XML Prague 2014

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 14–16, 2014

XML Prague 2014 – Conference Proceedings

ISBN 978-80-260-5712-3

MathFlow

The MathML Toolbox Standard

MathFlow™ for leading XML editors

MathFlow for oXygen

MathFlow for FrameMaker

MathFlow for XMetaL

MathFlow for Arbortext

MathFlow Components

The building blocks you need to develop custom solutions requiring equation editing, display, formatting and other advanced math functionality. Design Science's extensive experience working with system integrators, VARs, and ISVs creating custom math solutions has resulted in a flexible and powerful collection of components and APIs covering most development environments.

MathFlow use cases:

Web collaboration Online assessment authoring and delivery, math-enabled blogs, wikis, whiteboards, message boards and forums, live tutoring, virtual classroom and homework help applications, web meeting, instant messaging and chat applications

Web publishing workflows Editorial and production workflows, technical documentation, training and help authoring, elearning authoring tools, accessible content production and delivery

Desktop publishing workflows Editorial and production workflows for journals, textbooks and ebooks, prepress editorial and production services, production of technical documentation, training manuals and help systems

Integrated desktop applications Content conversion software and services, mainstream word processing and presentation apps that target education, science, and technology, scientific analysis, calculation, graphing and simulation, flashcards, tutoring, teaching, scientific visualization

Attend the session
A MathML Progress Report,
Sunday, February 16, 2014,
12:10 PM, presented by
Autumn Cuellar

For more information please visit www.dessci.com

Design Science, Inc. • 140 Pine Avenue, 4th Floor • Long Beach • California • 90802 • USA • 562.432.2920 • 562.432.2857 (fax) • info@dessci.com • www.dessci.com

Design Science
How Science Communicates™

<oXygen/> XML Editor

The Complete XML Development and Authoring Solution

XML Authoring

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

- **Visual XML Authoring**
- **Single Source Publishing**
- **Multiplatform Desktop and Web**
- **Highly Customizable**
- **Supported by Major CMSs**

XML Development

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

- **Intelligent XML Editing**
- **Visual Schema Modeling**
- **XSLT and XQuery Debugging**
- **XML Databases Support**
- **Web Services Analyzer**

www.oxygenxml.com

Availability

<oXygen/> is available in three editions: **<oXygen/> XML Author** for content authors, starting from 349 USD, **<oXygen/> XML Developer** for XML developers, starting from 349 USD and **<oXygen/> XML Editor** for XML developers and content authors, starting from 488 USD.
For Academic/Non-Commercial use **<oXygen/> XML Editor** is available at a discounted price of 98 USD.
All editions can run as a standalone application or as an Eclipse IDE plugin, on Windows 8, Windows 7, Vista, XP, 2000, Mac OS X, Linux and Solaris.

The Internationalization Tag Set (ITS) 2.0

Leading the Multilingual Web to Its Full Potential

ITS 2.0 improves speed and quality throughout the entire multilingual content production cycle. Benefits include:

ITS 2.0 savings in translation time and costs (Source: Study by Linguaserve and Cocomore, with the Spanish Tax Authority and VDMA)

- **Standardized metadata** across platforms and languages
- Reduction in time through **increased efficiency**
- **Cost savings** in translation management
- **Faster and more fine-grained communication** between all actors (e.g., webmasters, translators, localization project manager)
- Easy **format-independent integration of technologies** for automated processing of human language (e.g., machine translation)
- Contribution to **quality assurance** everywhere: content creation, translation, or post editing

ITS 2.0 video channel: http://www.youtube.com/user/W3CITS20

ITS Interest Group: http://www.w3.org/International/its/ig/

Development and Participants

ITS 2.0 was developed by the MultilingualWeb-LT Working Group at the W3C. The MultilingualWeb-LT project received funding from the European Commission (project name LT-Web) through the Seventh Framework Programme (FP7) in the area of Language Technologies. Grant Agreement No. 287815. All contributors listed at: **http://www.w3.org/TR/its20/#acknowledgements**

W3C® · DFKI · Trinity College Dublin · University of Limerick / Ollscoil Luimnigh · cocomore · DCU · ENLASO® · IJS · linguaserve · Software Lucy and Services · Microsoft · MORAVIA · VistaTEC · Vysoká škola ekonomická v Praze (VŠE) · Adobe · Logrus · Tilde · SAP®

Table of Contents

General Information

Date

Friday, February 14th, 2014 (preconference day)
Saturday, February 15th, 2014
Sunday, February 16th, 2014

Location

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

Organizing Committee

Petr Cimprich, *Xyleme*
James Fuller, *MarkLogic*
Vít Janota, *Xyleme*
Jirka Kosek, *xmlguru.cz & University of Economics, Prague*
Pavel Kroh, *pavel-kroh.cz & Macness.com*
Mohamed Zergaoui, *ShareXML.com & Innovimax*

Programm Committee

Robin Berjon, *W3C*
Petr Cimprich, *Xyleme*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *freelance consultant*
Felix Sasaki, *DFKI / W3C Fellow*
John Snelson, *MarkLogic*
Eric van der Vlist, *Dyomedea*
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)

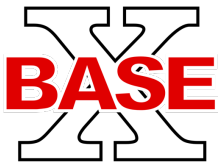
Sponsors

Gold Sponsor

Mark Logic Corporation (http://www.marklogic.com)

Sponsors

Design Science (http://www.dessci.com)
oXygen (http://www.oxygenxml.com)
MultilingualWeb-LT (http://www.w3.org/International/multilingualweb/lt/)
Mercator IT Solutions Ltd (http://www.mercatorit.com)
le-tex publishing services (http://www.le-tex.de/en/)
BaseX (http://basex.org/)
OverStory Consulting Ltd (http://www.overstory.co.uk/)



Preface

This publication contains papers presented during the XML Prague 2014 conference.

In its nineth 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 14–16 February 2014 at the campus of University of Economics in Prague. XML Prague 2014 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 conference starts with a pre-conference day which provides space for various XML community meetings in three parallel tracks. During the weekend classical single-track format is used and papers from it are published in the proceeedings. Additionally, we coordinate, support and provide space for W3C XSLT and XQuery working group meetings collocated with XML Prague.

Last but not least—this year Web has 25th anniversary. It is good time to look both back and forward to the future of the most important computing and communication platform.

We hope that you enjoy XML Prague 2014.

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

Distributed Extensibility: Finally Done Right?

Robin Berjon
W3C
<robin@berjon.com>

Abstract

It has long been a frequent goal of markup — and often document technologies in general — to enable extensibility by arbitrary third parties. However, all attempts to date have fallen largely short of their promises.

XML Namespaces do enjoy a modicum of (much reviled) success in this space, but they have managed this at the cost of a severe limitation in their power, covering only distributed extensibility in naming. Hopes that this would provide a solid foundation atop which richly extensible documents would be built have not come to fruition.

A new contender has recently entered this fray: Web Components. Trying to learn from past mistakes they offer rich, if complex, extensibility functionality, notably in behaviour and styling.

This presentation will look at how Web Components work, what they offer (and have so far declined to offer) in terms of distributed extensibility, and show how they can be put to work to enable innovative behaviour in documents.

1. Introduction

Distributed Extensibility is, at heart, a simple notion. It is the property of languages that can be extended by any party using them, with limited coördination — ideally none.

However the details are gorier than the surface simplicity would suggest. There are many aspects to a language and proponents of a given Distributed Extensibility solution are only rarely clear as to extensions to which aspect or aspects they are enabling. Making the syntax extensible requires a different architectural approach from making the semantics extensible; and yet again different approaches for enabling extensibility for rendering, validation, protocols, or behaviour.

Nowadays when Distributed Extensibility is mentioned people typically think of the debate that surrounded bringing the sort of extensibility supported by XML Namespaces to HTML, but the discussion is much older. In fact, XML Namespaces were themselves the topic of heated debates when they were introduced, and before that the thread can be followed through HyTime or Architectural Forms.

Today the Web platform is in the process of adding the latest instalment in this saga: Web Components. Have we finally found the right solution to this problem, or is this only yet another step in history? If the latter, is it at least a step forward?

2. XML Namespaces

XML Namespaces are a simple mechanism that make it possible to bind prefixes to arbitrary strings, and then use those prefixes on element and attribute names (and sometimes on other language-dependent values) so as to produce names that have a strong guarantee of uniqueness, being qualified names consisting of a `{arbitrary-string, local-name}` tuple. The distributed nature of this mechanism is reinforced by the common convention of using URLs for the arbitrary string, which, since they include a domain name, can be attributed some form of ownership. For our purposes it is sufficient to take that at face value and not enter a discussion of URNs or of the permanence of ownership in domain name attributions. In practice, the guarantees are in fact sufficient, and were it not there are solutions that can enable it.

Namespaces therefore allow anyone to produce their own XML vocabulary without having to coördinate with anyone else who may also be defining an XML vocabulary. What's more, constructs relying on namespaces can be mixed in the same document, without there being a risk of mistaking one for another that may have the same local name.

While that is a useful property, we are however forced to note that it is one of very limited power. In truth, even though one may mix namespaced vocabularies inside a given document, namespaces provide us with no information whatsoever as to how those vocabularies should be processed when mixed.

As soon as one tries to process a mixed namespaces document one needs to be aware of the specific processing rules enforced by a given language, to know for instance if recognised elements inside of containers belonging to an unknown vocabulary need to be processed or ignored (as well as many other such permutations). This issue is compounded by the fact that common schema languages for XML, most notably XML Schema, tend to operate on a default assumption that no extensibility is taking place. More general solutions to this problem, such as NVDL, never became broadly popular.

So XML Namespaces do indeed provide a syntactic hook for distributed extensibility, but nothing more. Styling, behaviour — even semantics — are left as an exercise to the implementer. In practice, while XML Namespaces have shown to indeed be useful in some cases (e.g. writing XSLT stylesheets that don't process what they don't mean to by mistake) their stringent limitations have strongly contributed to disqualifying them as a Distributed Extensibility solution for the Web platform. The requirements of the latter are, in fact, a fair bit more complex than just syntax.

3. Web Components

Over the past 25 years of the Web, there have been multiple attempts at defining a Distributed Extensibility system for the Web platform. This includes both versions of XBL as well as the several languages that were defined (and abandoned) as part of SVG. The author has, in fact, been promoting one or the other at regular intervals at the XML Prague conference.

The latest comer in this illustrious family of attempts is known as Web Components. Web Components essentially started from the ground up on the primitives that serve as integration points for the platform, informed by decades of experience adding new elements to HTML.

Web Components are defined in a series of documents:

- Explainer[1]. A primer document that contains a high level overview of Web Components, as well as an indication of what might soon be added to the system.
- Templates[2]. Recently integrated into HTML, the `template` element makes it possible to deploy parsed but inert (no script processing or resource loading) document fragments that can then be used to define custom elements in terms of existing ones (or more generally can support code reuse as per common practices of templating).
- Shadow DOM[3]. The Shadow DOM is a way of turning the DOM document tree into essentially something that resembles more a forest. Any element can have shadow branches attached to it, recursively. These are invisible to normal DOM traversal, but participate in rendering and behaviour. This makes it possible to support encapsulated extensibility in a shared document.
- Custom Elements[4]. Captures how new elements are defined and registered with the platform so that they can be used, by providing an implementation that extends common constructs.
- Imports[5]. A simple mechanism to import arbitrary groups of dependencies (typically script, style, and templates) into a document. Web Components are expected to be commonly defined in such importable bundles.

At heart, the principle of custom elements in Web Components is articulated on a simple notion: the integration hook for all things related to elements in the Web platform is the `HTMLElement` interface (or its derivates). Be it the parser, `Document.createElement()`, styling, behaviour, or scripting, all interactive aspects of the platform operate through this common integration point.

