a `column` value [16] for specifying that the header (or footer) is omitted at column breaks but not page breaks.

```

width: 100%;
border-collapse: collapse;
}
thead, tfoot {
background-color: #f0fff0;
}
td {
border: thin solid black;
}
.omit {
-ah-table-omit-header-at-break: true;
-ah-table-omit-footer-at-break: true;
}
</style>
...
<table>
  <thead>
    <tr><td>thead</td></tr>
  </thead>
  <tfoot>
    <tr><td>tfoot</td></tr>
  </tfoot>
  <tbody>
    <tr><td>1</td></tr>
    ...
    <tr><td>10</td></tr>
  </tbody>
</table>

<table class="omit">
...
</table>

```

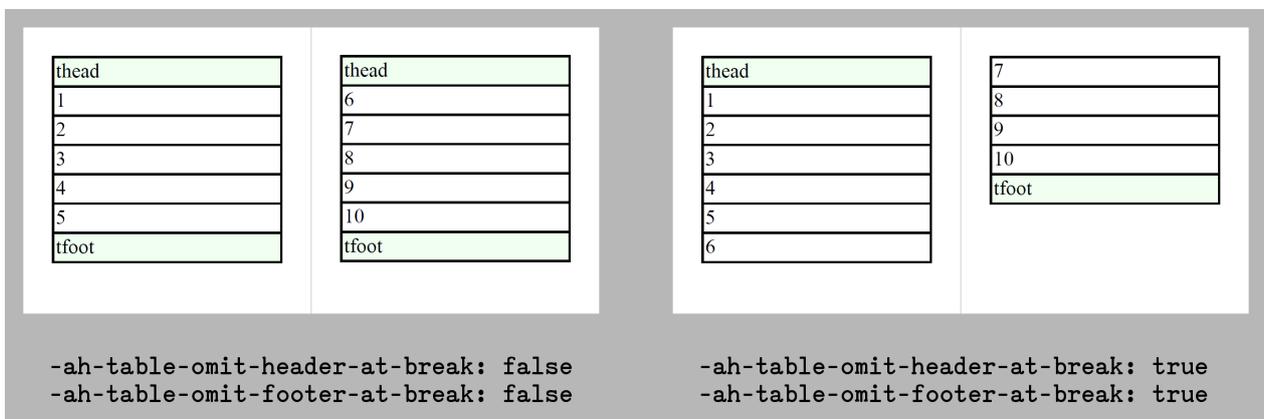


Figure 2. -ah-table-omit-header-at-break and -ah-table-omit-footer-at-break

2.3. Partial implementation

2.3.1. Footnotes

Footnotes in AH Formatter are both a partial implementation of XSL-FO features for CSS and a partial implementation of CSS features for XSL-FO.

XSL-FO footnotes are generated using the `fo:footnote` and `fo:footnote-body` formatting objects plus an `fo:inline` for the footnote citation. The contents of the `fo:inline`, as well as any corresponding number or symbol on the footnote itself, are expected to be included in the XSL-FO document. Example 4 shows XSL-FO markup for an `fo:block` containing an `fo:footnote`.

Example 4. XSL-FO footnote

```
<fo:block>
XML 文書をきれいに表示・印刷するための仕様である XSL 1.1 に対応しており<fo:footnote>
  <fo:inline baseline-shift="super" font-size="0.75em">(3)</fo:inline>
  <fo:footnote-body>
    <fo:block font-size="0.9em" text-indent="0em">
      <fo:inline baseline-shift="super" font-size="0.75em">(3)</fo:inline>
      詳細はオンラインマニュアルの「XSL 仕様の実装状況」を参照してください。
    </fo:block>
  </fo:footnote-body>
</fo:footnote>
、また W3C で策定作業中の CSS Level 3 のページ媒体向け仕様によるレイアウト指定のページ
組版にも対応しています。
</fo:block>
```

CSS footnotes are generated by setting the `float` (or `-ah-float`) property value to `footnote` [17][18]. The `::footnote-call` and `::footnote-body` pseudo-elements specify the styling for the footnote citation and footnote, respectively. Example 5 shows CSS styles and HTML markup for the same footnote text. Figure 3 shows

Example 5. CSS footnote

```
.footnote {
  -ah-float: footnote;
  margin-left: 3em;
}

::footnote-call {
  content: "("counter(footnote)");";
  font-size: 8pt;
  vertical-align: super;
}
```

```

::footnote-marker {
  content: " ("counter(footnote) " ) ";
  font-size: 8pt;
  vertical-align: super;
}
...
<p>XML 文書をきれいに表示・印刷するための仕様である XSL 1.1 に対応しており
<span class="footnote">詳細はオンラインマニュアルの「XSL 仕様の実装状況」を参照して
ください。
</span>、また W3C で策定作業中の CSS Level 3 のページ媒体向け仕様によるレイアウト指定
のページ組版にも対応しています。</p>

```

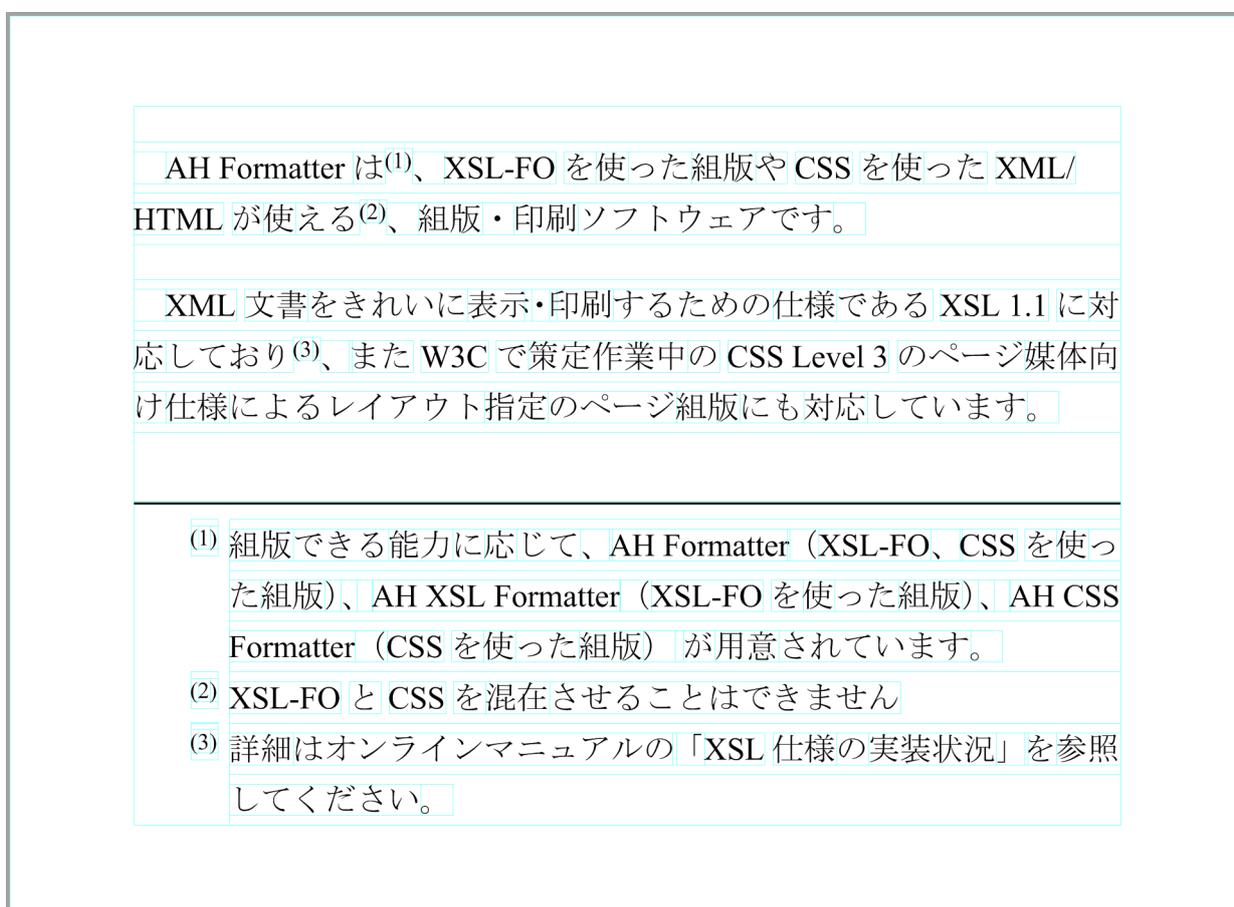


Figure 3. CSS footnotes

So far, so good: XSL-FO and CSS have similar concepts for the components of a footnote, so much of the existing XSL-FO implementation of footnote processing in the layout engine would have been reusable for the CSS implementation. However, there are also differences that preclude simply reusing the XSL-FO machinery for CSS:

- In XSL-FO, the “footnote-reference-area”, which contains the formatted footnotes, is implicitly present in every `fo:region-body` [19], whereas in CSS, the page area used to display footnotes is defined using an `@footnote` rule [20], which might not be defined for every page (since the example user agent stylesheet is only informative).
- In CSS, the `max-height` property applies to the footnote area (unless it is the last page of the document), whereas in XSL-FO, there are no controls over the height of the implicit footnote-reference-area.
- In CSS, the number or symbol for the footnote is expected to be generated by the CSS formatter using the value of the predefined “footnote counter” [21], whereas in XSL-FO, the number or symbol is expected to be present in the XSL-FO document since it is expected to be generated during the XSLT stage.

The additional CSS features of maximum height for the footnote area and generating the footnote number or symbol within the formatter have been reimplemented for XSL-FO as `axf:footnote-max-height` [22] and `<axf:footnote-number>` [23], respectively.

2.4. Different surface syntax

2.4.1. Running headers and footers

XSL-FO provides more control over the position and content of running headers and footers than is currently defined for CSS. As such, it was possible to implement both CSS named strings [24] and CSS running elements [25] using the existing machinery provided by the layout engine.

The XSL-FO `fo:simple-page-master` [26] can have four ‘outer’ regions that can be used for running headers and footers, as Figure 4 shows. The content for the running headers and footers generated on a page come from `fo:static-content` [27] in the current `fo:page-sequence`, and, as Figure 5 shows, different `fo:static-content` can be directed to specific regions on particular page masters. The `fo:static-content` can contain block-level formatting objects that contain hard-coded text (‘static’ static-content, if you like) but can also contain `fo:page-number`, which renders as the current page number, and `fo:retrieve-marker`, which renders as content from `fo:marker` formatting objects from among the formatting objects that generated the areas on the current page (or on a preceding page).

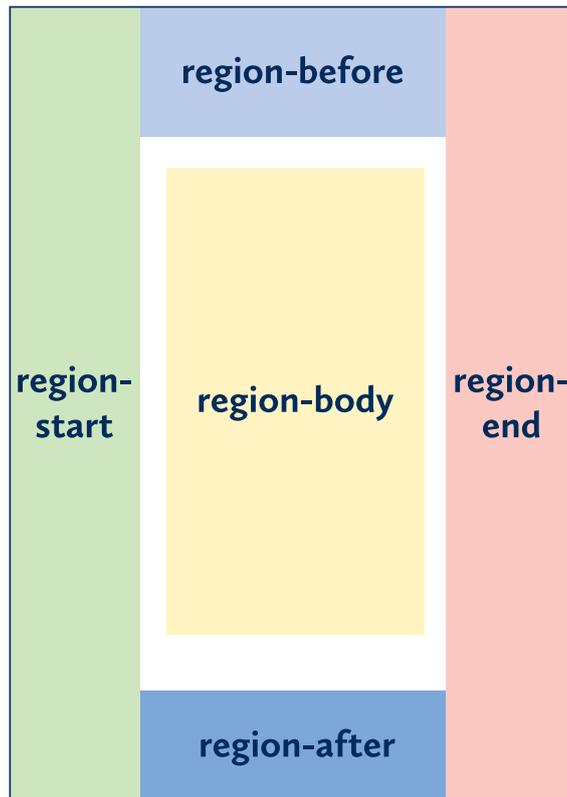
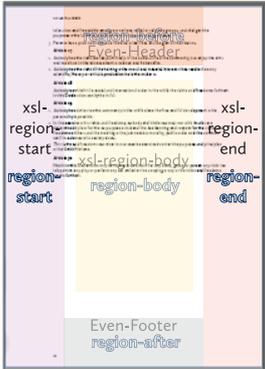


Figure 4. fo:simple-page-master regions

First
Even
Odd
Even



```

<fo:static-content flow-name="Even-Header">
  <fo:block>UDHR in Unicode</fo:block>
</fo:static-content>
<fo:static-content flow-name="First-Outside">
  <fo:block-container reference-orientation="270">
    ...
  </fo:block-container>
</fo:static-content>
<fo:static-content flow-name="Odd-Outside">
  <fo:block-container reference-orientation="270">
    ...
  </fo:block-container>
</fo:static-content>

```

```

<fo:static-content flow-name="Odd-Header">
  <fo:block>...</fo:block>
</fo:static-content>
<fo:static-content flow-name="Even-Footer">
  <fo:block><fo:page-number/></fo:block>
</fo:static-content>
<fo:static-content flow-name="Odd-Footer">
  <fo:block><fo:page-number/></fo:block>
</fo:static-content>
<fo:flow flow-name="xsl-region-body" id="sco">
  ...
</fo:flow>

```

Figure 5. fo:static-content directed to regions on page masters

CSS provides two mechanisms for generating running headers and footers: named strings and running elements. Named strings copy text from the document and/or stylesheet to one of 16 ‘margin boxes’ around the edges of the page, shown in Figure 6. Since the machinery implemented for running headers and footers in XSL-FO includes the ability to place areas from block-level formatting objects in the outer regions, it was possible to implement the margin boxes for the named strings using the same machinery despite the differences in syntax between the two stylesheet languages.

The content of named strings comes from elements on the current page (or a previous page), but there may be multiple instances of that type of element on the page. The `string()` function [28], which copies the value of the named string, has an optional second parameter for indicating whether the first, last, or another instance should be used in the margin box. The XSL-FO `fo:retrieve-marker` has a similar facility through the `retrieve-position` property, so, again, the named string selection could be implemented based on the machinery available for XSL-FO.

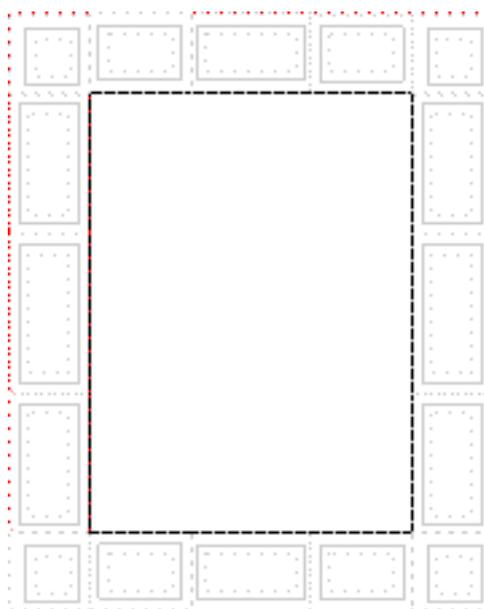


Figure 6. CSS margin boxes

CSS running elements expand upon named strings by removing entire elements from the normal flow and making them available for placement in a margin box so that, for example, a book title in a running head may include an italic word. Again, `fo:retrieve-marker` can retrieve more than just text, so running elements could also be implemented based on the machinery implemented for XSL-FO.

2.5. Don't implement

2.5.1. Page selection

XSL-FO and CSS have different but largely compatible mechanisms for specifying page sizes, but they differ markedly in how pages are selected for use during processing.

CSS has `@page` rules [29] for specifying page sizes, etc. `@page` rules can have `:left`, `:right`, `:first`, and `:blank` pseudo-classes so a CSS formatter can use different `@page` rules in different contexts. `@page` rules may also have a name [30], and a CSS formatter will, if necessary, force a page break to ensure that an element is formatted on a page from an `@page` rule with a name that matches the element's `page` property value. Example 6, from the CSS Paged Media Module Level 3 Working Draft [31], shows the styles and markup for two tables that are rendered on landscape pages (which could be the same page, if they would both fit) followed by a `<p>` that is rendered on the "narrow" page type. A CSS formatter would have to force a page break after the second table, if necessary, as part of switching to a new page size.

Example 6. CSS named page

```
@page narrow { size: 9cm 18cm }
@page rotated { size: landscape }
div { page: narrow }
table { page: rotated }
...
<div>
<table>...</table>
<table>...</table>
<p>This text is rendered on a 'narrow' page</p>
</div>
```

In contrast, an XSL-FO document has `fo:simple-page-master` for defining page size, etc., and the sequence of page masters to use for an `fo:page-sequence` is defined in an `fo:page-sequence-master`. XSL-FO has `fo:repeatable-page-master-alternatives` and `fo:conditional-page-master-reference` [32] for determining which page master to use for an odd, even, first, last, rest, or any and/or blank or not blank page in a page sequence. The number of pages produced by an `fo:page-sequence` can be fixed, the `fo:page-sequence` can be set to start or end on either an odd or an even page, and there can be a maximum number of repeats set for any of the sets of page master alternatives specified in the `fo:page-sequence-master`, but there is no facility in XSL-FO for breaking to an alternative page master based on anything in the XSL-FO markup contained by the `fo:page-sequence`.

These differences in their page selection mechanisms between CSS and XSL-FO over when to switch page types/page masters have meant that it is not practical to reimplement one for the other.

3. Conclusion

The extent to which Antenna House has and hasn't been able to merge the features of XSL-FO and CSS by implementing parts of one as extensions in the other illustrates both that neither stylesheet language specification is complete and that there are some fundamental differences of approach that either can't be bridged or aren't worth being bridged.

Bibliography

- [1] Antenna House Formatter V6. <http://www.antennahouse.com/antenna1/formatter/>
- [2] <http://www.antennahouse.com/>
- [3] https://www.w3.org/TR/xsl11/#fo_page-sequence-master
- [4] <http://www.antennahouse.com/product/ahf60/docs/>
- [5] <http://www.antennahouse.com/product/ahf60/docs/ahf-focss6.html>
- [6] https://drafts.csswg.org/css-syntax/#at_rule
- [7] <http://www.antennahouse.com/product/ahf60/docs/ahf-ext.html#ext-element-conf>
- [8] <https://www.w3.org/TR/REC-CSS1/#list-style>
- [9] CSS Counter Styles Level 3 <https://www.w3.org/TR/css-counter-styles/>
- [10] <https://www.w3.org/TR/xslt20/#number>
- [11] <https://www.w3.org/TR/xsl11/#d0e29313>
- [12] <https://drafts.csswg.org/css3-tables/>
- [13] <http://www.w3.org/TR/REC-CSS2/tables.html#width-layout>
- [14] <https://www.w3.org/TR/xsl11/#table-omit-header-at-break>
- [15] <https://www.w3.org/TR/xsl11/#table-omit-footer-at-break>
- [16] <http://www.antennahouse.com/product/ahf60/docs/ahf-ext.html#axf.table-omit-footer-at-break>
- [17] <https://www.w3.org/TR/2011/WD-css3-gcpm-20111129/#footnotes>

- [18] <http://www.antennahouse.com/product/ahf60/docs/ahf-float.html#FootnoteCSS>
- [19] https://www.w3.org/TR/xsl11/#fo_region-body
- [20] <https://drafts.csswg.org/css-gcpm/#footnote-area>
- [21] <https://drafts.csswg.org/css-gcpm/#footnote-counters>
- [22] <http://www.antennahouse.com/product/ahf60/docs/ahf-ext.html#axf.footnote-max-height>
- [23] <http://www.antennahouse.com/product/ahf60/docs/ahf-ext.html#footnote-number>
- [24] <https://drafts.csswg.org/css-gcpm/#named-strings>
- [25] <https://drafts.csswg.org/css-gcpm/#running-elements>
- [26] https://www.w3.org/TR/xsl11/#fo_simple-page-master
- [27] https://www.w3.org/TR/xsl11/#fo_static-content
- [28] <https://drafts.csswg.org/css-gcpm/#using-named-strings>
- [29] <https://www.w3.org/TR/css3-page/#at-page-rule>
- [30] <https://www.w3.org/TR/css3-page/#using-named-pages>
- [31] <https://www.w3.org/TR/css3-page/>
- [32] https://www.w3.org/TR/xsl11/#fo_conditional-page-master-reference

Virtual Document Management

Ari Nordström

<ari.nordstrom@gmail.com>

Abstract

The paper describes a proposed solution to the lack of proper identification and versioning of documents passing through a series of loosely connected systems, resulting in a lack of traceability, the duplication of information and a host of other problems.

The solution is a passive tracking system that logs transaction events occurring when a document passes through each system and uses them to build a workflow and versioning history of the document in the tracking system. This versioning information can then be made available to, and used by, the participating systems to locate and query past versions, effectively creating what can be described as a “virtual document management system”.

1. Introduction

The idea to this paper originated partly from an earlier one I wrote for Balisage, and partly from the very real needs of a client. The Balisage paper was all about introducing a version management layer on top of the eXist-DB XML database's rather crude versioning module, to provide the principles of versioning management capable of separating the unnecessary versions that would happen whenever saving a document from the meaningful ones; very quickly there will be lots of versions of which most are of little interest later.

The significant versions, therefore, will be very difficult to find, severely limiting the usability of the module.

My client's needs, on the other hand, center on no real version management to begin with. The document editing and publishing workflow would perhaps be best described as “distributed”, comprising several systems between which a document would be sent on its way to being published in one or more of the multitude of systems outputting the content¹. The problem here is that there is no single source system, no central system in control. Once published, there is no way to reliably trace a publication to its source to determine if there is a later and updated version of that document.

¹The systems range from several editing environments to a complex, multi-step publishing chain that enriches and converts the information, ending with several different legacy output systems as well as a brand new one being developed.

1.1. Version Management Concepts

While this is not the place to offer a detailed discussion of version management basics, it is nevertheless important to introduce and define a few key concepts. None of these is new or original, but since the definitions and their use vary depending on whom you ask, I will offer mine here:

- Anything can be versioned. Here, I will mostly infer XML documents, but the paper applies to every kind of content.
- A new version happens when there is a significant change to the old. What “significant” means may vary, of course, but at the core we are talking about any kind of change to the information content or structure. A spelling fix is an update, as is, of course, a reorganisation of a section or an added paragraph.
- Two different translations should not be seen as separate versions but rather different renditions of the same basic document, much like GIF or JPG renditions of an image.
- Nor should two different output formats of the same document (say, PDF and HTML) be seen as different versions. There might be good reasons to keep track of them, but if their contents are the same, presentation should not matter when determining their version.
- Moving a document along document workflow stages (for example, “editing”, “reviewing”, “translated”, “approved” or “published” should not automatically result in new versions. A document could easily pass through them all without a single change; on the other hand, multiple versions might be required for a single stage such as “editing”. Therefore, *workflow handling, including lifecycle handling, should be kept separate from version handling.*
- Links, from paper-cross-references to hyperlinks, images or content inclusions, should include the exact version and rendition² of the target.
- Version *labels* are just that, labels. They help readability but are not in themselves important. At their core, they are simply numerals starting from 1.

Importance can be attached to them by introducing various business rules; usually, the business rules help clarify workflow and lifecycle stages. For example, “1.0” will frequently represent an approved and published version (implying that “0.9” is a draft), “1.1” a derivative of that version, usually without any major changes to functionality, and so on.

1.2. The Semantic Document

Following my definitions, above, I would like to very briefly discuss the identification of documents, or more generically, resources.

²Meaning translation and, when relevant, the presentation.

A filename is seldom unique or usable as a document identifier, so it is useful to create an abstraction for the document ID. Me, I'm partial to URNs, as they are straight-forward to use when implementing the above while allowing one to retain some of the readability of a filename.

Imagine a “document” as a container of information about some subject. A base version of that document, disregarding versioning, workflow, presentation or language, might be identified as follows:

```
urn:x-myurn-ns:r1:mydocs:00001
```

What I'm saying here is simply that my document is, in this URN namespace, uniquely identified as “mydocs:00001”. It's an abstract document and only identifies the actual information, the document semantics. Adding a rendition language to the identifier could then be done like this:

```
urn:x-myurn-ns:r1:mydocs:00001:en-GB
```

This says that this particular rendition of the contents is in British English.

Documents change over time, however, so to track that change, we introduce an a version label to the identifier:

```
urn:x-myurn-ns:r1:mydocs:00001:en-GB:1
```

Now, we can reliably track change to the document:

```
urn:x-myurn-ns:r1:mydocs:00001:en-GB:1  
urn:x-myurn-ns:r1:mydocs:00001:en-GB:2  
urn:x-myurn-ns:r1:mydocs:00001:en-GB:3  
...
```

And if a version 10, say, was finally approved, we could translate that version to Finnish and identify the translation like so:

```
urn:x-myurn-ns:r1:mydocs:00001:fi-FI:10
```

If accepting the concepts as outlined here, it follows that this Finnish translation is identical to the British English `urn:x-myurn-ns:r1:mydocs:00001:en-GB:10` document.

2. The Problem

So, the problem in a nutshell:

Large numbers of XML documents are created, edited and published without a single, central source, a proper identification, or proper version handling. Instead, they are moved from one system to another in a complex and multi-ended publishing chain, where no single system has control over a document in the sense that it can control an identifier namespace and uniquely identify a document passing through a step as the same one that went through an earlier step.

The various systems can be seen as a pipeline of loosely connected black boxes (see Figure 1) where the information is enriched and converted to other formats, eventually being published in Word ML on a customer PC or as content in a system intended to eventually be used as a central repository.

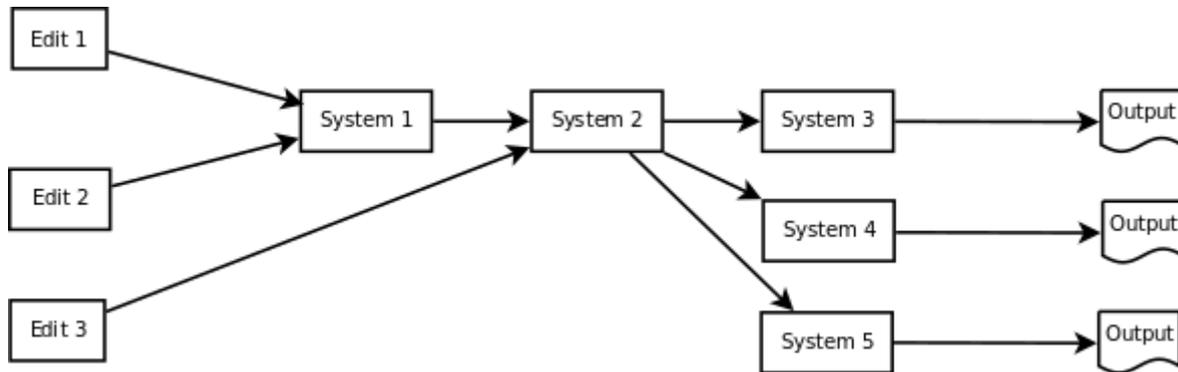


Figure 1. Loosely Connected Black Boxes

The individual steps have control over the content while inside the black box representing the current step, but usually there is no way to query or track the content before or after the step in the publishing chain³. They tend to be fetched from a shared folder and delivered to another, and so there is no way to reliably track a single document through the publishing chain.

Yet, traceability is exactly what is required by the business; it must be possible to trace a document back to the previous steps, including the source, to find out if a later version exists, and what the changes are.

As noted, many steps along the way have some degree of control over the document *within that step*. For example, many of the first steps in the chain⁴, where documents are authored⁵, are database-driven. There is not necessarily any actual versioning—a change to an existing document is frequently done directly to the document at hand and so no old version is kept—but the system knows what changes are done to what document. It's just that no other later step, no black box in the publishing chain, can reliably access or keep track of that information.

Also note that the content is led through different participant systems depending on the desired output which further serves to hide the tracks of a publication.

It is easy to get lost in the relative complexities of each system and come to the (wrong) conclusion that fixing the problem is done by adding versioning or ID handling in one of the participating systems. For example, some of the steps do

³The illustration is merely for illustration purposes; the actual situation is not depicted exactly as-is anywhere in this paper.

⁴Plural form; we are talking about several authoring systems.

⁵Helpfully labelled as “Edit”.

have ID mapping, where one set of IDs is mapped to another in an attempt to preserve the history of a document.

This, of course, is just wrong. The problem, to put it simply, happens because a) there is no global, unique and persistent identification of the resources, which means that b) there can be no versioning of the resources⁶.

3. The Solution

While I would usually propose a single-source document management system to handle the situation—something that can uniquely identify and version handle every piece of content, and introduce workflows to maintain complete control over the document throughout the publishing chain—this is, for a number of reasons, not feasible.

Therefore, I instead suggest introducing an external system, a passive observer, to track the transaction events that occur when a document passes through the publishing chain, from one system to the next. This system would log every transaction in every output and every input of every black box, identifying as much information of the transaction as possible and so adding any available data about the document itself, but also about the participating system, from the system ID to timestamps to local identifiers, any processing (such as identity transforms, enrichment, updated structural IDs, etc) happening between the input and the output, and so on.

The idea here is that by logging events happening to a document throughout the publishing chain, the tracking system is able to construct a versioning and workflow history for that document, noting where the document came from and identifying changes being done to the document while it is being published. The log can then be used to trace the document back to its origins.

Also, when an existing document in a source system is updated and published again, this too is passed on to the tracking system that, based on earlier transaction information, recognises that the new event happens to a document already being tracked, and will add to the versioning history for that document.

Ideally, the transactions should be logged in an XML document that models a workflow and versioning history for a resource, enabling easy access and manipulation, preferably in an XML database such as eXist.

3.1. Versioning XML

My Balisage paper (see [1]) proposes a versioning structure that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<resource>
```

⁶How can you version handle something you can't identify?