[1] http://w3c.github.io/webcomponents/explainer/
[2] http://www.w3.org/html/wg/drafts/html/master/scripting-1.html#the-template-element
[3] http://w3c.github.io/webcomponents/spec/shadow/
[4] http://w3c.github.io/webcomponents/spec/custom/
[5] http://w3c.github.io/webcomponents/spec/imports/

Web Components therefore make it possible to declare a binding between a new element name and an implementation for it that inherits from `HTMLElement`, give it a shadow DOM on which templated content can be attached (with styles scoped to the shadow), and the whole thing can be imported in one go, with a variety of additional hooks (for styling and scripting) exposed.

An example may speak more than an abstract description. Let us say that big red buttons are now all the rage on the Web, and that instead of having to constantly reuse `div` elements with a special class or data-* attribute, and ad hoc scripting and styling, we wish to build a nicely reusable big red button component that will be properly encapsulated, and be safely integratable into any HTML document.

First, we define a page that will be the integration point for this component, and in this page we define a template for the element's shadow DOM. That template notably features some CSS styles, and these will be scoped to the shadow, ensuring that they don't conflict with other styles in use on the importing page.

Example 1. The brb-component.html page

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>Big Red Button Component</title>
  </head>
  <body>
    <template id='brb-tmpl'>
      <style>
        .brb {
            background: #f00;
            font-family:  Impact;
            color:  #fff;
            text-align: center;
            cursor: pointer;
        }
      </style>
      <div class='brb'><content></content></div>
    </template>
    <script src='brb-component.js'></script>
  </body>
</html>
```

Note how the `template` element contains a `content` element. When the template is used as the shadow DOM for a `big-red-button` element, any content of the latter will be interleaved where the `content` element appears, thereby making it possible for content from the "real" DOM to be intermeshed with the shadow DOMs.

Then we implement the component itself in JavaScript.

Example 2. The brb-component.js component implementation

```
(function () {
    var cs = document._currentScript || document.currentScript
    ,   doc = cs.ownerDocument
    ;
    var brbProto = Object.create(HTMLElement.prototype, {
        _size: { value: 100, writable: true }
    ,   _div: { writable: true, configurable: true }
    ,   size:   {
            enumerable:     true
        ,   get: function () { return this._size; }
        ,   set: function (val) {
                var s = this._size = parseInt(val || "100", 10)
                ,   st = this._div.style;
                st.width = s + "px";
                st.height = s + "px";
                st["border-radius"] = Math.floor(s / 2) + "px";
                st["font-size"] = Math.floor(s / 3) + "px";
                st["line-height"] = Math.floor(s / 2) + "px";
            }
        }
    ,   createdCallback:    {
            value:  function () {
                var tmpl = ▶
document.importNode(doc.getElementById("brb-tmpl").content, true);
                this._div = tmpl.querySelector("div.brb");
                this.createShadowRoot().appendChild(tmpl);
                this.size = this.getAttribute("size") || 100;
            }
        }
    ,   attributeChangedCallback:   {
            value:  function (name, oldVal, newVal) {
                if (name === "size") this.size = newVal;
            }
        }
    });
    window.BigRedButton = document.registerElement("big-red-button", { ▶
prototype: brbProto });
}());
```

The details may be slightly daunting at first sight, but what is going on is actually simple. We are extending the prototype of `HTMLElement` with our own behaviour, that includes hooks that are called when the element is constructed (`createdCallback`) so that its attributes can be processed and its rendering set up or when its attributes change (`attributeChangedCallback`) so that setting attributes

has the expected, live, effect. Additionally, we are giving the element a `.size` property that can be manipulated directly in script.

The overhead of the above can seem like much for such a simple component; but in truth it does not grow by that much for more complex elements.

With the above we have a fully functional component: we've effectively added a new element to the platform that can now behave exactly as if it were defined as part of the HTML standard.

We can then now just import the component into a regular page, and interact with it as if it were any other element:

Example 3. A page using the Big Red Button component

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    <title>My Rockin' Page</title>
    <link rel='import' href='brb-component.html'>
  </head>
  <body style='background: #fff'>
    <h1>My Rockin' Page</h1>
    <p>I am so hot I spontaneously unbreak symmetry.</p>
    <big-red-button size="100" id='attr'>Attr</big-red-button>
    <big-red-button size="200" id='brb'>Push Me!</big-red-button>
    <script>
      var step = 10
      ,   brb = document.getElementById("brb")
      ,   running = false
      ;
      function animate () {
            brb.size += step;
            if (brb.size >= 300 || brb.size < 200) step = -step;
            if (running) requestAnimationFrame(animate);
      }
      brb.onclick = function () {
          if (running) running = false;
          else {
              running = true;
              requestAnimationFrame(animate);
          }
      };
      document.getElementById("attr").onclick = function () {
          brb.setAttribute("size", brb.size === 200 ? "500" : "200");
      };
    </script>
```

```
  </body>
</html>
```

As we can see above, we simply import the component using a `link` element, just use the `big-red-button` element, and then in the script interact with it in the most natural manner.

Such elements can be nested and they will successfully interact. This is a powerful way of building reusable markup pieces for the Web. Web Components provide a powerful distributed extensibility method of specifying behaviour and styling.

They do, however, have some limits. Semantics can, up to a point, be captured by using ARIA in the shadow DOM but that is restrictive. Going beyond that, only the name can provide key semantic information, and since that name is not namespaced it may not prove unique. Web Components make it a required that custom element names must contain a "-" and the expectation is that people will use their own prefixes that way, e.g. `robin-button`.

While perfectly operational, such ad hoc syntactical extensibility may not satisfy the XML user who wants to be able to obtain an arbitrary document and to extract the information in it without requiring context and without fear of name clashes. Can the two be married?

4. XML Components?

Web Components are defined for an HTML world, where namespaces have no currency. However, if there were interest, it would be possible to have the very same concepts extended to an XML world. This could be done at, in fact, relatively little cost. It would require two things.

First, a way of importing components that works for XML would need to be provided. That is quite simple: all that is required is a `<?xml-import>` processing instruction that can replace HTML Imports in terms of syntax but work the same way.

Second, where today we have `document.registerElement(elName, elDefinition)` that just matches on the element name, we could easily have `document.registerElementNS(elNamespace, elLocalName, elDefinition)` that would work for namespaced documents.

Readers will not be surprised to hear that there are currently no plans for such technology to be defined and incorporated into Web browsers. It is, however, a relatively low-hanging fruit and were someone interested in producing a general-purpose XML processing tool built atop a Web browser it would be relatively easy to wire in the above based on the existing code.

In the meantime, Web Components can also be used to implement XML languages in HTML more easily than is done today. For example, XForms is currently

usually handled by converting it completely to HTML, which in turn can require a certain amount of hacking to keep everything aligned and usable when on the client. With Web Components a two-pronged approach could be deployed. First, implement a very simple, generic transformation that simply converts every element in the XForms namespace into one with no namespace but with its local name prefixed with "xforms-". Then, implement the behaviour of all the xforms-* elements using components. The result will be a lot easier to use, deploy, and mix with other content than existing solutions.

5. Conclusion

Distributed Extensibility is a complex topic and we have not covered its full breadth here. However, we have shown that it is possible to find a solution that would work for all of the syntactical, rendering, and behaviour aspects, which in turn is more than has usually been supported to date.

It is an exaggeration to say that Web Components are Distributed Extensibility "finally done right", but they certainly bring a major piece to the puzzle, what's more a piece that can be made to play nicely with others.

The web needs “XML: The Good Parts”

Robbert Broersma
Frameless
<robbert@frameless.io>

Yolijn van der Kolk
Frameless
<yolijn@frameless.io>

1. Introduction

A new generation of web development frameworks on the rise provide a comprehensive starting point for creating web applications. They combine a basic templating engine, 'two-way data bindings' for form elements, and automatic recalculation of calculated values. The most popular amongst these frameworks are AngularJS[1], Ember.js[2], Knockout[3], and Polymer[4].

Fueled by the development of a suite of new web standards, Polymer is exploring markup-based declarative web applications ("everything is an element") using a simultaneously developed JavaScript implementation of the new Web Components standards. [1]

The increasing adoption of declarative frameworks might finally change the mindset of web developers in ways that make them appreciate the groundwork that standards like XPath, XForms and XSLT have laid.

Using the momentum of Web Components, dubbed the 'declarative renaissance'[5], a rebound of declarative XML technologies is possible when they are re-imagined to fill voids in JavaScript frameworks that are widely deployed today.

What could web development look like with full-fledged templating and query engines?

2. Death of a browser technology

Web development is shifting towards developing declarative applications, and frameworks are evolving slowly towards the same feature sets that have been designed for the XPath, XForms and XSLT standards a long time ago. If these shared

[1] http://angularjs.org/
[2] http://emberjs.com/
[3] http://knockoutjs.com/
[4] http://www.polymer-project.org/
[5] https://www.youtube.com/watch?v=PPJAKLq2ZsY

philosophies are growing more popular, then why aren't the XML technologies themselves appealing to web developers?

XML has become an unspeakable technology, a future averted by WHATWG pragmatists, a cumbersome toolset that has no place on the web.

While few web developers will have actual experience with XML, the image of XML has irreversibly been damaged by the disappointment of a previous generation.

The decline of the technology can be seen in many things: using AJAX definitely doesn't mean you're using XML anymore. [2] XMLHttpRequest is now mostly being used for exchanging JSON, hardly anyone still uses XHTML, the new DOM specification no longer considers attributes to be a Node, [3] and Chrome even intents to fully remove XSLT 1.0 support. [4]

Perhaps one of the main reasons for disappointment is that this family of extensible standards have been the least extensible parts of browsers: implementations cannot be extended to support new features. An XSLT 1.0 processor in the browser cannot be made to support the more recent `<xsl:for-each-group>`. Using variables in DOM XPath evaluators is technically impossible, and so is creating a polyfill for XPath 3.0 functions. If one feature is missing from an XML standard that is essential for your use case, on the web you cannot use the standard at all.

Not all should be considered lost: XML technologies that didn't make into browsers, or have since disappeared, now have a second chance.

We will show what can be done to cut down on complexity, and how we can give developers more flexibility. We will argue why Web Components should not settle for less than the power of XSLT 2 templates, and how users of popular web frameworks are missing out on the good parts of the XML platform.

3. Custom elements reincarnated

The extensible web manifesto pleads to design new web standards with low-level extensibility in mind, opening up possibilities for web applications that were previously reserved for browser makers and browser extensions. [28] The new DOM standard is specifically designed to allow to be influenced and extended using JavaScript. [5] [6]

Web Components aim to transparently provide web applications with full control over rendering and processing unknown elements in HTML.

4. Enable mix and match

With the Shadow DOM [7] 'mix and match' of markup vocabularies has never before been so close to being practical. CSS styles are contained within their respective components, so the theming of one widget library doesn't affect the layout of another. Events and focus transparently handle components as one Node in the DOM tree, even though internally they actually may consist out of dozens of elements.

The future seems bright indeed.

To prevent conflicts with elements that later might be added to the HTML standard, all custom elements must be prefixed using a dash, e.g.: `<custom-menu>`. Supposedly, it will not be long before someone will write a little script called AutoPrefixer that for convenience will implement something equivalent to element namespaces, to reduce typing and improve readability of the code.

```
<polymer-ui-menu>
  <polymer-ui-menu-item icon="settings" label="Settings"></polymer-ui-menu-item>
  <polymer-ui-menu-item icon="dialog" label="Dialog"></polymer-ui-menu-item>
  <polymer-ui-menu-item icon="search" label="Search"></polymer-ui-menu-item>
</polymer-ui-menu>
```

The above could then become:

```
<ui-menu inherit-prefix="polymer-">
  <ui-menu-item icon="settings" label="Settings"></ui-menu-item>
  <ui-menu-item icon="dialog" label="Dialog"></ui-menu-item>
  <ui-menu-item icon="search" label="Search"></ui-menu-item>
</ui-menu>
```

It's a trick that is remarkably similar to how XML works, where using the special `xmlns` attribute essentially defines a default prefix for all elements. Usually however, instead of being a notable convenience, namespace declarations make writing XML documents from scratch a bother. It needn't be though: not using hard-to-remember URLs would make namespaces as simple as writing `import pickle` in another language. Like the HTML doctype declaration was simplified to `<!DOCTYPE html>`, and since relative URLs in `xmlns` aren't being resolved anyway, [8] we might as well start using predictable namespaces values and make our lives easier.

```
<menu xmlns="polymer-ui">
  <menu-item icon="settings" label="Settings"/>
  <menu-item icon="dialog" label="Dialog"/>
  <menu-item icon="search" label="Search"/>
</menu>
```

Doesn't that look handsome, compared to the first example?

The above illustrates another syntax inconvenience of HTML compared to XML. Custom elements in HTML cannot be self closing. Within the possibilities of HTML parsing this seems like a reasonable trade-off, but compared to XML it is rather inconvenient.

Namespaces could also be useful to prevent clashes across libraries providing user interface widgets, and prevent the need for verbose element names that are required to prevent clashes. Libraries that register custom HTML elements should also register those elements in a namespace, providing maximum ease of use as well as flexibility for those who require it.

Clashes between vocabularies already exist. In this example an SVG element, inside the new `<template>` element introduced by Web Components, becomes an HTML element:

```
<template id="vertical-bar">
  <rect ... height="{value}"></rect>
</template>
```

Because of how HTML is parsed, [22] the namespace of the `<rect>` element will be the HTML namespace instead of SVG, breaking the constructed image. This would make implementing templates for the following example impossible, unless XHTML syntax is used and a namespace prefix explicitly defines `<svg:rect>` in the SVG namespace.

```
<bar-graph>
  <vertical-bar value="42"></vertical-bar>
</bar-graph>
```

For namespaced custom elements to work, the Shadow DOM must also define `document.registerElementNS()`. Registering these elements must not require a dash in the element name when the namespace is not the HTML or SVG namespace.

5. Using XHTML imports

For reasons outlined above, Web Components might benefit from using XHTML syntax in external template files. A major reason for not using XML syntax on the web doesn't apply for these files: when a server framework uses string concatenation to create XML instead of tree serialization there is a big risk of well-formedness errors, causing the page not to load at all. HTML imports are most likely hand-coded, and syntax errors can be caught during development just as easily as one would with JavaScript files.

There are significant drawbacks though: ampersands, and the less-than and greater-than signs need to be escaped as `&amp;`, `&lt;` and `&gt;`. When handcoding an HTML or XML file, unlike escaping < and >, it isn't very intuitive to escape ampersands that are part of URL query strings, in the `img src` attribute for example. For `script` blocks it is even a bigger issue, because escaping renders the script completely unreadable.

Overcoming these disadvantages is necessary to make switching to XHTML worthwhile for developers. On the web, XML is best served with error recovery. An initiative like XML Error Recovery [23] is essential to increasing adoption of XML. [21]

6. Templates will drive the web

Templates are at the core of most web pages. Historically templates are processed server side, but increasingly additional content is provided with the servers only sending the raw data and scripts from the web page implementing templates to present it.

Conditionally showing validation warnings next to form inputs. A list of auto-complete suggestions. The latest tweets. Showing the number of unread e-mails between parentheses after "Inbox", or not anymore when the last unread mail is opened.

7. Powered by queries

Data-driven templates need to specify what data sets to iterate trough, and what values to present on the screen. For tabular data like relational databases and CSV files using SQL makes sense, for traversing DOM trees using CSS selectors and the more powerful XPath can be used.

For JSON a path-based language (like XPath, based on UNIX file paths) makes sense as well. [9] In reality however, queries don't traverse data that is parsed from a JSON string, that would guarantee the structure is in fact a tree. So called 'JSON structures' really are regular JavaScript arrays and objects, and can potentially contain cyclic references.

Handlebars is one example of a template engine that traverses over JavaScript structures. It implements the popular Mustache template syntax, [10] self-described as 'logicless templates': that means templates without if/else statements, and no loops. Ironically these are in fact the only things they do offer, most notably these engines are lacking query expressions: there are no comparison operators and no arithmetic operators.

Here's the syntax, in a nutshell: `{{#person}} … {{/person}` is an instruction to loop over a dataset and repeat the contained template, equivalent to `<for-each select="person">` in XSLT. `{{^person}}Your address book is empty{{/person}}` is an if-not statement, and simply `{{name}}` renders text output, like `<value-of select="name"/>` does.

The output of any template is essentially limited to the data that can be targeted using the query language. When an advanced query language is not a significant part of a template engine, complex selecting and filtering must occur in a preprocessing step.

8. Taking advantage of XPath

Most query languages in JavaScript template engines evolve as part of a template engine, and semantics are adapted to support more use cases while earlier versions

are already widely put in production use. Unfortunately absence of a diligent design process will lead to quirky and undesirable behavior, such as the number value `0` evaluating to the boolean `true` in the case of Handlebars. [11]

Query engines for JavaScript structures are often bootstrapped by compilation to JavaScript function bodies, relying on `eval()` to implement the query. This shortcut gives these scripts the advantage of a smaller code size, offloading the workload to the browser. Especially in early versions of such query engines, there is significant risk for script injection [12] and constantly auditing security will remain of the essence.

Reliance on the conversion to JavaScript, and the pressure to keep the codebase small in general, has lead to an unexpressive and makeshift syntax [13] of the expressions that define what templates do. In Mustache selecting the *n*-th item from an array is not even possible: only statically numbered indexes can be selected, like accessing the first item using `list.0`. [14]

The meager semantics of the expression language ensure that the essence of queries will move to helper functions implemented in JavaScript, diluting the foremost value of using a template: separation of concerns. When describing what is to be computed can be defined in a query expression, the essence of the template is captured on the spot.

The semantics of the query language should not be left to chance. Also, developers should not be so restricted in their templates that for what would be a basic comparison in XPath, they need to resort to writing dozens of lines of code in JavaScript. [24]

When so many users of template engines could use more advanced queries, why not take advantage of an existing, fully documented language that too has been designed for tree structures? A language designed around the exact use cases of all these template frameworks, accompanied by tens of thousands of unit tests. We shouldn't hold back the web by waiting the next ten years for makeshift solutions to mature, instead we should be looking over the horizon, starting from a high point that XPath has already reached.

9. To the benefit of everyone

Since JavaScript on the web is mainly used to power user interfaces, it is surprising that Unicode string handling was never part of the language, or built into the core of most libraries. String lengths are inaccurate, which is especially problematic for validating form inputs. Sorting will cause 'Motörhead' to be listed after 'Mott the Hoople', reversing strings can even move diacritics to a different letter: 'daehr̈otoM'. [15]

Being inclusive to a wide range of cultures has always been an important aspect in the design of many W3C standards, because the web is always facing a larger audience than just John Doe. [19] Both developers using XPath and XSLT and the

visitors of their sites enjoy the benefits of having full Unicode support, whether it comes to input validation, sorting, string modifications or number formatting.

Both those who employ a library and the developers of what essentially are user interface libraries face a choice: sacrifice the belief that it is okay to have a 'light-weight' framework, or choose a policy of excluding individuals from all over the world, like from Finland [26] or from Turkey. [27]

Support for Unicode can add considerably to the download size of libraries. [16] When widely used libraries had never considered it an option to not support Unicode, browsers wouldn't have had the luxury to wait until 2013 [17] to deploy the ECMAScript Internationalization API [18] and offer significant bandwidth savings to their users by introducing new Unicode APIs.

Unicode that just works, localized formatting for numbers, currency, dates and time zone corrected times should be within reach to anyone developing a template. Based on thorough research to provide a publishing platform for content around the world, XSLT and XPath are powerful tools that need a facelift but then really need to find their way back to the web.

10. Design for developers

Starting with what web developers already know and expect, XPath should be extended to more conveniently support common use cases. At least all basic JavaScript functions should have readily-available alternatives in XPath. The following string functions need to be introduced, for example: `string-split()`, `string-reverse()`, `string-replace()`, `string-repeat()` and `trim()`. To encourage consistent behavior, all functions that are available in XPath should be available from JavaScript as well.

To align with expectations of JavaScript developers, the else-expression in `if/then/else` should be made optional. The API should allow to add new functions to XPath that can be used without namespace prefix, because namespaced functions... you can't explain that! Of course the math functions must become available without the `math:` prefix too. [25]

On the template front there are idiosyncrasies too, that we needn't expose a new generation of developers to. For example: the `regexp` attribute on the `analyze-string` instruction should not allow attribute value templates by default, because those will obviously break the regular expression syntax: when `{` and `}` need to be escaped, that will render the expression unreadable and make it impossible to just copy over existing patterns.

What would templates look like if they were designed for use in HTML in the first place?

```
<xsl:for-each-group select="blogpost/tag" group-by=".">
  <xsl:sort order="descending"/>
  <li>
    <xsl:value-of select="."/>
```

```
  </li>
</xsl:for-each-group>
```

In HTML the same template would involve a lot less typing:

```
<for-each select="blogpost/tag" sort reversed group-by=".">
  <li>{{.}}</li>
</for-each>
```

This is the kind of template syntax that is easy to get started with and easy to remember, and still offers the power of XSLT.

11. Instantly add interactivity

When developers would declaratively implement the interactive parts of web pages, they can not only move the burden of DOM manipulation to the template engine, but also adding event listeners and cleaning up afterwards. Being relieved of these duties is already hugely being appreciated by users of libraries such as D3.js[6].

Implementing an actual ticking clock in SVG would become child's play:

```
<svg viewBox="0 0 1000 1000" width="150" height="150">
 <g transform="translate(500, 500)">
  <path stroke="black" stroke-width="20" d="M 0 0 L 0 -325"
        transform="rotate({hours-from-dateTime(current-dateTime()) * 30 +
                           minutes-from-dateTime(current-dateTime()) div 2})"/>
  <path stroke="black" stroke-width="20" d="M 0 0 L 0 -450"
        transform="rotate({minutes-from-dateTime(current-dateTime()) * 6})"/>
  <path stroke="red"   stroke-width="5" d="M 0 0 L 0 -450"
        transform=
          "rotate({floor(seconds-from-dateTime(current-dateTime())) * 6})"/>
 </g>
</svg>
```

The most common user interface widgets can easily be implementing declaratively using XPath-driven templates. Paging of search results can be implemented by simply applying templates to `subsequence(result, $page * $pageSize, $pageSize)`. Not only can browsing be implemented by simply updating the `$page` variable, more search results will automatically be rendered when the `$pageSize` preference for the number of items per page changes.

A list of autocomplete suggestions could highlight the search keyword matches in bold using the `analyze-string` templates. Using XForms-like bindings between the data model and HTML form inputs, a visitor of a webshop could narrow down the search results using slider inputs for minimum and maximum price:

```
<input type="range" ref="$search/price/min" min="0" max="{max($results/price)}">
```

[6] http://d3js.org

With no additional code, adjusting the sliders should trigger the removal of the results that don't fall into the selected price range, as well as trigger rendering results that we're previously limited to another page by the `$pageSize` maximum.