```
<!-- Base URN -->
<base></base>

<!-- Integer version 1 -->
<version>
  <rev>1</rev>
  <url lang="en-GB"></url>
  <metadata>
    ...
  </metadata>

  <!-- 1st decimal version -->
  <version>
    <rev>1</rev>
    <url></url>
    <metadata>
      ...
    </metadata>
  </version>

  <!-- 2d decimal version -->
  <version>
    <rev>2</rev>
    <url lang="en-GB"></url>
    <metadata>
      ...
    </metadata>
  </version>
</version>

<!-- Integer version 2 -->
<version>
  <rev>2</rev>
  <url lang="en-GB"></url>
  <metadata>
    ...
  </metadata>

  <!-- Stage 1 (decimal versions) -->
  <version>
    <rev>1</rev>
    <url></url>
    <metadata>
      ...
    </metadata>
```

```
</version>
<version>
  <rev>2</rev>
  <url></url>
  <metadata>
    ...
  </metadata>
</version>
</version>
</resource>
```

This identifies a base version of the document in `base` (compare this to the semantic document in Section 1.2) and then uses nested `version` elements to identify every new version, including URLs and metadata about the respective versions. Implied here is versioning on two levels, much like what software versions tend to look like (i.e. “1.0”, “1.1”, “1.2”, “2.0”, “2.1”, etc) where the first-level versions describe major updates according to some business rules, and the second-level versions describe minor updates within these.

The `metadata` structure contains information about a specific version of the resource. This could include time stamps, titles, users, or basically anything that is deemed to be of interest.

My proposed passive tracking system should use something similar to the Balisage versioning markup, above, but with some differences:

- The publishing chain would need to track several sets of metadata per version or workflow stage: the output of one system would track one set of metadata (about that system) while the input of the next system in line would track another (about that system). The document itself would likely be unchanged, however.
- If the output and input versions are stored somewhere, these would be pinpointed with their separate URLs (pointing at the folders used by the respective system) but appended to the same document version. note that while these versions exist for some time, during processing, they are not persistent.

The result, ideally, is a versioning and workflow history of a resource based on the events happening to it along the publishing chain. It can by no means replace a complete version *handling* system unless the “versions” that happen along the way are saved and retrievable later.

Of course, the tracking system could easily store any relevant contents of the tracked document, from a skeletal XML structure to the complete document itself, at any (transaction event) point in the publishing chain, most usefully, perhaps, when logging the first and last steps of the publishing chain. This would in effect create a versioning system of sorts, a virtual document management system where changes to the content can be tracked and queried.

3.2. Tracking Changes

So, to illustrate a simple publishing chain, let's assume that an existing document A passes through a publishing chain with three systems as follows:

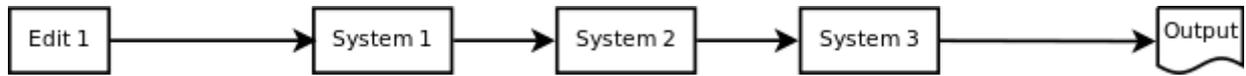


Figure 2. Publishing Chain

Each of the systems performs some kind of transformation, the details of which are unimportant for the purposes of this example.

The following happens:

1. The document is published from *Edit 1*, causing a transaction event registered by the tracking system. The tracker logs the originating system's (*Edit 1*) metadata, including the document ID for A as used by it, and possibly a URL to the source document, if available. The tracking system could also store the document itself, and add that URL to the versioning XML
All this is added to the existing versioning XML for document A.
2. *System 1* receives the file and causes another transaction event. The tracker logs System 1 metadata, adding it to A's versioning XML. This causes a new decimal version to be created.
3. *System 1* finishes its task(s) and sends the document off to the next system, *System 2*, again causing a transaction event. The tracker logs updated document A metadata, and possibly the transient URL, in the version created in the preceding step.
If storing the updated document, the tracking system will also add a URL and additional metadata to the versioning XML.
4. *System 2* receives the file, causing another transaction event. The tracking system creates a new decimal version and adds metadata about the receiving system (*System 2*), as well as (maybe) a transient URL. Again, the tracking system might also store that version and add the URL and additional metadata to the versioning XML about document A.
5. *System 2* finishes its task(s), causing a transaction event that adds metadata to the decimal version created in the preceding step, as well as (maybe) a transient URL. And, as before, the tracking system might store the document, adding another URL and more metadata.
6. The *System 2* output causes a transaction event, logging updated document A metadata, and possibly the transient URL, in the version created in the preceding step. And again, as before, the tracking system might store the document, adding another URL and more metadata.

The file is then sent to *System 3*.

7. System 3 receives the file, causing a transaction event. The tracking system again creates a new decimal version and adds metadata about the receiving system (System 3), as well as (maybe) a transient URL. Again, the tracking system might also store that version and add the URL and additional metadata to the versioning XML about document A.
8. System 3 happens to be the final instance. It processes the file, causing a transaction event that adds metadata and possibly a URL to the version created in the previous step.
9. The output is stored and logged. This would cause yet another transaction event, where the system metadata, document metadata and publication metadata would be stored, adding, of course, URLs where so required.

Note

Here, various additional steps could be taken to ensure that the publication is tracked, regardless of who (and where) the end user is so that the document's versioning XML as created above can be reliably queried later.

Note

The question of *when* a new version should be triggered, above, is partly a philosophical one, but also something that could depend on the system that causes the transaction event. For various reasons, one system might not cause a new version (in case of identity checks or similar) while another one would (for example, when transforming a document). It should be remembered that while the publishing chain is a mass of distributed systems where one does not know much about the other, the purpose of each one is reasonably well defined.

3.3. Updated Versioning XML

The Balisage versioning XML, ideally, needs an update to properly handle the publishing chain described above, most importantly to allow the coupling of multiple metadata and URL pairs within a single version so the URL and its associated metadata can be grouped; the above set of transactions relies on several sets of metadata without the content changing..

The publishing chain described above (Section 3.2) would result in versioning XML like this⁷:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="file:/home/ari/Dropbox/XMLPrague/2016/virtual-document-
```

⁷The Relax NG schema that describes the tracking system's versioning XML is not done as of this writing. Specifically, the metadata structure that is required is far from being ready.

```
management/rng/version-xml.rnc" type="application/relax-ng-compact-
syntax"?>
<map>
  <resources>
    <resource>
      <!-- Versionless information here -->
      <metadata>About the semantic document</metadata>
      <!-- Base identifier for document A -->
      <base>urn:x-versions:r1:001</base>

      <!-- Existing version -->
      <version>
        <rev>1</rev>
        <!-- V 1.0 revisions go here -->
      </version>

      <!-- New version -->
      <version>
        <rev>2</rev>
        <doc>
          <metadata>Output from EDIT 1, EDIT 1 SYSTEM</
metadata>
          <url>URL TO EDIT 1 VERSION</url>
        </doc>

        <version>
          <rev>1</rev>
          <doc>
            <metadata>SYSTEM 1 input metadata</metadata>
            <url>URL to SYSTEM 1 INPUT</url>
          </doc>
          <doc>
            <metadata>SYSTEM 1 output metadata</metadata>
            <url>URL to SYSTEM 1 OUTPUT</url>
          </doc>
        </version>
        <version>
          <rev>2</rev>
          <doc>
            <metadata>SYSTEM 2 input metadata</metadata>
            <url>URL to SYSTEM 2 INPUT</url>
          </doc>
          <doc>
            <metadata>SYSTEM 2 output metadata</metadata>
            <url>URL to SYSTEM 2 OUTPUT</url>
```

```
        </doc>
    </version>
    <version>
        <rev>3</rev>
        <doc>
            <metadata>SYSTEM 3 input metadata</metadata>
            <url>URL to SYSTEM 3 INPUT</url>
        </doc>
        <doc>
            <metadata>SYSTEM 3 output metadata</metadata>
            <url>URL to SYSTEM 3 OUTPUT</url>
        </doc>
        <doc>
            <metadata>METADATA ABOUT PUBLICATION</metadata>
            <url>URL to PUBLICATION</url>
        </doc>
    </version>
</version>
</resource>
</resources>
</map>
```

Let's walk through this. Here's the semantic document, identified using a document ID that is guaranteed to be unique within the tracking system, and the existing version:

```
<!-- Versionless information here -->
<metadata>About the semantic document</metadata>
<!-- Base identifier for document A -->
<base>urn:x-versions:r1:001</base>

<!-- Existing version -->
<version>
    <rev>1</rev>
    <!-- V 1.0 revisions go here -->
</version>
```

When a new version of the document is published, the tracking system logs a new virtual version by creating a new integer version:

```
<version>
    <rev>2</rev>
    <doc>
        <metadata>Output from EDIT 1, EDIT 1 SYSTEM</metadata>
        <url>URL TO EDIT 1 VERSION</url>
    </doc>
```

Then, for each input to a new system, and the system's corresponding output, the tracking system adds a new decimal version:

```
<version>
  <rev>1</rev>
  <doc>
    <metadata>SYSTEM 1 input metadata</metadata>
    <url>URL to SYSTEM 1 INPUT</url>
  </doc>
  <doc>
    <metadata>SYSTEM 1 output metadata</metadata>
    <url>URL to SYSTEM 1 OUTPUT</url>
  </doc>
</version>
```

Until we reach the final publication:

```
<doc>
  <metadata>SYSTEM 3 output metadata</metadata>
  <url>URL to SYSTEM 3 OUTPUT</url>
</doc>
<doc>
  <metadata>METADATA ABOUT PUBLICATION</metadata>
  <url>URL to PUBLICATION</url>
</doc>
```

Here, I've deliberately made the distinction between the last system's output and the actual publication to highlight that in the latter one, it would be easy to add information about *where* the publication is used, providing further help when tracing a document. This could be handled using a list such as this XML, or it could be a list of values in a referenced (but wholly separate) database.

4. Implementation and Use

An XML database should make it easy to build the XML-based versioning and workflow structures, and to query them later, but it can, of course, also be used to store the tracked content itself⁸, allowing queries into the different versions. The system needs to be a passive observer, in that it cannot control or change any of the content it tracks; however, the tracked systems all require some additional functionality that allows them to trigger the transaction events.

4.1. Transaction Events and Logged Metadata

The tracking system should be a service subscribed to by the participant systems. While it would probably be possible to track a resource through a publishing

⁸Making it into an almost full-fledged version management system.

chain with only some of the steps logging transactions⁹, ideally, every system in the process should subscribe to the service.

Essentially, the tracking functionality would be a service that logs any useful metadata about the originating system (system ID, timestamps, users, etc), the document (ID in the originating system, URL, etc), as well as any other useful information about the transaction, for example, automated comments or information entered by a user when initiating the transaction or event. Once a transaction was completed, the information would be passed along to the tracking system, initiating a change to an existing versioning XML, or perhaps a new versioning instance altogether, depending on how the event was triggered.

Obviously, if a system only has a single purpose (i.e. “convert XML to PDF”, the type of event it triggers should be clear. A system intended for multiple types of processing, depending on the context¹⁰, would benefit from including the purpose in the logged metadata.

4.2. Using the Versioning Information

The versioning information can be queried by any of the participating systems, for example, to present an overview of the document history or to track specific changes to a document, based on, say, a structural ID or a fixed XPath expression. It should be easy to present the versioning XML in HTML format, perhaps with links added to the older versions, or with diffing functionality added.

Of course, any stored metadata about the changes in each version can also be presented to the user.

I tend to see the information as a virtual document management system, virtual because none of the subscribing systems control it and because the documents identified by the tracking system are *not real*; they are all reverse-engineered from the available information and only available indirectly.

Since the versioning and workflow information is reverse-engineered, it also implies that the information could be wrong and we could, in fact, risk adding a transaction event to the wrong resource in the tracking system or create two versioning trees for what should have been a single document. Exactly how, and by what, an event is triggered and what is included in the log, is crucial but hampered by the fact that the participant systems have limited, and differing, capabilities.

Also consider the case where a document is sent to two *parallel publishing outputs* where the requirements of one output differ from the other's, forcing the editor updating the document in the originating system (*Edit 1* in Section 3.2) to change the contents slightly for the two outputs. In effect, this causes a “fork”, a

⁹Using timestamps, etc; an identity transform that changes document identifiers or other metadata would otherwise risk covering the tracks.

¹⁰Such as the originating system.

split of the contents that happens because of the differing requirements. Without a document management system, it is very difficult to keep track of, and update, the two variants.

In order to log the two events as a fork of a single version rather than two separate versions¹¹, some business logic and some additional markup is required to handle the fork in the versioning XML. A separate version tree should probably be created for the fork and link the original and the fork to each other, for example, by adding an ID/IDREF pair¹²:

```
<version idref="id-fork">
  <rev>2</rev>
  <doc>
    <metadata>Output from EDIT 1, EDIT 1 SYSTEM</metadata>
    <url>URL TO EDIT 1 VERSION</url>
  </doc>

  <version>
    <rev>1</rev>
    ...
  </version>
  ..
</version>
```

The @idref would point at another resource in the XML, one with a new base identifier and a separate versioning history:

```
<resource id="id-fork">
  <!-- Versionless information here -->
  <metadata>Info about the forked document A</metadata>
  <!-- Base identifier for document A fork -->
  <base>urn:x-versions:r1:002</base>

  <!-- Existing version -->
  <version>
    <rev>1</rev>
    <!-- V 1.0 revisions go here -->
  </version>
  ...
</resource>
```

Or, with a separate versioning XML instance, an external reference. Adding more business logic, it might be useful to use the base identifier of the target (i.e.

¹¹This is

¹²Or something much cooler, an extended XLink linkbase that connects the two. There, I managed to include XLink in this paper, too.

version/@ref="urn:x-versions:r1:002") and allow later processing to determine what is returned by a query to locate the fork.

5. End Notes

There are several points to be made about the proposed system. In no particular order:

- The paper really mostly describes what should be labelled as a workflow tracking system rather than a versioning one, even though it will be able to keep track of every version *published* by the editing system.
- To overcome that weakness and actually keep track of what's happening *inside* an editing system, while a document is being drafted, the system would have to “publish” drafts, too, and allow these to generate suitable transaction events.

In fact, the tracking system could be used to add version handling capabilities to a single system, simply by adding functionality to output (“publish”) a document using different publication flags (“draft”, “review”, “approved”, etc), store the outputs, and use the events to build a versioning XML document.

- Of course, for full versioning capabilities regardless of the way the tracking system is used, any significant draft versions would also have to be saved by the tracking system¹³.
- I've tried to look for solutions similar to mine out there but found surprisingly little of note, although Eliot Kimber's work, presented at Balisage 2015 (see [2]), is well worth reading and touches upon similar ideas.

Interestingly, having talked to my client, other parts of the company have touched upon similar ideas—passive trackers that log information from the subscribing systems—but these ideas have yet to come to pass.

5.1. Future Work

For the immediate future—pre-conference—I hope to do a demo of some kind, quite possibly using eXist-DB only and faking some transaction events within an eXist application. Another option would be to write a proof of concept in eXist, enriching its current versioning module, but that will take longer.

In the longer term, I hope to convince my client of the usability of the approach. If accepted, the XML format is likely to change and adapt to their particular requirements (the metadata being a case in point).

¹³My Balisage paper (see [1]) discusses useful approaches when creating a versioning strategy.

Bibliography

- [1] Nordström, Ari. "Multilevel Versioning." Presented at Balisage: The Markup Conference 2014, Washington, DC, August 5 - 8, 2014. In Proceedings of Balisage: The Markup Conference 2014. Balisage Series on Markup Technologies, vol. 13 (2014). doi:10.4242/BalisageVol13.Nordstrom01.
- [2] Kimber, Eliot. "Hyperdocument Authoring Link Management Using Git and XQuery in Service of an Abstract Hyperdocument Management Model Applied to DITA Hyperdocuments." Presented at Balisage: The Markup Conference 2015, Washington, DC, August 11 - 14, 2015. In Proceedings of Balisage: The Markup Conference 2015. Balisage Series on Markup Technologies, vol. 15 (2015). doi:10.4242/BalisageVol15.Kimber01.

Define and Conquer

Using Semantic XML for Functional Software Specifications

Dr. Patrik Stellmann
<patrik.stellmann@gdv-dl.de>

Abstract

The XML universe provides several standards and tools for technical documentation.

This paper introduces an approach to functionally specifying software with XML tools and provides an overview of the challenges that we solved. An essential part is to use a highly specialized semantic XML schema customized for the specific enterprise architecture. Thus, the general idea of "highly specialized semantics" is introduced and some hints on how to define and maintain such a schema are given.

This paper is based on over three years of experience using the combination of oXygen XML Editor and DITA. The concept and ideas should be applicable to other XML based documentation standards and editors as well.

Keywords: XML, DITA, oXygen XML editor, XSLT, Schema, Schematron, Functional specification

1. Introduction

Writing good functional software specifications is challenging and mistakes are often very expensive to fix. Still such documents are often written with editors like MS Word that provide very limited authoring assistance – especially when compared to modern integrated development environments.

Being a system architect an essential part of my job is writing functional specifications. Additionally, I'm responsible for maintaining the book of procedures that we provide for our customers. The book of procedures describes the XML interfaces of our web services. It is written in DITA XML.

The framework we are currently using is a mixture of DITA and DocBook: it uses DITA elements and the class concept. But it uses XInclude instead of DITA maps. After realizing the disadvantages of this approach (e.g. not being able to use DITA-OT for publishing) we are currently working on the conversion to real DITA and refactoring the added features as open source plugins¹ for oXygen XML editor² and DITA-OT³.

From a developers point of view our framework can be seen as a programming language for system architects. It not only helps to write a consistent technical document. It also provides assistance in designing a good architecture.

2. Highly Specialized Semantics

A functional software specification has to contain all information required for implementing, testing and operating the software. But in contrast to source code this information is provided in natural language which can hardly be interpreted by computers.

DITA and other XML standards for technical documents, such as DocBook already define semantic elements on a level of detail required for authoring and publishing. But this doesn't help when trying to automatically *interpret* the actual content.

DITA additionally provides elements for the programming domain⁴, for example parameter lists. But these elements alone are usually not sufficient for the following reasons:

- They are not sufficiently specific (e.g. it's not clear, what kind of parameters are described).
- They are not sufficiently fine-grained (within the parameter description you might want to define the exact data type).
- They provide no context (the parameters of what interface are described).

The specialized elements are derived from more generic ones. Our schema specializes elements on all levels. For example, we specialize parameter elements to add domain specific meaning. We also specialize topic elements to structure the definition of a software component. As a result for every component of our software architecture and the individual properties we have specialized DITA elements.

Figure 1 illustrates how we use our highly specialized XML to specify a login service for an enterprise service bus (ESB). Figure 2 shows the corresponding XML code.

¹DITA-SEMIA (DITA Semantic Information Architecture) is a joint development by GDV DL (www.gdv-dl.de) and parson (www.parson-europe.com) being published on github: github.com/dita-semia

²www.oxygenxml.com

³<http://www.dita-ot.org/>

⁴DITA programming elements: <https://docs.oasis-open.org/dita/v1.0/langspec/pr-d.html>

2.4 ESB-Service: Login

Provides functions for login and logoff.

Namespace	ESB-Services/Common
Type	Windows Webservice
DB Access	<ul style="list-style-type: none"> ▪ DBROOT.LOGIN_TOKEN (<i>read/write</i>) ▪ DBROOT.USER (<i>read</i>)
Required Components	<None>

Overview of the Functions

- **Login**

Validates the user and password and returns a login token on success. (Details: see *Function: Login*.)

- **Logoff**

This function causes the passed login token to become invalid. (Details: see *Function: Logoff*.)

2.4.1 Function: Login

Validates the user and password and returns a login token on success.

DB Access	<ul style="list-style-type: none"> ▪ LOGIN_TOKEN (<i>write</i>) ▪ USER (<i>read</i>)
ESB Calls	<None>

2.4.1.1 Request

XML Interface

<code><LoginRequest></code>	
Username (String, required)	Name of the user.
Password (String, required)	Password of the user.

2.4.1.2 Workflow

The function checks if `Username` is valid according to `USER.USERNAME` and `Password` matches with `USER.PASSWORD`. If this is the case a new login token is generated and stored in a new dataset in `LOGIN_TOKEN`. `LOGIN_TOKEN.VALID_TO` is computed according to `USER.LOGIN_PERIOD`.

2.4.1.3 Antwort (Response)

Figure 1. Sample Functional Specification

```

<EsbService xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="urn:gdvdl:demo-ph.xsd"
  id="HD3-5ZQ-Q5">
  <title><Key>Login</Key></title>
  <body>
    <shortdesc id="GF3-5ZQ-Q5">Provides functions for login and logoff.</shortdesc>
    <Header>
      <Namespace>ESB-Services/Common</Namespace>
      <Type/>
      <ListDbAccess/>
      <ListRequiredComponents><None/></ListRequiredComponents>
    </Header>
    <ListFunctionOverview/>
  </body>
<EsbServiceFunktion id="JG3-5ZQ-Q5">
  <title><Key>Login</Key></title>
  <body>
    <shortdesc id="I33-5ZQ-Q5">Validates the user and password and returns a login token on success.</shortdesc>
    <Header>
      <ListDbAccess>
        <DbTable Access="write" Namespace="DBROOT">LOGIN_TOKEN</DbTable>
        <DbTable Namespace="DBROOT">USER</DbTable>
      </ListDbAccess>
      <ListEsbCalls><None/></ListEsbCalls>
    </Header>
  </body>
<Request id="K33-5ZQ-Q5">
  <title/>
  <body>
    <Overview/>
    <Structure id="EJ3-5ZQ-Q5">
      <Definition id="JJ3-5ZQ-Q5">
        <Root id="OJ3-5ZQ-Q5">
          <Name/>
          <Field id="CCJ-4SY-Q5">
            <Definition>
              <Key>Username</Key>
              <Type><Datatype Value="String"/><Presence Value="required"/></Type>
            </Definition>
            <Description><p id="SB4-VSY-Q5">Name of the user.</p></Description>
          </Field>
          <Field id="TXK-5SY-Q5">
            <Definition>
              <Key>Password</Key>
              <Type><Datatype Value="String"/><Presence Value="required"/></Type>
            </Definition>
            <Description><p id="TCY-VSY-Q5">Password of the user.</p></Description>
          </Field>
        </Root>
      </Definition>
    </Structure>
  </body>
</Request>
<Workflow id="NK3-5ZQ-Q5">
  <title/>
  <body>
    <p id="V5W-JTY-Q5">The function checks if
    <KeyRef KeyType="XML-Field" Namespace="ESB-Services/Common/Login/Login/Request">Username</KeyRef>
    is valid according to
    <KeyRef KeyType="DB-Column" Namespace="DBROOT/USER">USER.USERNAME</KeyRef> and
    <KeyRef KeyType="XML-Field" Namespace="ESB-Services/Common/Login/Login/Request">Password</KeyRef>
    matches with
    <KeyRef KeyType="DB-Column" Namespace="DBROOT/USER">USER.PASSWORD</KeyRef>. If
    this is the case a new login token is generated and stored in a new dataset in
    <KeyRef KeyType="DB-Table" Namespace="DBROOT">LOGIN_TOKEN</KeyRef>.
    <KeyRef KeyType="DB-Column" Namespace="DBROOT/LOGIN_TOKEN">LOGIN_TOKEN.VALID_TO</KeyRef>
    is computed according to
    <KeyRef KeyType="DB-Column" Namespace="DBROOT/USER">USER.LOGIN_PERIOD</KeyRef>.</p>
  </body>
</Workflow>
<Response id="GL3-5ZQ-Q5">

```

Figure 2. Sample Functional Specification Markup

3. Solved Challenges

Writing technical documents is a complex task in general. Often style guides are supposed to help the authors to write consistent content – especially when several authors are working on the same document.

Writing functional software specifications is no different. But as an additional difficulty the focus of the author – in this case being a system architect – should be on the actual architecture and design that needs to be created – not on writing a document.

Modern integrated development environments like Eclipse or MS Visual Studio provide great authoring assistance for writing software. This includes the following features:

- Content completion
- Online help
- Syntax highlighting
- Real-time validation
- Quick fixes

Our highly specialized semantic XML is the starting point of an integrated development environment for system architects. The following sections illustrate how XML improved our workflow of functionally specifying software and partially even the following stages of our software development cycle.

3.1. Authoring Assistance

In our company there are several guidelines a system architect has to follow when functionally specifying software. The guidelines describe documents structure, layout and even the actual software architecture. Providing these guidelines in a separate document had only limited success as it is difficult to keep them in mind all the time.

Our customized framework assists the author and enforces some of the rules of the style guide. The following use case illustrates how the framework guides the author. Assume that the author wants to add the parameter `Gender` to a software component.

As the author starts to write, immediate authoring assistance is provided by the XML schema itself. The highly specialized schema provides dedicated elements for all relevant architectural components – e.g. for the accepted types of a field. The author does not have to write huge blocks of unstructured text but can pick the required elements. To insert new elements in oXygen's author mode the author just has to press enter and a list of all allowed elements is displayed. In our use cases it would look like shown in Figure 3.

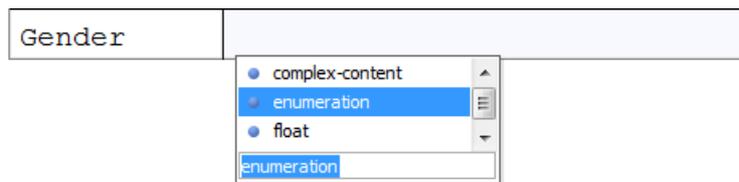


Figure 3. Authoring Guidance by Element Choices

After choosing the required type – in this case an enumeration – Oxygen will automatically add all mandatory child elements. In case of an enumeration this is a list of key-name-pairs with a single entry. The architect immediately sees the fields he has to fill as shown in Figure 4.

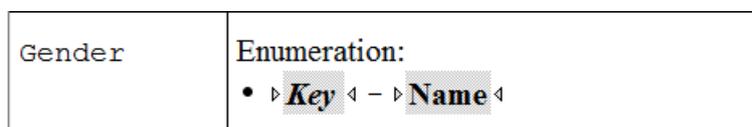


Figure 4. Authoring Assistance by Mandatory Elements and Static Content

The framework adds static content. In our use case the prefix "Enumeration:" and the dash between the key and name elements are added. The architect does not have to think about how he should describe the enumeration and if he should use a colon or a dash between the key and the name.

And finally, it is possible to provide some pure architectural hints for the author that are relevant for this use case. For instance, an automatically generated text can provide information about how to model an enumeration, as shown in Figure 5.

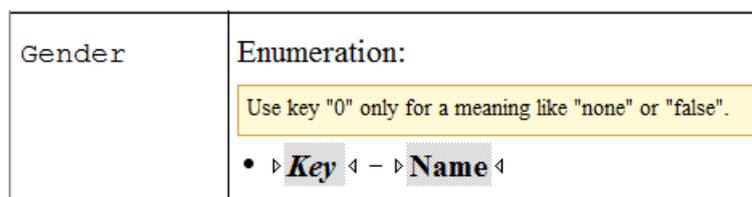


Figure 5. Authoring Assistance by Hints

In oXygen these hints can easily be activated and deactivated by changing the current CSS style so experienced authors can hide it.

3.2. Identifiers and Enumeration Values

A functional software specification usually contains several references to other architectural entities. For example, the specification of a web service references the data base tables it accesses. Using the exact value is obviously very important and still it is very hard to ensure consistency. This holds especially true as functional specifications tend to be modified several times.

Our framework with its highly specialized semantic XML automatically recognizes all identifiers within the specification. For example, the definition of the parameter "Gender" of type enumeration as shown in Figure 6 defines the key "Gender" of type "Parameter" as well as the two enumeration keys "1" and "2". The corresponding XML code is shown in Figure 7.

 Gender	Enumeration: <ul style="list-style-type: none"> •  1 - Male •  2 - Female
--	---

Figure 6. Definition of a Parameter as Enumeration

```

<Parameter id="LXH-FJ3-25">
  <Key>Gender</Key>
  <Datatyp>
    <Enumeration>
      <Value id="ZHN-FJ3-25">
        <Key>1</Key>
        <Name>Male</Name>
      </Value>
      <Value id="XWW-3J3-25">
        <Key>2</Key>
        <Name>Female</Name>
      </Value>
    </Enumeration>
  </Datatyp>
</Parameter>

```

Figure 7. Markup of Parameter Definition

To refer to an identifier the architect just inserts a specialized `KeyRef` element. Depending on the key type the layout can be adapted automatically, for example to use italics or mono-space font or to display some prefix and suffix characters.

When inserting such a specialized `KeyRef` element the author can just write the key (e.g. "Gender") and will get a warning, if the value is undefined or ambiguous. The author can then open a dedicated dialog that lists all defined keys and can select the one he needs to reference. This also helps to identify the key if the actual value is not known in advance.

When referencing enumeration values the authors are often facing the challenge that on the technical level the key is important. But the name is essential for understanding its meaning in the first place. Thus, for enumeration values we used to add the name in parentheses next to the key. But this increases the risk of inconsistency. Since the enumerations are specified as pairs of key and name in another chapter of the specification it is possible to add the name automatically.

Our framework checks and ensures consistency and guarantees that the author always refers to the correct key.

In our example a specification of a condition would look as shown in Figure 8.

If the field  `TypeOfPerson`  of structure  `Customer`  has the value  `2`  `(Organization)` the field  `Gender`  must be empty.

Figure 8. Referencing Identifiers an Enumeration Values

In this example the name of the enumeration value and the brackets around the Customer structure were added by the framework. The gray colored text is only visible for the author and provides information about the KeyRef element. The three question marks ("???) indicate an unknown reference while the three number signs ("###") indicate an ambiguous reference. The red or green link symbols visualize if the KeyRef element is valid. Additionally there are schematron rules that check the validity.

3.3. Redundancy

As a software developer you are used to avoid redundancies as they usually lead to inconsistencies and increased maintenance costs. However, as a reader you often prefer duplicated content over having to follow a cross reference – especially when you don't want to read the complete functional specification but only the part that appears to be relevant for you.

With highly specialized semantic XML it is possible to automatically interpret the content. This allows the definition of schematron rules that ensure consistency. And in several cases this is already a sufficient solution. But it only avoids the risk of inconsistencies but doesn't eliminate the additional costs for maintenance.

DITA already provides the conref ("content reference") concept. The DITA conref mechanism allows to reference DITA content and instead of displaying a link the content itself is displayed. But this works only for the same type of content. For example, when using the same bullet list twice. It is not possible to reuse the information itself, for example when a bullet list and a table contain similar information. More complex scenarios, like reusing textual information for figures, are obviously also not possible with current DITA mechanisms.

To solve this problem our framework provides a newly developed mechanism. The XSLT-Conref adds an XSL transformation to the existing conref concept. So before the referenced content is displayed it is transformed. XSLT-Conref accepts any content as input that is sufficiently structured to be automatically interpreted and can produce any output that is supported by the editor and publishing tool. The most important output formats are DITA XML and SVG for graphical content.

One sample use-case is a graphical sequence diagram that illustrates the interaction of a software component with others, see Figure 9. The sequence diagram is generated using the textual description of the software component and collects references to functions of the component.

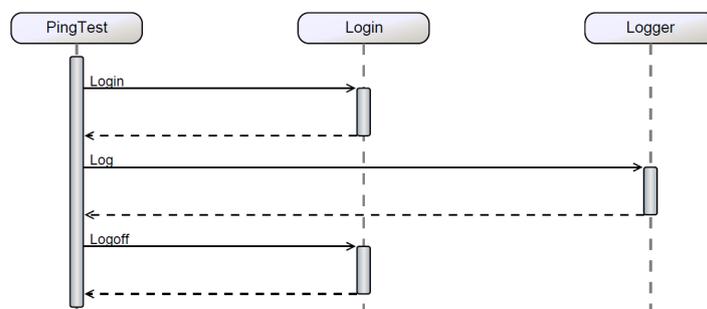


Figure 9. Sequence Diagram

Also note that such overviews are not only generated when the document is being published as PDF or HTML. The XSLT-conref is resolved in the XML editor and already available for the author when writing the document. This helps to keep an overview of the architecture when designing it.

3.4. Importing External Data

When designing a software architecture you usually do not have only textual content. For instance, in our case it is very common that a change request requires modifications in the data model. There are dedicated tools for working with data bases and we use Toad Data Modeler which produce structured output. While it is handy to use dedicated tools for different jobs it is desirable to have only a single document for the complete specification. Importing the data model information into our DITA document by XSLT-Conref is our solution for this challenge..

Since the data model itself is stored in XML, it is quite simple to convert it into generic DITA content with a section for each entity and a table describing the individual columns with their type information and description. Our framework with the XSLT-Conref allows easy import of the data model without the need for every architect to write his own XSLT scripts.

Since the XSLT-Conref can also be combined with the key mechanism described in Section 3.2 you can even automatically register all tables and columns as individual keys. That allows to reference them in a comfortable way.

3.5. Filtering

Filtering content based on some profiling attributes (e.g. audience) is a DITA standard feature. By using highly specialized semantic XML, these attributes can often be set already by the schema. E.g. the list of database tables might not be

relevant for the customer. By using a specialized element `ListDbAccess` with the attribute default `audience` set to "internal" the author doesn't have to tag it manually.

3.6. Work Item Tracking

After the functional specification is finished the implementation is often done with the support of a work item tracker. For that, the content of the document needs to be split into smaller snippets and imported as tasks into the tracking software. Additionally, it is important that the work items and the document are linked in both directions. When a work item is assigned to a developer or tester he needs to be able to identify the corresponding section in the specification. And the other way around, when reading the specification you should be able to easily check the state of the related work item.

We have solved this challenge by adding specific marker elements to the specification, usually directly below the title of a section. An XSLT transformation then generates a list of work items. The generated list can be imported directly into the tracking software.

The work item and the related section are already logically linked by having the same title. Additionally, the work item can contain a link directly to the section in the published PDF or HTML. Since each work item has a unique ID within the tracking software it can be referenced in the document. The link in the specification points to the web interface of the tracking software.

The schema defines a dedicated element for each type of work item. Thus, each marker element can contain specific additional fields required by the tracking software or for consistency checks. For instance, each work item needs to be linked to a package and when changing the data model the related data base objects need to be identified.

A sample for the work item type "database-modification" could look like Figure 10. The marker element has a reddish color due to its tagging as "internal information" (as described in Section 3.5). Content that can't be edited by the author is highlighted with a gray background color.

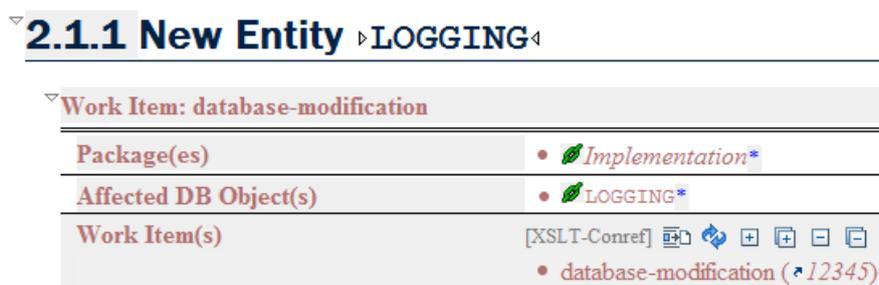


Figure 10. Work Item Marker

Note that within the marker element the author only has to fill in the name of the package and the list of related data base objects. All other content is generated by the framework: The title and the labels are static and the reference to the work item is generated with XSLT-Conref as already described in Section 3.3. The id of the tracking software is read from a separate file that has been generated during the import process. Content that can't be edited by the author is highlighted with a gray background color.

The generated work item for this sample would have the title "CR0123 – 2.1.1 New Entity LOGGING" where "CR0123" is the ID of the change request.

There are also scenarios where multiple work items are required to track the implementation of one section in the documentation. Of course our framework provides marker elements for that as well. The system architect simply adds a marker of the specific type and the correct list of work items is generated.

An example of a section with multiple work items is the implementation of a stored procedure that is installed with two separate scripts. The document looks as shown in Figure 11.

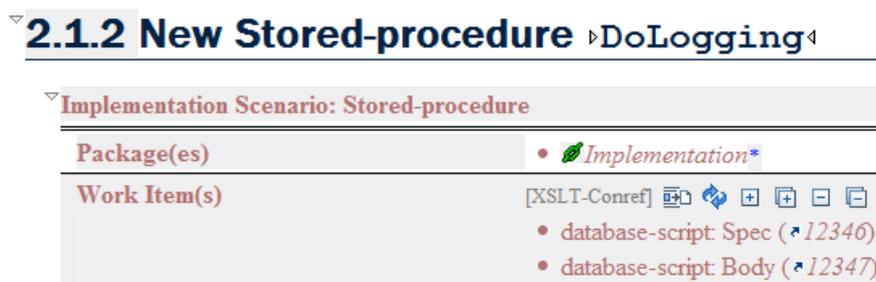


Figure 11. Marker for a Realization Scenario

The resulting work items have the following titles:

- CR0123 – 2.1.2 New Stored-procedure DoLogging: Spec
- CR0123 – 2.1.2 New Stored-procedure DoLogging: Body

3.7. Architectural Data

A functional software specification should contain all the information that is required to implement, test and operate the software. Due to using fine-granular semantic XML a lot of this information is available in a form that can be extracted by an XSLT script.