Because web pages must not become unresponsive when something unexpected happens, the semantics of some XPath functions and XSLT instructions should be amended for interactive queries not to cause fatal exceptions but instead to fail more gracefully, like returning `NaN` or the empty sequence, or execute a fallback template.

12. Automatically optimizing performance

Using an extensive and expressive query language instead of the one-dimensional queries offered by Mustache, has more advantages than just increased flexibility in handling data structures.

The greatest benefit will come from programmatically analyzing expressions for powering reactive templates: they can be used to determine reasons for recalculation and to automatically create dependency trees so recalculations can be performed in optimal order. Expressions could even be split up into several parts, limiting the impact of updating one variable to recalculation of only affected expressions, always taking the most direct path to correctly updating the template output.

Combine aforementioned logic and parts of the query can also be allowed to return values asynchronously, transparently making use of Promise return values, [20] and finish rendering the dependent templates when the data arrives.

Even without reactive templates there is much to gain. There already is a lot of experience with automatically optimizing templates based on query expressions, such as fusion of tree traversal loops, branch elimination and hoisting (parts of) expressions out of loops.

13. Brought to you by Frameless

Since the summer of 2011 the authors of this essay have been working on a framework to make the declarative web reality: Frameless[7]. Designed as a multistage project, work started out with implementing the latest versions of XSLT and XPath in JavaScript, followed by repurposing the engine as an interactive template engine, while staying compliant with the tens of thousands of test cases from existing test suites.

Frameless is software implemented in JavaScript that aims to bring powerful features from existing web standards together in the browser, and explores ways to combine the declarative real-time data bindings from XForms with the powerful templates of XSLT.

[7] http://frameless.io/

We are happy to announce that all improvements and dreams that have been put forward here are already being enjoyed in the Frameless labs: the future is now.

14. Conclusion

Web Components are a big step forward in providing the web platform with the extensibility that was imagined for XML. Because browsers lacked APIs to extend standards from the XML platform, the technologies failed to live up to expectations of providing distributed extensibility and were abandoned.

Now that declarative programming is being appreciated more by the web development community, a lot of value and experience can be found in these sidetracked browser technologies, although some refurbishing will be in order to go from interesting to desirable.

We must re-imagine existing technologies with a ease of use in mind: a leaner syntax and a powerful JavaScript API. What would XSLT look like with HTML syntax? What would XPath be like when it was designed by JavaScript developers?

Compromised have to be made: on the HTML side namespaced elements must stay first class citizens, requiring changes to the Shadow DOM working draft specification. From the XML side, versions of XForms and XSLT should be distilled that adhere to HTML syntax conventions. XPath must radically improve the extensibility by and the interaction with JavaScript.

There is a future where declarative web applications will make lives of developers much better, but only if together we start learning from the past and stop pretending things don't need to get more complex before they get easier.

Bibliography

[1] W3C, Introduction to Web Components http://www.w3.org/TR/components-intro/

[2] Adaptive Path, Ajax: A New Approach to Web Applications https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php

[3] WHATWG, DOM Standard: Attr interface http://dom.spec.whatwg.org/#interface-attr

[4] Adam Barth, Intent to Deprecate and Remove: XSLT https://groups.google.com/a/chromium.org/forum/#!searchin/blink-dev/xslt/blink-dev/zIg2KC7PyH0

[5] Alex Russel, Real Constructors & WebIDL Last Call http://infrequently.org/2011/10/real-constructors-webidl-last-call

[6] W3C, Web IDL: NamedConstructor http://www.w3.org/TR/WebIDL/#NamedConstructor

[7] W3C, Shadow DOM http://www.w3.org/TR/shadow-dom/

[8] W3C, Plenary Ballot on relative URI References In namespace declarations http://www.w3.org/2000/09/xppa

[9] Stefan Goessner, JSONPath - XPath for JSON http://goessner.net/articles/JsonPath/

[10] Mustache: logic-less templates http://mustache.github.io/

[11] Handlebars: 0 is true https://github.com/wycats/handlebars.js/issues/608

[12] Mario Heiderich, A wiki dedicated to JavaScript MVC security pitfalls https://code.google.com/p/mustache-security

[13] AngularJS filter module http://docs.angularjs.org/api/ng.filter:filter

[14] Mustache: Accessing Array item by index in template https://github.com/janl/mustache.js/issues/158

[15] Mathias Bynens, JavaScript has a Unicode problem http://mathiasbynens.be/notes/javascript-unicode

[16] Steven Levithan, XRegExp Unicode addon http://xregexp.com/plugins/#unicode

[17] Peter Beverloo, Chrome 24 beta http://blog.chromium.org/2012/11/a-web-developers-guide-to-latest-chrome.html

[18] ECMAScript Internationalization API http://www.ecma-international.org/ecma-402/1.0/

[19] W3C, Personal names around the world http://www.w3.org/International/questions/qa-personal-names

[20] Domenic Denicola and Brian Cavalier, Promises/A+ specification http://promisesaplus.com/

[21] Anne van Kesteren, XML5's Story http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[22] WHATWG, HTML: tree construction http://www.whatwg.org/specs/web-apps/current-work/multipage/tree-construction.html

[23] W3C, XML Error Recovery Community Group http://www.w3.org/community/xml-er/

[24] Sharon DiOrio, Angular Filters Beyond OrderBy and LimitTo https://www.youtube.com/watch?v=L4FJ_kuO9Rc&t=4m49s

[25] W3C, XPath: Trigonometric and exponential functions http://www.w3.org/TR/xpath-functions-30/#trigonometry

[26] Oona Räisänen, Wanted: Valid last name https://twitter.com/windyoona/status/427176843158888449

[27] Tex Texin, Internationalization for Turkish: Dotted and Dotless Letter "I" http://www.i18nguy.com/unicode/turkish-i18n.html

[28] The Extensible Web Manifesto http://extensiblewebmanifesto.org/

In Consideration of improvements to XProc

Norman Walsh
MarkLogic Corporation
<norman.walsh@marklogic.com>

Abstract

XProc: An XML Pipeline Language has been gaining adoption steadily, if slowly. Even among its fans, the observation has been made that some aspects of the language frustrate new users. Recently, the XML Processing Model Working Group, the working group at the W3C responsible for the continued development of XProc, has drafted a new requirements document for V.next of the language.

The principle focus of V.next is usability improvements. These range from the relatively simple, obvious syntactic shortcuts, to the relatively audacious, removing entire language features determined to be more trouble than they are worth.

This paper reviews the current state of the art in XProc language design with an eye towards explaining and amplifying the efforts of the working group on the one hand, and on the other, encouraging members of the XML community to voice their concerns.

Keywords: XML, Pipelines

1. Introduction

User and implementor experience with XProc 1.0 has exposed a number of ways in which the XProc language could be improved. The Working Group's focus for V2.0 is on usability improvements.

Broadly speaking, the requirements can be divided into two groups. The first group, necessary enhancements, covers items that address specific, identified shortcomings in the current language. The second group, additional enhancements, covers items that are either deemed lower priority or more experimental.

2. Necessary enhancements

2.1. Allow attribute value templates

Formally, the way that options are passed to a step is through a `p:with-option` mechanism analogous to `xsl:with-param` in XSLT. However, there's a "shortcut" syntax that allows step options to be expressed concisely as options:

```
<p:declare-step version="1.0" name="main"
                xmlns:p="http://www.w3.org/ns/xproc">
  <p:output port="result"/>

  <p:load href="doc.xml"/>

</p:declare-step>
```

The shortcut syntax is foiled whenever the value of the option must be computed by the pipeline. In that case, users must fall back on the formal syntax:

```
<p:declare-step version="1.0" name="main"
                xmlns:exf="http://exproc.org/standard/functions"
                xmlns:p="http://www.w3.org/ns/xproc">
  <p:output port="result"/>

  <p:load>
    <p:with-option name="href" select="resolve-uri('doc.xml', exf:cwd())"/>
  </p:load>

</p:declare-step>
```

This is frustrating especially when the "computation" is a simple variable reference. Allowing option shortcuts to contain XSLT-style attribute value templates (AVTs) would simplify many pipelines. Additionally, allowing AVTs in other places, such as the `href` attribute on `p:document`, are worth considering.

XSLT 3.0 introduces a feature which allows expressions in curly braces to be evaluated in element content. This feature is similar to the facility provided by the `p:template` step. Extending XProc to support curly braces in a manner consistent with XSLT 3.0 will be considered.

2.2. Align with XPath 3.0 technologies

Some will remember that when the XProc 1.0 effort began, it was possible (perhaps naïvely) to imagine that it would finish *before* XPath 2.0 and related specifications.

Supporting XPath 1.0 therefore seemed prudent. Today XPath 1.0 no longer seems relevant; it adds complexity to the specification and is unlikely to be implemented. XPath 1.0 support will be removed from XProc.

XProc 2.0 will be defined on top of the XPath data model.

2.3. Allow arbitrary XDM values in variables

XProc 1.0 restricts the values of variables, options, and parameters to be only strings. This has proven to be an inconvenient limitation. Consistent with defining XProc on top of the XDM, XProc 2.0 will allow variables, options, and parameters to have any XDM value insofar as possible. XProc 2.0 will also allow the required types of variables, options, and parameters to be specified.

2.4. Add explicit flow handling

If a pipeline runs XQuery on a document that has been processed with XInclude, it follows logically that the XInclude step must run before the XQuery step. (Ignoring questions of parallel processing and optimization.)

This is how the majority of steps in a pipeline are coordinated. The processor computes a "dependency graph" and insures that for a step "A", all the steps that it depends on are run before "A" itself. However, there are cases where no such explicit dependency exists.

Consider a step that creates a directory on the filesystem. It might be necessary to run this step before an XSLT step, but there's no obvious way in which the create directory step produces a document that the XSLT step consumes, there's no "data flow dependency" between the two steps. (It is sometimes possible to contrive such a dependency, but it's ugly and distracting.)

There's no standard XProc 1.0 mechanism to express this kind of dependency, but there are implementation-defined extensions in most implementations. XProc 2.0 will provide a standard mechanism.

2.5. Simplify parameters

The parameter mechanism in XProc exists to support XSLT parameters. Consider a pipeline that uses XSLT to apply the DocBook stylesheets:

```
<p:declare-step version="1.0" name="main"
                xmlns:p="http://www.w3.org/ns/xproc">
  <p:input port="source"/>
  <p:output port="result"/>

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

```
</p:declare-step>
```

The first suprising thing about that pipeline is that it's erroneous. There are some complicated syntactic and semantic shortcuts, but in principle the pipeline you have to write is this one:

```
<p:declare-step version="1.0" name="main"
                xmlns:p="http://www.w3.org/ns/xproc">
  <p:input port="source"/>
  <p:input port="parameters" kind="parameter"/>
  <p:output port="result"/>

  <p:xslt>
    <p:input port="stylesheet">
      <p:document href="docbook.xsl"/>
    </p:input>
    <p:input port="parameters">
      <p:pipe step="main" port="parameters"/>
    </p:input>
  </p:xslt>

</p:declare-step>
```

The purpose of all this complexity is to address the problem that arises because the DocBook stylesheets have hundreds of parameters. Asking the pipeline author to enumerate all of them in every case is not practical. At the same time, telling the user running the pipeline that none of them are available seems fairly hostile.

However, experience with parameters in XProc 1.0 reveals that the parameter port mechanism is too complicated. It causes user confusion and introduces syntactic complexity not justified by its function.

Something else must be done.

2.6. Integrate non-XML documents into pipelines

While I'm inclined to see everything in the world through the lens of XML, experience has shown that real-world pipelines often involve non-XML documents. Several workarounds have been invented for special cases. The limitation that V1.0 can only pass XML between steps makes some pipelines difficult, if not impossible, to write.

Providing the ability to allow non-XML documents to flow between steps opens up the possibility of writing simple pipelines to work with images, JSON, Turtle, EPUB, etc.

Conceptually, this is not a difficult change to understand; but the details are not obviously straightforward. It seems uncontroversial that non-XML documents should be able to flow through the identity step and that it is an error to attempt to

add an attribute to one. It's less clear if the string-replace step, for example, can operate on non-XML documents.

Once again, at a conceptual level, replacing strings in non-XML documents is straightforward. But the existing string-replace step uses XPath to identify where the replacement should occur. That's not going to work for non-XML documents. Other non-XML formats, for example JSON, have different addressing mechanisms. Does the string replace operation have to support multiple mechanisms, or are several "flavors" of string replace step necessary?

2.7. Support a variety of syntactic simplifications

XProc 1.0 offers relatively few default behaviors, requiring instead that pipelines specify every construct fully. User experience has demonstrated that this leads to very verbose pipelines and has been a constant source of complaint. XProc 2.0 will introduce a variety of syntactic simplifications as an aid to readability and usability, including but not limited to:

- `<p:pipe step="name"/>` binds to the primary output port of the step named "name".
- `<p:pipe port="secondary"/>` binds to the "secondary" port of the step on which the default readable port occurs.
- `<p:input port="portname" href="..."/>` is a shortcut for a nested `p:document`.
- `<p:input port="portname"/>` is a shortcut for a nested `p:empty`.
- Allow `p:inline` to be optional.
- Allow curly brace expansion in `p:inline` (with an attribute to control whether or not that behavior is enabled)
- Provide a `select` attribute on `p:for-each`/`p:viewport`
- Change all steps with a single non-primary output to have a single primary output
- Consider harmonizing `p:viewport-source` and `p:iteration-source`
- Add an AVT `value` attribute to options, parameters, and variables (to be used instead of `select`)

3. Additional enhancements

3.1. Associate arbitrary metadata with documents

Adding metadata to documents is a natural thing for pipelines to do, either for subsequent use by the pipeline or for eventual output. For example, the serialization options provided in an XSLT stylesheet could be carried forward to the eventual

serialization of the result document by the pipeline. In XProc 1.0, there's no way to maintain that association.

This metadata is a natural place to store information like the media type of (especially) non-XML documents. It may also be possible to store serialization parameters (such as ones specified by an XSLT stylesheet) in the metadata for subsequent use.

3.2. Support steps with a dynamic number of ports

While most steps have a predetermined and static number of inputs and outputs, this is not universally the case. In XProc 1.0, a putative `p:eval` step which could run a dynamically constructed pipeline, for example, suffers from the limitation that the signature of the `p:eval` step usually differs from the signature of the evaluated pipeline.

3.3. Provide improved status information

XProc 1.0 provides scant support for reporting the status of a pipeline and providing aid to users attempt to debug pipelines. Implementation-defined extensions have demonstrated that some additional facilities, such as a `p:message` step, would be an aid to users.

3.4. Provide a mechanism for importing user-defined functions

Experience with user-defined functions in XQuery and XSLT reveals that they can be a powerful addition to the language. Providing some feature that allowed users to extend the vocabulary of functions available in, for example, the test expressions on `p:when` elements would greatly simplify some pipelines.

Such a mechanism might take the form of the ability to load extension functions defined in, for example, XQuery, or it might include adding the ability to define functions in XProc.

3.5. Enhance try/catch

Support for catching errors in XProc 1.0 is limited to a simple `p:try`/`p:catch` pair, which catches and handles all errors uniformly. To align XProc with modern languages, the try/catch mechanism could be extended to support the ability to catch specific errors and possibly with the addition of a “finally” construct.

3.6. Consider using XDM everywhere

In addition to supporting XDM values in variables, options, and parameters, XProc 2.0 might allow XDM values in more places, such as allowing p:for-each to iterate over a sequence of strings or integers.

3.7. Simplify p:viewport and allow it to have multiple outputs

The use of an optional, single p:output binding in p:viewport creates confusion for users. The binding is used both to connect the inner workings of the viewport and as the name of the output port as seen from the outside.

In addition, the fact that viewport can produce only a single result means that for some tasks, multiple passes are required, using a combination of `p:viewport` and `p:for-each`. Consider the task of changing image references in an XHTML document from .svg to .png and generating the sequence of .png images. In XProc 1.0, this requires a `p:viewport` and a `p:for-each`.

Adding an explicit `p:viewport-result` allows us to remove the confusion between the input and the name of the output. Allowing multiple outputs allows us to collapse the `p:viewport` and `p:for-each` logic into a single step.

```
<p:viewport
  name? = NCName
  match = XSLTMatchPattern>
    ((p:viewport-source? &
      p:viewport-result? &
      p:output* &
      p:log?),
     subpipeline)
</p:viewport>
```

The viewport-result connects the transformation inside the viewport back into the source document over which viewport is operating. The transformed document always appears on a port named 'result'. Any other outputs are simply sequences analagous to `p:for-each`. It's a static error to name one of those outputs 'result'.

3.8. Provide a way to specify the base URI of a document

The base URI of a *document* created by the `p:inline` element is the base URI of the `p:inline` element. Specifying an `xml:base` attribute on the root element of the document does not help as that only applies to that element and its decendants.

Additionally, in some pipelines, it is desirable to be able to change the base URI of documents produced by other steps. No convenient mechanism exists in XProc V1.0 to satisfy these requirements.

XSLT 3.0 Streaming for the masses

Abel Braaksma
Abrasoft
<abel@abrasoft.net>

Abstract

Streaming in XSLT is often considered an elite technique that's only understood and mastered by a happy few. However, streaming is everywhere nowadays, with twitter and news feeds, big data processing, log listeners, facebook, or any social media board, forums, streaming media like movies, music etc. Does it really have to be so hard to process streams with XSLT? The answer: it is not. It turns out that if you follow ten simple rules you can already apply streaming to many common scenarios. Once you understand those rules, it is a small step to move forward to more complex scenarios.

This paper first gives you a firm grasp about when to use streaming. And once you need streaming, it introduces step-by-step approach that is easy to memorize and master. And for more complex and corner cases not covered by the ten basic rules, it provides a complete visual flowchart that covers all the complex streaming rules that are otherwise so hard to follow from just the specification text. Bottom line, this paper gives you a head start when you consider using streaming with XSLT 3.0

Keywords: XML, XSLT, XPath, streaming, XSLT-30

1. Disclaimer

This paper discusses the new rules as they are defined in the most recent public Working Draft, which is, as of this writing, currently in Last Call. The latest version can be found at [27]and the version used for this paper is [28]. When this paper refers to XPath, it uses the most recent version of XPath 3.0, which is currently in Proposed Recommendation state [25]. The latest version of XPath 3.0 can be found at [24]. The related XPath Functions and Operators specifications used is the Proposed Recommendation [9], for which [8] holds the latest version.

Since none of these specifications is currently a W3C Recommendation, it is possible that details mentioned in this paper will change in the future, or are removed altogether from the specifications.

2. Errata and updates

After this paper is presented and published at XML Prague 2014[2], I will regularly update it when the XSLT and related standards evolve, or when new insights lead to a better set of common streamability rules. Those updates, and also any errata for this paper can be found at http://exselt.net/papers. From the same site, you can also download several versions of the flowcharts, like a one-page wallpaper version, a condensed multi-page version for printing on A4 or Letter format paper and an online interactive version. With time, I hope to create an online application that can create a flowchart of any streaming XSLT construct.

3. A brief history of streaming

It is hard to find the first occurrences of streaming in computing. The analogy with river flows must have been so obvious, that the word *stream* found its appearance in computing related articles even before the ENIAC[3] was build in the mid 1940s. Using the analogy with rivers, one could argue that the first data streams were Morse code send over the wire. However, the term stream as we know it now, is used to signify the delivery method of data, which differentiates between inherent nonstreaming delivery systems such as books, and CDs, and inherent streaming systems, such as broadcasting radio or television through the ether. When we apply the term streaming, we usually mean that data, which would otherwise be only available in a nonstreaming way, is made available in a streaming way. It was [21] who did the necessary ground-breaking work in the 1920s to split a signal in multiple signals without loss, which he called *multiplexing*. He needed a way to stream analogue data to multiple peers and did his invention while working for the US Army as a Major General. Squier later patented his invention in 1922. [1] describes how he used it to build a device that allowed bringing music to multiple devices at once and dubbed it Muzak, a trademarked term still in use today.

It is his invention that ultimately led to streaming in computing as we know it today. Some of the earliest uses of streams is in the [10] language, in the form of `READ`, `WRITE INPUT TAPE` instructions. But it those "streams" were more comparable to a Morse sequence of bits, because the analogy with broadcasting did not yet apply, that is, there was no way yet to read the input into multiple programs at the same time. But FORTRAN paved the way for other languages such as C which in turn inspired the inventors of the Unix operating system, itself build in C in the early 1970s, to generalize access to I/O by means of standard input, output and error streams. Around the same time we see the term *stream* pop up in early standards

[2] XML Prague 2014, see http://xmlprague.cz.

[3] The ENIAC was the first ever general purpose computer, see, among others, The ENIAC Story: http://ftp.arl.mil/~mike/comphist/eniac-story.html.

about networking, such as [2] in 1968 and the *New Host-Host protocol* by [5] et al in 1970.

Those early advances in streaming techniques and standards for networking and file and memory I/O cleared the way for media streaming in the 1980s for media streaming, as defined and pioneered in the [11] standard by many researchers. The MPEG standard defined a file structure and a delivery method that did not require the knowledge of the whole file, it did not even require the start of the file, hence allowing reading of the file in chunks, without requiring to read or download the whole file at once.

The techniques applied in that standard still form a basis of our understanding of streaming nowadays and the notion of having a limited amount of the stream in memory at any given time, allowing to start in the middle or jump forward (and in MPEG also backward) through the stream and it is that notion that also applies to streaming with XML data: an XML stream has no direct knowledge of its beginning or end and may start in the middle, as long as the streaming provider makes sure that the structure is coherent XML, that is, if you were to start "in the middle", you still require a full [23], with access to the ancestor axis all the way to the root node, but no free-ranging (up and down the stream) access to all preceding or all forward nodes. It is this limitation that requires a strict set of rules to allow processing of such streaming XML with XSLT, that are explained in the next sections.

4. Why streaming?

Not all data manipulation scenarios require streaming. Where and when you should switch to a streaming scenario may depend on many factors. This section briefly describes what factors may come into play.

4.1. Input data size

The most obvious use-case for using a streaming aproach is the size of data. As a rule of thumb, if the data that needs to be read does not fit in memory, or if the data that needs to be written does not fit in memory, or both, you would choose a streaming approach. In fact, streaming is your only option in these use-cases. Whether or not data fits in memory depends on many factors. Your processor of choice, the XML reader that creates the XDM in memory, the platform it is run on and the capabilities of your physical computing environment, like bitness, memory or a fast swap disk, together make it either possible or impossible to process a given large XML file with XSLT.

Often it will be clear from the start whether the document is too large to fit into memory, but in the case of doubt, you can also calculate it. The following example calculation may need different constants for your processor of choice. Suppose you have an 8GB RAM computer, but you're stuck with a 32-bit processor, your maxim-

um available memory will be 2, sometimes 3GB. If your input is 400MB in size, and the chosen XML reader will require three to four times the size of the input XML and the required output is estimated to be half the size of the input, the maximum total required memory is 400 x 4 + 200 x 4 = 2,400MB. Assuming the operating system, other programs and the processor itself also require memory, this relatively small amount of input XML needs to be processed in a streaming way to prevent out of memory issues.

4.2. Streamed input

But size is not the only factor in play. Not all data can always be read at once. Suppose you have a logging feed from your website and your stylesheet wants to send a report on each serious error that is reported, there's no way you can read all the data at once, simply because it is not yet available. In such scenarios, where the input is already streamed and your stylesheet needs to continue processing virtually indefinitely, you should use a streamed processing approach. In this scenario, you could consider the stylesheet to operate as a certain event dispatcher: when the event of an error occurs, it will do something.

4.3. Output streaming

It is important to distinguish between input streaming and output streaming. The XSLT specification is very thorough when it comes to input streaming and all rules that allow streamability are written in such a way that they also allow output streaming, but whether or not a processor actually does output streaming is not required by the specification. Before you begin using streaming, you should check your processor whether it can do output streaming, or whether it will keep the output in memory until the processing is completed. If so, you should not use that processor for scenarios that run indefinitely, where intermediate output is a requirement.

Some scenarios require little or no input, but the output is too large to fit in memory. Suppose you want to calculate all the possible permutations of a given input string. The output in such case can be huge, and then again, streaming is your best option (on a side note, the XSLT 3.0 specification says little about output buffering, but it requires that it runs in constant memory when streaming is enabled, which also applies to the writing result sets).

Output streaming has another compelling use-case: chained, or piped streaming, which can be applied to scenarios where multiple passes are required over the streamed input document. For instance, suppose you want to sort an XML file using streaming. It is not possible to use xsl:sort with streamable nodes (the reason is that it requires multiple back and forth readings to do the sorting), but several algorithms exist that allow streamed sorting, which usually requires the output of a first, partial

sort, to be applied again as input to a new pass, until the full stream is sorted. For these and related scenarios to work, your processor or your XML Pipeline [26] environment must have a mechanism of using the streamed output of one pass as streamed input for the next pass.

4.4. Unparsed text input streaming

This paper discusses streaming of XML, but XPath 3.0 introduces a new function, `fn:unparsed-text-lines`[4], which takes an external resource as input and parses it line by line. The original intend of that function was to allow unparsed data to be streamed, however the Working Group at some point decided to not formalize this requirement. When your intend is to do streaming of unparsed input, you should check the capabilities of your processor to find out whether it can do streaming using this function, or whether you would requite an extension function.

At the moment of this writing, the XSLT 3.0 specification does not provide an additional function for doing streaming of unparsed text input.

5. XSLT 3.0 streaming basics

There is one golden rule while doing streaming with XSLT: each instruction can have a maximum of one downward expressions[5]. Remember this rule. Memorize it. It is important, as we will come back to this rule again and again.

5.1. Guaranteed streamability

Streaming is all about *guaranteed streamability*[6]. Knowing that your stylesheet rules, when they apply to streaming, are guaranteed streamable is important, because it means it will be processed in a streaming way on every streaming processor, that is, on any processor that supports the streaming feature. It is possible that individual vendors have created ways to allow a broader group of constructs or expressions to be streamable, but that is out of the scope of this paper. Guaranteed streamabitlity is well defined in the XSLT 3.0 specification, but the rules are complex. This paper shows an alternative approach to the rules, which will fit most scenarios and are easier to follow.

You might wonder why you would bother with guaranteed streamability. After all, you could just write the stylesheet the way you always did, initialize streaming

[4] See section 14.8.6 of XPath Functions and Operators: http://www.w3.org/TR/xpath-functions-30/#func-unparsed-text-lines.

[5] The specification does not use this terminology, instead it uses the term *consuming expression*, which *consumes* the input tree. Iin practice it is easier to visualize what happens by considering the term *downward expression* which was often used in informal discussions during meetings.

[6] See section 19.10 in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#dt-guaranteed-streamable.

as described in one of the following sections and hope for the processor to do its best and process as much as possible using streaming. After all, wasn't XSLT written with streamability in mind in the first place? The answer to this question is yes and no. In the course of defining the standard, we've found that many approaches towards streamability exist. Just like watching a movie on Youtube, where you can scroll back to a previous point, there have been papers in the past describing streamability of processing XML that can stream both up and down the tree. Given such an approach, it would theoretically be possible to use any expression with streaming.

However, the working group has decided that such a broad definition of streaming was not feasible. In fact, the complexity of supporting different kinds of streaming and formalizing it would prove to be too hard in practice, let alone hoping for implementers to implement such complexity in their processors. By choosing for a limited approach, namely, forward-only streaming (give or take a few exceptions, which we will get to in the next sections), the result is already quite complex, but doable for implementers and, given some help with tutorials here and there, understandable for programmers.

This does not preclude implementers from using smarter algorithms to allow more complex constructs to be streamable, or to even allow forward and backward streaming. The term *guaranteed streamability* is meant to denote the subset of rules that is formalized in the specification and that all processors that claim to support streaming, must adopt. As a consequence of this, it is a necessary evil for programmers learning streaming to also adopt and understand *guaranteed streamability analysis*[7], at least to the level where it can provide you with enough tools to create usable stylesheets for your scenarios.

5.2. How to find out whether your processor supports streamability

Before we even attempt to use streaming, we should find out whether our processor supports it. This can be done similarly to how you would check for the supported XSLT version, or whether the processor supports schema-awareness.

If you are uncertain whether your processor supports streaming, or whether the license for your processor entitles you to using it, you can use the new streaming system property `xsl:supports-streaming`[8], which takes the value `yes` or `no` depending on whether the processor supports it. For instance, the following snippet will output "Supports streaming" or "Does not support streaming" depending on your processor's capabilities:

[7] See section 19 of XSL Transformations 3.0, which deals with streamability analysis: http://www.w3.org/TR/xslt-30/#streamability.

[8] See section 20.3.4 in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#function-system-property.