The extracted architectural data is used during the whole software development cycle. Extracted architectural data is used for the following tasks:

- Generate the definition of data types (e.g. enumerations) in source code.
- Generate drafts for new software components.

This allows the developer to immediately start with the implementation of the functional logic instead of creating new files and required base classes.

- Generate the complete XML schema for a web service interface.
- Import the data (e.g. enumerations) into the data base to be used as targets in foreign keys to ensure data consistency.
- Generate drafts for XML messages for test cases.
This allows the tester to delete the not required elements and he doesn't have to start from scratch.

For all these tasks the consistency between functional specification and the extracted data is ensured by an automated conversion. Of course it occurs that during the implementation or another stage of the development cycle some architectural data turns out to be incompatible. This can be either because the existing environment is different from the expectation, the developer has misinterpreted the document or the functional specification and, thus, the extracted architectural data is simply wrong. But in all these cases there is a feedback mechanism that allows the system architect to solve the problem early on.

4. Defining a Framework with fine-granular Semantics

The benefits described in the previous section don't come without a cost. Defining a framework that support fine-granular semantics requires several components:

- **Schema**
Depending on the tools, documentation standard and personal experience this could be defined with DTD, XSD or Relax-NG. From our experience Relax-NG appears to be the most efficient one for defining highly specialized semantics.
- **Schematron Rules**
To ensure consistency between separate but related content you might need schematron rules. These can be integrated into the schema definition or in a separate file.
- **CSS**
Using Oxygen as authoring environment you need to define CSS rules to make your semantic XML look the desired way when editing it. (Other editors might require some other way of customization.)
- **Output Transformation**
Using DITA as underlying documentation standard the publishing (e.g. PDF or HTML) is usually done using the DITA-OT. In this case you have to adapt XSLT scripts to handle your specific semantic elements.
- **XSLT Scripts**
Depending on the framework additional XSLT script might be required for data extraction or to resolve XSLT-Conref elements.

Note that the definition of a highly specialized schema is not meant to compensate flaws of a standard. In fact, I believe that this kind of high specialization is something that a general standard simply can't ever provide. This is due to the customization that has to match the very specific organizational and architectural structure. According to this requirement the specific documentation framework always has to be created or adapted according to individual needs. And since the software architecture and organizational structure is usually refined and developed further over time, maintenance of the documentation framework is a continuous task as well.

One side-effect is not to be underestimated: Most likely you will get a much deeper understanding of your actual base architecture when defining a documentation framework for it. In fact, it repeatedly occurred that I had difficulties to model some aspect of our architecture in our documentation framework. But this was always due to the lack of a clear design of our architecture. Describing the architecture as an XML schema helped to discover deficits in the architecture itself.

In the following sections I will give some hints for creating and maintaining such a documentation framework.

4.1. Attribute Defaults

When defining a highly specialized semantic XML schema you will probably have several cases when some content is static and should not be editable by the author. This content can easily be added by CSS rules (when authoring) or XSL templates (when publishing). However, this way you always have to modify several code files of your framework when creating new elements.

DITA already provides an attribute `spectitle` that allows the specification of a title for various elements within the schema. Our framework extends this concept by additional attributes to create prefix or suffix content for several block and inline elements or to define the complete header row of a table. It also allows to specify context sensitive content by using XPath expressions or provide some default content that is only being displayed when the author didn't enter some explicit content.

The basic CSS rules and XSLT templates need to be implemented only once to evaluate these generic attributes. As a result, in most cases the schema designer just has to add an attribute with a default value when adding new elements into the schema without the need to modify and test any additional code.

Default values are also used for the XSLT-Conref and implicit key definitions. The `ListFunctionOverview` element within the body of the service is shown in Figure 12 with expanded attribute defaults: the class attribute and the URI for the XSLT script to be executed. The `Key` element within the title of the login service is shown in Figure 13: it contains attributes to specify the type of the key and XPath expressions for the root as well as the namespace and the description.

```
<ListFunctionOverview
  class      = "+ topic/section GDVDL-Web-Service-d/ListFunctionOverview "
  xcr:xsl    = "urn:gdvdl:web-service:ListFunctionOvervie.xsl"/>
```

Figure 12. Expanded ListFunctionOverview Element

```
<Key
  class      = "+ topic/topic reference/reference GDVDL-Component-d/Key "
  ikd:type   = "Component"
  ikd:root   = "parent::title/parent::*"
  ikd:namespace = "body/Header/Namespcae"
  ikd:desc   = "body/shortdesc">Login</Key>
```

Figure 13. Expanded Key Element

4.2. Mandatory Choices

It is a common scenario that an element should allow exactly one of a list of options. For instance, a "conditions" element might be allowed to contain either a "list" element or the element "none". This can be easily modeled in any schema language. But the problem is, that after the author has inserted the conditions element the XML code is no more schema valid until he inserts the next element – either `list` or `none`. But there are several mechanisms that don't work with a none schema valid file including schematron validation, XSLT-Conref and key referencing.

As our framework requires schema valid documents, we choices in a different way. In our framework the child elements of choices are always optional according to the schema. A schematron rule to checks if the element is empty. This way you will still ensure the same resulting structure but won't force the author into a situation with an invalid document and none working authoring assistance.

4.3. Schema Changes

A highly specialized semantic schema is more likely to be changed than a conventional schema as it is closely intertwined with the software architecture. Changes in the architecture might require changes in the highly specialized schema. New optional elements are usually no problem but it also occurs that new mandatory elements need to be added or old elements have to be removed. When adapting the schema also existing documentation has to be adapted to be valid. This can be a problem when you don't have access to all files – or maybe don't even know about all files that use your schema.

So far we have identified the following options to solve this problem:

- Adapt the schema to the new requirements.
- Make all schema changes backward compatible.

- Create a new schema version for future documents.

When adapting the schema an XSL transformation can convert the old documentation to match the new schema. So whenever an author has a schema invalid file after updating the framework he can trigger the automated transformation of his files with just a mouse click. But from my experience it is very difficult to write an XSLT script that reliably performs the required modifications.

Making the schema backward compatible means, that all new elements have to be optional and all obsolete elements still need to be valid. To make sure that new content is written according to the new requirements and old content is adapted at some time there are some additional steps to be taken:

- Add schematron rules that report missing new elements and the existence of obsolete elements preferably together with a quick fix.
- Hide obsolete elements from the list of proposals when inserting a new element in oXygen.
- Automatically add new elements that are meant to be mandatory when the parent element is being inserted in oXygen.

When creating new schema versions a schematron rule could check if a file uses the latest version of its schema. But since we are currently using XInclude, all files of a document need to use the identical single schema. Thus, every modification would require a new schema version. This will change when using DITA maps. With DITA maps it is possible to use a different schema for each file and, thus, create schemas that define only a single kind of topic – for instance an ESB service. So for the future I expect this approach to be the best choice.

Currently, we are using a combination of the first two approaches: When the changes of the schema affect a major or critical part of the existing files, the changes are made backward compatible. Otherwise the schema is modified together with the documents.

4.4. Special Cases

From my experience it is either inefficient or even impossible to model all possible scenarios in the schema. But when a specific scenario is not modeled in the schema the authors might try to avoid the semantic rules. For instance, if the title element of an itemized list is the only element accepting free text in that context then it might happen that a full explanation is inserted there. Simply because the author could not find another element to store this information.

To avoid suffocating your authors with your highly specialized schema you should ensure to also allow sufficiently generic elements. For instance, our DITA derived framework allows at the end of a list of specialized topics any number of generic topics. And in some places (e.g. within a list of function calls where each

entry is meant to be a key reference) I have even created explicit "special-case" elements that can be inserted.

5. Conclusion

A framework with highly specialized semantic XML schema and the corresponding standards and tools significantly improve the way of functionally specifying software. Several concepts from the field of technical writing can be applied here as well.

From a developers point of view such a framework can be seen as a programming language for system architects. The syntax allows comfortable authoring – comparable to that of programming languages in an integrated development – and to a great extent automatic processing. But additional to other ordinary programming languages you can generate publications to be read by humans that fulfill the requirements of professional technical documents.

We have already successfully applied most of these concepts into our software development cycle and even new system architects quickly adapted to our style of functionally specifying software.

6. Future Work

Currently the focus is on converting the current framework to DITA maps and DITA-OT. In parallel the schema is continuously being refined and adapted according to the further development of our base software architecture.

Additional enhancements we are planning are:

- *Documentation Maintenance*

After a change request has been implemented the functional specification needs to be integrated into the overall documentation. By writing the specifications in the same way as our documentation this is basically some copy&paste action. This merge task is expected to be automatable.

- *Linking Test Cases*

To ensure consistency between automated test cases and the specifications the idea is to link the test cases with the corresponding topic, section or list item by using the already existing `id` attribute.

- *Generate Framework from Style Guide*

Another idea is to define the framework within a single specialized DITA document using all the concepts described here. From this document all the files defining the framework would be generated: the schema, Schematron rules, CSS and XSLT scripts for publication.

Subjugating Data Flow Programming

R. Alexander Miłowski

MarkLogic, Inc.

<alex.milowski@marklogic.com>

Norman Walsh

MarkLogic, Inc.

<norman.walsh@marklogic.com>

Abstract

XProc 1.0 is data flow language and W3C recommendation that provides the ability to describe steps for processing XML documents for some end purpose. While there has been some adoption, it has not been as successful as we would have liked. In this paper, we examine the issues surrounding the usability of XProc 1.0 pipelines, rethink our end goals, and describe a proposal for a new direction.

1. What is Data Flow Programming?

When processing data, developers often conceptualize a work flow through which the data and their associated files are processed. A work flow can be considered an abstract sequence of steps that are chained together to perform some useful process. Each step, in turn, consumes and produces some set of artifacts which are often files stored within some repository. Chaining these steps together produces a work flow that is data-driven.

These work flows can be broadly categorized as control-flow oriented or data-flow oriented. When control-flow oriented, each step in the work flow is orchestrated by some control language or dependency semantics. Meanwhile, when data-flow oriented, each step in the work flow is invoked when its inputs “arrive” at the step. While either are sufficient for implementing work flows in general, there is a preference for data-flow oriented technologies within certain domains (e.g., scientific data processing) [1].

Data-flow programming languages have a long heritage and are often associated with graph visualization tools for visualizing the connections of inputs and outputs. Tools like Taverna [2] and Kepler [3] have blended the distinction between the visualization, the tool, and the underlying data-flow engine for scientific computing. These systems consist of a sequence of step-wise operations where outputs are matched to inputs to perform operations on data sources that are often Web services.

Each step executes some kind of script (e.g., Python), component language (e.g., XSLT [4]), or operation (e.g., HTTP GET) to interact with data and services. In many cases, the artifacts produced by the step are not realized except within the engine to facilitate the connection between steps. That is, output artifacts are not saved unless explicitly requested to do so.

XProc 1.0 [5] is an example of a data-flow programming language. XProc provides facilities for describing the processing of XML data as a sequence of steps that operate on inputs and produce outputs. The language facilitates connecting the steps to describe the overall process as a flow of data between steps.

An XProc implementation is responsible for turning these descriptions and their necessary referenced ancillary files (e.g., XSLT transforms, XQuery [6], etc.) into an executable pipeline. Data (i.e., XML documents) can then be attached to the declared inputs of this data flow to cause steps to execute in the particular ways specified and subsequently cause outputs to be produced.

Further work on XProc has been in the direction of generalizing this data-flow specification language to allow XProc to process any kind of data and not just XML documents. These changes allow XProc to be considered in various application spaces where tools like Taverna and Kepler have been utilized in the past as well as unexplored applications. Yet, these tools have been oriented towards programmers who develop steps in a programming language such as Python.

This brings into question whether XProc's use of an XML specification language has the right focus for these new consumers. Moreover, for some existing users, an XML syntax may be inconvenient or obscure the connections within a particular pipeline. That is, having an XML syntax for “programming” pipelines may not be the right design center.

2. Rethinking XProc

XProc 1.0 was specifically designed to describe the relationships between inputs, outputs, and steps. Specifically, XML syntax allows defaulting connections between outputs of preceding siblings and the inputs of immediately following siblings. The intent was to make simple pipelines easier to specify.

Unfortunately, the language is used to describe a data flow graph and the connection (edges) within this graph must be made to fit within the constraints of a tree due to the use of an XML syntax. As such, simple meets and joins result in a more verbose syntax.

Complicating things further, an author's use of a pipeline may evolve over time and modifying a pipeline can become increasingly difficult. Just adding another step between processes may require more than a simple insertion. Unnamed steps may be required to be named and referenced just to make the step connections.

Even further, small manipulations of outputs to adjust for inputs require insertion of a step. There is not always an easy mechanism for referencing an output and using XPath (or XQuery) to manipulate a result as preparation for input to another step. While this is truly another step, the need for a verbose construct for what may be a simple manipulation leaves the user wanting for simplicity.

Finally, these and other issues of verbosity make the language difficult to use for the uninitiated and cumbersome for the experienced user. There exists a gap between simple and complex pipelines in XProc that is difficult to bridge. Even though some of that gap is conceptual, the syntax stands in the way of guiding a developer through the use of the language.

The end result of successfully using XProc is often rewarding. The overall process and what it accomplishes is often impressive, useful, and provides improved methods for packaging and deployment. Yet, the path towards that solution was fraught with errors and treachery; bringing the use of XProc into question.

This experience yields several questions:

1. Do users need data flow programming concepts?
2. Do users need to be explicitly aware that they are using pipelines and data flow programming?
3. Is there a better syntax for XProc?

In our opinion, the answer to (1) above is a resounding “yes”. Data flow programming is useful and practical concept for manipulating documents as artifacts produced by a chain of processes. Many specifications are designed to be layered (e.g., XInclude before XSLT) and so there is an implicit assumption of some basic level of data flow within applications.

Yet, in considering (2), it is unclear whether data flows and the connections between steps need to be as explicit as described in XProc 1.0. Our experience with users indicate that, in part, the gap in understanding simple versus more complex pipelines has a lot to do with a lack of understanding that data flows between adjacent sibling steps via the implicit connection of primary input and output ports.

Finally, for question (3) and considering the goal of attempting to simplify XProc for users, a syntax more akin to a functional language that allows direct use of XPath would allow simple manipulations (e.g., projections) to be directly specified without inserting steps. Modest modifications to a pipeline could be entertained by simply adjusting or adding a path or predicate expression. The result might be a more compact syntax that would facilitate a better understanding of the flow of data.

3. XProc as an XQuery Function Library

One consideration is that XProc's utility is possibly really in the steps it provides. If XProc's steps were available as a set of functions, they could be used directly in a variety of programming languages. This would allow the orchestration to take advantage of the native language's capabilities for manipulating XML and other data.

As manipulating XML data is a major focus of the language, use of XPath within some language is a requirement. While one could conceive of utilizing XPath within various programming languages, one obvious choice to consider is XQuery. By choosing XQuery, orchestrating processing is then a task of describing a sequence queries to execute in a particular XQuery programming context.

Many of the steps provided by XProc have a large set of options in addition to the inputs and outputs they process. As a function library, the signature of these step functions would likely be complicated with many parameters that must be specified. This would make the invocation of a step function difficult.

For example, XSLT has two inputs ("source" and "stylesheet"), parameters that would be specified as a map, and four options ("initial-mode", "template-name", "output-base-uri", "version"). This would likely end up with a seven-parameter function invocation where the order of the parameters is important and fraught with user error.

Example 1. XSLT Step Function Invocation

```
p:xslt(doc("doc.xml"), doc("stylesheet.xsl"),
       map { "toc" : true() }, (), (), (), "2.0")
```

Ideally, we'd like to default or omit option values but XQuery and XPath do not allow this as of version 3.1. We'd like to be able to do as in other languages (e.g., Python) and have named parameters with defaults. Then a user can just name a parameter they are using and omit the rest.

Example 2. XSLT Step Function Invocation with Named Parameters

```
p:xslt(doc("doc.xml"), doc("stylesheet.xsl"),
       $parameters := map { "toc" : true() },
       $version := "2.0")
```

While the invocation is not necessarily more compact, the meaning of the invocation is much more clear. The lack of positional parameters helps prevent the user from making simple errors, defaults are implied by omission rather than required to be remembered, and options can be specified in any order as they are today in XProc 1.0.

The invocation of a step as function implies that it returns the output as some sequence type. As a step can return multiple outputs and can also return a se-

quence of documents on a particular output port, within the constraints of XPath 3.0, we can conceive of two alternative representations: a map or a function.

A map has the advantage that each output port name would be a key of the map and each associated value the output. As maps are immutable, the map would need to be constructed and returned. As such, the result of the step would need to be computed and placed into the map. Constructing the map “by hand” is somewhat tedious and seems likely to interfere with parallelism.

A function has the advantage of allowing some aspects of parallelism. When the function is invoked, the output port name is passed and the result of the computation by the step is returned. That deferred request for the sequence of results enables some parallelism as requested by the user.

With the new => operator in XQuery 3.1, the output of a step invocation can be implied as the first argument of the right-hand side. This allows explicit chaining as shown in Example 3 where the primary result of `p:xinclude` is the first parameter to `p:validate-with-xml-schema`. Again, we take explicit advantage on the features of XQuery to improve usability.

Example 3. Example Pipeline

```
declare variable $source as external;

p:xinclude($source)
=> p:validate-with-xml-schema(doc("schema.xsd"))
=> p:xslt(doc("style.xsl"))
```

Moreover, users can now declare their own steps by implementing them in XQuery as long as they return the right kind of construct. Annotations may be able to be used to differentiate between “regular functions” and “step functions”. Yet, the distinct declaration of a step is lost in the syntax of the XQuery.

An unfortunate consequence of this approach is that we have lost the explicit description of the data flow graph. While a clever implementation may be able to tease apart those steps that are able to be run in parallel, the most likely outcome is execution will wait for the step to finish. As a result, pipelines will necessarily take a more tree-like graph structure that fan out from root of execution.

In such a resulting language, the data flow aspects of XProc is subjugated to the control of the XQuery processor. The ability for such a processor to defer executing and synchronized flows between steps affords the ability for the data flow to have its natural parallelism. As such, to have parallelism, we need to consider the problem the other way around and have the data flow in control of the overall pipeline.

4. Parallelism: What is it good for?

One of the original promises of XProc was that the steps could be run in parallel:

An XML Pipeline should not inhibit a sophisticated implementation from performing parallel operations, lazy or greedy processing, and other optimizations [7].

Yet, time has shown that users seem to care little about this requirement. Development of parallel XProc implementations has stalled, most current implementations have no parallel execution, and no users seems to have noticed. Simply, the uses of XProc have not in so far dictated this as a required enhancement to the existing implementations being used.

By making the connections between steps (i.e., blocks of execution) more explicit, the user is declaring to the processor the dependencies between major portions of processing. The value is not necessarily embedded in just enabling parallelism. The value is in separating concerns and allowing for alternative routes for an artifact (an output) to be processed.

By discarding any attempt at supporting parallelism of steps we are possibly discarding a very important distinction: the outputs of steps are results which may be reprocessed, inspected, logged, etc. Moreover, the connections between steps are insertion points for easy modification of the behavior of the pipeline.

Regardless, users always care about performance. Whether a pipeline runs within or exceeds an expected time period is very important in many applications. So, while surfacing control over parallelism may not be important to the pipeline author, the underlying effects produced by enabling parallelism most certainly is important. Hence, we want to remove the need to for the author to think about parallelism whilst creating constructs than enable its use by the implementation.

We want to retain the data flow aspects of XProc and so we recast our goals as follows:

1. From the perspective of the data flow language, a step is a black box that takes inputs and produces outputs.
2. A step has a distinct signature of inputs, outputs, and options (parameters) that can mapped to various implementations and/or functions within a domain language (e.g., XQuery).
3. The connections between steps produce artifacts that should be able to be easily inspected and re-purposed.
4. The context of a step invocation should facilitate modifications and the outputs of steps are places where future choices can be made for modification of the pipeline.
5. Steps may be able to be executed in parallel and still yield equivalent results. The data flow language should not exclude that possibility.

5. Subjugating Steps in Data Flows

The overall goal of describing a data flow is describing the graph of inputs and outputs between steps. The nodes in the graph are steps with a certain number of inward and outward arrows. The graph has explicit nodes that represent inputs (generators) and outputs (sinks). Over the edges of the graph flows data that drives the pipeline processes.

An example of this conceptual model is shown in Figure 1 where an input document is validated by one of two schemas depending on a version attribute. Afterwards, the result is transformed and the result of the transformation is the result of the pipeline. The encoding of this in XProc V1.0 is shown in Example 4

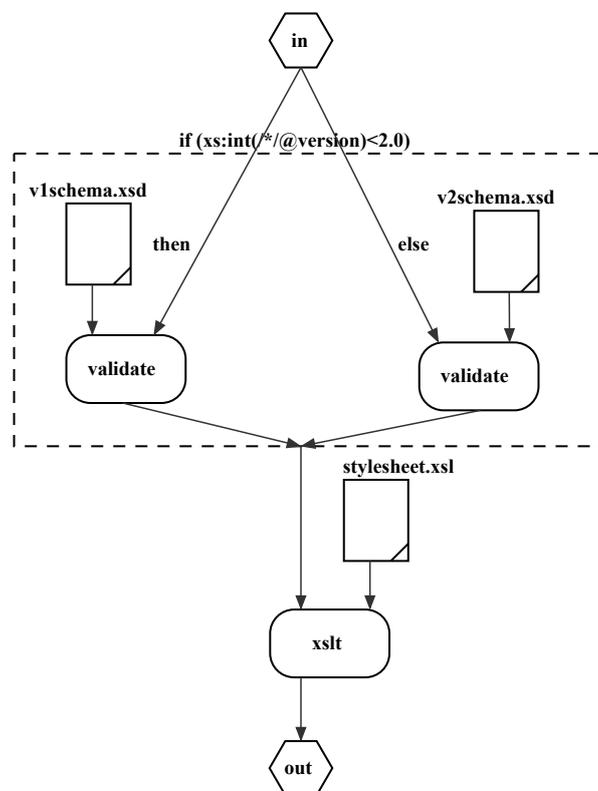


Figure 1. XProc Example 3 - Conceptual

Example 4. XProc, Example 3 in XProc V1.0

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc" version="1.0">
  <p:choose>
    <p:when test="xs:decimal(*/@version)<2.0">
      <p:validate-with-xml-schema>
        <p:input port="schema">
          <p:document href="v1schema.xsd"/>
        </p:input>
      </p:validate-with-xml-schema>
    </p:when>
  </p:choose>
  <p:xslt>
    <p:input port="stylesheet">
      <p:document href="stylesheet.xsl"/>
    </p:input>
  </p:xslt>
</p:pipeline>
```

```
</p:when>

<p:otherwise>
  <p:validate-with-xml-schema>
    <p:input port="schema">
      <p:document href="v2schema.xsd"/>
    </p:input>
  </p:validate-with-xml-schema>
</p:otherwise>
</p:choose>

<p:xslt>
  <p:input port="stylesheet">
    <p:document href="stylesheet.xsl"/>
  </p:input>
</p:xslt>

</p:pipeline>
```

In XProc 1.0, the dominate construct is the step. Most constructs are implemented by steps, contain steps, or declare new steps that can be re-used elsewhere. The connections between steps are either implicit (as in Example 4) or described via various annotations of names and references to those names.

The result is the design center for XProc 1.0 is the nodes of processing within the graph and not the edges that describe flow of data. This violates goals (3) and (4) from the previous section. There is literal or no element-level constructions that represents a connection in the XML syntax in XProc 1.0.

Further complicating things is that a step is not a black box. A step can describe a reusable portion of a graph and then act like a step. Thus the terminology can be confusing as everything must be a step but everything isn't really always a step. This complicates achieving goals (1) through (4) as well.

In the conceptual graph, there are two basic flows:

1. source \rightarrow validate \rightarrow transform \rightarrow result
2. source \rightarrow if (v1) then validate-v1 else validate-v2 \rightarrow result

We need a data flow language that subjugates steps and focuses on the describing the data flow and its connections. It needs to be clear from the syntax how to discern the shape of the graph, the connections between steps, and any kinds of guards or special processing that might happen to control the flow of data.

Steps should be able to be treated as black boxes with a specific signature. The data flow language should not need to know much about a step other than its signature to be able to describe to an implementation how to invoke a step. This

kind of implementation agnosticism will enable the same data flow language to be used in radically different environments.

6. Überproc: a hammer for all your nails

The fundamental question at this point is whether we've lost our way in amongst our musing of new and better ways to specify pipelines. For some, XProc is another and more complicated way of doing what Makefiles or Ant build scripts can already accomplish. For others, it is a frustrating but useful way to specify complex processes on documents.

Example 5. A Simple Example

```
xproc version = "2.0";

"doc.xml" → xinclude()
          → [$1,"stylesheet.xsl"] → xslt()
          » "result.html"
```

Consider the example shown in Example 5 that processes a fixed file with XInclude and transforms it to HTML with XSLT. The pipeline data flow starts with the URI reference to the input document on line 3 that is fed into the XInclude() step via the chain operator (a right arrow). Following the XInclude step is a binding of the result and the stylesheet as the inputs for an XSLT transformation. Finally, on the last line, the output is sent to a particular file (a URI) via the append operator (a double greater than). The flow of data from source to result is clear and stated in a compact syntax.

In Example 4 is an original example from the XProc 1.0 specification of an example pipeline that validates using two different schemas based on a version attribute in the document source. This example has been reworked a new syntax in Example 6 that will now be described in brief.

Example 6. XProc, Example 3 Reworked

```
xproc version = "2.0";

inputs $source as document-node();
outputs $result as document-node();

$source → { if (xs:decimal($1/*/@version) < 2.0)
             then [$1,"v1schema.xsd"] → validate-with-xml-schema() » @1
             else [$1,"v2schema.xsd"] → validate-with-xml-schema() » @1
           }
          → [$1,"stylesheet.xsl"] → xslt()
          » $result
```

At the top we have a declaration of the inputs and outputs of the pipeline. As a top-level module, this example can be invoked by a processor. When the processor does so, it must provide an input document to process as `$source` and optionally provide binding (e.g., a file name) for output document `$result`.

The purpose of a pipeline document is to describe a data flow. While step signatures can be declared, we are not currently addressing the issue of describing steps and their implementation any further. This is consistent with XProc 1.0 where steps that are not pipelines in themselves are opaque.

The main construct used to define a data flow is a step chain. A step chain starts with a set of input port bindings is followed by a sequence of applications of the chain operator (`->` or `→ U+2192`).

For example, a simple invocation of XInclude is:

```
$in → xinclude()
```

The XInclude step has a single input port named `source` and so that port name can be explicitly specified:

```
source=$in → xinclude()
```

A second step can be easily added to the chain in a similar way. When a preceding step produces output, we can refer to it positionally:

```
$in → xinclude()  
    → [source=$1,stylesheet="stylesheet.xsl"] → xslt()
```

In this case, a list of port bindings is specified by name. The positional variable `$1` refers to the first output of the preceding step (XInclude). The literal for `"stylesheet.xsl"` is a URI reference for loading a stylesheet. The chain operator passes along this information as input bindings to the XSLT step invocation.

We can shorten the invocation to:

```
$in → xinclude()  
    → [$1,"stylesheet.xsl"] → xslt()
```

Finally, to assign results to an output port, we use the append operator (`>>` or `» U+226B`). The append operator can only be preceded by a step chain on the left side. On the right is a variable reference (or list of variables for multiple outputs), a literal for storing data, or positional output port reference (e.g., `@1`, `@2`, etc.).

```
$in → xinclude()  
    → [$1,"stylesheet.xsl"] → xslt()  
    » $out
```

In this new syntax, the use of variables on right side of an append operator is two-fold. First, when there is a declared output port of the same name, the operator sends the output of the chain to that port. All references to such output ports are merged as they occur.

Second, when an port variable is not already declared, the variable is used to signal a connection within the graph. Conceptually, we can think of such a port variable as being a box into which we place the output. We then take the box to where it is used as input to other step chains and “unpack copies” of what is contained in the box.

A variable reference from an append operator may be used as input for multiple step chains:

```
$in → xinclude()
    >> $included
$included → [$1,"stylesheet.xsl"] → xslt()
          >> $out
$included → [$1,"summary.xsl"] → xslt()
          >> $out
```

We can also build up more complex flows by block expressions. A block expression is contained in curly brackets and can contain constructs like conditionals:

```
$in → { if ($1/doc/cheese='cheddar')
        then consume() >> @1
        else reject() >> @1
      }
    >> $out
```

A block expression always has a set of positional inputs and outputs that are addressable as $\$n$ and $@n$, respectively. Unlike XProc 1.0, what is contained within the block expression does not have to have uniform outputs. Instead, whatever flow executes can append outputs to the positional outputs of the block expression. If no such append operations occur, the block expression has empty output.

Various XProc 1.0 “compound steps” translate to operators. For example, iteration over input sequences is now a simple `!` operator that can be inserted into any step chain:

```
$in//section ! { [$1,"chunk.xsl"] → xslt() >> @1 } >> $chunks
```

where the sections are iterated over by the block expression to transform each section with XSLT.

Also, in the previous example is a projection. Expressions can now be applied to inputs to produce new sequences without invoking a step to do so. In the example, the very first operator turns the input into a sequence of `section` elements.

XProc 1.0 has a “viewport” compound step that was not generally well understood by users. The operation has replacement semantics over certain matching sub-trees. In the new syntax, the operator is called “replace” to more directly match its semantics. It applies a block expression and expects a replacement as a result on the first positional port.

For example, replacing sections by their transformation is specified as follows:

```
$in → replace ($1//section) {  
    [$1,"chunk.xml"] → xslt() » @1  
}  
» $out
```

Moreover, with this new concept and syntax we can also innovate and pull in more familiar constructions from other pipelining systems. For example, the `tee` program can be added as its own operation (`tee` or `⊥` U+22A4). It has the same semantics of sending a single input to two places. It has the nice result of allowing inserting small step chains within larger ones:

```
$in → xinclude() ⊥ { $1 » "included.xml" }  
→ [$1,"stylesheet.xml"] → xslt()  
» $out
```

In this example the output of `XInclude` is serialized to a document (`included.xml`) and is also sent to `XSLT` to be transformed. Any block expression can follow the tee operator. This allows complex flows to be built without assigning outputs to intermediary variables.

Finally, it is useful in any programming languages to build expressions and assign them to variables for reuse and clarity. A data flow language such as `XProc` has the task of understanding the relationship between the variable binding and the possible interactions with the inputs the variables may be built from. These dependencies predicate what parts of the flow must be completed before the value of the expression can be built.

A simple let clause is allowed within block expressions and must contain a lexically scoped block expression itself. Within the block expression or each subsequent variable binding, preceding and ancestor variables may be used in expressions. These variable declarations share the same scoping rules as port variable references.

For example, the previous example that checks for a version attribute for validation could use a variable to retrieve the version:

```
$in → {  
  
    let $version := xs:decimal($1/*/@version) {  
        if ($version < 2)  
            then [$1,"schema1.xsd"] → validate-with-xml-schema() » @1  
            else if ($version < 3)  
                then [$1,"schema2.xsd"] → validate-with-xml-schema() » @1  
            else fail("No schema available")  
    }  
  
}
```

Finally, there are many more aspects of XProc both from the requirements for 2.0 and functionality of 1.0 that have not been addressed here. Solutions for literal documents, binary and non-XML documents, templating, re-use mechanism, step declaration, and other features have been worked through. Their descriptions can be found in the forthcoming proposal for this syntax.

7. A Complete Example

This example was translated from an existing XProc 1.0 pipeline that inserts data in a MarkLogic database. The purpose of the pipeline is to perform two updates. One update inserts one document for each weather report element. The other update generates a set of queries that update the position of the weather stations in the database. Both of these updates are driven by the same input data.

Example 7. Database Import Example

```
xproc version = "2.0";

import "marklogic.xpl";
namespace s="http://weather.milowski.com/V/APRS/";
namespace ml="http://example.com/extensions/marklogic"

option $xdb.user as xs:string;
option $xdb.password as xs:string;
option $xdb.host as xs:string;
option $xdb.port as xs:string;

inputs $data as document-node();
outputs $records as xs:integer,
        $positions as xs:integer;

$data/s:aprs/s:report[@type != 'encoded']
  → $1/s:aprs/s:report[@type != 'position']
  → $1/s:aprs/s:report[not(@error)]
  » $filtered

$filtered
  → replace (/s:aprs/s:report) {
    let $uri := 'http://weather.milowski.com/station/' ||
               $1/*/@id || '/' ||
               $1/*/@id/@received || '.xml'
    {
      $1 → ml:insert-document(
        user=$xdb.user, password=$xdb.password,
        host=$xdb.host, port=$xdb.port,
```

```
        uri=$uri)
    data { <inserted/> } >> @1
    }
}
→ $1/*/inserted → count()
>> $records

$filtered
→ [$1,"make-position-update.xsl"] → xslt()
→ replace (/queries/query) {
    $1 → ml:adhoc-query(
        user=$xdb.user, password=$xdb.password,
        host=$xdb.host, port=$xdb.port)
    >> @1
}
→ $1/*/query → count()
>> $positions
```

8. The Future of XProc

XProc started as a mechanism to process XML documents. Since its origins, the Web and the systems that are built upon its technologies have matured and extended far beyond the reaches of XML. Yet, the needs of data processing are even more important than when the working group started.

We have the explicit charge in version 2.0 to address the needs of processing data in a heterogeneous environment. Data of many kinds of formats needs to be processed by pipelines. For XProc to survive in this ecosystem, the standard may benefit from adapting a new perspective.

The interesting position of the proposal in this paper is that XProc is now about processing data and is agnostic to the kind of data and steps. If that data happens to be semi-structured data (e.g. XML, HTML, JSON), we can envision a world where XQuery-like expressions allow the flow to make decisions based on those expressions. Simply stated, checking a version on a JSON object should be no more difficult than doing the same on an XML document.

By focusing on the data flow, we are step agnostic. A valid implementation strategy for a product in a JSON-only world would be to implement no XML-related steps. A similar position might be useful for log processing or relational databases.

Admittedly, such specialization ignores the very interesting aspects of positioning XProc to process data from a variety of data sources and formats. By doing so, XProc matches more directly the reality of the enterprise: data comes in all shapes and sizes. We want to accept and process all data as it is, enriching it via

stepwise processes, and produce more useful information via data flow programming.

References

- [1] *Enabling scientific data on the web*, R Alexander Miłowski, University of Edinburgh, 2014-11-27 <https://www.era.lib.ed.ac.uk/handle/1842/9957>
- [2] *The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud*, Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble, *Nucleic Acids Research*, 2013-05-02, doi:10.1093/nar/gkt328 <http://nar.oxfordjournals.org/content/early/2013/05/02/nar.gkt328>
- [3] *Kepler: an extensible system for design and execution of scientific workflows*, I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, *16th International Conference on Scientific and Statistical Database Management*, 2004 pp. 423-424
- [4] *XSL Transformations (XSLT) Version 2.0*, Michael Kay, W3C, 2007-01-23 <http://www.w3.org/TR/xslt20/>
- [5] *XProc: An XML Pipeline Language*, W3C, 2010-05-11, Norman Walsh, Alex Miłowski, and Henry S. Thompson <http://www.w3.org/TR/xproc/>
- [6] *XQuery 3.0: An XML Query Language*, W3C, 2013-10-22, Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson <http://www.w3.org/TR/xquery-30/>
- [7] *XML Processing Model Requirements and Use Cases*, W3C, 2006-04-11, Alex Miłowski <http://www.w3.org/TR/xproc-requirements/>

Schematron QuickFix

Octavian Nadolu
oXygen XML Editor

<octavian_nadolu@oxygenxml.com>

Nico Kutscherauer
data2type GmbH

<kutscherauer@schematron-quickfix.com>

Abstract

Fixing XML validation errors can be challenging for many users, especially if they are not very familiar with the syntax and structure of XML. For many years, development tools have provided ways to allow users to select actions that automatically fix reported issues for certain programming languages (such as Java, C, etc.). This functionality is usually called "Quick Fixes". In a similar way, XML tools provide Quick Fixes for XML validation errors. For instance, Eclipse has included XML Quick Fixes for over 10 years. Another example of this idea is the spell checking functionality, which provides a list of possible corrections and allows the user to select one of them as a replacement for an incorrect word.

The validation of XML documents against DTD, XML Schema, or RELAX NG schema provides a limited set of possible problems and is usually only able to detect basic structural errors (such as a missing element or attribute) and the corresponding automatic fixes are usually rather straightforward. A more interesting case would be if you are using Schematron to identify issues in XML documents, as the fixes in this case may range from trivial to very complex and there is no automatic way of fixing them.

Schematron solves the limitation that other types of schema have when validating XML documents because it allows the schema author to define the errors and control the messages that are presented to the user. Thus the validation errors are more accessible to users and it ensures that they understand the problem. These messages may also include hints for what the user can do to fix the problem, but this creates a gap because the user still needs to manually correct the issue. This could cause people to waste valuable time and creates the possibility of making additional errors while trying to manually fix the reported problem. Providing a Quick Fix functionality for Schematron validation errors will bridge this gap, saving time and avoiding the potential for causing other issues.

Two years ago, the idea of Schematron QuickFix (SQF) was discussed during the XML Prague conference and it started to take shape. It has now reached a point where we have a draft specification available, a W3C community group dedicated to XML Quick Fixes¹, and two independent SQF implementations. The first draft of the Schematron QuickFix specification was published in April 2015 and it is now available on GitHub² and within the W3C "Quick-Fix Support for XML Community Group".

Schematron QuickFix defines a simple language to specify the actions that are used to fix the detected issues, layered on top of XPath and XSLT, and integrated within Schematron schemas through the Schematron annotation support.

In this session, we will present various use cases that are solved with Schematron QuickFixes, ranging from simple to complex, sometimes involving changes in multiple locations within a document, or even in external documents. We will also discuss the language and challenges related to the SQF implementation. Join us to learn how SQF can be useful in your next XML project!

1. Introduction

Helping users to solve errors in XML documents has always been a challenge. There are various solutions to automatically generate proposals that fix the errors and to present them to the user. The fix proposals can be generated by the validation engine, or based on the error messages, or on error codes that we get from the validation engine. However, a language that can allow the developer to define fixes is more powerful and flexible.

Schematron has become more and more popular in the XML world. Companies are using Schematron to enforce business rules on their documents and to verify the quality of their documents. Schematron is a language that defines rules for the structure and content that an XML document should follow.

Schematron QuickFix (SQF) has been developed as an extension of the Schematron language. It is a language that allows developers to define fixes for the Schematron validation errors.

2. Validation Errors

From the user's point of view the validation error can be described by three things: the validation message, the location of the error (system ID and position in the document), and how to fix that error.

¹ <https://www.w3.org/community/quickfix/>

² <http://schematron-quickfix.github.io/sqf>

Description	System ID	Location
❗ cvc-identity-constraint.4.3: Key 'keyref1' with value 'harris' not	\\Samples\personal\personal-schema.xml	60:13
❗ cvc-id.1: There is no ID/IDREF binding for IDREF 'robert.taylor'.	\\Samples\personal\personal-schema.xml	12:28
❗ cvc-id.1: There is no ID/IDREF binding for IDREF 'harris'.	\\Samples\personal\personal-schema.xml	21:23

Figure 1. Validation Errors

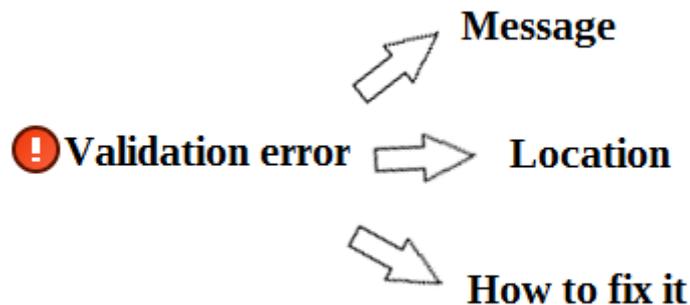


Figure 2. Validation Error Description

When an XML document is validated against DTD, XSD or RNG schema, the error messages refer more to the XML syntax of the document and are not easily understood by the user. To fix this type of errors, the user must understand the validation messages, check the location of the problems, and determine what operations must be done.

For the validation of an XML document against a Schematron schema, the errors are in fact failed constraints or business rules and are meant to be easily understandable by the user. The Schematron developers can better explain what operations should be done in order to fix a problem because they control the error messages. Also, the location of the problem can be specified better in the Schematron schema.

However, in both cases, to fix the error the user must do the operations manually and this can result in generating even more errors. The automation of the error-fixing process will help the user to solve the problem faster and with fewer or no errors.

A good example of error fixing is a spell checker. A spell checker presents errors but also offers a set of solutions to fix them. Similarly, solutions can be generated for XML validation errors.

3. Fixing Validation Errors

Over the course of time various IDEs (such as Eclipse or IntelliJ IDEA) have implemented fixes for XML validation errors and helped the user to solve the errors

by offering fix proposals. The fix proposals can be implemented directly in the validation engine, or it would be better to have an implementation that does not depend on the validation engine. A way to solve this is by analyzing the validation message, error code, and the error location that the engine provides.

We can say that the validation errors can be split into two categories:

- Predefined – Defined in the validation engine. This is the case when an XML document is validated against a DTD, XSD or RNG schema.
- Custom – Defined by the user. This is the case when an XML document is validated against a Schematron schema.

3.1. Fixing Predefined Errors

For the predefined errors, fixes can be provided automatically based on the message, error code (if there is one), and location of the error. A limitation of this approach could be that each validation engine might provide its own predefined messages and error codes. Therefore, you will need an implementation for each validation engine.

However, by providing fix proposals for these types of errors, the user will be helped with not only solving the problem, but also understanding it. The messages from the validation processor are often difficult to understand for novice users. For example, when an ID definition is missing, you might get the following error message:

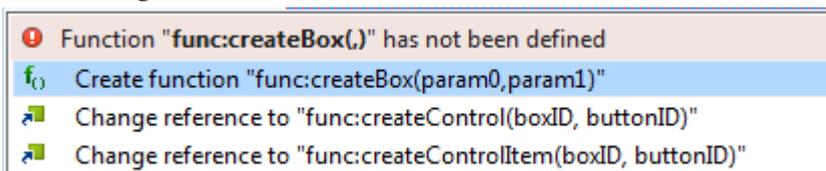
cvc-id.1: There is no ID/IDREF binding for IDREF 'robert.taylor'.

It would be more appropriate to have a more understandable message, such as: "There is an invalid ID reference: 'robert.taylor'. Would you like to change it to the similar ID: 'robert.taylor'?"

Another example could be when an XSL document is validated and we have an undeclared function. This might cause the following error to be presented:

XPST0017 XPath syntax error at char 0 on line 1802 near {...x(\$boxID, func:getButtonId(...): Cannot find a matching 2-argument function named {http://www.oxygenxml.com/doc/xsl/functions}createBox()

In this case, a more appropriate way to present the error might be: "The function 'func:getButtonId()' has not been defined. Would you like to create this function or change the reference to a function with a similar name?"



3.2. Fixing Custom Errors

When an XML document is validated against a Schematron schema, we obtain customized errors. The errors are defined in the Schematron schema using the `sch:assert` and `sch:report` elements. In this case, it is difficult, and almost impossible, to generate a fix based on the error message and location.

Therefore, a solution to propose fixes for this type of errors is to define the fixes directly in the schema. A Schematron developer can create fixes and associate them with the *assert* or *report* message.

```
<sch:rule context="title">
  <sch:report test="exists(b)" sqf:fix="resolveBold">
    Bold element is not allowed in title.</sch:report>
  <sqf:fix id="resolveBold">
    .....
  </sqf:fix>
</rule>
```

These fixes can be defined using the annotations support from Schematron. Schematron allows elements and attributes from other namespaces to be added as annotations in the schema. Thus, these annotations will not interact with the default validation of the Schematron schemas and they will be ignored by Schematron processors that do not support them.

4. Schematron Quick Fixes

To allow users to create fixes for the Schematron error messages, the Schematron QuickFix (SQF) language was created as an extension of Schematron. Using the SQF language, users can define fixes for *assert* or *report* error messages.

The Schematron QuickFix has been defined as a simple but powerful language. It defines some basic operations that need to be implemented by the processor.

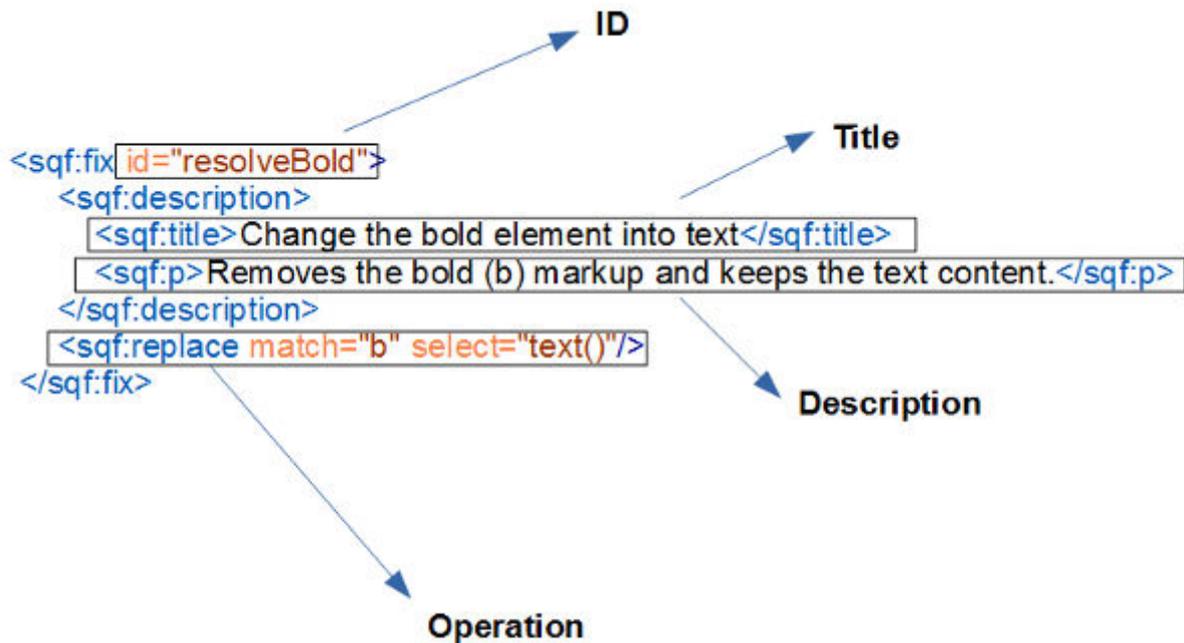


Figure 3. Simple Schematron QuickFix example

The operations can be done with precision in a specified place and you do not need to modify the entire document.

The first draft of the Schematron QuickFix specification was published in April 2015 on the W3C Quick-Fix Support for XML community group³ page.

4.1. SQF Benefits

The Schematron schema can be used to validate any type of XML document. Thus, business rules or constraints can be defined for projects containing DITA, DocBook, XHTML, or TEI documents, and also for stylesheets or XML Schemas.

For DITA, DocBook, XHTML, or TEI documents, simple styling rules can be imposed, such as:

- The title should not contain bold.
- A list should contain more than one list item.

Also, more complex rules can be added, such as:

- Ensure that the table layout is correct.
- Text needs to be normalized (NFC).

For XSLT, XSD, or RNG documents, you can define coding styles, such as:

- The names of the variables must not contain '-', and it is recommended to use the camel case format.

³ <https://www.w3.org/community/quickfix/>

- The names of templates and functions should not exceed a specified length.

Some of these rules can be solved very easily, but a less experienced user might still make mistakes and add other errors. On the other hand, a user with experience might be able solve them rather easily, but might need to perform a few operations.

For more complex problems (for instance, problems that will update multiple nodes in the document or make complex conversions), it would be better to have an action to do this automatically.

4.2. SQF Implementations

An SQF fix consists of a set of operations that must be performed in an XML document. These are basic operations (add, delete, replace, and string replace) that need to make precise changes in the document. This means that when a fix is applied, only the affected part of the XML document should be changed and the DOCTYPE declaration, entities, etc. must be preserved.

There are two types of implementations that can be used to execute Schematron QuickFixes:

- Using an engine (implemented in Java, C, or other language) that collects the fixes during the Schematron validation process, and performs the modification precisely (using the engine programming language) when the fix is applied.
- Using an XSLT engine that generates a set of XSLT scripts during the Schematron validation process, and these scripts are applied when the fix is executed.

5. Schematron QuickFix Language

As shown in the example above (see Figure 3), the QuickFix (defined by the `sqf:fix` element) is structured as follows:

- *ID* – Used to reference the QuickFix by `sch:assert` or `sch:report` elements.
- *Title and description* – The *title* is used for the name of the QuickFix. The optional *description* can be used to give the user additional information regarding the QuickFix.
- *Operation* – The *Activity Elements* that specify the actions of the QuickFix.
- *Additional features* – There are also some additional features that had no place in our simple example (such as *User Entries* and `use-when` conditions).

5.1. Reference a QuickFix

To reference a QuickFix, the `sch:assert` or `sch:report` element needs a `sqf:fix` attribute with the ID of the QuickFix. In many cases there are multiple possible

solutions for one error, and the `sqf:fix` attribute is able to reference more than one QuickFix. In this case, the `sqf:fix` attribute should contain a list of those QuickFix IDs (separated with whitespace) that should be referenced:

Example 1. Reference to Multiple QuickFixes

```
<sch:rule context="title">
  <sch:report test="exists(b) "
    sqf:fix="resolveBold deleteBold">
    Bold element is not allowed in title.</sch:report>
  <sqf:fix id="resolveBold">
    <sqf:description>
      <sqf:title>Change the bold element into text
    </sqf:title>
      <sqf:p>Remove the bold (b) markup and keep the
        text content</sqf:p>
    </sqf:description>
    <sqf:replace match="b" select="text()"/>
  </sqf:fix>
  <sqf:fix id="deleteBold">
    <sqf:description>
      <sqf:title>Delete the bold element</sqf:title>
      <sqf:p>Remove the bold (b) markup including the
        text content</sqf:p>
    </sqf:description>
    <sqf:delete match="b"/>
  </sqf:fix>
</sch:rule>
```

5.1.1. Scope

The referenced QuickFix needs to be in the scope of the `sch:report` or `sch:assert` element. It is in the scope if the `sqf:fix` element is a child of the same `sch:rule` element (which contains the `sch:report` or `sch:assert` element) or if the QuickFix was defined globally. To define a QuickFix globally, the top-level element `sqf:fixes` should contain one or more `sqf:fix` elements. QuickFixes that are specified in this way are available for all `sch:report` or `sch:assert` elements of the schema.

Example 2. Global QuickFixes

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
  queryBinding="xslt2">
  <sch:pattern>
    <sch:rule context="title">
```

```

    <sch:report test="exists(b) "
      sqf:fix="resolveBold deleteBold">
      Bold element is not allowed in title.</sch:report>
    </sch:rule>
  </sch:pattern>
  <sqf:fixes>
    <sqf:fix id="resolveBold">
      <sqf:description>
        <sqf:title>Change the bold element into text
        </sqf:title>
        <sqf:p>Remove the bold (b) markup and keep the
          text content</sqf:p>
      </sqf:description>
      <sqf:replace match="b" select="text()"/>
    </sqf:fix>
    <sqf:fix id="deleteBold">
      <sqf:description>
        <sqf:title>Delete the bold element</sqf:title>
        <sqf:p>Remove the bold (b) markup including the
          text content</sqf:p>
      </sqf:description>
      <sqf:delete match="b"/>
    </sqf:fix>
  </sqf:fixes>
</sch:schema>

```

5.1.2. QuickFix Groups

To avoid long lists of IDs, it is possible to reference a *QuickFix group*. A QuickFix group is a set of QuickFixes. A reference to a QuickFix group is equal to a reference to each QuickFix in this group.

Example 3. Reference to a QuickFix Group

```

<sch:report test="exists(b) "
  sqf:fix="bold">
  Bold element is not allowed in title.</sch:report>
<sqf:group id="bold">
  <sqf:fix id="resolveBold">
    <sqf:description>
      <sqf:title>Change the bold element into text
      </sqf:title>
      <sqf:p>Remove the bold (b) markup and keep the
        text content</sqf:p>
    </sqf:description>
    <sqf:replace match="b" select="text()"/>
  </sqf:fix>

```

```
<sqf:fix id="deleteBold">
  <sqf:description>
    <sqf:title>Delete the bold element</sqf:title>
    <sqf:p>Remove the bold (b) markup including the
      text content</sqf:p>
  </sqf:description>
  <sqf:delete match="b"/>
</sqf:fix>
</sqf:group>
```

To reference a QuickFix group, the `sqf:group` element also has an `id` attribute. For the `sch:assert` or `sch:report` elements, there is no difference between referencing a QuickFix or QuickFix group.

A QuickFix group can also be defined globally (in an `sqf:fixes` element) or locally (in an `sch:rule` element).

5.2. Title and Description

The title (`sqf:title` element) is very important for a QuickFix. It is a challenge for the developer to create short titles that provide the user with enough information to understand what will happen when the QuickFix is executed. The developer can also deliver more information by using `sqf:p` elements.

Also, as with Schematron error messages, the title and optional description can be customized by using the Schematron elements `sch:value-of` or `sch:name`.

Example 4. Customized Title and Description

```
<sqf:fix id="delete">
  <sqf:description>
    <sqf:title>Delete the <sch:name/> element</sqf:title>
    <sqf:p>The <sch:name/> element is misplaced in the
      <sch:name path=".." /> element.</sqf:p>
    <sqf:p>This QuickFix will delete the <sch:name/>
      element<sch:value-of select="
        if (./node())
        then ' with all its content.'
        else '.'" />
  </sqf:p>
</sqf:description>
  <sqf:delete />
</sqf:fix>
```

This QuickFix deletes the context node of the error (matched by the `sch:rule` element). This is a common solution, and if specified in this way, it can be reused in various contexts.

5.3. Activity Elements

After the description, the developer must specify what the QuickFix should actually do. To define actions, the developer can choose between four types of *Activity Elements*.

The developer can also add any number of Activity Elements to one QuickFix. Each Activity Element is executed separately, in the context of the node where the error occurred (matched by the `sch:rule` element).

All Activity Elements have a `match` attribute to select nodes by XPath, relative to the context node of the error. These nodes are called *anchor nodes*. The kind of processing of the anchor nodes depends on the type, additional attributes, and the content of the Activity Element. An Activity Element that has no `match` attribute selects the context of the error as its anchor node.

There are four types of Activity Elements:

- `sqf:delete`
- `sqf:replace`
- `sqf:add`
- `sqf:stringReplace`

5.3.1. Delete Nodes

The delete action is the simplest type of Activity Element. The `sqf:delete` element deletes the anchor nodes.

Example 5. Delete the Error Context

```
<sch:rule context="p">
  <sch:report test="normalize-space(.) = ''"
    sqf:fix="delete">
    <sch:name/> element without text is not allowed.</sch:report>
  <sqf:fix id="delete">
    <sqf:description>
      <sqf:title>Delete the <sch:name/> element</sqf:title>
    </sqf:description>
    <sqf:delete/>
  </sqf:fix>
</sch:rule>
```

The empty `p` element is deleted by the QuickFix `delete`.

Example 6. Delete Nodes by Using the Match Attribute

```
<sch:rule context="title">
  <sch:report test="comment()"
    sqf:fix="deleteComment">
```

```
    Comments are not allowed in the <sch:name/> element.</sch:report>
<sqf:fix id="deleteComment">
  <sqf:description>
    <sqf:title>Delete the comment.</sqf:title>
  </sqf:description>
  <sqf:delete match="comment()"/>
</sqf:fix>
</sch:rule>
```

All comments in the `title` element are deleted by the QuickFix `deleteComment`.

5.3.2. Replace Nodes with New Content

The Activity Element `sqf:replace` replaces each anchor node with new content. There are three methods to create new content in SQF. In some cases, they can also be combined.

5.3.2.1. Replace by Using SQF Attributes

The combination of the attributes `node-type` and `target` creates exactly one node. The `node-type` attribute specifies the type of the node with the values: `element`, `attribute`, `processing-instruction`, `pi`, `comment`, or `keep`. The value `pi` is the short version of `processing-instruction`. The value `keep` is provided to create the same type of node as the anchor node.

If the value of the `node-type` attribute is not `comment`, the `target` attribute is required to specify the name of the new node. The attribute value is analyzed like an attribute value template (as defined in XSLT: <https://www.w3.org/TR/xslt20/#attribute-value-templates>). An XPath expression, which is marked with curly brackets (`{XPath}`), is evaluated to generate the node name. After evaluation, the value needs to be a valid `xs:QName`.

You can combine this method of specifying the value or content of the new node with one of the other two methods of creating new content. If the attributes `node-type` and `target` create an element, the new content generated by using the second method (described below) becomes the new content of this element. Otherwise, the new content is transformed into an atomic value (attribute value, comment value, etc.) in the same way as child nodes of `xsl:attribute` elements are transformed to attribute values in XSLT.

5.3.2.2. Replace by Using XPath

The `select` attribute of the Activity Element creates new content by using XPath. The given XPath expression is evaluated in the context of the anchor node. The return value is used to generate new content. Returned nodes are copied and atomic values are transformed to text nodes.

If the `select` attribute is set, the Activity Element must be empty.

Example 7. Create Nodes with the SQF Attributes and Copy Nodes with XPath

```
<sch:rule context="b">
  <sch:report test="ancestor::b"
    sqf:fix="italic">
    Bold in bold is not allowed.</sch:report>
  <sqf:fix id="italic">
    <sqf:description>
      <sqf:title>Change it to italic.</sqf:title>
    </sqf:description>
    <sqf:replace target="i" node-type="element" select="node()"/>
  </sqf:fix>
</sch:rule>
```

The recursive `b` element is replaced by an `i` element. The content is copied.

5.3.2.3. Replace by Using New Content

If the `select` attribute is omitted, the content of the Activity Element is used to create new content. The content is evaluated in the same way as the content of an `xsl:template` element in XSLT 2.0. That also means that any element that is not in the XSLT namespace is handled as a Literal Result Element. Exceptions are the SQF element `sqf:keep` and the Schematron elements `sch:let`, `sch:value-of`, and `sch:name`. The Schematron elements are handled in the same way as in Schematron. The `sqf:keep` element will copy nodes that are selected by the XPath expression in the `select` attribute.

The initial context of this "template" will be the anchor node.

Example 8. Create Nodes with New Content

```
<sqf:fix id="italic">
  <sqf:description>
    <sqf:title>Change it to italic.</sqf:title>
  </sqf:description>
  <sqf:replace>
    <i>
      <sqf:keep select="node()"/>
    </i>
  </sqf:replace>
</sqf:fix>
```

This QuickFix example performs the same actions as the previous example (Example 7), but uses a different way.

5.3.3. Add the New Content

To insert new content without replacing existing nodes, the Activity Element `sqf:add` is used. It creates new content in the same way as the `sqf:replace` element, but the new content is inserted relative to each anchor node instead of replacing it.

To specify the exact position, the `sqf:add` element can have a `position` attribute to indicate that the new content is inserted after (value `after`), before (value `before`) the anchor node, as the first (value `first-child` – default value), or as the last child (value `last-child`) of the anchor node. If the new content is an attribute, the `position` attribute should not be used because it is always added automatically as an attribute of the anchor node.

Example 9. Add Nodes

```
<sch:rule context="h2">
  <sch:assert test="preceding::h1"
    sqf:fix="addH1">
    A h2 should not be used without a h1 before.</sch:assert>
  <sqf:fix id="addH1">
    <sqf:description>
      <sqf:title>Add a h1 element before the h2 element.</►
sqf:title>
    </sqf:description>
    <sqf:add node-type="element" target="h1" position="before"/>
  </sqf:fix>
</sch:rule>
```

Directly before the `h2`, a new `h1` is inserted. Because there is no content defined for the new element, the `h1` is empty.

5.3.4. Replace Substrings

The Activity Element `sqf:stringReplace` is a special case. There is a restriction for the anchor nodes in that they must be text nodes. These text nodes are analyzed by a regular expression provided in the `regex` attribute. Each substring of the anchor text node that matches to the regular expression is replaced by new content. The new content is created in the same way as the `sqf:replace` element, although the attributes `target` and `node-type` are not available for the `sqf:stringReplace` element.

Example 10. Replace Substrings

```
<sch:report test="matches(., '____')"
  sqf:fix="form">
  More than three underscores in a row shouldn't be used.</sch:report>
```

```

<sqf:fix id="form">
  <sqf:description>
    <sqf:title>Replace the misused characters by a form element.</▶
sqf:title>
  </sqf:description>
  <sqf:stringReplace regex="___+">
    <form/>
  </sqf:stringReplace>
</sqf:fix>

```

5.4. Additional Features

5.4.1. User Entry

For some solutions of an error, it is impossible to define a QuickFix without getting more information from the user. For instance, if the Schematron error is that the `title` element is empty (and it should not be). For this case, the solution would be to define a new title. A predefined QuickFix for this issue is impossible because there is an unlimited number of possible titles. Therefore, additional input from the user is needed.

For this case, one or more *User Entries* can be defined for any QuickFix. The User Entry acts like a parameter whose value is set by the user during the execution of the QuickFix.

Example 11. User Entry

```

<sch:rule context="title">
  <sch:assert test="normalize-space(.) != ''"
    sqf:fix="title">
    A title shouldn't be empty.</sch:assert>
  <sqf:fix id="title">
    <sqf:description>
      <sqf:title>Set a title</sqf:title>
      <sqf:p>This QuickFix will set a title by using a
        User Entry.</sqf:p>
    </sqf:description>
    <sqf:user-entry name="title">
      <sqf:description>
        <sqf:title>Please enter the new title.
        </sqf:title>
      </sqf:description>
    </sqf:user-entry>
    <sqf:replace target="title" node-type="element"
      select="$title" />
  </sqf:fix>
</sch:rule>

```

The `sqf:user-entry` element has a `name` attribute and contains an `sqf:description` element. The `name` attribute specifies the name of the User Entry. For XPath expressions of all Activity Elements, a variable is now available that has the name of the User Entry, and the variable can be used to access the value of the User Entry.

The `sqf:description` element is used to define a title (`sqf:title`) and optionally an additional description (`sqf:p` elements) of the particular use-case of the User Entry.

5.4.2. Use-when Condition

In some cases, the usefulness of a QuickFix depends on the context of the error. For one error, the QuickFix might make sense, but for another error of the same kind, it might be useless because it would create another error.

Example 12. Useless QuickFix

Schematron schema:

```
<sch:rule context="title">
  <sch:report test="exists(b) "
    sqf:fix="resolveBold deleteBold">
    Bold element is not allowed in title.</sch:report>
  <sch:assert test="normalize-space(.) != ''">
    A title shouldn't be empty.</sch:assert>
  <sqf:fix id="resolveBold">
    <!--...-->
  </sqf:fix>
  <sqf:fix id="deleteBold">
    <sqf:description>
      <sqf:title>Delete the bold element</sqf:title>
      <sqf:p>Remove the bold (b) markup including the
        text content</sqf:p>
    </sqf:description>
    <sqf:delete match="b"/>
  </sqf:fix>
</sch:rule>
```

XML instance:

```
<article>
  <section>
    <title><b>This title should be bold</b></title>
  </section>
  <section>
    <title>This title should be bold<b/></title>
  </section>
</article>
```

Both titles will produce the same error, but for the first title, the QuickFix `deleteBold` is useless because after its execution the `title` element would be empty and would produce another kind of error (caused by the `sch:assert` element).

To avoid such a subsequent error, the `use-when` condition helps. The XPath expression in the `use-when` attribute of the `sqf:fix` element is a condition to provide the QuickFix:

Example 13. Use-when Condition

```
<sqf:fix id="deleteBold"
  use-when="node() [normalize-space(.) != ''] except b">
  <sqf:description>
    <sqf:title>Delete the bold element</sqf:title>
    <sqf:p>Remove the bold (b) markup including the
      text content</sqf:p>
  </sqf:description>
  <sqf:delete match="b"/>
</sqf:fix>
```

The QuickFix `deleteBold` is proposed to the user only if the `title` element contains text other than whitespaces and has a `node` that is not a `b` element.

The `use-when` attribute is also available for each Activity Element, so the developer is able to specify use-when conditions for each single Activity Element.

6. Projects Using SQF

There are some projects available that use the SQF language to propose fixes:

- Dynamic Information Model (DIM) project (<https://github.com/oxygenxml/dim>) – Uses SQF to propose fixes for the Schematron rules.
- TEI (<http://wiki.tei-c.org/index.php/Category:Schematron>) – A page that contains Schematron schemas and SQF that can be used to determine and fix various problems in TEI documents.
- <oxygen/> DITA framework – A built-in framework in oXygen XML Editor for DITA documents that contains a set of Schematron schemas and SQF fixes that can be used to impose rules and propose fixes to solve errors.
- <oxygen/> User Manual (<https://github.com/oxygenxml/userguide>) – A public version of the oXygen XML Editor user guide that provides an example where SQF rules have been implemented on a DITA project.

7. Conclusions and Future Plans

The Schematron QuickFix language is useful for XML projects because it offers proposals to solve the Schematron validation errors and warnings. Thus, the users understand the problem better, make fewer mistakes, and solve the problem in less time. The language is very simple, has just a few elements, and can be easily adopted by the Schematron developers to create quick-fix proposals.

From the implementations perspective, SQF should not be difficult to implement because the language has only four basic operations (add, delete, replace, and string replace) that must be supported.

In the near future, we plan to publish the second draft of the Schematron QuickFix specification that will contain new things such as how you can execute fixes on other documents, new definitions, and examples for the SQF elements.

We intend to update the specification and add other elements or change their behavior, based upon discussions and feedback that we have received on the SQF GitHub project⁴. For example, the *call-fix* element will be able to reference a group of operations and support will be added to allow the developer to generate the fixes dynamically.

⁴ <https://github.com/schematron-quickfix/sqf>

Validating office documents in the publishing production workflow

Andrew Sales

Andrew Sales Digital Publishing Limited

<andrew@andrewsales.com>

Abstract

This paper will present an innovative, open-source approach to verifying the quality of content, by applying business rules to the markup that underlies OOXML and ODF. These are normally onerous to author in full, as they entail (in the case of word-processing documents) inferring structure from a sequence of styled paragraphs and other items. It will show how these rules can be expressed declaratively and succinctly, using a schema language. It will also demonstrate how this language can be interpreted by a processor to produce human-readable reports, including as annotations in the original source. Finally, it will outline how this information can potentially be put to use in correcting defects in content, via the Schematron QuickFix (SQF) framework.