```
<xsl:template name="xsl:initial-template">
      <xsl:value-of select="
          if(system-property('xsl:supports-streaming') eq 'yes')
          then 'Supports streaming'
          else 'Does not support streaming'" />
</xsl:template>
```

Note: this snippet uses the new special name `xsl:initial-template`[9] as name for the template, which means nothing more that this template is the entry point for the transformation, regardless whether an input document is provided or not.

5.2.1. How non-streaming processors deal with streamability

What happens when you process a stylesheet with a processor that does not support streaming, and the stylesheet itself contains templates that are defined in a streamable mode (see next section), or they contain `xsl:stream`[10] instructions? The answer is simple: when a processor does not support streaming, it should try its best to process the stylesheet in a normal way, as if each streaming construct was non-streaming. It will simply ignore the guaranteed streamability analysis required by streaming processors.

If your stylesheet requires streaming, it is very well possible that a non-streaming processor will blow itself up, for instance because it runs out of memory. You can require streamed processing if you feel that is a requirement for your stylesheet by raising an error (a new feature of XPath 3.0), by using an `xsl:message` with `terminate="yes"` instruction, by stopping processing in the initial template, by using the new `xsl:assert`[11] instruction or by using an `xsl:use-when` expression, all with a test on the system property `xsl:supports-streaming`.

Here is one way to do that in the initial template:

```
<xsl:template name="xsl:initial-template">
    <xsl:choose>
        <xsl:when test="system-property('supports-streaming') eq 'yes'">
            <xsl:stream href="streamable-source.xml">
                 <!-- streamable instructions -->
            </xsl:stream>
        </xsl:when>
        <xsl:otherwise>
            <xsl:message terminate="yes">
                Sorry, this stylesheet requires
```

[9] See section 10.1 in XSL Transformation 3.0: http://www.w3.org/TR/xslt-30/ [http://www.w3.org/TR/xslt-30] (search the document for `xsl:initial-template`).

[10] See section 18.1 of XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#element-stream.

[11] Assertions are a new and powerful feature, that help specifically in testing scenarios. Whether or not an assertion is run can be switched on or off when initiating the transformation. For more information, see XSL Transformations 3.0, section 22.2: http://www.w3.org/TR/xslt-30/#assertions

```
                a streaming processor
            </xsl:message>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

5.3. How to initiate streaming

There are several ways to tell the processor that you want to use streaming. You can set the default mode to streaming, which means that the processor will have to process the principal input document through streaming, you can tell the processor to use streaming on a specific mode, you can merge in a streaming way or you can use the new `xsl:stream` instruction.

5.3.1. Using xsl:stream to initiate streaming

Most commonly, streaming is initiated with this new `xsl:stream` instruction, which takes an uri in the `href` attribute that is resolved to the document you want to stream. The `xsl:stream` instruction works the same as the `fn:doc` function in regards to uri resolution, except for the fact that it is not stable. That means, for multiple invocations it can return different nodes, whereas the `fn:doc` function would always return the same document for the same uri. This is an important difference and a consequence of streamability: the input document cannot be read in memory all at once, hence it requires multiple read-requests to the source. How the stream is read and how the nodes are given to the XSLT processor are not part of the XSLT specification. Typically, a processor will have a streaming XDM implementation, which is in principle the same as a normal XDM implementation, except that it will only have a limited set of nodes and a limited set of accessors at any given time.

5.3.2. Using xsl:mode to initiate streaming

Another way to initiate streaming is by using the new `xsl:mode`[12] declaration and setting its streamable attribute to `yes`. The `xsl:mode` declaration defines the properties in which a certain mode can operate, among other things, whether or not a mode can be used within streamable processing.

When you have an `xsl:mode` declaration without the `name` attribute present, the declaration applies to the default default mode (the default mode can be specifically selected with `xsl:apply-templates` and the `mode` attribute set to the special name

[12]Modes can now be specified declaratively, with features as streamable, whether to ignore or copy non-matched nodes, to warn on non-matched or doubly matched nodes, or whether typing is strict or lax etc. See section 6.6.1 on Declaring Modes in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#declaring-modes

`#default`, the default of the defaults, if none is specified, is the unnamed mode). If you define `streamable="yes"` to the default mode, the processor *must* process the default input document using streaming.

When you declare a named mode to be streamable, and the initial mode of the stylesheet is set to that named mode, the processor *must* process the default input document using streaming as well. However, a streamable mode does not always mean that any nodes going through that mode are streamed. This happens for instance when you would apply the result of an `fn:doc` function to that mode. Because the `fn:doc` function returns a non-streamed stable document, the whole document is already in memory and the streamable property of the mode does not have any effect. However, processors are allowed to optimize processing other input using streaming, as long as it doesn't affect the way the stylesheet would have operated without streaming.

Any mode that is declared with `streamable="yes"` is required to be guaranteed streamable. Processors are not required to raise an error when the mode is not guaranteed streamable, but they are required to inform the user if it is not, for instance, by showing a warning (which in turn, the processor is allowed to switch off at user option).

5.3.3. Using xsl:merge to initiate streaming

The new `xsl:merge` is different from other instructions in that it allows the user to inform the processor that you want the documents-to-be-merged to be merged using streaming. The current state of the specification however contains a bug[13] that conflicts with stability and streaming rules, but once the bug is fixed, the third option to initiate streaming is by means of the `xsl:merge` instruction.

To initiate merged streaming, you specify the `streamable` attribute of `xsl:merge-source`[14], which is a child instruction of the `xsl:merge` instruction, and set it to `yes`. It is possible mix both streamed and non-streamed sources together, in other words, you are not required to set all `xsl:merge-source` instructions to the same streamable attribute value.

This type of streaming is very specific, as it is only intended to be used when merging multiple sources into one. This is very hard to do using just the `xsl:stream` instruction and the strict streaming rules, not in the least because you are not allowed to move streamed nodes around (as explained in a later section in this document). This paper will not go into the details of streamed merging, as it is considered an advance concept. However, the flowcharts later in this paper do have the `xsl:merge`

[13] This particular bug was raised on the public bugzilla against the latest working draft, see: https://www.w3.org/Bugs/Public/show_bug.cgi?id=24343

[14] For more info on streamable merging, see section 15.4 of XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#streamable-merging

instruction, which should give you enough information to start doing streamable merging if your use-case requires so.