The paper builds on work presented at XML London 2015 and available on GitHub at <https://github.com/AndrewSales/schematron4word>.

Keywords: Schematron, validation, OOXML, ODF, SQF

1. Background

The paper previously presented[1] on work in this area focused on the idea of applying quality assurance to styled word-processing documents as the basis of sound data capture. Some publishers need to do it this way, on grounds of convenience or cost — often legacy workflows will exist. Whereas macros may have been used to perform this kind of in-application validation historically, the availability of office formats as XML now means XSLT and thereby Schematron are viable, and arguably more maintainable and more portable, alternatives.

Beyond validation, it is also possible to insert error messages at the location of the fault into the original document, in the form of comments for editorial review.

This approach has been shown to work in both OOXML and ODF, and in other office documents, for example spreadsheets.

However, writing this kind of Schematron schema, even with the aid of libraries of abstract patterns, can entail long-winded and convoluted constraints because of the nature of the task: attempting to impose an implied structure on a

series of styles and other objects. Since a better fit would appear to be a schema language for styled documents, this approach was proposed. A schema language has already been devised; the present paper pursues the challenge of a processor for this language which can be used to validate styled documents and so enable the display of any errors *in situ* in the original, with the potential to fix them (semi-)automatically.

2. A processor for style schemas

Francis Cave's "style schema"[2] seeks to express word-processing document constraints using a custom schema language. It is close to RELAX NG in format (by which means it is also specified), with some document-specific additions, such as sections and body/header/footer, as shown in this excerpt (as RNC for readability):

```
Section = element Section { Body, Header?, Footer? }
Header = element Header { (Para | Table)+ }
Footer = element Footer { (Para | Table)+ }
Para =
  element Para {
    (Drawing
      | DocProperty
      | Text
      | Tab
      | Bookmark
      | Comment
      | ParaAnyOf)*,
    style_att
  }
ParaAnyOf =
  element OneOrMore {
    element Choice {
      (Drawing
        | DocProperty
        | Text
        | Tab
        | Bookmark
        | Comment
        | ParaAnyOf)+
    }
  }
```

Applying these schema-specified constraints to an office document requires a custom processor to interpret them.

The earlier paper proposed generating Schematron from the style schema to do this, as that technology had been used so far successfully with manually-auth-

ored constraints. Certain things are quick and easy to achieve with Schematron in this way, such as enumerating all the allowed styles for a document, and, with a little more interpretation, which are mandatory and in what order. This is a good starting point. However, choice and optionality can make assertions lengthy and complex and with a style schema of any length or complexity in itself, the number of requisite assertions could grow considerably.

Moreover, there is the problem of maintaining context. In a Schematron schema expected to constrain allowed sequences of styles, you must elaborate the possible contexts (in `rule/@context`), and where styles are re-used in a schema these can be many.

Faced with these considerations, a bespoke processor is appealing as it would offer most control and be designed specifically for the task, but is a significant undertaking. With a formal expression of the grammar in place, utilities such as REx¹ can produce a Java class or XSLT which could be adapted to suit². This would entail generating the EBNF for a given style schema (the productions will vary from schema to schema) and some customisation of the resulting parser to report an error's XPath.

However, validating documents against a grammar clearly has several well-established precedents in XML processing, so these perhaps present a more convenient route. If the style schema can be successfully transformed into one of the existing XML schema languages, then it should be possible to exploit pre-existing parsing technology.

3. An implementation: DTD + SAX handler

As a first cut, the tried-and-tested DTD was selected. They may seem rather outmoded in the schema era, but DTDs still have much to offer. The transform was reasonably straightforward: the limitations and simplicity of DTD syntax are a boon in this respect. Also, SAX³ libraries provide a convenient way to harvest an XPath to the location of an error. As a quick win, this approach seems promising.

The general approach is:

1. transform the style schema to a DTD, declaring style names as pseudo-elements;
2. transform the styled source to pseudo-elements;
3. set up a validating parser that reports the XPath to any errors encountered in the styled source transformed in the previous step;

¹<http://www.bottlecaps.de/rex/>

²For an illuminating discussion of validators generated in this way, see Tony Graham's paper on validating XSL-FO[3].

³Simple API for XML, <http://sax.sourceforge.net/>

4. insert annotations into the styled source by way of an identity transform, using the XPaths reported in the previous step.

3.1. Transforming the style schema to a DTD

For this, we treat each distinct stylename declared in the schema as an element to declare in the DTD output:

```
<!-- each element that can have a stylename is declared as an element -->
<xsl:template match="sts:Para | sts:Table | sts:Text" mode="declare">
  <xsl:text><!ELEMENT </xsl:text>
  <xsl:sequence select="if(@styleID) then asdp:get-stylename(.) else ►
name()"/>
  <xsl:text> (#PCDATA</xsl:text>
  <xsl:if test="$declare-built-in-inline-elements">
    <xsl:text>|%built-in-inline;</xsl:text>
  </xsl:if>
  <xsl:text>)*>
</xsl:text>
</xsl:template>
```

Built-in styles (such as bold, italic etc) are declared in a static, pre-built module:

```
<!ENTITY % built-in-inline "b|i|ul|sub|sup|sc|url">
  <!ENTITY % auto-generated SYSTEM "auto-gen.dtd">
  %auto-generated;
```

where `auto-gen.dtd` is the automatically-generated part, e.g:

```
<!ELEMENT Document ((Para.articlehead,
(Para.bodytext
)+, ((Para.bibhead,
(Para.bib
)+))?, Footer))>

<!ELEMENT Para.Footer (#PCDATA|%built-in-inline;)*>
<!ATTLIST Para.Footer xpath CDATA #REQUIRED>

<!ELEMENT Para.articlehead (#PCDATA|%built-in-inline;)*>
<!ATTLIST Para.articlehead xpath CDATA #REQUIRED>

<!ELEMENT Para.bodytext (#PCDATA|%built-in-inline;|Text.bibref)*>
<!ATTLIST Para.bodytext xpath CDATA #REQUIRED>

<!ELEMENT Text.bibref (#PCDATA|%built-in-inline;)*>
<!ATTLIST Text.bibref xpath CDATA #REQUIRED>

<!ELEMENT Para.bibhead (#PCDATA|%built-in-inline;)*>
<!ATTLIST Para.bibhead xpath CDATA #REQUIRED>
```

```
<!ELEMENT Para.bib (#PCDATA|%built-in-inline;|Text.bibdate|Text.bibnum) *>
<!ATTLIST Para.bib xpath CDATA #REQUIRED>

<!ELEMENT Text.bibdate (#PCDATA|%built-in-inline;)*>
<!ATTLIST Text.bibdate xpath CDATA #REQUIRED>

<!ELEMENT Text.bibnum (#PCDATA|%built-in-inline;)*>
<!ATTLIST Text.bibnum xpath CDATA #REQUIRED>

<!ELEMENT Footer (Para.Footer)>
```

Character styles (in style schema parlance, `Para/Text[@styleID]`) receive similar treatment and are inserted into the mixed content model, as in `Para.bodytext` and `Para.bib` above. The purpose of attribute `xpath` is discussed in the next section.

3.2. Transforming the source document styles to pseudo-elements

This approach of course necessitates transforming the input document too, so that it can still be usefully validated. To achieve this, we treat each styled paragraph, inline style or object (such as tables, drawings etc) as a pseudo-element. The transform therefore has to do two things:

- create elements from styles that correspond with those declared in the DTD generated from the style schema; and
- provide a route back to the location in the source document, so that a meaningful report of the error can be made.

The former can be taken from the `w:styleId` attribute in the (e.g. paragraph or run) properties,⁴ with the structure type (paragraph/character style, header, footer) prepended:

```
<Para.bib><Text.bibnum>[2]</Text.bibnum>
<url address="https://www.oasis-open.org/standards">https://www.oasis-
open.org/standards#opendocumentv1.2</url>. Retrieved ►
<Text.bibdate>2015-03-08</Text.bibdate>.</Para.bib>
```

For the latter, XPaths to the nodes in the original are inserted as an attribute:

```
<Para.bib xpath="/w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-
section[1]/wx:sub-section[1]/w:p[3]"> [...]
```

and the character styles mentioned above carried through:

⁴Although it will be absent in some cases, such as default styles, and is addressed e.g. by `Para.Normal` (appropriately enough perhaps for a phantom stylename) to denote Word's default paragraph style.

```
<Text.bibnum xpath="/w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-section[1]/wx:sub-section[1]/w:p[2]/w:r[1]">[1]</Text.bibnum>
```

This is still a direct translation of the flat (in this case, WordProcessingML) format; no attempt is made to infer structure at the stage of generating these pseudo-elements.

3.3. Validator with error location reporting

A validating instance of the Apache Xerces2-J⁵ SAXParser is created with some `org.xml.sax.helpers.DefaultHandler` methods overridden:

```
private ArrayList<ParseError> errs;
private ParseError error;
private Stack<String> context;

Validator(){
    errs = new ArrayList<ParseError>();
    context = new Stack<String>();
}

public void startElement(String uri, String localName, String qName, ►
Attributes atts) throws SAXException {
    context.push(atts.getValue("xpath"));

    if(error != null){
        error.setXPath(context.peek());
        errs.add(error);
        error = null;
    }
}

public void endElement(String uri, String localName, String qName) ►
throws SAXException {
    String xpath = context.pop();

    if(error != null){
        error.setXPath(xpath);
        errs.add(error);
        error = null;
    }
}

public void error(SAXParseException e) throws SAXException {
```

⁵<http://xerces.apache.org/xerces2-j/>

```
error = new ParseError(e.getMessage());
}
```

A stack is maintained so that errors reported just before an `endElement` event (typically about the completeness of the element just ended) have their XPath location reported correctly. A complete document of errors is emitted on `endDocument()`, e.g.

```
<errors>
  <error location='/w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-
section[1]/w:p[10]'>Element type "Para.Normal" must be declared.</error>
  <error location='/w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-
section[1]/wx:sub-section[1]/w:p[1]'>Element type "Para.Heading2" must ►
be declared.</error>
  <error location='/w:wordDocument[1]/w:body[1]/wx:sect[1]/wx:sub-
section[1]/w:p[1]'>The content of element type "Document" must ►
match "(Para.articlehead,Para.bodytext+, (Para.bibhead,Para.bib
+)?,Footer)".</error>
</errors>
```

Note that the final error reported in this example pertains to the root element ("Document" is not complete when the document ends), so the XPath reported here is by default associated with the *first paragraph* in the body of the original document, to enable it to be displayed when rendered.

3.4. Inserting errors into the source

To present these errors to the user we are faced with two challenges, one technical, the other semantic. The first is relatively trivial: adapt the existing `errs2xsl.xsl` to accept an error document emitted by the custom validator. Here, we simply test for that document's presence via a parameter passed to the stylesheet and process any errors it contains just as though they had come from the Schematron SVRL. The second is more significant to the user, as we are showing them error messages designed for markup, but they are looking at *styles* in an unstructured environment. In this case, a crude re-wording in the XSLT of the parser-specific messages suffices, as a limited variety of errors is expected. So, for instance the error message `Element type "Para.Normal" must be declared. becomes unrecognised style "Normal" when rendered in the UI.`

3.5. Filling in the gaps with Schematron

So much for checking the implied structure of the document. But there remains the issue of the expected content of the individual pseudo-elements. For example, the style schema expresses that a given style should have fixed text via the text content of `Text`, as in the case of `bibhead`, which style should cue the start of the bibliography:

```
<Define name="bibhead">
  <Para styleID="bibhead">
    <Text>References</Text>
  </Para>
</Define>
```

This can be expressed by generating a rule such as:

```
<rule context='Para.bibhead'>
  <let name='fixed-text' value='"References"'/>
  <assert test='. = $fixed-text'><name/> should have the text ►
content '<value-of select='$fixed-text'/'>'; got '<value-of select='.'/'>
>'</assert>
</rule>
```

Further extension of the style schema format is desirable, to introduce more data-typing, for instance by the simple addition of XSD types. For example, to constrain the content of `bibdate` to a valid date it would be nice to be able to specify this:

```
<Para styleID="bib">
  <Text styleID="bibdate" type='xs:date' />
  <Text styleID="bibnum" />
</Para>
```

and have the corresponding constraint applied automatically by Schematron in much the same way as the previous example.

4. Post mortem

There are certainly limitations to this approach, in that you probably still need to hand-craft some Schematron rules, but so it goes; the majority of the work in checking the structure implied by styles is done by the style schema validator, and some constraining of text content by auto-generated Schematron.

Another consideration is that the main focus for the schema language as it stands is *documents* in the strict sense, as opposed to say spreadsheets or presentations. Other kinds of office document may benefit, but it is hoped that at least the concept is more or less proven.

What we lose, as indicated above, is the user-defined or customisable error message, or at least something which has more meaning in the style-centric world. Instead we are reliant on interpreting and re-wording XML parser messages.

There is also no inherent support yet in the style schema format for cross-references or "co-occurrence constraints". In their absence, the constringer is again reliant on custom Schematron.

And it goes without saying that this is merely one implementation: other schema languages are of course available.

5. Fixing the source

When this technique was first applied as part of a journals workflow, there was no Schematron involved, only pure XSLT to validate and in some cases re-process the document to fix basic errors. Now, we have Schematron Quick Fix,⁶ which offers a formal approach and implementation.

In this case, the question is what fixes can be usefully applied and where to apply them.

The same difficulty besets what can be fixed as it does all of this validation activity: namely that we are inferring structure from markup that may imply it, rather than validating an already instantiated structure. This means a certain amount of guesswork is required about the document author's intentions. Under these circumstances, caution is advised in amending the source, but unambiguous cases (such as the wrong boilerplate text given in a particular style) should prove uncontroversial, not to mention very useful.

In order to allay editorial concerns about fixing less certain cases, perhaps it would be worthwhile implementing them as part of the change tracking markup, as in this (ODF this time) snippet:

```
<office:text>
  <text:tracked-changes>
    <text:changed-region xml:id="ct179952656" ►
text:id="ct179952656">
  <text:deletion>
    <office:change-info>
      <dc:creator>SQF</dc:creator>
      <dc:date>2016-01-24T00:00:00</dc:date>
    </office:change-info>
    <text:p text:style-name="Standard">Bibliography</►
text:p>
  </text:deletion>
</text:changed-region>
  <text:changed-region xml:id="ct179952552" ►
text:id="ct179952552">
  <text:insertion>
    <office:change-info>
      <dc:creator>SQF</dc:creator>
      <dc:date>2016-01-24T00:00:00</dc:date>
    </office:change-info>
  </text:insertion>
```

⁶<http://www.schematron-quickfix.com/>

```
    </text:changed-region>
  </text:tracked-changes>
  <text:p text:style-name="Standard">
    <text:change text:change-id="ct179952656"/>
    <text:change-start text:change-id="ct179952552"/>▶
  >References<text:change-end text:change-id="ct179952552"/>
  </text:p>
</office:text>
```

Here, the original document has been amended – the bibliography heading corrected – albeit "non-invasively". This approach would mean that, once rendered, the editor can use the built-in review tools to accept or reject the change, or otherwise address the issue.

6. Next steps

There is still clearly more implementation work to be done on both the schema and processor side. If this technique is of wider interest, contributions to the effort are very welcome. The focus has been on word-processing mostly in this instance, but it is conceivable that style schemas for other office documents could prove useful.

Bibliography

- [1] *The application of Schematron schemas to word-processing documents*. Andrew Sales. XML London 2015. June 6-7th, 2015. doi:10.14337/XMLLondon15.Sales01.
- [2] *A style schema for word-processing documents*. Francis Cave. February 2015. Personal communication.
- [3] *Validating XSL-FO with Relax NG and Schematron*. Tony Graham. XML London 2015. June 6-7th, 2015. doi:10.14337/XMLLondon15.Graham01.

I would like to express my gratitude to those fellow-sufferers who have kindly allowed their brains to be picked over time: Alex Brown, Francis Cave, Jirka Kosek, Horst Kucharczyk and Kourosch Mojar.

Data Just Wants to Be Format-Neutral

Steven Pemberton
CWI, Amsterdam

Abstract

Invisible XML is a technique for treating any parsable format as if it were XML, and thus allowing any parsable object to be injected into an XML pipeline. The parsing can also be undone, thus allowing roundtripping.

This paper discusses issues with automatic serialisation, and the relationship between Invisible XML grammars and data schemas.

1. Abstraction and Representation

All numbers are abstractions. There is no *thing* that is the number 3, you can't point to it, only to a representation of it. The best that we can say is that the number three is what it is that three apples and three chairs have in common.

Given the right context, we understand that "CXXVII", "127", "7F", "1111111" and "one hundred and twenty-seven" are all representations of the same number: the underlying concept is identical. We choose the representations we use either through familiarity, or for convenience. For instance, while it is relatively easy to add numbers expressed in roman numerals together, it is very hard to multiply them; binary representations are used in computers because the electronics needed to manipulate them are much simpler.

And so it is with data representations in general. To take an example, there is no *essential* difference between the JSON

```
{"temperature": {"scale": "C"; "value": 21}}
```

and an equivalent XML

```
<temperature scale="C" value="21"/>
```

or

```
<temperature>
  <scale>C</scale>
  <value>21</value>
</temperature>
```

since the underlying abstractions being represented are the same. We choose which representations of our data to use, JSON, CSV, XML, or whatever, depending on habit, convenience, or the context we want to use that data in.

On the other hand, having an interoperable *generic* toolchain such as that provided by XML to process data is of immense value. How do we resolve the con-

flicting requirements of convenience, habit, and context, and still enable a generic toolchain?

2. Invisible XML

Invisible XML [ixml] is a method for treating non-XML documents as if they were XML, enabling authors to write documents and data in a format they prefer while providing XML for processes that are more effective with XML content.

The essence of Invisible XML is based on the observation that, looked at in the right way, an XML document is no more than the parse tree of some external form, so that all that is needed is to parse the external form using some general-purpose parsing algorithm, and then serialise the resulting parse-tree as XML.

To take a very simple example, imagine a grammar for a very simple expression language that allows such expressions as:

$a \times (3+b)$

The grammar could look like this:

```
expression: ^expr.
expr: term; ^sum; ^diff.
sum: expr, "+", term.
diff: expr, "-", term.
term: factor; ^prod; ^div.
prod: term, "*", factor.
div: term, "\div", factor.
factor: ^letter; ^digit; "(", expr, ")".
letter: ^["a"-"z"].
digit: ^["0"-"9"].
```

The format used is a 1-level van Wingaarden grammar [vwf], a variant of BNF [bnf]. Each rule consists of a non-terminal to be defined, followed by a colon, and a *definition* followed by a full-stop. A definition consists of a number of *alternatives* separated by semicolons. Each alternative consists of a list of *non-terminals* and *terminals* separated by commas. A terminal is enclosed in quotes. An alternative, as a shorthand, may also consist of a range of characters enclosed in square brackets.

The only thing that needs to be explained here is the use of the "^" symbol, which marks non-terminals in the parse tree that are required to show up in the final XML serialisation. To illustrate, if we parse the example expression " $a \times (3+b)$ " with this grammar we would get the following parse tree:

```
^expr
|  term
|  |  ^prod
|  |  |  term
|  |  |  |  factor
```

```

|   |   |   |   |   ^letter
|   |   |   |   |   |   ^"a"
|   |   |   |   |   |   "x"
|   |   |   |   |   |   factor
|   |   |   |   |   |   |   "("
|   |   |   |   |   |   |   |   expr
|   |   |   |   |   |   |   |   |   ^sum
|   |   |   |   |   |   |   |   |   |   expr
|   |   |   |   |   |   |   |   |   |   |   term
|   |   |   |   |   |   |   |   |   |   |   |   factor
|   |   |   |   |   |   |   |   |   |   |   |   |   ^digit
|   |   |   |   |   |   |   |   |   |   |   |   |   |   ^"3"
|   |   |   |   |   |   |   |   |   |   |   |   |   |   "+"
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   term
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   factor
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   ^letter
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   ^"b"
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   ")"

```

Serialising this as XML, retaining all nodes, would give the following XML instance:

```

<expr>
  <term>
    <prod>
      <term>
        <factor>
          <letter>a</letter>
        </factor>
      </term>
      x
    </prod>
    <factor>
      (
        <expr>
          <sum>
            <expr>
              <term>
                <factor>
                  <digit>3</digit>
                </factor>
              </term>
            </expr>
            +
          </sum>
        </expr>
      </factor>
    </prod>
  </term>
  <factor>
    <letter>b</letter>
  </factor>
</expr>

```

```
        </term>
      </sum>
    </expr>
  )
</factor>
</prod>
</term>
</expr>
```

However, serialising it retaining only the marked nodes gives us the following XML:

```
<expr>
  <prod>
    <letter>a</letter>
    <sum>
      <digit>3</digit>
      <letter>b</letter>
    </sum>
  </prod>
</expr>
```

3. Serialisation

Since in general the input form and the generated XML are isomorphic, returning the generated XML to its original format is just a process of serialisation, nothing that a suitable bit of XSLT couldn't do, or even CSS in some simple cases.

For instance, to take an example from the original paper, where a piece of CSS

```
body {color: blue; font-weight: bold}
```

is parsed into XML 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>
```

Rejoicing in the possibility of formatting CSS with CSS, the following simple bit of CSS would return the XML back into regular CSS format:

```
block::before {content: "{"}
block::after  {content: "}"}
name::after  {content: ":"}
property::after {content: ";"}
```

The original paper also shows how to produce an alternative XML serialisation of the CSS snippet using attributes:

```
<css>
  <rule>
    <selector>body</selector>
    <block>
      <property name="color" value="blue"/>
      <property name="font-weight" value="bold"/>
    </block>
  </rule>
</css>
```

which could be round-tripped with the following piece of CSS:

```
block::before {content: "{"}
block::after  {content: "}"}
property::before {content: attr(name) ":" attr(value) ";"}
```

However, considering the XML above for the expression, it is harder (a combinatorial problem) to round-trip the XML using CSS because of the loss of context caused by eliding intermediate nodes like `term`, `factor` and `expr`. For instance, if `<sum>` is a direct child of `<prod>`, then it must have been enclosed in brackets in the original expression, and therefore the serialisation must include brackets around such `<sum>s`, but you can only infer it, and it is impossible to infer if the original had *two* pairs of brackets around it.

An alternative option to such inference is to regard the grammar of a format as a specification of a presentation language for the parse-tree of that format, and write a suitable program that walks the tree hand-in-hand with the grammar.

4. Serialisation by Tree Walking

If you have the parse tree that was used to generate the XML serialization, then serialising it back to its original form is trivially easy: the parse tree is traversed depth first, and each time a terminal symbol is reached, it is copied to the output:

```
serialise(t)=
  for node in children(t):
    select:
      terminal(node):
```

```
output (node)
nonterminal (node) :
    serialise (node)
```

However, in the general case you will not have the original parse-tree, and so life is harder. Because of the lack of context referred to earlier, caused by the elision of intermediate nodes in the parse tree, you essentially have to recreate the parse-tree. This can be done by 'parsing' the XML serialisation using the original grammar.

5. Earley Parsing

In the literature, the Earley parsing algorithm [earley] is often referred to as a "state chart" parsing algorithm [aho]. However, from a modern computing perspective, it is more useful to see it as a serialised parallel parsing processor.

Each rule, such as

```
sum: expr, "+", term.
```

represents, in Unix terms, a process. The right hand side is a series of 'instructions' for matching the input, starting at the current position. These are executed sequentially.

However, if a right hand side has several alternatives (separated by ";" in xml grammars), such as

```
factor: ^letter; ^digit; "(", ^expr, ")".
```

then the processed is 'forked' (again in Unix terminology) to produce a sub-process for each alternative, each processing from the same start position.

The processes are put in a queue, ordered on the position in the input they are parsing from. All processes for position n are run before processes for position $n+1$ (not essential, but reduces the need for keeping the whole input around during processing).

If a process successfully matches an input symbol, it is paused and added to the queue for position $n+1$.

If a process reaches the end of its 'instructions', it *succeeds* (terminates successfully, and returns to its parent rule).

If a process meets an input symbol it wasn't expecting, it *fails* (terminates unsuccessfully, and returns to its parent rule).

For a rule with more than one alternative, if one or more succeeds, the rule itself succeeds, otherwise it fails.

More than one alternative can succeed if the grammar is ambiguous. For instance, with the simple grammar:

```
div: "i"; div, "÷", div.
```

the string

$i \div i \div i$

can be parsed in two ways, essentially either as

$\text{div}(i, \text{div}(i, i))$

or

$\text{div}(\text{div}(i, i), i)$

or in other words, either as

$i \div (i \div i)$

or as

$(i \div i) \div i.$

The whole process ends when all the sub-processes have terminated; if the top-level process succeeds, then you have successfully parsed the input, and otherwise not.

There is one other issue: if a rule has already been queued for a particular position, it is not added a second time, instead being linked to the already-queued version.

6. Parsing a Parse Tree

Parsing a parse tree is a similar procedure. The top level rule must be matched against the XML tree. A 'marked' terminal in the grammar must be present in the XML, as must a marked nonterminal, which is then further treated as a nonterminal in the original algorithm. A non-marked terminal is assumed to be present. Finally an unmarked nonterminal is treated the same as any nonterminal in the original algorithm.

This parsing will produce a parsetree that can then be used for serialisation as described above.

The only thing to note is that parsing the parsetree can also produce an ambiguous result. For example, suppose an expression grammar allowed the use of several sorts of brackets, where the brackets had no separate semantic meaning, so that as well as

$a \times ((b+1) \times (c+1))$

you could also write

$a \times (\{b+1\} \times \{c+1\})$

with the following grammar fragment:

`factor: ^letter; ^digit; "(", expr, ")"; "{", expr, "}".`

Since the brackets do not appear in the final serialised parsetree, there is no way to tell from it if an original bracketed expression had been

(b+1)

or

{b+1}

since they both produce the identical serialisation. This implies that while round-tripping will be semantically identical, it won't necessarily be character-by-character identical. If this is not wanted, then to overcome it, effective information about the elided characters *has* to appear in the serialisation. For instance by using rules like:

```
factor: ^letter; ^digit; ^pexpr; ^bexpr.  
pexpr: "(", expr, ")".  
bexpr: "{", expr, "}".
```

7. Representation Neutrality

A major consequence of Invisible XML is that the external representation of any format is relatively unimportant: it is the data represented that matters, and in particular the resulting parse-tree. This means from the point of view of IXML that any external representation of a format is equivalent, as long as it has the same parse tree.

Take for instance the syntax of an ixml grammar, a part of which looks like this:

```
ixml: (^rule)+.  
rule: @name, colon, definition, stop.  
definition: (^alternative)+semicolon.  
alternative: (term)*comma.  
term: symbol; repetition.  
...  
name: (letter)+.  
colon: ":".
```

As long as the resulting serialised parsetree is the same, we could easily choose another format for the grammars. For instance:

```
<ixml> ::= (^<rule>)+  
<rule> ::= @<name> <define-symbol> <definition>  
<definition> ::= (^<alternative>)+<bar>  
<alternative> ::= (<term>)*  
<term> ::= <symbol> | <repetition>  
...  
<name> ::= "<" (<letter>)+ ">"  
<define-symbol> ::= "::~"  
<bar> ::= "|"
```

(Note that these two grammar fragments, both describe *and* use the format described).

The only repercussion this has on Invisible XML is during the delivery, we not only have to say what the syntax is of the document that we are parsing, but also what syntax of that syntax is, if it is not the standard one. So for instance the mediatype could look like:

```
application/xml-invisible; syntax=http://example.com/syntax/css; ►  
in=http://example.com/syntax/invisible-xml-alt
```

8. Normalising Grammars

So what is the resulting parse tree of a particular ixml grammar? The way to find out is to process the grammar in the following way. For each symbol in every rule:

1. if it is an implicit terminal delete it
2. if it is a refinement, replace it with the definition of that refinement enclosed with brackets, unless this refinement is already a part of it (i.e. the refinement is recursive).

and then delete all rules that are no longer used.

So for example, for the expressions grammar, we would end with:

```
expr: (^letter; ^digit; ^prod; ^div; ^sum; ^diff).  
sum:  (^letter; ^digit; ^prod; ^div; ^sum; ^diff),  
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).  
diff: (^letter; ^digit; ^prod; ^div; ^sum; ^diff),  
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).  
prod: (^letter; ^digit; ^prod; ^div; ^sum; ^diff),  
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).  
div:  (^letter; ^digit; ^prod; ^div; ^sum; ^diff),  
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).  
letter: ^["a"-"z"].  
digit:  ^["0"-"9"].
```

which has eliminated all refinements, and all non-productive terminal symbols.

This resulting parse-tree definition is essentially a definition of the data-structure for the internal representation of the document, in other words a form of schema.

As another example, take this fragment of the ixml grammar for itself:

```
ixml: (^rule)+.  
rule: @name, colon, definition, stop.  
definition: (^alternative)+semicolon.  
alternative: (term)*comma.  
term: symbol; repetition.  
symbol: terminal; ^nonterminal; ^refinement.
```

```
terminal: ^explicit-terminal; ^implicit-terminal.repetition: ^one-or-more; ^zero-or-more.
```

The first rule:

```
ixml: (^rule)+.
```

doesn't change.

```
rule: @name, colon, definition, stop.
```

becomes:

```
rule: @name, ":", (^alternative)+semicolon , ".".
```

by substituting all the definitions for the refinements. This is then processed again to give:

```
rule: @name, ":", (^alternative)+";" , ".".
```

and finally the (unmarked) terminals are deleted:

```
rule: @name, (^alternative)+.
```

The rule

```
definition: (^alternative)+semicolon.
```

becomes by a similar process:

```
definition: (^alternative)+.
```

(but will be later deleted, as it is no longer used).

The rule

```
alternative: (term)*comma.
```

becomes by a similar process

```
alternative: (symbol; repetition)*.
```

which then can be processed again to give

```
alternative: (terminal; ^nonterminal; ^refinement; ^one-or-more; ^zero-or-more)*.
```

and then one final time to give

```
alternative: (^explicit-terminal; ^implicit-terminal; ^nonterminal; ►  
^refinement; ^one-or-more; ^zero-or-more)*.
```

and so on.

So our final parse-tree description for this grammar fragment is:

```
ixml: (^rule)+.
```

```
rule: @name, (^alternative)+.
```

```
alternative: (^explicit-terminal; ^implicit-terminal; ^nonterminal; ►  
^refinement; ^one-or-more; ^zero-or-more)*.
```

```
one-or-more: (^alternative)+; (^alternative)+, ^separator.
zero-or-more: (^alternative)+; (^alternative)+, ^separator.
separator: ^explicit-terminal; ^implicit-terminal; @nonterminal; ►
@refinement.
symbol: ^explicit-terminal; ^implicit-terminal; ^nonterminal; ►
^refinement.
terminal: ^explicit-terminal; ^implicit-terminal.
explicit-terminal: @string.
implicit-terminal: @string.
nonterminal: @name.
refinement: @name.
attribute: @name.
```

From a data-structuring point of view, these are type definitions, semicolons representing unions, commas representing structs, and repetitions representing lists.

9. Subsets

A corollary of the observation above that any external representation of a format is equivalent, as long as it has the same parse tree, is that if a format has a normalised grammar that is a subset of another normalised grammar, and the same root node, then the first language is compatible with the second (but not the other way round).

10. Data Conversion

Since external representation is no longer important, it would be easy to transform one format to another, as long as their normalised grammars are compatible. So transforming an ixml grammar to one in a different representation is as simple as parsing it with one grammar and serialising it with another.

11. Conclusion

XML has provided us with a standard data-representation layer, and a standard processing pipeline. With a relatively small addition we can open up the pipeline to all structured documents, making XML truly ubiquitous.

References

- [1] Pemberton, Steven. "Invisible XML." Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). doi:10.4242/BalisageVol10.Pemberton01.

- [2] Backus-Naur Form, http://en.wikipedia.org/wiki/Backus-Naur_Form
- [3] S. Pemberton, 1982, Executable Semantic Definition of Programming Languages Using Two-level Grammars, [http://www.cwi.nl/~steven/vw.html\[vw\]](http://www.cwi.nl/~steven/vw.html[vw])
- [4] https://en.wikipedia.org/wiki/Earley_parser
- [5] Aho, AV, and Ullman, JD, "The Theory of Parsing, Translation, and Compiling", Prentice-Hall, 1972, ISBN 0139145567

From XML to RDF step by step: Approaches for Leveraging XML Workflows with Linked Data

Marta Borriello

Vistatec

<marta.borriello@vistatec.com>

Christian Dirschl

Wolters Kluwer Germany

<cdirschl@wolterskluwer.de>

Axel Polleres

<axel.polleres@wu.ac.at>

Phil Ritchie

<philr@vistatec.ie>

Frank Salliau

<frank.salliau@ugent.be>

Felix Sasaki

<felix.sasaki@dfki.de>

Giannis Stoitsis

<stoitsis@agroknow.com>

1. Introduction

1.1. Motivation

There have been many discussions about benefits and drawbacks of XML vs. RDF. In practice more and more XML and linked data technologies are being used together. This leads to opportunities and uncertainties: for years companies have invested heavily in XML workflows. They are not willing to throw them away for the benefits of linked data.

In XML workflows XML content is

- Generated, from scratch or based on existing content;
- processed, e.g.: validated, queried, transformed; and
- stored in various forms, e.g.: as XML, in a different format (e.g. PDF / HTML output); and

- potentially input to other XML or non-XML workflows.

Each part of the workflow may include huge amounts of XML data. This can be XML files themselves, but also additional related items like: XSLT or XSL-FO stylesheets for transformation or printing, XQuery based queries, or XML Schema / DTD / Relax NG schemata for validation etc.

For many potential users of linked data, giving up these workflows is not an option. Also, changing even a small part of the workflow may lead to high costs. Imagine one element `linkedDataStorage` added to an imaginary XML document:

Example 1. XML document with linked data embedded in an element

```
<myData>
  <head>...</head>
  <body>
    <linkedDataStorage>...</linkedDataStorage> ...
  </body>
</myData>
```

Adding this element to an XML element may break various aspects of the workflow, like:

- Validation: the underlying schema does not understand the new element.
- Transformation: a transformation may expect a certain element as the first child element of body. If `linkedDataStorage` is the first child, the transformation would fail.

One may argue that good XML schema will leave space for expansion using lax validated parts or by accepting attributes from other namespaces, for example. Nevertheless, in practice we have to work with a lot of existing XML Schemas, related tooling and workflows. So creating extension points and deploying lax validation may not be an option in real life.

Whereas the strict validation against schemas on the one hand is in the XML world often seen as a feature of the toolchain, on the other hand, such adding of elements and schemaless integration of different (parts of) datasets is actually one of the main “selling points of RDF”. However, note that on the contrary, even in the RDF world, users are starting to demand tools for stricter schema validation, which has recently lead to the foundation of a respective working group around RDF Data Shapes in W3C.²² So, overall there seems to be lots to learn for both sides from each other.

This paper wants to help with XML and RDF integration to foster incremental adoption of linked data, without the need to get rid of existing XML workflows.

²²See <http://www.w3.org/2014/data-shapes/>.

We are discussing various integration approaches. They all have benefits and drawbacks. The reader needs to be careful in deciding which approach to choose.

1.2. The Relation to RDF Chimera

In her keynote at XML Prague 2012 and a subsequent blog post, Jeni Tennison discussed RDF chimera²³. She is arguing that for representing RDF, syntaxes like RDF/XML or JSON or JSON-LD should be seen as a means to achieve something - a road, but not a destination. An example is a query to an RDF data store, and the outcome is represented in an HTML page.

The goal of our paper is different. We assume that there is existing content that benefits from *data integration* with linked data - without turning the content into a linked data model. Let's look at an example: imagine we have the sentence Berlin is the capital of Germany!. There are many linked data sources like DBpedia²⁴ that contain information about Berlin; it would add an enormous value to existing content (or content creation workflows) if such information could be taken into account. This does not mean - like in the case of RDF chimera - to give up the XML based workflows, but to provide means for the data integration. In this view we can see the linked data process as a type of enrichment, hence we call the process *enriching XML content* with linked data based information.

1.3. Background: The FREME Project

FREME²⁵ is an European project funded under the H2020 Framework Programme. FREME is providing a set of interfaces (APIs and GUIs) for multilingual and semantic enrichment of digital content. The project started in February 2015, will last for two years and encompasses eight partners. The partners provide technology from the realm of language and data processing, business cases from various domains, and expertise in business modeling. This expertise is of specific importance since both language and linked data technologies are not yet widely adopted. The challenge of XML re-engineering for the sake of linked data processing is one hindrance that needs to be overcome to achieve more adoption.

FREME provides six e-Services for processing of digital content:

- e-Internationalisation based on the Internationalisation Tag Set (ITS) Version 2.0.
- e-Link based on the Natural Language Processing Interchange Format (NIF) and linked (open) data sources.
- e-Entity based on entity recognition software and existing linked entity datasets.

²³ <http://www.jenitennison.com/2012/06/30/rdf-chimera.html>

²⁴ <http://dbpedia.org/about>

²⁵ See the FREME project homepage at <http://freme-project.eu/> for more information.

- e-Terminology based on cloud terminology services for terminology management and terminology annotation web service to annotate terminology in ITS 2.0 enriched content.
- e-Translation based on cloud machine translation services for building custom machine translation systems.
- e-Publishing based on cloud content authoring environment (for example e-books, technical documentation, marketing materials etc.) and its export for publishing in Electronic Publication (EPUB3) format.

This paper will not provide details about the services - examples and more information on FREMOTE can be found at <http://api.freme-project.eu/doc/current/>

All e-services have in common that XML content is a potential input and output format: via FREMOTE, XML content can be enriched with additional information, to add value to the content. But FREMOTE is only one example: many other linked data projects involve companies working with linked data content.

2. Business Case Motivation Examples

2.1. The Case of Agro-Know and Wolters Kluwer - Linked Data in XML Publishing Workflows

Agro-Know is data oriented company that helps organisations to manage, organise and open their agricultural and food information. One of the main activities of Agro-Know is the aggregation of bibliographic references from diverse sources to support online search services like AGRIS²⁶ of the Food and Agricultural Organisation of the United Nations. Agro-Know is doing so by aggregating metadata records from data providers such as journals, small publishers, universities, research centers, libraries and national aggregators. The metadata aggregation workflow of Agro-Know includes parts for metadata analysis, harvesting, filtering, transformation, enrichment, indexing and publishing. The main goal of applying the different steps of the workflow is to end up with a well formatted and complete metadata record that is compatible to the metadata standard for agricultural sciences, namely AGRIS AP²⁷. The majority of the metadata records that are collected are in XML following several metadata formats such as DC, AGRIS AP, DOAJ, MODS, MARC 21 etc. The processed metadata records are published in AGRIS AP, JSON and RDF.

Within such metadata aggregation workflow, Agro-Know is facing several challenges related to the enrichment of the metadata records. One such example is non-structured information about authors that in many cases is not following

²⁶ <http://agris.fao.org/>

²⁷ <http://www.fao.org/docrep/008/ae909e/ae909e00.HTM>

an authority file and includes additional information in the same XML tag like affiliation, email and location. This information cannot be automatically transformed to structured information and remaining in the tag, it reduces the quality of provided filtering options in the search functionality of the online service. In addition to that, since an authority file is not used on the data provider side, this results in an ambiguity problem as the same author may appear with many different variations of the name. Another problem is the absence of subject terms in the metadata records from a multilingual vocabulary such as AGROVOC²⁸, that consists of more than 32.000 terms available in 23 languages. Including AGROVOC terms in the metadata records can semantically enhance the information and can enable better discovering services at the front end application.

To solve such problems, Agro-Know is using the FREME e-services in order to improve a) the online services that are offered to the end users and b) the semantics of the metadata records that is provided to other stakeholders of this data value chain, such as publishers. The main goal will be to add the structured information in the XML records by keeping the level of intervention at a minimum level in order to eliminate the revisions required in the existing workflows. Examples of how an XML part referring to authors can be enriched using FREME e-services is presented in the table below. In this case, including the ORCID²⁹ identifier may help in disambiguation but also in the enrichment of the information as we can show to the end user of the online service additional valuable information directly retrieved from ORCID.

Before FREME	Result of deploying FREME
<pre><dc:creator> <ags:creatorPersonal> Stoitsis, Giannis, Agroknow </ags:creatorPersonal> </dc:creator></pre>	<pre><dc:creator> <ags:creatorPersonal>Stoitsis, Giannis</ags:creatorPersonal> <nameIdentifier schemeURI= "http://orcid.org/" nameIdentifierScheme= "ORCID">0000-0003-3347-8265 </nameIdentifier> <affiliation>Agroknow</affiliation> </dc:creator></pre>

²⁸ <http://aims.fao.org/vest-registry/vocabularies/agrovoc-multilingual-agricultural-thesaurus>

²⁹ <http://orcid.org/>

<pre><dc:subject> <ags:subjectClassification scheme="ags:ASC"> <![CDATA[J10]]> </ags:subjectClassification> </dc:subject></pre>	<pre><dc:subject freme-enrichment= "http://aims.fao.org/aos/agrovoc/c_426 http://aims.fao.org/aos/agrovoc/c_24135 http://aims.fao.org/aos/agrovoc/c_4644 http://aims.fao.org/aos/agrovoc/c_7178"> <ags:subjectClassification scheme= "ags:ASC"><![CDATA[J10]]> </ags:subjectClassification> </dc:subject></pre>
---	--

Wolters Kluwer is a global information service provider with businesses mainly in the legal, tax, health and financial market. The fundamental transformation process from folio to digital and service offerings that is currently disrupting the whole industry requires also more efficient and streamlined production processes. In the last ten years, the industry has very much focused on XML based publishing workflows, where all relevant information resides as metadata within the documents, which are structured according to proprietary DTDs or XML schemas. The industry is slowly starting to adapt linked data principles, because they offer the required flexibility, scalability and information interoperability that XML or also relational database models do not offer. As a first step, metadata is extracted from the documents and stored in parallel in graph databases. Already this step requires a major shift in technology as well as business culture, because the focus and added-value moves away from pure content to data and information.

Wolters Kluwer Health is customer of Agro-know and has integrated its XML delivery channel for enriched scientific references mainly in the area of agriculture. Agro-know is offering more and more added value services using linked data principles and in this course reduces the traditional XML-based delivery pipeline step by step in order to stimulate usage of the superior channels. This change causes major challenges at the customer's side. Semantic Web technology and standards are not yet common solutions in publishing. Therefore technical infrastructure as well as skills have to be developed in order to get things even started. This requires a certain transition period, where the old delivery channel remains stable and the customer can prepare the changes.

In such a scenario, Wolters Kluwer recommends that the source provider enables the customer to locally keep his old production workflow in place as long as it is needed. This could be achieved e.g. by making the conversion script available as open source. In addition, a proper documentation about the differences from old to new is also vital for success. Ideally, a direct communication flow between vendor and customer would help to lower concerns and accelerate uptake of the new process.

2.2. The Case of Vistatec - Linked Data in XML Localization Workflows

Vistatec is a leading provider of digital content translation and locale adaptation services for international businesses. These businesses use a variety of Vistatec multilingual services to publish a range of content types including: corporate communications; marketing collateral; e-commerce catalogues; and product, travel destination, and leisure experience descriptions.

Vistatec's production processes are highly automated and based on XML standards for systems integration, process component interoperability and the capture and use of metadata.

The localization process, content types and end consumers of the content all benefit greatly from FREME semantic enrichment and entity discovery and linking e-services.

The successful adoption of technology hinges upon the ease of use. Vistatec has adapted its open source XLIFF³⁰ editor, Ocelot³¹, to consume FREME e-services in a transparent and optimal way using configurable pipelines.

The table below summarizes the steps of a typical localization workflow and the benefits that can be realized through the use of FREME e-services:

Process Step	FREME e-service	Benefit
Conversion of native document to Extensible Localization Interchange File Format	e-Internationalization	Define translatable portions of the document.
Translation	e-Terminology and e-Entity	These services help linguists to identify and use appropriate translations suitable for the subject domain.
Semantic enrichment	e-Link	Suggest information resources which relate to the subject matter of the content.

³⁰ https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xliff

³¹ http://open.vistatec.com/ocelot/index.php?title=Main_Page

Content publication	e-Pub	Incorporation of markup for entity definitions and hyperlinks to information resources which relate closely to the subject matter of the content.
---------------------	-------	---

A tutorial from the FREME documentation³² shows how XLIFF can be processed via FREME e-Services. We process the following XLIFF element, using the example sentence from a previous section:

```
<source>Berlin is the capital of Germany!</source>
```

The e-Entity service identifies Berlin as a named entity with a certain type and a unique URI. The result of the process is not stored in XML, but as a linked data representation including **offsets** that point to the string content. The URI

```
<http://freme-project.eu/#char=25,32>
```

identifies the entity, offsets and entity related information. A complete linked data representation, using the turtle syntax, looks as follows.

Example 2. Linked data representation using offsets for pointing to existing XML content

```
<http://freme-project.eu/#char=25,32> ...
(1) nif:anchorOf      "Germany"^^xsd:string ;
(2) nif:beginIndex   "25"^^xsd:int ;
(3) nif:endIndex     "32"^^xsd:int ; ...
(4) itsrdf:taClassRef <http://nerd.eurecom.fr/ontology#Location>;
(5) itsrdf:taIdentRef <http://dbpedia.org/resource/Germany>.
```

The linked data statements expressed in above representation are:

- The annotation is (1) anchored in the string Germany.
- The annotation (2) starts at character 25 and (3) ends at character 32.
- The annotation is related to (4) the class URI <http://nerd.eurecom.fr/ontology#Location>.
- The entity is (5) uniquely identified with the URI <http://dbpedia.org/resource/Germany>.

The example should make clear why we are not looking at a data conversion task (like in the discussion on RDF chimera), but at a data integration task. We don't aim at changing the XLIFF source element, but at relating it to the information

³²See <http://api.freme-project.eu/doc/0.4/tutorials/spot-entities-in-xliff.html>

provided via the linked data representations. The data integration approaches discussed in Section 3 are ways to achieve this goal.

2.3. The Case of iMinds - Linked Data in Book Metadata

iMinds, Flanders' digital research hub, conducts research on book publishing in close collaboration with the Flemish publishing industry. An important aspect in book publishing is book metadata. As the volume in published books increases, and as the book industry becomes more and more digital, book metadata also becomes increasingly important: book publishers want their books to be found on retail sites. Correct and rich metadata is a prerequisite for online discoverability.

Within the publishing value chain, stakeholders have since long agreed to use a standard format for book metadata: ONIX for Books³³. This XML-based standard has been adopted worldwide and is used by publishers to communicate book product information in a consistent way in electronic form.

ONIX for Books is developed and maintained by EditEUR³⁴, the international group coordinating development of the standards infrastructure for electronic commerce in the book, e-book and serials sectors. The ONIX for Books standard and corresponding workflow are solidly embedded in the publishing retail chain, from publisher to distributor to retailer. Migrating to other technologies, e.g. using linked data, requires a substantial investment which stakeholders in this industry are very reluctant to make.

We do not recommend to substitute this standard with a fully fledged linked data approach. However, we find there are cases where linked data can be beneficial as enrichment of existing ONIX metadata. The example below shows a possible usage of linked data in ONIX.

Author information as linked data: An ONIX file typically contains information about the author(s) of a book. These are called contributors (other types of contributors are illustrators, translators etc.).

Below you can find a block of metadata with information about the author of a book, Jonathan Franzen. A possible enrichment with linked data might be to insert the link to the authority record about Jonathan Franzen on viaf.org, via the Entity tag. Please note that these tags are not valid within the current ONIX schema and are used here merely as an example of a possible enrichment.

This enrichment may prove useful in several ways:

- disambiguation of the author (using the VIAF identifier); and
- providing a link to more information on the author.

³³See <http://www.editeur.org/83/Overview/>

³⁴See <http://www.editeur.org/>.

Example 3. A potential approach for embedding linked data identifiers into ONIX

```
<Contributor>
  <NameIdentifier>
    <NameIDType>
      <IDTypeName>Meta4Books ContributorID</IDTypeName>
      <IDValue>65097</IDValue>
    </NameIDType>
  </NameIdentifier>
  <ContributorRole>A01</ContributorRole>
  <SequenceNumber>1</SequenceNumber>
  <NamesBeforeKey>Jonathan</NamesBeforeKey>
  <KeyNames>Franzen</KeyNames>
  <Entity>
    <URI>http://viaf.org/viaf/84489381/</URI>
  </Entity>
</Contributor>
```

3. Approaches for Linked Data Integration in XML Workflows

The following approaches are a non exhaustive list. The aim is to provide examples that are currently in use and show their benefits and drawbacks. The examples are anchored in the business cases described in Section 2. The structure of the approach description is always as follows:

- Name the example;
- Explain what actually happens with some XML code snippets; and
- Explain drawbacks and benefits, both from a linked data and an XML processing point of view.

3.1. Approach 1: Convert XML to Linked Data

What actually happens: XML is converted into linked data. The XML content itself is not touched, but an additional set of data, i.e. a linked data representation is created.

```
<xs:element name="lingualityType">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="monolingual"/>
      <xs:enumeration value="bilingual"/>
      <xs:enumeration value="multilingual"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

During the mapping of XML to linked data several decisions have to be taken. Sometimes information is being lost. A real example is the mapping of META-SHARE to linked data³⁵. META-SHARE provides metadata for describing linguistic resources.

Using this element could look like this, expressing that a resource is multilingual, e.g. a multilingual dictionary:

```
<lingualityType>multilingual</lingualityType>
```

A linked data model that represents this data type could look as follows:

Example 4. Linked data model that represents parts of the META-SHARE schema

```
### Class
ms:linguality
  rdf:type owl:ObjectProperty ;
  rdfs:domain ms:Resource ;
  rdfs:range ms:Linguality .
### Property
ms:Linguality
  rdf:type owl:Class .
#### Instances
ms:monolingual a ms:Linguality.
ms:bilingual a ms:Linguality.
ms:multilingual a ms:Linguality.
```

This statement expresses the same information like in the XML representation: a given resource, e.g. a dictionary, is multilingual.

What are the benefits: The benefit of this approach is that existing XML workflows don't need to be changed at all. The linked data representation is just an additional publication channel for the information. In this sense, the approach is similar to the RDF chimera discussion. It is however still different, since the conversion from XML to RDF involves RDF focused data modeling and aims at integration with further linked data sources.

The additional RDF representation can be integrated with other linked resources without influencing the XML workflow. This is what actually is being done with the META-SHARE case: via the LingHub portal³⁶, META-SHARE and other types of metadata for linguistic resources is converted to RDF and being made available. A user then can process integrated, multiple linked data sources without even knowing that there is an XML representation available.

³⁵See http://www.lrec-conf.org/proceedings/lrec2014/pdf/664_Paper.pdf for more details on the META-SHARE case and on how the conversion from XML to linked data was achieved.

³⁶See <http://linghub.lider-project.eu/>.

What are the drawbacks: The approach requires a completely new tool chain, aiming at linked data integration based on XML (or other format based) data sources. The existing setup, e.g. used to analyze the XML structures with XQuery or to transform it via XSLT, cannot be used. A query like “Give me all linguistic resources that are multilingual” can be executed via XPath easily in the XML representation. In standard XQuery there is no bridge to SPARQL. However, work has been done to create this bridge, see Section 5.

Another drawback of this approach is that it is not always possible to convert XML completely into RDF - in particular, RDF has no facility for representing mixed content, an essential part of processing textual, human language content. A takeaway of Section 1.2 is that data should always be in the format that best fits its representation, and URIs can serve as a bridge between formats. The usage of URIs for bridging between RDF and XML is discussed in Section 3.5.

3.2. Approach 2: Embedd Linked Data into XML via Structured Markup

What actually happens: linked data is embedded into HTML. Various syntaxes can be used, e.g. JSON-LD, RDFa, or microdata. This approach is deployed e.g. in schema.org³⁷; see the schema.org homepage for various markup examples.

We take again the sentence Berlin is the capital of Germany from a previous section. The integration of information from Wikipedia with the string Berlin can be achieved as follows:

```
<a itemscope itemtype="http://schema.org/Place" itemprop="url"
href="https://en.wikipedia.org/wiki/Berlin">Berlin</a>
```

With this approach search engines will interpret the link `https://en.wikipedia.org/wiki/Berlin` as a machine readable link to integrate additional information about the place Berlin, taken from Wikipedia as a data source. The item being the source of data integration is identified as being of type place via the URI `http://schema.org/Place`.

What are the benefits: the approach works well in situation in which the hooks for data integration can be embedded into XML or HTML content. The search engine optimization scenario via schema.org is the prototypical case for doing this. The embedding may also be done in a dedicated markup part for metadata using the JSON-LD or other linked data syntaxes, without changing the text content; see for details Section 3.4.

What are the drawbacks: RDFa and Microdata changes the content and includes new markup. That may not be an option for XML tool chains that don't “understand” the new markup, e.g. lead to validation errors. JSON-LD or turtle may have the same issues: where should a tool store this data in the XML structure if no general metadata location is available?

³⁷ <http://schema.org/>

3.3. Approach 3: Anchor Linked Data in XML Attributes

What actually happens: an identifier is embedded in the XML structure. This identifier serves as a bridge between the XML and RDF structures. The below example uses the `its:taClassRef` attribute to store the identifier.

```
<source ...>
  <mrk ...its:taIdentRef="http://dbpedia.org/resource/Berlin">
    Berlin</mrk> is the capital of Germany!</source>
```

The data integration task, i.e. fetching additional information from linked data sources about Berlin, can be executed relying on this information. The outcome may then be stored directly in the XML source. Below we assume that the population was fetched using the DBpedia information.

```
<source ...>
  <mrk ...its:taIdentRef="http://dbpedia.org/resource/Berlin">
    Berlin</mrk> (population: 3517424) ...</source>
```

For different purposes, separate linked data queries could be set up. They rely on the same identifier `http://dbpedia.org/resource/Berlin`.

What are the benefits: using an XML attribute that is already available in the format in question means that no new types of markup is needed. That is, existing XML toolchains can stay as is, including validation or transformation processes.

What are the drawbacks: the data integration is postponed. The completed integration, if needed, needs to choose one of the other approaches discussed in this paper. Also, the data integration does not leave a trace. Further processing steps in the (XML) toolchain cannot identify that the string (`population: 3517424`) is a result of a data integration process.

3.4. Approach 4: Embed Linked Data in Metadata Sections of XML Files

What actually happens: many XML vocabularies have metadata sections that may contain arbitrary content. This is also true for XLIFF discussed in Section 2.2. The outcome of the linked data processing could be stored in such a section.

What are the benefits: Compared to Section 3.2, the size of the content itself is not growing with additional, linked data related markup.

What are the drawbacks: There is no per se relation to the content. Like in Example 2, one may create pointers to the XML content, here using character offsets. But the pointers may be fragile, if one thinks e.g. of reformatting, insertion or deletion of content or markup. In addition, some linked data syntaxes may interfere with XML validation or well formedness constraints.

3.5. Approach 5: Anchor Linked Data via Annotations in XML Content

What actually happens: A generalized approach of Section 3.3 means that linked data is stored separately from XML structures and that there is a reference from linked data to the XML content in question. In Section 2.2, we were using character offsets. The W3C Web Annotation Data Model³⁸ allows to realize such anchoring. Character offsets are just one way of anchoring the annotation. One can also use XPath expressions, see the following example.

Example 5. Anchoring annotations in XML via the Web annotation data model

```
{ "id": "http://example.com/myannotations/a1",
  "type": "Annotation",
  "target": { "type": "SpecificResource",
    "source": "http://example.com/myfile.xml",
    "selector": { "type": "FragmentSelector",
      "conformsTo": "http://www.w3.org/TR/xpath/",
      "value": "/xlf:unit[1]/xlf:segment[1]/xlf:source/xlf:mrk[1]" },
    "itsrdf:taIdentRef": "http://dbpedia.org/resource/Berlin",
    "itsrdf:taClassRef": "http://schema.org/Place",
    "http://dbpedia.org/property/population" : "3517424" } }
```

The XPath expression in above linked data representation (which uses the JSON-LD syntax) selects the XLIFF mrk element from the example in Section 3.3.

What are the benefits: In addition to the approach 3, here we are able to add the linked data information in the separate annotation, e.g. the population of Berlin; there is no need to change the content itself. If needed for certain applications, we can use this annotation approach to generate others. URIs pointing to the content are an important aspect of such format conversions. The beforehand mentioned ITS 2.0 specification shows an example of 1) generating linked data annotations anchored in XML content³⁹, and 2) integrating the separate annotations into the markup content⁴⁰. The beforehand described F2F framework deploys this approach in its ITS enabled e-Internationalisation⁴¹.

What are the drawbacks: the resolution of linked data information potentially can be computationally expensive, see e.g. lot's of XPath expressions to compute for annotations. Also, if the source content changes, the anchoring mechanism may not work anymore. Some mechanisms are more robust (e.g. XPath expressions), some may be more precise (e.g. the character offset based anchoring).

³⁸ <http://www.w3.org/TR/2015/WD-annotation-model-20151015/>

³⁹ <http://www.w3.org/TR/its20/#conversion-to-nif>

⁴⁰ <https://www.w3.org/TR/its20/#nif-backconversion>

⁴¹ <http://api.f2f-project.eu/doc/0.4/knowledge-base/eInternationalization.html>

4. Relating Business Cases and Integration Approaches

The following table relates the three business cases and the various integration approaches.

Business case	Integration approaches being considered	Actual current or experimental praxis
Linked data in XML publishing workflows	Approach 1: convert XML into linked data	XML workflow kept, linked data conversion scripts to be made available
Linked data in XML localization workflows	Approach 3: anchor linked data in XML attributes; Approach 4: embed linked data in metadata sections of XML files	No established practice in localisation industry
Linked data in book metadata	Approach 4: embed linked data in metadata sections of XML files	No established practice in localisation industry

It becomes obvious that industries take different approaches towards linked data integration. This can be explained with availability of native linked data tooling, knowledge about its usage, and complexity and potential costs of adapting existing XML workflows.

5. Routes to bridge between RDF and XML

As for **Approach 1** (converting XML into linked data), in fact existing XML transformation tools like XSLT and XQuery could be used out of the box with the caveat that the result is mostly tied to the RDF/XML representation, that has various disadvantages, foremost verbosity. More "modern" RDF serializations like Turtle or JSON-LD cannot be created out of the box by XML tools straightforwardly, plus additional filtering of querying on the resulting RDF triples needs to be encoded directly into the XML toolchain, which might be easier solvable in the RDF world itself, e.g. using SPARQL. Likewise, as for **Approach 2**, we have already identified, that the XML toolchain is not tailored to process and consume RDF or similar meta-data formats natively. We face the same problem in **Approaches 3-5**, where RDF-like sources are just linked out of XML content, without the necessary toolchain tightly coupled to XML tools that could process the RDF content natively.

So, there is certainly a gap to bridge here in terms of tooling, but recently, that partially seems to change: there are academic approaches to encode SPARQL into XML processors, such as encoding SPARQL to XSLT or XQuery, cf. e.g. Fischer et al. (2011) and Groppe et al. (2008). Plus there are actually some XML processors like SAXON start supporting SPARQL natively, cf. <https://developer.marklogic.com/learn/semantics-exercises/sparql-and-xquery>.

Alternatively, given that SPARQL actually can produce XML or JSON (among others) as output format⁴², it is possible to directly consume the results of SPARQL queries in XML tools, however more complex use cases need some scripting around this, plus intermediate results for an overall transformation need to be stored and processed separately, ending up in a heterogeneous tool-chain, comprising XML tools, SPARQL processors and potentially even another scripting language on top.

Additionally, there are new “hybrid” but integrated toolchains arising that try to combine the two worlds of XML and RDF in a “best-of-both-worlds” approach: most prominently, we’d like to mention as an example here the XSPARQL project⁴³, that aims at integrating SPARQL into XQuery in a compilation approach: that is, queries on RDF data or to a remote SPARQL endpoint serving Linked Data can be embedded into an XQuery. For instance, the following query transforms geographic RDF data queried from the file <http://nunolopes.org/foaf.rdf> into a KML file:

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

<kml xmlns="http://www.opengis.net/kml/2.2">{
  for $name $long $lat from <http://nunolopes.org/foaf.rdf>
  where { $person a foaf:Person; foaf:name $name;
    foaf:based_near [ a geo:Point;
    geo:long $long;
    geo:lat $lat ] }
  return <Placemark>
    <name>{fn:concat("Location of ", $name)}</name>
    <Point>
      <coordinates>{fn:concat($long, ",", $lat, ",0")}</coordinates>
    </Point>
  </Placemark> }
</kml>
```

The boldface part of the above query is actually SPARQL syntax embedded into XQuery. An XSPARQL compiler translates this query to a native XQuery that del-

⁴²See <http://www.w3.org/TR/sparql11-overview/#sparql11-results> for details.

⁴³See <http://xsparql.sourceforge.net> for details.

egates these query parts to a native SPARQL query processor. More details on this approach can be found in Bischof et al. (2012).

Note that also the other way, i.e. transforming XML data into RDF is supported in this approach, by allowing SPARQL's CONSTRUCT clauses in the return clause of an XQuery, as shown in the following short example, which transforms XML data from Open Streetmap to RDF:

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
prefix kml: <http://earth.google.com/kml/2.0>
let $loc := "WU Wien" for $place in doc
  (fn:concat("http://nominatim.openstreetmap.org/search?q=",
    fn:encode-for-uri($loc),
    "&format=xml"))
let $geo := fn:tokenize($place//place[1]/@boundingbox, ",")
construct { <http://www.polleres.net/foaf.rdf#me>
foaf:based_near [ geo:lat {$geo[1]}; geo:long {$geo[3]} ] }
```

More recently, XSPARQL has been extended to cover the recent SPARQL1.1 specification, plus support for JSON as input and output format (by internally representing JSON in a canonical XML representation have been added, cf. Dell'Aglio et al. (2014).

While XSPARQL, which has actually started as a W3C member submission in 2009⁴⁴, is just one possible route, the authors believe that joint efforts in standardization bodies to bridge the gaps between RDF and XML in order to enable such transformations and integrated tooling in a standard way should be further pursued.

6. Conclusion

This paper discussed the motivation for integrating RDF and XML. We looked at various business case scenarios that can benefit from this integration. We then discussed several approaches to realize the integration. Finally, we looked into technical solutions that integrate the actual XML and RDF technology stacks.

A reviewer of this paper suggested consider XProc for integrating RDF and XML workflows. XProc 2.0⁴⁵ will have the ability to pass information other than XML data between steps; it would be possible to pass RDF data between XProc steps and have filtering and processing steps for that RDF data. This would allow processing of XML data with XML tools (XSLT, XQuery), while tracking and also processing RDF data with e.g. SPARQL or XSPARQL. This approach sounds promising but has not been explored in this paper, so we leave it to future work.

⁴⁴See <http://www.w3.org/Submission/2009/01/>.

⁴⁵ <https://www.w3.org/TR/xproc20/>

A next steps could be to discuss the integration approaches in a broader community, e.g. in a dedicated forum like a W3C community group. This could also help to move the forehand described XML - RDF standardization work forward. Such standardization has been discussed in the past. It is the hope of the authors of this paper that it brings new insights to this discussion, with the real-life needs from actual applications who more and more are in need of the integration.

7. Acknowledgments

The creation of this paper was supported by the FREME project under the Horizon 2020 Framework Programme of the European Commission, Grant Agreement Number 644771.

Bibliography

- [1] Bischof, S., S. Decker, T. Krennwallner, N. Lopes and A. Polleres. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics (JoDS)*, 1(3):147-185, 2012.
- [2] Dell'Aglio, D., A. Polleres, N. Lopes and S. Bischof. Querying the web of data with XSPARQL 1.1. In *ISWC2014 Developers Workshop*, volume 1268 of *CEUR Workshop Proceedings*. CEUR-WS.org, October 2014.
- [3] Fischer, P., D. Florescu, M. Kaufmann and D. Kossmann D (2011). Translating SPARQL and SQL to XQuery. In: *Proceedings of XML Prague'11*, pp 81–98.
- [4] Groppe S., J. Groppe, V. Linnemann, D. Kukulenz, N. Hoeller, C. Reinke (2008). Embedding SPARQL into XQuery/XSLT. In: *SAC'08*. ACM, New York, pp 2271–2278.

Promises and Parallel XQuery Execution

James Wright

<james.jw@hotmail.com>

1. What is it?

Its a simple library which implements the promise¹ pattern as seen in many other languages and frameworks. Most notably those in the javascript community, such as jQuery, Q.js and even EMCAScript 6.

The pattern resolves around the idea of deferred execution through what is called a *deferred*. When an action is deferred, it returns a function, known as a *promise* that when executed at a later time, will perform and return the results of the work it deferred.

Since the work is deferred and can be executed at an arbitrary time. There is the ability to attach further processing at a later date but prior to actual execution, via callback functions. This may sound confusing and hard to imagine, but I 'promise' the examples that follow will make it clearer. No pun intended.

2. Why?

The main driver behind implementing the promise pattern was to realize *parallel* execution of XQuery code within a single query. The overall benefit is to reduce execution time for longer costlier queries, as well as reduce overall complexity of solutions. Additionally, this pattern is a means for building what I call *durable pipeline logic*. If this sounds enticing, keep reading!

I hope that through sharing my findings as well as illustrating the pattern, other XQuery developers, implementations and even the language itself can begin to introduce robust *async* functionality into their offerings.

2.1. Methods of Parallism

As it stands there a several ways to achieve parallel processing with XQuery. Some systems implement specific functionality with *async* methods, such as *ex-sit-db's util:eval-async* or *Marklogic's spawn*, both which accept an inline XQuery expression, string or file URI.

Alternatively a scheduling approach or batch driven process can be leveraged, where a command script schedules or spawns query execution in external processes.

Although all these approaches are effective, they often leave the domain of a single query and generally do not provide robust error handling mechanisms.

¹ <https://api.jquery.com/category/deferred-object/>

Additionally in my experience, they often lead to complex multi teired scripts which are more prone to failure and costlier to maintain.

Another approach exists, and has been used with great success for years, and its called the promise pattern.

3. Introducing the Promise Pattern

The idea behind the promise pattern is simple. It provides a mechanism to 'defer' a piece of work, resulting in a new object, called a 'Promise'. A promise is usually a non executed piece of work, which has the added benefit of accepting callbacks.

Before I dive any futher, boot up BaseX and follow along!

3.1. Install Steps

The module which makes the promise pattern happen can be found here: <https://github.com/james-jw/xq-promise>

3.1.1. Dependencies

This module is dependent on BaseX².