5.3.4. Using fn:unparsed-text-lines to initiate unparsed text streaming

After the three ways to initiate streaming for XML documents (or any document available as a streamable XDM), you may wonder whether it is possible to stream unparsed text input. Unfortunately, the answer is both *yes* and *no* as we already saw in section 3.4. You can use `fn:unparsed-text-lines` the same way you are used to use `fn:unparsed-text`. If you want the largest chance on your processor optimizing it for streaming, either apply templates on it (yes, in XSLT 3.0 you can apply templates on other things than nodes[15]) or use an `xsl:for-each`. Since each context item will be an atomic value and since there is no way to navigate from one line to another, the rules discussed in this paper do not apply. Perhaps it is a good idea, however, to set the modes you use for parsing unparsed text to `streamable="yes"`, just in case, hoping the processor will understand the hint. But remember, the processor is *not required* to apply streaming processing of unparsed text, so it is probably a good idea to simply try it out with your processor, or to check your processor's manual.

There is still discussion going on both inside the working group as outside the working group[16] as to whether we should formalize streaming of unparsed input. It is obvious from the discussions that many people expect `fn:unparsed-text-lines` to behave in a streaming way, but the specification says nothing to confirm that. Well, perhaps apart from a note in the current XSLT 3.0 specification that says that it facilitates streaming[17], and a large note in an earlier draft of the specification[18], quote:

> *"This function has been added in XSLT 2.1 for three reasons: to do the line splitting in a way consistent with the rules applied during XML parsing; to do it without*

[15] Matching non-nodes is done with the new `.[Expr]` syntax, which is called a predicate pattern. It contains an expression that returns a boolean and takes as input the current item, which can be a node, any atomic type, a map or even a function item. See The Meaning of a Pattern in section 5.6.3 of XSL Transformations 3.0 for a more detailed description or predicate patterns: http://www.w3.org/TR/xslt-30/#pattern-semantics

[16] In September 17, 2013, Wendell Piez wrote a message to the XSL Mailing List of Mulberry Technologies Inc., in which he laid out a common scenario of parsing large CSV files, grouping them and outputting them as XML. See: http://xsl-list.markmail.org/thread/pwuzpcvdoi7eam4h. On the same list, more discussions have arisen about using unparsed text in a streamable way. Quite recent, Dimitre Novatchev wrote "it seems not obvious how cases involving very large unparsed text files have been addressed", in answer to a discussion on streaming unparsed text, see: http://markmail.org/message/jcvjh54fyuav3y4h.

[17] See section J.2 Changes in Other Related Specifications: http://www.w3.org/TR/xslt-30/#xpath-changes-since-2.0

[18] See section 19.2.2, The unparsed-text-lines function, of the June 2010 Working Draft of XSLT 2.1: http://www.w3.org/TR/2010/WD-xslt-21-20100511/#unparsed-text-lines

recourse to regular expressions (which is likely to be more efficient), and to make it easier for processors to read the input file line by line, which is likely to use less memory."

But none of these notes have made it to a formal section of the current specification yet.

5.3.5. Other means of initiating or using streaming

There are two other instructions that have a `streamable` attribute. Those instructions are `xsl:accumulator` and `xsl:attribute-set`. The meaning of using `streamable="yes"` on these two instructions is slightly different than the meaning on `xsl:merge-source` and `xsl:mode`. For modes and merging, setting streamable to `yes` actually forces the processor to take a streaming approach for the initial source document or the input merge sources. In the case of `xsl:mode` it also allows the mode to be used when called or applied to from any streaming scenario, for instance from under an `xsl:stream` instruction, but when the mode is used without streaming, it acts as any other mode. The same applies to accumulators and attribute-sets. If you intend to use them with streaming, you must specify that on their declaration. However, they themselves do not instantiate streaming. Conversely, when `streamable="yes"` is not specified on either an attribute-set or an accumulator, you are not allowed to use them from a streamed template or instruction.

As with any other feature, it is possible that processors extend the capabilities provided by the standard and create their own extension functions, instructions or declarations that can initiate streaming. For example, the EXPath HTTP Client [13], as implemented in Java as an extension for Saxon by Florent George [14] supports a limited amount of streaming [19]. If you are interested in using this EXPath [6] extension, James R. Fuller has provided a good tutorial in [15] for working with web services using XSLT and this EXPath extension. Another one worth mentioning is the Saxon `saxon:stream` extension, which was dubbed *Burst mode streaming* [4].

Neither these, nor other extensions will be covered in this paper, as the focus of this paper is on the new streaming facitilities provided by the XSLT 3.0 specification and most of these extensions were provided because earlier XSLT standards did not support streaming out of the box. With the new streaming features becoming an integral part of new streaming processors such as [7] or existing processors such as [20], legacy streaming extensions, which typically only work on one processor, are no longer necessary.

[19] A brief mention of this support was on the public EXPath mailing list by Florent George, see: http://lists.w3.org/Archives/Public/public-expath/2013Oct/0010.html

6. Rules of thumb for streaming with XSLT 3.0

This section describes rules that can be deducted from the complex rules as outlined in section 19 of the XSLT 3.0 specification. The rules are deliberately simple and easy to follow. When the rules do not work for your situation you may well need a more complex solution. It is the intend of these rules to cover most of the common scenario's found when processing streaming XML with XSLT, but there is no intention to be complete. For that purpose, see the next section, where a set of more or less complex flowcharts can be used to analyze any streaming scenario for guaranteed streamability assessments.

The constructs in the following sections, when not specifically specified otherwise, assume that they are an immediate child of a streamable template[20] or an immediate child of the `xsl:stream` instruction. For instance, if the text has an example saying that the following construct is streamable:

```
<xsl:for-each select="foo">
    <xsl:value-of select="@bar" />
</xsl:for-each>
```

it means that it is *guaranteed streamable* when used within a streamable template rule such as this:

```
<xsl:mode streamable="yes" />
<xsl:template match="somenode">
    <xsl:for-each select="foo">
        <xsl:value-of select="@bar" />
    </xsl:for-each>
</xsl:template>
```

In this example, the `xsl:mode` declares the unnamed mode streamable, hence any template rule that doesn't specify another mode, must also be streamable. The whole example above is *guaranteed streamable* and uses only rule #1 and rule #2. The pattern in a streaming template rule is not as free as a pattern in a non-streaming pattern. See rule number 3 for streamable patterns and how to construct them.

In those examples where an `xsl:template` is used, a further unspecified attribute `mode="streaming"` assumes that an `xsl:mode` declaration exists with `name="streaming"` and `streamable="yes"`. Where a template rule is used with `mode="non-streaming"` this means that again a mode declaration is assumed, this time however with `name="non-streaming"` and `streamable="no"`.

[20] A streamable template is a template that is in the scope of a mode with `streamable="yes"` as outlined earlier in this paper. When any mode is streamable and your template uses the `#all` special mode name, that template rule must also be guaranteed streamable.

6.1. Rule 1: each template rule can have a maximum of one downward expression

Each template rule that is in a streamable mode can contain a maximum of one downward expression[21] in all of its immediate children. This is the most important rule to remember. A downward expression can be a child selection expression, a descendant-or-self expression or any of these with a motionless filter expression (see Rule 3).

The effect of this rule is that a streaming stylesheet will most typically contain many very small template rules, that move through the streamed input document one tiny step at the time. Streaming is all about thinking in one direction: down. Make sure to craft your rules in such a way that you never have to look back, because once you have visited a node, you cannot visit it again.

The following trivial example selects `author` elements from a book. It is guaranteed streamable because it has at most one downward expression in the first child of the template rule.

```
<xsl:template match="book" mode="streaming">
    <xsl:copy-of select="author" />
</xsl:template>
```

The following example is slightly more complex, but by just glancing over it, it is clear that the first template has one downward child select expression and the second template concatenates some literals with the value of the `name` element.

```
<xsl:template match="book" mode="streaming">
    <xsl:apply-templates select="author" mode="#current"/>
</xsl:template>

<xsl:template match="author" mode="streaming">
    <xsl:value-of select="'Author: ' || name || '&#xA;'" />
</xsl:template>
```

It is allowed to have more than one construct as immediate children under a template rule, as long as at most one of these children consumes the input stream, that is, contains a downward expression. Of course, if there is no downward expression at all, it is also guaranteed streamable.

This simple and basic rule is easy to get wrong, however. For example, consider changing the content of the second template as follows:

```
<xsl:value-of select="'Author: ' || firstname || ' ' || surname" />
```

[21] The specification talks about at most one *consuming* expression. In the flowchart later in this paper, the term *consuming* is used instead of *downward expression*. But most consuming expressions become consuming because they are a downward expression. Later rules and the flowchart help define the term *consuming* to a greater extend.

Surely, this can be streamable, right? Yes, you *might* be right if you consider this to be streamable. The `select` expression of `xsl:value-of` contains two downward expressions. Even though it is only one expression, you should always split your expression into single steps, in this case the steps are the first literal expression `'Author: '`, the child-select expression `firstname` the second literal expression `' '` and the second child-select expression `surname`.

If this were to be processed using streaming, ignoring the rules for a moment, the processor would first find the element `firstname`, reads and parses its content, and positions the read pointer at the end of that element; then it will search for the element `surname`, reads and parses its content and positions the read pointer at the end of that element (effectively, the closing tag). This *could* work, if you knew beforehand that `surname` always comes after `firstname`. However, a processor cannot possibly know that.

Even in the even that the processor would know the order of the elements, for instance in the presence of a schema, it still validates a principle of XSLT processing and that is, that the order of processing does not matter. In other words, if the processor were to decide in all its wisdom that finding `surname` first and then `firstname`, it would be no problem in a non-streaming situation, but in a streaming environment, there is no way the processor can go back. Hence it is vital that we are lenient with the processor and think along.

So how would we fix our stylesheet? As mentioned in the beginning of this section, a stylesheet that uses streaming typically contains many very small template rules. Assume that in this particular situation, as an author of the stylesheet we know that the order of `firstname` and `surname` is always the same and in that respective order, we can write the whole stylesheet as follows:

```
<xsl:mode on-no-match="shallow-skip"
          name="streaming"
          streamable="yes" />

<xsl:template match="book" mode="streaming">
    <xsl:apply-templates select="author" mode="#current"/>
</xsl:template>

<xsl:template match="author" mode="streaming">
    <xsl:apply-templates select="*" mode="#current"/>
</xsl:template>

<xsl:template match="firstname" mode="streaming">
    <xsl:value-of select="'Author: ' || ." />
</xsl:template>

<xsl:template match="surname" mode="streaming">
```

```
    <xsl:value-of select="' ' || . || '&#xA;'" />
</xsl:template>
```

This may look much more like the older XSLT 1.0 style of writing stylesheets, where in the absence of more advanced constructs, it was not uncoming to have many short template rules. In a way, using streaming feels a bit like going back to XSLT 1.0 ways of doing things, however, many powerful functions from XSLT 3.0 and XPath 3.0 are available that were not available at the time of XSLT 1.0.

Note, as a convenience, I added the new `on-no-match` attribute to the mode declaration, which was covered in depth in an earlier paper from my hand, Lazy processing of XML in XSLT for Big Data [3]. It is a shortcut for removing the nodes that we are not interested in from the output stream.

You may wonder how this particular use-case is supposed to be written if the order of `surname` and `firstname` were the other way around. And indeed, it is not trivial at first how to do this, but in some of the following rules we will come back to this example and provide alternative streamable approaches for this scenario.