3.1.2. Automatic Install

Install as an EXPath Module using the following BaseX command:

```
repo:install("https://raw.githubusercontent.com/james-jw/xq-promise/▶  
master/dist/xq-promise-0.8.1-beta.xar")
```

3.1.3. Manual Install Steps (Windows)

1. Install BaseX
2. Copy the `xq-promise-x.x.x-Beta.jar` into your `basex\lib` directory
3. As well as the the `xq-promise.xqm` file into your `basex\repo` directory
4. Install the `xq-promise.xqm` module using the following command

```
basex REPO INSTALL xq-promise.xqm
```

3.1.5. Declaration

To use the module in your scripts simple import it like so:

```
import module namespace promise = 'https://github.com/james-jw/xq-  
promise';
```

² <http://www.basex.org>

4. The Basics of a Promise

The `xq-promise` module, in its current iteration, includes several methods for creating and interacting with promises.

4.1. defer

The first and most important function is `defer`.

```
defer($work as function(*),
      $arguments as item()*,
      $callbacks as map(*,function(*)*)
      as function(map(*,function(*)*))
```

The signature may look daunting but the pattern is simple. Use the `defer` method to defer a piece of work for later execution by passing in a function item and the future arguments as a sequence.

Lets see how this works with a 'Hello World' example:

```
import module namespace promise = 'http://github.com/james-jw/xq-
promise';
let $greet := function($name) {
  'Hello ' || $name || '!'
}
let $promise := promise:defer($greet, 'world')
return
  $promise
```

In the above example, we defer the execution of `$greet` until after the return of `$promise`. Upon execution of the script we should see `hello world!`.

But wait! If you examine the output. The value returned is:

```
function (anonymous)#0.
```

This is not at all what we want! Yet this is where the power of the promise pattern hopefully starts to be realized. Formost, as mentioned prior, a promise is a function. To retrieve its value, it must be called. Change the last line of the above example as follows:

```
$promise()
```

With the above modification we get the expected answer: `Hello world!`

So far we have deferred a simple piece of work and then learned how to execute it at a later. Now let me introduce the real power of the promise³ pattern with callbacks

³ <https://api.jquery.com/category/deferred-object/>

5. Callbacks

A callback is a function which will be executed on the success or failure of some deferred work.

6.1. Adding callbacks

For example:

```
let $extractListItems := function ($res as map(*)) { $res?list?* }
let $error := function ($err) {
  trace($err, 'Failed: ') => prof:void()
}
let $retrieve := p:defer($worker, ($req, $uri))
  => p:then(parse-json(?))
  => p:fail($error)
let $extract = p:then($retrieve, $extractListItems)
return
  $extract()
```

Note the calls to `then` and `fail` using the arrow operator. These calls add additional callbacks to the callback chain on the promise returning from `defer` resulting in an augmented promise which is stored in `retrieve`. Because of this, concise chaining can be accomplished!

Also note how the `$extractListItems` callback is appended to the `$retrieve` promise, resulting in a new promise `$extract`, which when executed, will initiate the full chain of callbacks!

6.1.2. Chaining Helper Functions

Four methods, two of which were just demonstrated, matching the callback event names exist for attaching callbacks in a chain fashion leveraging the new `=>` (arrow) operator in XQuery 3.1. For example:

```
let $retrieve := p:defer(http:send-request(?, ?), ($req, $uri))
  => p:then(parse-json(?))
  => p:then($extractListItems)
  => p:always(trace(?))
  => p:done(file:write-text(?, $path))
  => p:fail($error)
return
  $retrieve()
```

7.1. then

The `then` callback will be invoked upon success of deferred execution. It acts as a pipeline function for transforming the response over successive callback execu-

tions. Unlike the next two events, but similar to `fail`, this method can alter the pipeline result, and generally does.

7.2. `done`

Called on success.

This method has no effect on the pipeline result and thus its return value will be discarded. Its main purpose is for reacting to successful deferred execution as opposed to affecting its outcome like `then` does.

A common use case for `done` is logging.

7.3. `always`

Operates the same as `done`, except it also is called on the promise's failure, not only success. For example, if only an `always` callback is provided and an error occurs in the original deferred, the `always` callback will be provided the error details prior to the query ceasing with an exception. This allows for error logging or other duties. Error handling and mitigation; however, is relegated to the final callback `fail`

7.4. `fail`

This final callback `fail` is called if a deferred action fails.

A failure occurs if any deferred work or callback function throws an exception. The `fail` callback allows handling and potentially mitigating these errors during a fork-join, fork or even serial process. Without a `fail` callback an exception will go uncaught and cause the entire query to stop. In essence, adding a `fail` callback to a deferred chain, is equivalent to the `catch` in a `try/catch` clause.

Upon failure, the callback will be provided a `map(*)` containing the error details as well as some **additional** information:

```
map {
  'code': 'error code',
  'description': 'error description',
  'value': 'error value',
  'module': 'file',
  'line': 'line number',
  'column': 'column number',
  'additional': map {
    'deferred': 'Function item which failed. Can be used to retry the ►
request',
    'arguments': 'The arguments provided to the failed deferred.'
  }
}
```

Similar to a catch clause, the fail callback has the option of returning a value as opposed to propagating or throwing an error itself:

```
promise:defer($work)
=> promise:fail(function ($err) {
    if($err?code = 'XQPTY0005') then 'I fixed it!'
    else fn:error(xs:QName('local:error'), 'Unfixable error!')
})
```

In the above example we see that if the `$err?code` returned matches `XQPTY0005`. The error will be mitigated and the result will be the value `I fixed it!`. This is because, as stated, if a fail callback returns a value, the failure will be handled with value returned from the fail callback replaced in the pipeline.

If no suitable replacement value exists, but the error should simply be ignored. The fail callback would return the empty-sequence.

```
promise:defer($work)
=> promise:fail(function ($err) {
    if($err?code = 'XQPTY0005') then ()
    else fn:error(xs:QName('local:error'), 'Unfixable error!')
})
```

In this example the error code `XQPTY0005` will result in the empty sequence `()` and thus the error being ignored.

Alternatively, if the error code is not `XQPTY0005` and the failure cannot be ignored. Throwing an exception within the callback using `fn:error` would cause the entire fork and query to cease in addition to reporting the error in the console.

7.4.1. Understanding callback chains

When multiple callbacks are added to a promise, multiple levels of processing and error handling can be achieved. The order callbacks are added is important, especially in regards to `then` and `fail`. In order to help better explain, let's take the following example which chains up a series of callbacks:

```
promise:defer($work)
=> p:then(parse-json(?))
=> p:done(trace(?, 'Json Parsed: '))
=> p:fail(json-parse-error-handler(?))
=> p:then(transform-json(?))
=> p:fail(transform-error-handler(?))
=> p:always(trace(?, 'Result: '))
```

I am going to try and do a blow by blow of the above example. Hopefully it will be clear how order plays a role in callbacks and error handling.

In the above example, the `$work` variable we will assume returns a json string.

If the parsing of the json string in the first callback `parse-json`, throws an exception, the `json-parse-error-handler` will get an error map(*) as described

earlier and will have the opportunity to remedy the error. For example, it could try parsing the file differently.

Alternatively, should the parsing succeeded, the `done` callback would get the resulting parsed json object, and in this example, log it to the console with the tag:

```
Json Parsed: { ...
```

If a failure occurs but the `json-parse-error-handler` callback returns a value, or no error occurs at all with `parse-json` succeeding, the value from either case would be passed to the following `transform-json` function. Should this following transform succeed. The result will be seen in the console due to the `always` callback. If however; an error occurs, the `transform-error-handler` would be called with the appropriate error map(*) instead.

Alternatively, had the previous error handler `json-parse-error-handler` thrown an error instead of returning a value as demonstrated in the previous paragraph. The `transform-error-handler` would have been provided this error for mitigation as well.

In either case, this last error handler will either return a value, or fail. If it returns a value and thus resolves the error, the `always` callback will be provided this result, which will be logged to the console. Otherwise, if an exception is thrown, the `always` callback will still be called with the error details, but since no further error handlers exist in the chain, the exception will go uncaught and be thrown after the `always` callback completes its duties. This will result in the entire query being terminated.

7.5. when

Now that hopefully pipelines makes sense, Another critical method in the `promise`⁴ pattern is the `when` function.

```
when($promises asfunction(map(*,function(*)),
    $callbacks as map(*,function(*)*)
    asfunction(map(*,function(*)))
```

The purpose of `when` is to combine 2 or more promised actions into a single promise. This is extremely powerful. Like the `defer` method discussed earlier, the `when` method also returns a deferred promise, which accepts callbacks just the same.

For example:

```
let $write-and-return-users:= function ($name, $users) as item()* {(
    file:write($name, $users),
    $users
)}
let $extractDocName := promise:defer(doc(?), $doc-uri)
=> promise:then($extract-name)
```

⁴ <https://api.jquery.com/category/deferred-object/>

```
let $extractUsers := promise:defer(json-doc(?), $uri)
  => promise:then($extract-list-items)
let $users:= promise:when(($extractDocName, $extractUsers))
  => promise:then($write-and-return-users)
  => promise:fail(trace(?, 'Requesting users failed: '))
})
return
  $users() ! trace(?.?username, 'Retrieved: ')
```

In this example, we perform two deferred actions and then merge their results in the `$write-and-return-users` callback. Since this item is attached to the `when`'s promise on the `then` callback, its result will be seen on the call to `$users()`.

We could continue to attach callbacks as needed until we are ready. There is no limit.

7.6.1. Multiple Callbacks per event

Multiple callbacks, not just one, can be attached to each of the 4 events. For example:

```
(: same $req, etc.. from above :)
let $extract-links := function ($res) { $res//a }
let $promise := promise:defer($request, 'http://www.google.com')
  => promise:then(($extract-body, $extract-links))
  => promise:fail(trace(?, ('Execution failed!'))
return
  $promise()
```

Foremost, note the sequence of callbacks passed into `then`. Both of these will be called in order. The result of the first callback will be passed to the second. In this example, since `then` is a pipeline callback. The result will be all the links in the document.

Second, note the `fail` callback. It uses the power of XQuery 3.0 and function items⁵ to add a `trace` call when any part of the execution fails. How convenient!

Hopefully its starting to come clear how the `promise` pattern can be quite useful.

8. The Power of Promises and Parallel Execution

It should be clear now: how to defer work for later execution, what a promise is, and how to join multiple promises. It still may not be entirely clear what the benefit this pattern has in the context of XQuery; however that is about to change.

⁵ http://docs.basex.org/wiki/XQuery_3.0#Function_Items

8.1. fork-join

Let me introduce two last methods, and the whole reason I wrote this library.

```
fork-join($promises as function(*)*) as item()*
```

It is simple yet powerful. It accepts a sequence of promises, or single arity functions and executes them in a fork join fashion, spawning threads as needed depending on the work load, followed by rejoining the work on the main thread.

As seen earlier, promises can be used to build up a piece of work for later execution. With this ability, coupled with `fork-join`. Parallelized XQuery processing becomes a reality.

Lets see how we can use this capability by comparing a simple example involving making http requests. The example will use the promise pattern but not `fork-join` just yet.

```
import module namespace promise = 'https://github.com/james-jw/xq-
promise';
let $work := http:send-request(<http:request method="GET" />, ?)
let $extract-doc := function ($res) { $res[2] }
let $extract-links := function ($res) { $res//*[a[@href => ►
matches('^http')]] }
let $promises :=
  for $uri in ((1 to 5) ! ('http://www.google.com', 'http://►
www.yahoo.com', 'http://www.msnbc.com'))
  let $defer := promise:defer($work, $uri)
    => promise:then($extract-doc)
    => promise:done(trace(?, 'Results found: '))
  return
    promise:then($defer, $extract-links )
return
  $promises ! .()
```

In the above example, we use promises to queue up 25 requests and then execute them in order with:

```
$promises ! .()
```

If you run this example in BaseX GUI and watch the output window, you will see the requests come in as the query executes. This is due to the addition of the `trace? 'Results Found: ' callback`.

Also notice, only one request is executed at a time. Each request must wait for the full response and processing of the previous. This is a current limitation of BaseX, since by design it runs each query in its own single thread. There are several workarounds such as splitting up the work via a master query, or using a string concatenated XQuery expression to spawn another process. Although effective, all these workarounds require extra effort and multiple components. Additionally they leave the language's domain and the context of the current query..

Luckily, with the introduction of this module `xq-promise`. This is no longer the case! Lets change the previous example so it uses the newly introduced `fork-join` method to speed up the process, by splitting the requested work into multiple threads before returning the final joined value.

Luckily the previous example already used `defer` so the change is only one line. Replace:

```
$promises ! .()
```

which manually executes each promise on the main thread, with:

```
promise:fork-join($promises)
```

If you watch this execute in BaseX you will quickly see its executing much faster, with multiple requests being processed at once.

On my machine, the first example without `fork-join` took on average 55 seconds. With `fork-join` this time dropped to 6 seconds!

That is a clear advantage! Playing around with `compute size` and `max forks`, which I will introduce shortly, I have been able to get this even lower, to around 2 seconds!!

8.2. fork

In addition to `fork-join` is the simple `fork` method. The `fork` method operates much like `defer`, in that it returns a promise which accepts callbacks. Unlike `defer` however, the work its provided is executed immediately in a new thread. This is as opposed to being deferred for later execution.

For example, lets imagine we want to optimize the performance of a web response. During the processing, an external API is queried in addition other internal processing. The internal call is not dependend on the external call, until the end, and thus these operations can run in parrallel:

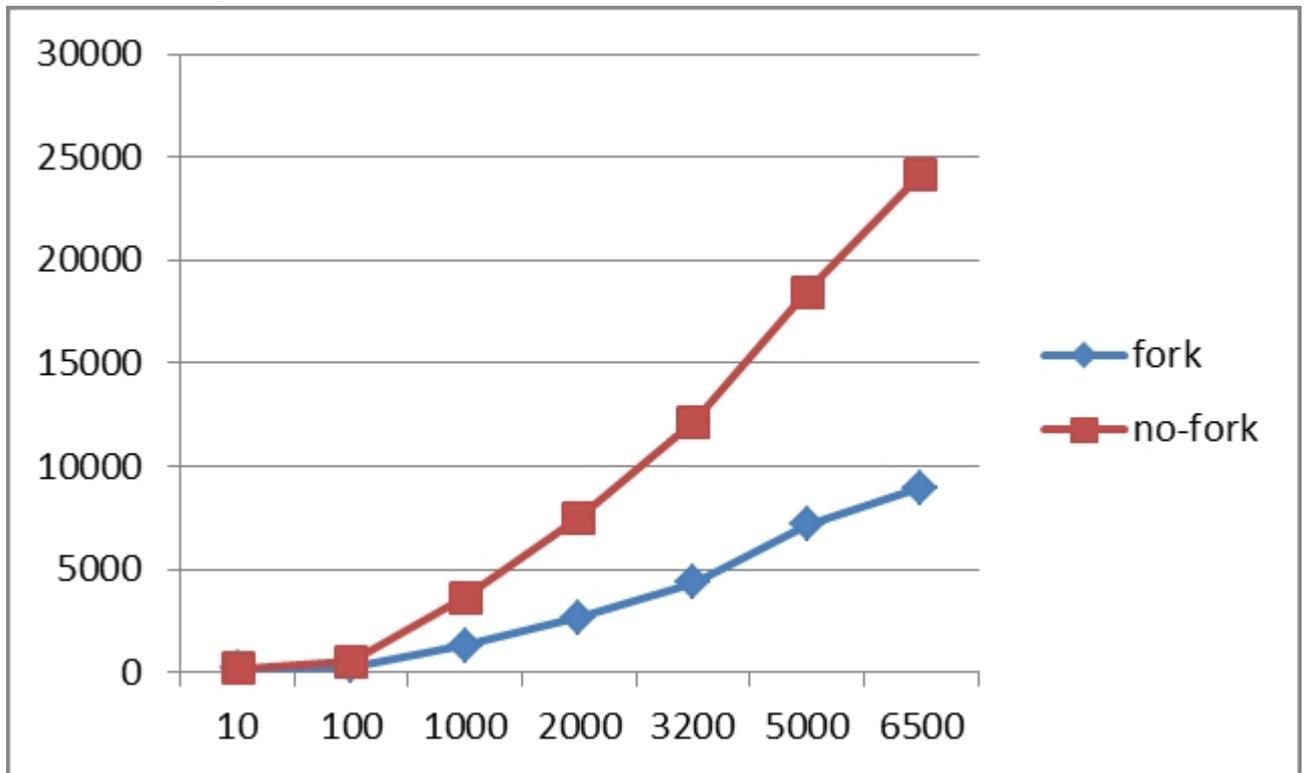
```
let $request := http:send-request($req, ?)
let $promise := promise:fork($request, 'http://myapi.com')
let $hardAnswer := some-heavy-work()
return
  ($hardAnswer, $promise())
```

In the above example, the work of sending the `http` request, and waiting for its response, will be forked immediately letting the main thread continue with computing the `$hardAnswer`. Once that is done, both it and the promise can be returned.

Hopefully its clear now what the use cases for both `fork-join` and `fork` are and how to use them!

9. Performance

Below is a chart showing execution with and without fork-join being employed. The Y axis shows the full execution time in milliseconds while the X shows the number of items processed.



With a Quad-Core machine, the execution time is cut to nearly a quarter. I'd expect it to be around an eighth on an 8 core machine.

Here is the xquery script used in the above test:

```
import module namespace promise = "https://github.com/james-jw/xq-  
promise";  
import module namespace geo = "http://expath.org/ns/geo";  
declare namespace gml = "http://www.opengis.net/gml";  
  
let $counties := db:open('DetailedCounties')//feature/gml:*  
let $first := $counties[1]  
return  
  promise:fork-join(  
    for $county in $counties  
    return promise:defer(geo:intersection($first, ?), $county)  
  )
```

10. Limitations

With any async process there are limitations. So far these are the only noticed limitations:

- Updating database nodes in a callback
- Using a transform clause in a callback

11. Implementation

This library is implemented for BaseX⁶ via the QueryModule⁷ class. It leverages Java 7's ForkJoinPool⁸ and RecursiveTasks⁹.

Here are three `java` source files as part of the implementation:

11.1. XqPromise

The XqPromise class implements QueryModule¹⁰ from the BaseX API and exposes the methods described earlier:

- `defer`
- `when`
- `fork-join`
- `is-promise`

11.2. XqDeferred

This class is at the core of the promise¹¹ pattern and represents a unit of work to perform in the future. It implements the XQFunction interface from the BaseX¹² API, and thus is an XQuery Function. The purpose of this function item is to defer the execution of work. Not only is it a function, but it also maintains the parameters it should leverage when called.

If called, it executes its work, with the provided arguments work.

11.3. XqForkJoinTask

Implements RecursiveTask¹³ and performs the forking process leveraging a fixed ForkJoinPool¹⁴

The pool size is determined by default by the number of CPU cores. The

⁶ <http://www.basex.org>

⁷ <http://docs.basex.org/javadoc/org/basex/query/QueryModule.html>

⁸ <https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

⁹ <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html>

¹⁰ <http://docs.basex.org/javadoc/org/basex/query/QueryModule.html>

¹¹ <http://www.basex.org>

¹² <http://www.basex.org>

¹³ <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html>

¹⁴ <https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

12. Moving Forward

Given the level of ease in executing this implementation in BaseX I am hopeful the same can be said for other implementations of XQuery 3.1.

With the addition of the Promise pattern to the XQuery repertoire, I could also see a possibility for the easy addition of an `async` keyword within the language using a transformation on the XQuery code.

For example, the following query, with the addition of the `'async'` keyword in the `for` block:

```
let $req := <http:request ...>
for async $uri in $uris
let $item := http:send-request($req, $uri)
return
  <something>{$item}</something>
```

could be transformed to the following query, leveraging the promise pattern:

```
let $req := <http:request ...>
let $for-func := function ($uri) {
  let $item := http:send-request($req, $uri)
  return
    <something>{$item}</something>
}

return p:fork-join(
  for $uri in $uris
  return
    p:defer($for-func, ($uri))
)
```

Notice how the structure is maintained. The transformation would simply require the wrapping of the entire `for` block in a `fork-join` call, followed by wrapping the contents of the `for` block contents in a function item.

13. Shout Out!

If you like what you see here please star the repo and find me on github¹⁵ or linkedIn¹⁶

14. Contribute

If you have any ideas to make it better, feel free to leave feedback, make a pull request, log an issue or simply question ask a question! Additionally if your inter-

¹⁵ <https://github.com/james-jw>

¹⁶ <https://www.linkedin.com/pub/james-wright/61/25a/101>

esting in making this work in another database system, Id be glad to provide any guidance and assistance in making that happen!

Happy forking!

Entities and Relationships in a Document Database

Charles Greer
MarkLogic Corporation
<cgreer@marklogic.com>

Abstract

Those working with NoSQL databases agree that some flavor of this technology will supplant the relational database in coming years. While the post-relational world to some looks like "polyglot persistence" in which enterprises use multiple database technologies, each in a best-fit scenario, there is an alternate emerging trend of the hybrid database, in which a single platform can handle workloads of varying types, and provide the benefits of central administration and systems architecture. We think the document-oriented database, with XML as a foundation, is the basis for the next generation of hybrid databases, because the document model is expressive enough to encapsulate simpler models within it. The final step in replacing older relational database technology is to encapsulate its native model and expose that model to new capabilities. So this paper explores how to work with Entities, Attributes, and Relationships on top of a document database and XML.

Keywords: XML, RDF, Data Integration, Enterprise Architecture

1. Introduction: Reseeding the Enterprise

I love the spectrum of "greenfield" and "brownfield" to characterize enterprise IT.

One one end of this spectrum are "greenfield projects." These are systems that are constructed from scratch, with no integration requirements or dependencies on other systems. "Greenfield" does not exist. On the other end is the "brownfield." A prototypical "brownfield project" grows up within a mature enterprise, in which integrations are *ad hoc* or point-to-point, and in which the complex interactions of systems and data can make change management slow and expensive. This is the starting state of all real enterprise projects.

So enterprise architects, who accept that all fields are brown, face a continual challenge to improve data services, to evolve architectures, to maintain the old while delivering the new. Proactive management of an IT portfolio requires focus on real requirements, systems architecture, data architecture, and on how to deploy applications.

In this essay I use outrageous metaphors about growing and cultivating enterprise IT, with emphasis on how to build new systems by seeding them from old ones.

- Dirt** This story's setting is a set of software systems. I call it dirt because it forms a substrate on which our information systems run. "Dirt" consists of databases and applications that move data around. In particular we call out the data hub, an architectural pattern that aggregates, normalizes, and disseminates enterprise data.
- Seeds** Entity/Attribute/Relationship diagrams (or simply E/R diagrams) pervade enterprise IT documentation. Here, we emphasize the pattern behind such diagrams, which is a method of decomposing complex types into tuples, and propose a declarative model for incremental enterprise data integration. E/R diagrams contain information about how the data needs to be used, how it propagates. Such E/R models can be very small.
- Trees** We use trees, specifically XML trees, to encode and persist our entity instances. Trees can encode source data from relational systems, and model-valid instance data together. Moreover, we know how to index and query trees.
- Fruit** The purpose of aggregating data and indexing it is to enable applications, for the data to provide value to the business as a whole. In this case, exposing entities and relationships via a SQL-like query language is one kind of fruit. Search over tree structures is another. Ideally such fruit is low-hanging.

So let's dig in more deeply to the dirt and explore data integration!

2. Dirt

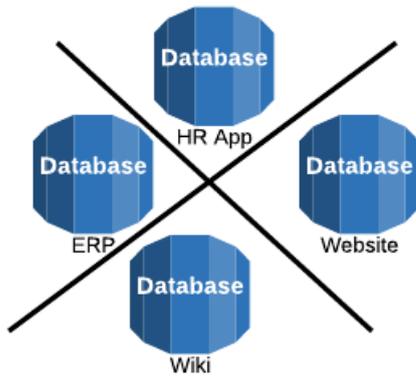
Substrate of Data Integration Efforts

In the beginning there was an ERP system that ran the business.

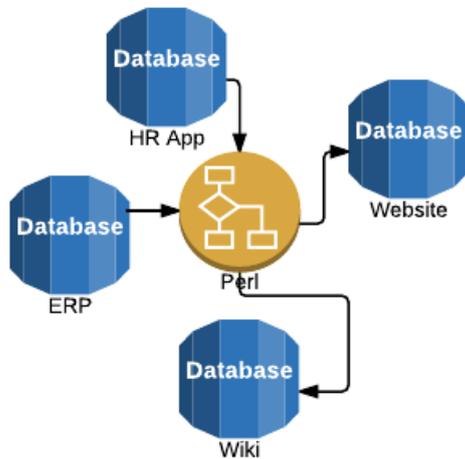
Then came the website that sold things.

Then there was the HR system, and the Wiki.

Each system had its own database, and they did not speak to, or of, one another.



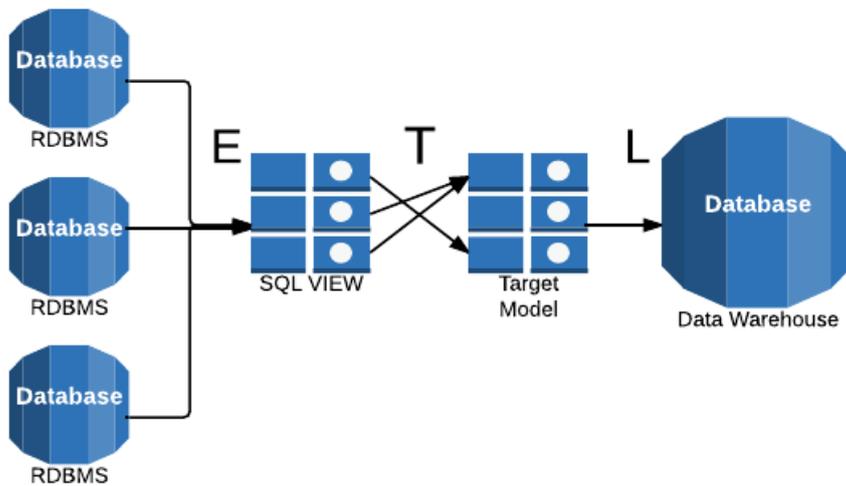
Then came perl. Perl could extract data from the ERP, massage it, send it to the website, from the website to the ERP, from the HR system, translate, to the website, and from the Wiki to hell.



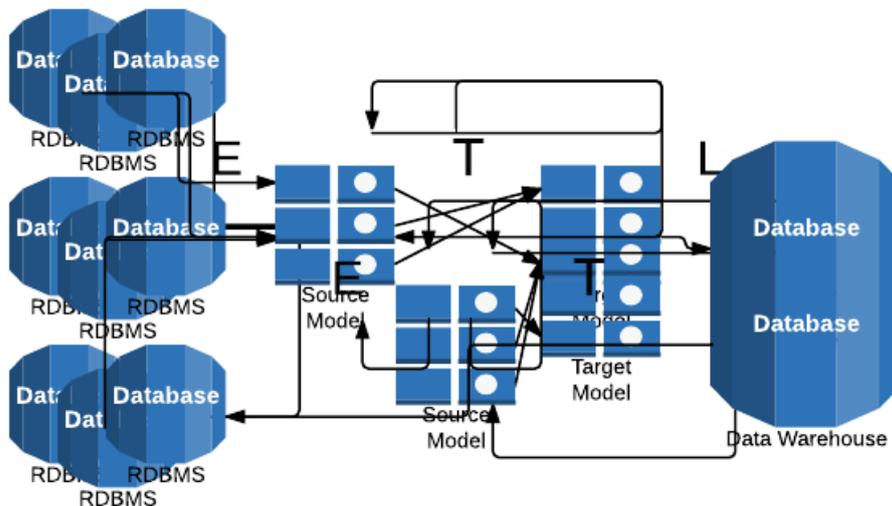
Then came the data warehouse, and all movement ceased.

This is the brownfield, the dirt in which we built the messy conglomerates of thousands of systems and data stores. They call them silos, and everybody wants what's in them. New applications all need the data from the old systems. And its all locked up in purpose-built relational systems and schemas.

A common pattern emerges, in which we extract data from source systems, get it ready for downstream ones, and load that data into a new one. ETL.

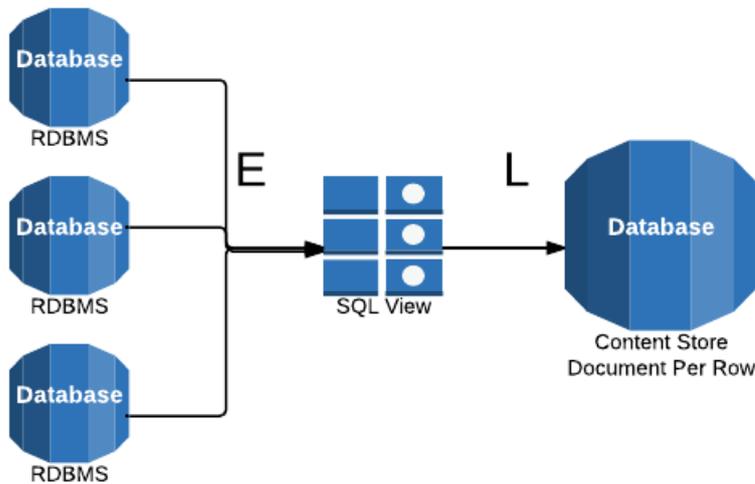


In the ETL process, businesses inevitably need to rework data models. The upstream transactional system manages data in one (relational) schema. The downstream one reworks the data into facts and dimensions. Another turns the data into web-ready denormalized tables. Oh, and there are sixteen transactional systems and three data warehouses. Each system requires a model, and the ETL layer provides the glue and the mapping logic between the disparate models.



Yes, it's a mess.

In contrast to this absurdist view of ETL, there is another type of data integration that really is simply data aggregation. A document database requires no schema up-front. With very simple exported structures (one row maps to one document) it's simple to move data from a SQL query result into a document-oriented database. A data hub using a schema-agnostic technology offers the promise of simple, but dumb, aggregation of data.



The source data from several databases arrives at a single new system, but with no model reconciliation. We can search it, but the meaning of the data has been lost in translation.

While this integration is cleaner looking than traditional ETL, and many enterprises have benefited from such a schemaless data aggregation, we can do better. We can cultivate meaning in the dirt, create the beginnings of a common emergent model across data sources. We'll call these initial chunks of meaning seeds, just to stretch the metaphor further.

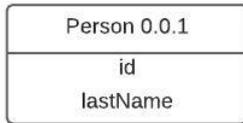
3. Seeds

Partial and incremental E/R models

Lacking in schemaless integration is a conveyance of meaning from source system to targets. The schemas from the source databases have not been recast in a common vocabulary, and as such there's not as much value in the aggregated data as there could be.

A simple metamodeling approach to data hubs might help. Let's say that we are interested in building an application whose domain is all of the people that I do business with, across many business units and relationship types. Though there may be duplicates and there may be differences in how people are represented, it's a reasonable dataset to want to extract, merge, and republish.

Say I want to search by last name. I'm going to create a very straightforward E/R diagram intended to "receive" instance data about people. It has just an id, and a last name. The very minimum I need to create an application that can search explicitly by `lastName`.

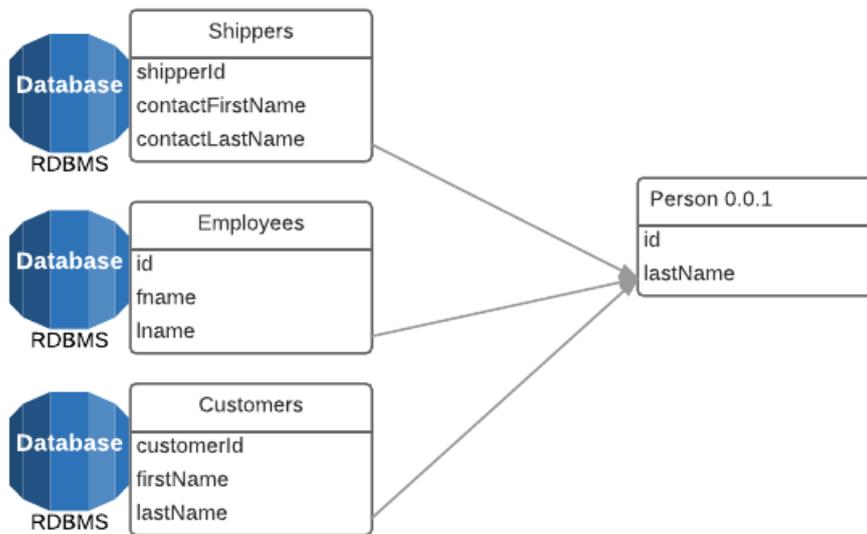


We can even store such entity types in a document database by making a document structure for them. How about a flavor of JSON Schema?

```
{
  "info": {
    "title": "Person",
    "version": "0.0.1"
  },
  "definitions": {
    "Person": {
      "properties": {
        "id": {
          "type": "string"
        },
        "lastName": {
          "type": "string"
        }
      }
    }
  }
}
```

For any given source system, there is an implicit relationship between that source and this new entity type. For each source schema, there is some mapping (or function) that can create an instance of `Person`. The point is that we are not particularly concerned with merging any more than absolutely necessary to meet the requirements of my integration today.

An example:



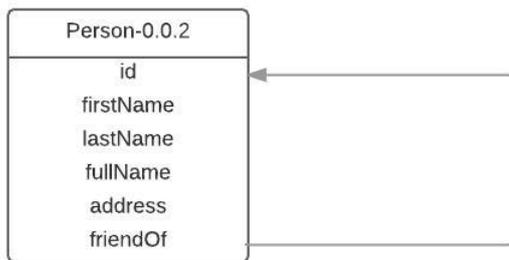
In this example, `lastName` in my entity type will be sourced from the columns `contactLastName`, `lname`, and `lastName` respectively.

```
// generated, then edited function stub
xquery version "1.0-ml";
module namespace person = "http://entities.org/Person/0.0.1";
(: create person from shipper document :)
declare function person:from-shippers($source as document-node())
mas map:map
{
    let $new-person := map:map()
    let $_ := map:put($new-person, "id", guid())
    let $_ := map:put($new-person, "lastName", data($source/▶
contactLastName))
    return $new-person
};
(: create person from employee document :)
declare function person:from-employees($source as document-node())
mas map:map
{
    let $new-person := map:map()
    let $_ := map:put($new-person, "id", guid())
    let $_ := map:put($new-person, "lastName", data($source/lname)
return $new-person
};
(: create person from customer :)
declare function person:from-customers($source as document-node())
mas map:map
{
```

```
let $new-person := map:map()
let $_ := map:put($new-person, "id", guid())
let $_ := map:put($new-person, "lastName", data($source/lastName))
return $new-person
};
```

Now it's one thing to present an entity type with two attributes. It's another to create a real data hub. But this process is iterative, and I just want to give you a taste. An entity type is versioned - we update the version with each change in order to keep track of the evolution of the integration.

Let's say that the next version of our entity type requires addresses, and also provides for a graph representation of people knowing each other. The E/R diagram now looks something like the following. It has several more attributes, and is in a relationship. Instances of Person may now refer to other instances.



I represent this new model in JSON thus:

```
{
  "info": {
    "title": "Person",
    "version": "0.0.2"
  },
  "definitions": {
    "Person": {
      "properties": {
        "id": {
          "type": "string"
        },
        "firstName": {
          "type": "string"
        },
        "lastName": {
          "type": "string"
        },
        "fullName": {
          "type": "string"
        }
      }
    }
  }
}
```

```
"address": {
  "type": "string"
},
"friendOf": {
  "$ref": "#/definitions"
}
},
"primary-key":["id"]
}
}
```

Maybe in six months we'll use another field from that source system. If the structure is still not appreciably changed, we can simply append to the new model, and update transforms that are used to fill it.

I was neglecting a good part. We are using a hybrid database, which stores tress, but can query triples and tuples too.

These JSON documents actually surface to triples to an RDF index. My document database happens to map XPath expressions to generated triples and to expose them to a SPARQL engine. The document above surfaces these triples:

```
@prefix es: <http://marklogic.com/entity-services#> .
@prefix doc: <http://example.org/> .
@prefix type: <http://example.org/Person-0.0.2/> .
@prefix prop: <http://example.org/Person-0.0.2/Person/> .
```

```
doc:Person-0.0.2 a es:EntityTypeDoc ;
  es:title "Person" ;
  es:version "0.0.2" ;
  es:definitions type:Person .
type:Person a es:EntityType ;
  es:version "0.0.2" ;
  es:property
    prop:id,
    prop:firstName,
    prop:lastName,
    prop:fullName,
    prop:address,
    prop:friendOf ;
  es:primaryKey prop:id .
prop:id a es:Property ;
  es:title "id" ;
  es:datatype "string";
  a es:PrimaryKey .
prop:firstName a es:Property ;
  es:datatype "string";
  es:title "firstName" .
prop:lastName a es:Property ;
```

```
    es:datatype "string";
    es:title "lastName" .
prop:fullName a es:Property ;
    es:datatype "string";
    es:title "fullName" .
prop:address a es:Property ;
    es:datatype "string";
    es:title "address" .
prop:friendOf a es:Property ;
    es:ref type:Person ;
    es:title "friendOf" .
```

These triples can be combine with external ones, such as those used in enterprise ontology management. Also, they can be queried alongside RDF reference data ingested from disparate sources.

We also declare performance-related information about the entity type model:

```
{
  "info": {
    "title": "Person",
    "version": "0.0.2"
  },
  "definitions": {
    "Person": {
      "properties": {
        ...
      }.
      "primary-key":["id"],
      "range-index":["lastName"],
      "word-lexicon":["fullName"]
    }
  }
}
```

What makes this incremental approach possible, is that in our systems architecture we had the ability to store not just the newly minted model, but also whatever sources came along with it. We instantiate the model, in other words, alongside its source within XML tree structures.

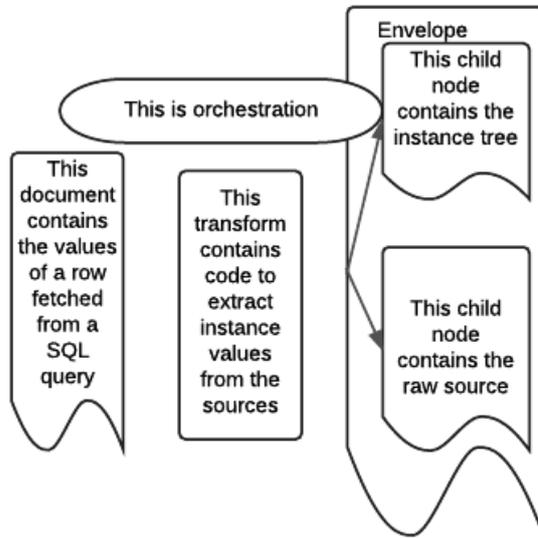
4. Trees

Real XML trees.

You've seen how we can use NoSQL databases to aggregate information, and I claim can make simple E/R diagrams more complex over time without disrupting the business. XML documents enable this kind of flexible combination of structures and schemas.

Within the data hub, we use a particular pattern to integrate data from upstream systems. This so-called "envelope pattern" wraps instance data together

with raw sources in a single XML document tree. This simple mechanism ensures that, no matter how minimal the entity type, the raw source is available for word searches, and even for retrieval by downstream systems. Thus the lineage of the data can be preserved by keeping the original document stored together with whatever instance data we've extracted.



Expressed in XML, we

1. Create a new wrapper document, the `<envelope>`
2. Create an XML serialization of an entity instance within the envelope.
3. Include the source document inside a child of `<envelope>`

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <Person>
    <id><123/id>
    <firstName>Barty</firstName>
    <lastName>Crouch</lastName>
    <fullName>Barty Crouch</fullName>
    ...
  </Person>
  <es:sources>
    ... original document here ...
  </es:sources>
</es:envelope>
```

XML can also make envelopes from other document-based sources: spreadsheets, .docx documents, or ESB messages.

So what can we do with an entity instance once its instantiated in a tree structure and an XML database?

Anything we can already do with trees!

We know how to index and search XML documents really well. With the added benefit of an entity type model, the structure of an entity instance is deterministic.

The envelope is a map of instance data to documents, preserving the fidelity of individual properties and data types. Moreover, since our model also asserted index definitions, it can generate a configuration that specifies indexes. Here is a configuration artifact that defines `range-path-index` definitions for an entity type:

```
{
  "database-name": "%%DATABASE%%",
  "schema-database": "%%SCHEMAS_DATABASE%%",
  "path-namespace": [
    {
      "prefix": "es",
      "namespace-uri": "http://marklogic.com/entity-services"
    }
  ],
  "range-path-index": [
    {
      "collation": "http://marklogic.com/collation/",
      "invalid-values": "reject",
      "path-expression": "/es:envelope/es:entity/Person/lastName",
      "range-value-positions": false,
      "scalar-type": "string"
    }
  ]
  ...
}
```

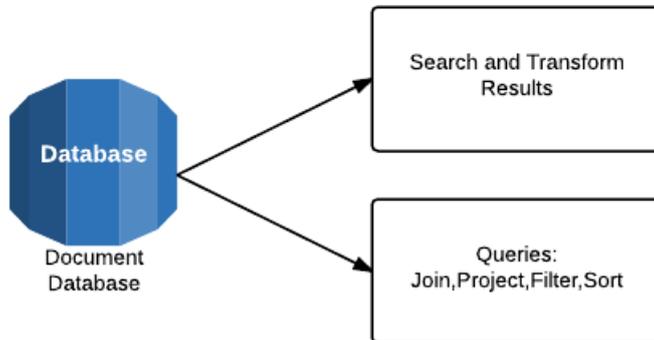
From the model, in other words, we can infer that the `fullName` property in this person record is always found at the XPath `/es:envelope/es:entity/Person/fullName`, and thus we can index just that property.

Ultimately, the E/R model that we created before, combined with a document-based persistence strategy, has enabled a data-driven way to accelerate application development.

5. Fruit

Denormalize, Index, Join

Data hubs in a large integration project combine document services with row or tuple-oriented ones. Application developers have access to instance data as documents, rows, or triples, in a manner consistent with the queryable entity type metadata.



One thing developers can make is search applications. Because their persisted structure is a simple XML tree, and because there's a trivial transformation of this instance data to JSON, web developers can manipulate instance data from a search interface with ease.

I've also got something else interesting. Entity instances look a lot like rows; they are flat structures, relationships being references to other flat structures. "Rows" imply that we should be able to access the entity data via some SPARQL or even SQL-like interface, one that projects, joins, filters and sorts. ODBC clients such as BI tools, web apps, and SPARQL clients alike can query across instance data, the entity model, and enterprise reference data all with one interface. In fact, in our futuristic database, this kind of query will work too:

```
SELECT
  a.lastName as lastName,
  friend.lastName as friendsName
FROM
  Person a, Person friend
WHERE
  a.lastName = "Smith" and
  a.friendOf = friend.id
```

Futuristic? Maybe. I make no commitments. But it would be tasty fruit, probably low hanging.

6. Hybrid Data Hubs

Behind the envelope pattern is a notion that we need not create a model in order to load data. However, type-conformant instance data can be materialized alongside source documents. There's just one system for operations and security.

Storing entities alongside source documents and reference data within XML trees is an approach to data aggregation and modeling that leverages a hybrid storage solution. The benefits of such an approach can be significant; operational systems in an enterprise are not, after all, insignificant with regard to support

services. For every database in production one needs a backup and disaster recovery plan, an audit and security policy, an administration layer, and of course a separate loading and egress strategy.

By using incremental and evolutionary entity-relationship models, overlaid upon a document database, an architectural team can leverage a single document-oriented database to manage a data hub. With this pattern we avoid the messy alternatives present in today's ETL scenarios.

Transforming JSON using XSLT 3.0

Michael Kay

Saxonica

<mike@saxonica.com>

Abstract

The XSLT 3.0 and XPath 3.1 specifications, now at Candidate Recommendation status, introduces capabilities for importing and exporting JSON data, either by converting it to XML, or by representing it natively using new data structures: maps and arrays. The purpose of this paper is to explore the usability of these facilities for tackling some practical transformation tasks.

Two representative transformation tasks are considered, and solutions for each are provided either by converting the JSON data to XML and transforming that in the traditional way, or by transforming the native representation of JSON as maps and arrays.

The exercise demonstrates that the absence of parent or ancestor axes in the native representation of JSON means that the transformation task needs to be approached in a very different way.

1. Introduction

JSON [2] has become a significant alternative to XML as a syntax for data interchange. The usually-cited reasons¹ include:

- JSON is simpler: the grammar is smaller. The extra complexity of XML might be justified for some applications, but there are many others for which it adds costs without adding benefits.
- JSON is a better fit to the data models of popular programming languages like Javascript, and this means that manipulating JSON in such languages is easier than manipulating XML.
- JSON is better supported for web applications (for example, for reasons that are hard to justify, JSON is not subject to the same security restrictions as XML for cross-site scripting).

However, some of the transformation tasks for which XSLT is routinely used (for example, hierarchic inversion) are difficult to achieve in general-purpose languages like JavaScript.

¹I include here only the reasons that I consider to be credible. Many comments on the topic also claim that XML is more verbose or that its performance is worse, but this appears to be folklore rather than fact.

XSLT 3.0 [4] (together with XPath 3.1 [5]) provides capabilities for handling JSON data. These capabilities include:

Two new functions `json-to-xml()` and `xml-to-json()` to convert between JSON and XML. These perform lossless conversion. The `json-to-xml()` function delivers XML using a custom XML vocabulary designed for the purpose, and the `xml-to-json()` function requires the input XML to use this vocabulary, though this can of course be generated by transforming XML in a different vocabulary.

Two new data types are introduced: maps and arrays. These correspond to the "objects" and "arrays" of the JSON model. In fact they are generalizations of JSON objects and arrays: for example, the keys in map can be numbers or dates, whereas JSON only allows strings, and the corresponding values can be any data type (for example, a sequence of XML nodes), whereas JSON only allows objects, arrays, strings, numbers, or booleans.

A new function `parse-json()` is provided to convert from lexical JSON to the corresponding structure of maps and arrays. (There is also a convenience function `json-doc()` which does the same thing, but taking the input from a file rather than from a string.)

A new JSON serialization method is provided, allowing a structure of maps and arrays to be serialized as lexical JSON, for example by selecting suitable options on the `serialize()` function.

While XSLT 3.0 offers all these capabilities², it does not have any new features that are specifically designed to enable JSON transformations — that is, conversion of one JSON structure to another. This paper addresses the question: can such transformations be written in XSLT 3.0, and if so, what is the best way of expressing them?

Note that I'm not trying to suggest in this paper that XSLT should become the language of choice for transforming any kind of data whether or not there is any relationship to XML. But the web is a heterogeneous place, and any technology that fails to handle a diversity of data formats is by definition confined to a niche. XSLT 2.0 added significant capabilities to transform text (using regular expressions); the EXPath initiative has added function libraries to process binary data[1]; and the support for JSON in XSLT 3.0 continues this trend. XSLT will always be primarily a language for transforming XML, but to do this job well it needs to be capable of doing other things as well.

2. Two Transformation Use Cases

We'll look at two use cases to study this question, in the hope that these are representative of a wider range of transformation tasks.

The first is a simple "bulk update": given a JSON representation of a product catalogue, apply a price change to a selected subset of the products.

²Some of these features are optional, so not every XSLT 3.0 processor will provide them.

The second is a more complex structural transformation: a hierarchic inversion. We'll start with a dataset that shows a set of courses and lists the students taking each course, and transform this into a dataset showing a set of students with the courses that each student takes.

For each of these problems, we'll look first at how it can be tackled by converting the data to XML, transforming the XML, and then converting back to JSON. Then we'll examine whether the problem can be solved entirely within the JSON space, without conversion to XML: that is, by manipulating the native representation of the JSON data as maps and arrays. We'll find that this isn't so easy, but that the difficulties can be overcome.

3. Use Case 1: Bulk Update

Rather than invent our own example, we'll take this one from json-schema.org:

```
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
      "length": 7.0,
      "width": 12.0,
      "height": 9.5
    },
    "warehouseLocation": {
      "latitude": -78.75,
      "longitude": 20.4
    }
  },
  {
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
      "length": 3.1,
      "width": 1.0,
      "height": 1.0
    },
    "warehouseLocation": {
      "latitude": 54.4,
      "longitude": -32.7
    }
  }
]
```

The transformation we will tackle is: for all products having the tag "ice", increase the price by 10%, leaving all other data unchanged.

First we'll do this by converting the JSON to XML, then transforming the XML in the traditional XSLT way, and then converting back. If we convert the above JSON to XML using the `json-to-xml()` function in XSLT 3.0, the result (indented for readability) looks like this:

```
[
<?xml version="1.0" encoding="UTF-8"?>
<array xmlns="http://www.w3.org/2005/xpath-functions">
  <map>
    <number key="id">2</number>
    <string key="name">An ice sculpture</string>
    <number key="price">12.50</number>
    <array key="tags">
      <string>cold</string>
      <string>ice</string>
    </array>
    <map key="dimensions">
      <number key="length">7.0</number>
      <number key="width">12.0</number>
      <number key="height">9.5</number>
    </map>
    <map key="warehouseLocation">
      <number key="latitude">-78.75</number>
      <number key="longitude">20.4</number>
    </map>
  </map>
  <map>
    <number key="id">3</number>
    <string key="name">A blue mouse</string>
    <number key="price">25.50</number>
    <map key="dimensions">
      <number key="length">3.1</number>
      <number key="width">1.0</number>
      <number key="height">1.0</number>
    </map>
    <map key="warehouseLocation">
      <number key="latitude">54.4</number>
      <number key="longitude">-32.7</number>
    </map>
  </map>
</array>
```

And we can now achieve the transformation by converting the JSON to XML, transforming it, and then converting back:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions">

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

  <xsl:param name="input"/>

  <xsl:output method="text"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-xml" select="json-to-xml(unparsed-
text($input))"/>
    <xsl:variable name="transformed-xml" as="document-node()">
      <xsl:apply-templates select="$input-as-xml"/>
    </xsl:variable>
    <xsl:value-of select="xml-to-json($transformed-xml)"/>
  </xsl:template>

  <xsl:template match="map[array[@key='tags']/string='ice']/▶
number[@key='price']/text()">
    <xsl:value-of select="xs:decimal(.)*1.1"/>
  </xsl:template>

</xsl:stylesheet>
```

Sure enough, when we apply the transformation, we get the required output (indented for clarity):

```
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 13.75,
    "tags": [
      "cold",
      "ice"
    ],
    "dimensions": {
      "length": 7,
      "width": 12,
      "height": 9.5
    },
    "warehouseLocation": {
      "latitude": -78.75,
```

```
        "longitude": 20.4
      }
    },
    {
      "id": 3,
      "name": "A blue mouse",
      "price": 25.5,
      "dimensions": {
        "length": 3.1,
        "width": 1,
        "height": 1
      },
      "warehouseLocation": {
        "latitude": 54.4,
        "longitude": -32.7
      }
    }
  ]
}
```

Now, the question arises, how would we do this transformation without converting the data to XML and back again?

Here we immediately see a difficulty. We can't use the same approach because in the map/array representation of JSON, there is no parent axis. In the XML-based transformation above, the semantics of the pattern `map[array[@key='tags']/string='ice']/number[@key='price']/text()` depend on matching a text node according to properties of its parent (a `<number>` element) and grandparent (a `<map>` element). In the map/array model, we can't match a string by its context in the same way, because a string does not have a parent or grandparent.

However, all is not lost. With a little help from a general-purpose helper stylesheet, we can write the transformation like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:jlib="http://saxonica.com/ns/jsonlib"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array" ►
  version="3.0">

  <xsl:param name="input"/>

  <xsl:output method="json"/>

  <xsl:import href="maps-and-arrays.xsl"/>

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

```
<xsl:template name="xsl:initial-template">
  <xsl:apply-templates select="json-doc($input)"/>
</xsl:template>

<xsl:template match=".[. instance of map(*)][?tags = 'ice']">
  <xsl:map>
    <xsl:sequence select="map:for-each(.,
      function($k, $v){ map{$k : if ($k = 'price') then $v*1.1
else $v }})"/>
  </xsl:map>
</xsl:template>
</xsl:stylesheet>
```

This relies on the helper stylesheet, `maps-and-arrays.xsl`, containing default processing for maps and arrays that performs the equivalent of the traditional "identity template" (called shallow-copy processing in XSLT 3.0): specifically, processing an array that isn't matched by a more specific template rule should create a new array whose contents are the result of applying templates to the members of the array; while processing a map should similarly create a new map whose entries are the result of applying templates to the entries in the existing map. Unfortunately the shallow-copy mode in XSLT 3.0 doesn't work this way; it has the effect of deep-copying maps and arrays.

For maps, we can write a shallow-copy template like this (it's not actually needed for this use case):

```
<xsl:template match=".[. instance of map(*)]" mode="#all">
  <xsl:choose>
    <xsl:when test="map:size(.) le 1">
      <xsl:sequence select="."/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:map>
        <xsl:variable name="entries" as="map(*)*"
          select="map:for-each(., function($k : $v) { map:entry($k,
$ v) } )"/>
        <xsl:apply-templates select="$entries" mode="#current"/>
      </xsl:map>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

This divides maps into two categories. Applying templates to a map with less than two entries returns the map unchanged. Applying templates to a larger map splits the map into a number of singleton maps, one for each entry, and applies

templates recursively to each of these singleton maps. In the absence of overriding template rules for any of these entries, the entire map is deep-copied.

To make it easier to write a template rule that matches a singleton map with a given key, we can define a library function:

```
<xsl:function name="jlib:is-map-entry" as="xs:boolean">
  <xsl:param name="map" as="item()"/>
  <xsl:param name="key" as="xs:anyAtomicType"/>
  <xsl:sequence select=". instance of map(*) and map:size(*) eq 1 and
map:contains($key)"/>
</xsl:function>
```

An overriding template rule can then be written like this:

```
<xsl:template match=".[jlib:is-map-entry(., 'price')]">>...</xsl:template>
```

Writing a shallow-copy template rule for arrays is a little bit trickier because of the absence of XSLT 3.0 instructions for creating arrays: we hit the problem of composability, where XPath constructs such as `array{}` cannot directly invoke XSLT instructions like `<xsl:apply-templates/>`; and we also hit the problem that the only way of iterating over a general array (one whose members can be arbitrary sequences) is to use the higher-order function `array:for-each()`.

One way to write it might be like this:

```
<xsl:template match=".[. instance of array(*)]">
  <xsl:sequence select="array:for-each(., jlib:apply-templates#1)"/>
</xsl:template>

<xsl:function name="jlib:apply-templates">
  <xsl:param name="input"/>
  <xsl:apply-templates select="$input"/>
</xsl:function>
```

But this has the disadvantage that tunnel parameters are not passed through a stylesheet function call; in addition, the current template rule and current mode are lost. We can get around these problems using this more complicated formulation, which uses head-tail recursion:

```
<xsl:template match=".[. instance of array(*)]" mode="#all">
  <xsl:choose>
    <xsl:when test="array:size(.) = 0">
      <xsl:sequence select="[]"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="head" as="item()*">
        <xsl:apply-templates select="array:head(.)" mode="#current"/>
      </xsl:variable>
      <xsl:variable name="tail" as="array(*)">
        <xsl:apply-templates select="array:tail(.)" mode="#current"/>
      </xsl:variable>
    </xsl:otherwise>
  </xsl:choose>
```

```
        </xsl:variable>
        <xsl:sequence select="array:join((array{$head}, $tail))"/>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>
```

The complexity here doesn't really matter greatly, because the code only needs to be written once.

Returning to our specific use case, of updating prices in a product catalog, the main limitation of our solution is that all the update logic is contained in a single template rule, which works for this case but might not work for more complex cases. The match pattern for the template rule matches a map that needs to be changed, and this matching can only consider the content of the map, not the context in which it appears. Moreover, the template body does all the work of creating a replacement map monolithically without further calls on `<xsl:apply-templates>`; it would be possible to make such calls, but the syntax doesn't make it easy.

4. Use Case 2: Hierarchic Inversion

In our second case, we'll look at a structural transformation: changing a JSON structure with information about the students enrolled for each course to its inverse, a structure with information about the courses for which each student is enrolled.

Here is the input dataset:

```
[{
  "faculty": "humanities",
  "courses": [
    {
      "course": "English",
      "students": [
        {
          "first": "Mary",
          "last": "Smith",
          "email": "mary_smith@gmail.com"
        },
        {
          "first": "Ann",
          "last": "Jones",
          "email": "ann_jones@gmail.com"
        }
      ]
    }
  ],
  {
    "course": "History",
```

```
    "students": [
      {
        "first": "Ann",
        "last": "Jones",
        "email": "ann_jones@gmail.com"
      },
      {
        "first": "John",
        "last": "Taylor",
        "email": "john_taylor@gmail.com"
      }
    ]
  }
],
{
  "faculty": "science",
  "courses": [
    {
      "course": "Physics",
      "students": [
        {
          "first": "Anil",
          "last": "Singh",
          "email": "anil_singh@gmail.com"
        },
        {
          "first": "Amisha",
          "last": "Patel",
          "email": "amisha_patel@gmail.com"
        }
      ]
    },
    {
      "course": "Chemistry",
      "students": [
        {
          "first": "John",
          "last": "Taylor",
          "email": "john_taylor@gmail.com"
        },
        {
          "first": "Anil",
          "last": "Singh",
          "email": "anil_singh@gmail.com"
        }
      ]
    }
  ]
}
```

```
    ]
  }
]
}]
```

The goal is to produce a list of students, sorted by last name then first name, each containing a list of courses taken by that student, like this:

```
[
  {
    "email": "ann_jones@gmail.com",
    "courses": [
      "English",
      "History"
    ]
  },
  {
    "email": "amisha_patel@gmail.com",
    "courses": ["Physics"]
  },
  {
    "email": "anil_singh@gmail.com",
    "courses": [
      "Physics",
      "Chemistry"
    ]
  },
  {
    "email": "mary_smith@gmail.com",
    "courses": ["English"]
  },
  {
    "email": "john_taylor@gmail.com",
    "courses": [
      "History",
      "Chemistry"
    ]
  }
]
```

As before, a stylesheet can be written that does this by converting JSON to XML, transforming the XML, and then converting back. The XML representation of our input dataset looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<array xmlns="http://www.w3.org/2005/xpath-functions">
  <map>
    <string key="faculty">humanities</string>
```

```
<array key="courses">
  <map>
    <string key="course">English</string>
    <array key="students">
      <map>
        <string key="first">Mary</string>
        <string key="last">Smith</string>
        <string key="email">mary_smith@gmail.com</string>
      </map>
      <map>
        <string key="first">Ann</string>
        <string key="last">Jones</string>
        <string key="email">ann_jones@gmail.com</string>
      </map>
    </array>
  </map>
  <map>
    <string key="course">History</string>
    <array key="students">
      <map>
        <string key="first">Ann</string>
        <string key="last">Jones</string>
        <string key="email">ann_jones@gmail.com</string>
      </map>
      <map>
        <string key="first">John</string>
        <string key="last">Taylor</string>
        <string key="email">john_taylor@gmail.com</string>
      </map>
    </array>
  </map>
</array>
</map>
<map>
  <string key="faculty">science</string>
  <array key="courses">
    <map>
      <string key="course">Physics</string>
      <array key="students">
        <map>
          <string key="first">Anil</string>
          <string key="last">Singh</string>
          <string key="email">anil_singh@gmail.com</string>
        </map>
        <map>
          <string key="first">Amisha</string>
```

```
        <string key="last">Patel</string>
        <string key="email">amisha_patel@gmail.com</string>
    </map>
</array>
</map>
<map>
    <string key="course">Chemistry</string>
    <array key="students">
        <map>
            <string key="first">John</string>
            <string key="last">Taylor</string>
            <string key="email">john_taylor@gmail.com</string>
        </map>
        <map>
            <string key="first">John</string>
            <string key="last">Taylor</string>
            <string key="email">john_taylor@gmail.com</string>
        </map>
    </array>
</map>
</array>
</map>
</array>
```

Here is the stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0"
  xmlns="http://www.w3.org/2005/xpath-functions"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions"
  expand-text="yes">

  <xsl:param name="input"/>

  <xsl:output method="text"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-xml" select="json-to-xml(unparsed-
text($input))"/>
    <xsl:variable name="transformed-xml" as="element(array)">
      <array>
        <xsl:for-each-group select="$input-as-xml//string[@key='email']" ►
group-by=".">
          <xsl:sort select="../string[@key='last']"/>
          <xsl:sort select="../string[@key='first']"/>
          <map>
```

```
<string key="email">{current-grouping-key()}</string>
<array key="courses">
  <xsl:for-each select="current-group()">
    <string>{../../../../*[@key='course']}</string>
  </xsl:for-each>
</array>
</map>
</xsl:for-each-group>
</array>
</xsl:variable>
<xsl:value-of select="xml-to-json($transformed-xml)"/>
</xsl:template>

</xsl:stylesheet>
```

Is it possible to write this as a transformation on the maps-and-arrays representation of JSON, without converting first to XML? The challenge is again that we can't use the parent axis to find the course associated with each student. Instead, the approach we will use is to flatten the data into a simple sequence of tuples containing the values that we need (last name, first name, email, and course), and then use XSLT grouping on this sequence of tuples. We'll represent the intermediate form as a sequence of maps.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0"
  xmlns="http://www.w3.org/2005/xpath-functions"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions"
  expand-text="yes">

  <xsl:param name="input"/>

  <xsl:output method="json"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="input-as-array" select="json-doc($input) " ▶
as="array(*)"/>
    <xsl:variable name="flattened" as="map(*)*">
      <xsl:for-each select="$input-as-array?*?courses?*">
        <xsl:variable name="course" select="?course"/>
        <xsl:for-each select="?students?*">
          <xsl:map>
            <xsl:map-entry key="'course'" select="$course"/>
            <xsl:map-entry key="'last'" select="?last"/>
            <xsl:map-entry key="'first'" select="?first"/>
            <xsl:map-entry key="'email'" select="?email"/>
          </xsl:map>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:variable>
  </xsl:template>
```

```
        </xsl:for-each>
    </xsl:for-each>
</xsl:variable>
<xsl:variable name="groups" as="map(*)*">
    <xsl:for-each-group select="$flattened" group-by="?email">
        <xsl:sort select="?last"/>
        <xsl:sort select="?first"/>
        <xsl:map>
            <xsl:map-entry key="'email'" select="current-grouping-key()"/>
            <xsl:map-entry key="'courses'" select="array{ current-group()?
course }"/>
        </xsl:map>
    </xsl:for-each-group>
</xsl:variable>
<xsl:sequence select="array{$groups}"/>
</xsl:template>

</xsl:stylesheet>
```

Interestingly, this technique of flattening the data into a sequence of maps (turning it into first normal form) and then rebuilding a hierarchy using XSLT grouping is probably a very general one; it could equally have been used for our first use case.

5. On the Question of Parent Pointers

I'm not sure if it was ever a conscious decision that XML structures should be navigable in all directions (in particular, in the parent/ancestor direction), while JSON structures should only be navigable downwards. It's not only the XDM model (used by XSLT and XPath) that makes this choice; the same divergence of approach applies equally when processing XML or JSON in Javascript. Both XML and JSON are specified primarily in terms of the lexical grammar rather than the tree data model, and it's not obvious from looking at the two grammars why this difference in the tree models should arise.

The ability to navigate upwards (and to a lesser extent, sideways, to preceding and following siblings) clearly has advantages and disadvantages. Without upwards navigation, a transformation process that operates primarily as a recursive tree walk cannot discover the context of leaf nodes (for example, when processing a price, what product does it relate to?), so this information needs to be passed down in the form of parameters. However, the convenience of being able to determine the context of a node comes at a significant price. Most notably, the existence of owner pointers means that a subtree cannot be shared: it is difficult to implement the `xsl:copy-of` instruction without making a physical copy of the affected subtree. This means that each phase of a transformation typically incurs cost proportional to document size. It is difficult to implement iterative transfor-

mations, consisting of small incremental changes to localized parts of the tree. This difficulty was reported a while ago [3] in a project that attempted to use the XSLT rules engine to perform optimization on the XSLT abstract syntax tree; the high performance cost of making small changes to the tree made this infeasible in practice.

The ability to navigate freely in the tree also seems to imply a need to maintain a concept of node identity (whereby two nodes that are independently created differ in identity even if they are otherwise indistinguishable). Node identity also comes at a considerable price, in particular by imbuing the language semantics with subtle side-effects: calling the same function twice with the same arguments does not produce the same result.

The model that has been adopted for JSON, with no node identity and no parent navigation, makes certain kinds of transformation more difficult to express, but it may also make other kinds of transformation (especially the kind alluded to, involving many incremental and localized changes to the tree structure) much more feasible.

6. Conclusions

From these two use cases, we seem to be able to draw the following tentative conclusions:

- Transformation of JSON structures is possible in XSLT 3.0 either by first converting to XML trees, then transforming the XML trees in the traditional way, then transforming back to JSON; or by directly manipulating the maps-and-arrays representation of JSON in the XDM 3.0 data model.
- When transforming the maps-and-arrays representation, the use of traditional rule-based recursive-descent pattern matching is inhibited by the fact that no parent or ancestor axis is available. This problem can be circumvented by first flattening the data – moving data from upper nodes in the hierarchy so that it is held redundantly in leaf nodes.
- The absence of built-in shallow-copy templates for maps and arrays is an irritation, but is not a real problem because these only need to be written once and can be imported from a standard stylesheet module.
- The lack of an instruction, analogous to `<xsl:map>`, for constructing arrays at the XSLT level is a further inconvenience; it means that data constructed at the XSLT level has to be captured in a variable so that the XPath array constructors can be used to create the array.
- Similarly, it would be useful to be able to invoke `<xsl:apply-templates>` as a function, to allow its use within the function supplied to `map:for-each()` or `array:for-each()` – preferably without losing tunnel parameters.

References

- [1] Binary Module 1.0 EXPath Module, 3 December 2013. <http://expath.org/spec/binary>
- [2] Introducing JSON <http://json.org>
- [3] Writing an XSLT Optimizer in XSLT Proc. Extreme Markup Languages, Montreal, 2007. Available at <http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html> and with improved rendition at <http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html>
- [4] XSL Transformations (XSLT) Version 3.0. W3C Candidate Recommendation, 19 November 2015. Ed. Michael Kay. <http://www.w3.org/TR/xslt-30>
- [5] XML Path Language (XPath) 3.1. W3C Candidate Recommendation, 17 December 2015. Ed. Jonathan Robie, Michael Dyck, and Josh Spiegel. <http://www.w3.org/TR/xpath-31>

Jiří Kosek (ed.)

**XML Prague 2016
Conference Proceedings**

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and AH Formatter.

1st edition

Prague 2016

ISBN 978-80-906259-0-7 (pdf)
ISBN 978-80-906259-1-4 (ePub)