6.1.1. What are downward expressions?

So far, this section assumed you would understand the concept of downward expressions, and more concretely, the concept of *one downward expression*. But it may not be so trivial as it looks. For instance, the expression `following::foo` is clearly a downward expression, but is not considered streamable at this time. The reason for it not being streamable is that with multiple templates, it is very hard to write rules that prevent a matching template to visit nodes multiple times. Currently, the streamability analysis is explicitly written to prevent analysis to go over the boundaries of the current construct, which makes it possible to statically analyze whether any given construct is streamable or not.

To understand why it is indeed not streamable, remember that we cannot visit a node twice, and have a look at this example:

```
<xsl:template match="book" mode="streamable">
    <xsl:apply-templates
        mode="#current"
        select="following-sibling::book[1]" />
</xsl:template>
```

This example clearly goes over each node only once. It doesn't actually do anything (let's assume it outputs the `@name` attribute), but it is a typical XSLT pattern for retaining state, in fact, for a long time it was even considered a streamable way of keeping a running total, see [18][22]. However, suppose we write this example slightly differently. In fact, let us introduce a common error seen with many beginning XSLT students:

[22] In particalur, the section in the mentioned paper called "Streaming with retained state".

```
<xsl:template match="book" mode="streamable">
    <xsl:apply-templates
        mode="#current"
        select="following-sibling::book" />
</xsl:template>
```

It is not always obvious to anyone starting out with XSLT what such an example does. It goes over each `book` element many times. In fact, the first `book` element is visited once, the second twice, the third four times, the fourth eight times etc, which is not compatible with streaming. While at first it seems trivial that the following-sibling axis is streamable, in practice it is very hard to come up with a rule that allows usage of the following-sibling axis, but prevents situations like the one above, which is legal XSLT, but illegal in streaming.

While the following and following-sibling axes may not have seem obvious, the other axes that are not streamable in any expression that operates on a streamed node, `preceding::` and `preceding::sibling::`, are much easier to understand: they force the processor to look back, and looking back is not allowed. These four axes together form the group of *intrinsicly non-streamable axis steps*.

6.2. Rule 2: each construct can have a maximum of one downward expression

This rule extends the previous rule and introduces the concept of a *construct*. In streaming, a construct is an instruction, a sub-instruction (such as `xsl:when`), an expression, a sub-expression (in `a | b` there are two subexpressions `a` and `b`) and declarations, if they are of influence to streamability. Typically, many constructs have a sequence constructor (examples are `xsl:for-each`, `xsl:sequence`, `xsl:with-param`, `xsl:template`, `xsl:attribute` etc). This sequence constructor is a very powerful means in XSLT and it allows us to nest instructions inside other instructions.

The essence of this rule is that any sequence constructor, no matter how deeply nested, may contain at most one downward expression in its immediate child. However, if that child has itself a sequence constructor, it is also allowed to have one downward expression. Here is an example:

```
<xsl:for-each select="author">
    <xsl:for-each select="books/book">
        <xsl:apply-templates select="summary" mode="#current" />
    </xsl:for-each>
</xsl:for-each>
```

This example, when placed inside a streamble template, in total contains three downward expressions: `author`, `books/book` and `summary`. However, each immediate child contains at most one downward expression, which makes it guaranteed

streamable. Once the read pointer is at the first `author`, the processor knows that it needs to process all children `books` and its children `book` (typically, a processor will go over all descendants and test whether the pattern `books/book` fits, rooted at the context node `author`); finally it will select the children of `book` that match the nametest `summary` and applies templates to it. Once done, it goes on to the next matching `book` until all books are processed. Once done with that, it will go on to the next matching `author`, until all authors are processed.

This rule does not apply to all constructs. In particular, it only applies to constructs that change focus, like `xsl:for-each`, `xsl:iterate`, `xsl:for-each-group`, `xsl:template`. It also applies to expressions, in that an expression such as `a/b/c` changes focus from one axis step to another. As a consequence, in our example, we were allowed to write `books/book`, as the first `books` changed focus from `author` to `books` and each was allowed to have at most one downward expression. Were we to write `books/book[isbn]`, the second half, `book[isbn]`, does not change focus (i.e., to `isbn`, the focus remains on `book` and the whole construct still returns a `book`) and we have two downward expressions in this path expression, which is not guaranteed streamable[23].

Here is a counter example that is not guaranteed streamable:

```
<xsl:for-each select="author">
    <xsl:for-each select="books/book">
        <summary isbn="{isbn}">
            <xsl:value-of select="summary" />
        </summary>
    </xsl:for-each>
</xsl:for-each>
```

The literal result element `summary` has an attribute value template that selects a child element `isbn`. Since a literal result element does not also change focus to this element, the nested `xsl:value-of` instruction cannot be processed using guaranteed streamability, because the processor cannot know whether it is available before or after the `isbn` element.

This particular example cannot be rewritten using nested constructs and maintaining guaranteed streamability. It cannot even be rewritten with `xsl:fork` (see next section), because the two elements `isbn` and `summary` are siblings of one another, and we actually want to nest them. There are, however, several other ways of rewriting this particular use-case. Sticking with the nested for-each loops (in itself not

[23]This rule is not entirely in line with the specification. In the specification, a filter expression is still a focus changing construct. However, that is because it changes the focus in A[B] to A inside the filter expression. But because a top level filter expression must be motionless, the rule still holds, and I consider the construct not changing focus because A[B] returns A, so the return focus is the same as the focus inside the filter. The few exceptions to this rule will not be discussed in this paper. For a more in depth view, see section 19.3, Determining the Context Item Type in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#determining-context-item-type

a good programming style, but we use it here illustratively), this is one way to rewrite it:

```
<xsl:for-each select="author">
    <xsl:for-each select="books/book">
        <xsl:variable name="items"
            select="*[self::isbn or self::summary]/string()" />
        <summary isbn="{$items[1]}">
            <xsl:value-of select="$items[2]" />
        </summary>
    </xsl:for-each>
</xsl:for-each>
```

The variable $items is assigned the string-value of the nodes (see Rule 5 for a more in-depth explanation on using variables in streaming) and the whole body of the inner `xsl:for-each` now has only one downward selection, the expression `*`.

You may have wondered why we don't write the select expression of the variable differently. Why using this verbose syntax? The reason is that if you were to write `(isbn, summary)/string()`, the expression contains two downward selections, which can potentially result in returning overlapping nodes, for instance in the expression `(section, section/para)`, with multiple nested sections this will return nodes that overlap[24], which is not compatible with streaming. While we could see that `(isbn, summary)` will never return overlapping nodes, the working group decided to not make the rules overly complex for this one particular expression, since there is an easy way to rewrite this type of expression.

The reason that the predicate, combined with the child-select expression is allowed and streamable, is because a nametest on the `self::` axis is considered motionless: a streaming processor does not need to look ahead to determine the name of the current node. See also Rule 9, about motionless filters.

6.3. Rule 3: Use motionless expressions where possible

A motionless expression, or construct, is one that doesn't require the processor to move from the current point in the input XML stream at all. At each current node, the following properties can always be requested:

- Nametests on the current node, like `name()`, `local-name` and `self::foo`. Example: `<xsl:element name="{local-name()}" />` creates a new element with the local name of the current element.

[24]Overlapping means: one or more nodes in the sequence contain nodes that are itself also part of the sequence. For instance, `<section><para></para></section>` and the expression `section | section/para` will return two items, the root `section` element and the child `para` element. Since `para` is already contained in `section`, these nodes are said to *overlap*. In streaming, overlapping nodes are not allowed, because it requires the processor to go over the input stream multiple times.

- Climbing the parent, ancestor or ancestor-or-self axes. Example: `*[ancestor-or-self::section]`, returns all elements that are children of a `section` element or that is itself a `section` element. Here the `*` makes this a downward expression, but the predicate is allowed because it is motionless.
- Visit the attribute axis. Example: `<xsl:value-of select="'Name' || @name || ', age: ' || @age || ', birth: ' || @birthyear` outputs concatenated strings of the values of the attribute nodes `name`, `age` and `birthyear` of the current context node.
- Any function or instruction that takes atomic values, unless the atomic values are created from serializing the content of a streamable node, in which case that operation itself is consuming and must be the only one in the construct. Examples: `current-dateTime()`, any math function, any stylesheet function that does not take a streamed node as input. Consider `string(.)`, this expression takes a node and returns the string-value. When used on itself, it is allowed, because it consumes the current node. Similarly, you can use `my:fibonacci(input)`, if the input parameter is declared as an atomic type, for instance `xs:integer`.
- Functions that are inspectional and that operate on the current node. Examples: `has-children(.), exists(.), nilled(.), not(.), in-scope-prefixes(.)` etc, which makes expressions such as `foo[has-children()]` streamable.
- Literal result elements. As long as literal result elements do no use a consuming expression among its children or in its attribute value templates, it is motionless.
- Any node creation instruction. If the content or an attribute value template in a node creation instruction contains a non-motionless expression, it is considered consuming, and you should apply Rule 1 or 2. Example: `<xsl:comment>My comment</xsl:comment>` is motionless, as is `<xsl:element name="@lastname" />`, which uses a motionless expression in its attribute value template. Conversely, `<xsl:copy-of select="author" />` is not motionless, but it contains only one downward expression, so it is allowed in streaming.
- Any variable reference is motionless. The reason this works is that it is not allowed to store a reference to a streamed node in a variable. As a result, any variable reference is motionless[25], even a variable reference that contains nodes, because allowed nodes can only be grounded nodes. Example: suppose you have declared a variable `<xsl:variable name="doc" select="doc('foo.xml')" />`, you can have multiple expressions operating on `$doc`, like `$doc//para[em | bold]`, which would otherwise not be streamable.

[25] A notable exception exists for grouping and merging, where it is possible to bind the grouping variable using `bind-group` or the source using `bind-source`. These variables behave differently, see the flowchart section in this paper for details.

Not all motionless expressions are grounded[26] and there is a notable exception for so-called climbing path expressions[27]: they are not allowed to be passed on. A climbing path expression is any of the attribute, the ancestor, the ancestor-or-self, the parent or the namespace axis. Consider for instance the third bullet point above, the attribute axis. The axis is motionless, but not grounded. To see what that means, consider the following example:

```
<xsl:template match="author" mode="streaming">
    <xsl:apply-templates select="@name" mode="#current" />
    <xsl:copy-of select="." />
</xsl:template>

<xsl:template match="*@" mode="streaming">
    <xsl:copy-of select="parent::*" />
</xsl:template>
```

At first sight, there is nothing wrong with this slightly contrived example. But if we look closely, we see that, assuming the example is considered streamable, the processor has to visit all the children of the `author` twice, because both template rules create a copy of that element. To prevent this from happening, streamability analysis uses the term *climbing* for expressions that are themselves motionless, but can potentially be used to move away from in a non-streamable way. As in the example above: when you move a node around that is climbing, anything can happen. Hence we consider the above example free-ranging. To rewrite the example in a streamable way, see Rule 6, or consider using `xsl:fork` as explained in the previous section.

To determine this for your own situation, consider whether the instruction or expression can return streamed nodes. If it can, it may not pass them on (for instance with `xsl:apply-templates` or a function defined with `xsl:function` that takes nodes as arguments) , because the processor cannot possibly predict what will happen with that node. The solution for this situation is commonly relatively simple: atomize the nodes, use `xsl:fork` or create grounded copies of the nodes (see Rule 6). When you need to pass an attribute on to another function or template, atomization is the way to go:

```
<xsl:template match="author">
   <xsl:value-of select="my:format-date(@publish-date/string())" />
</xsl:template>

<xsl:function name="format-date">
   <xsl:with-param name="date" />
   <xsl:value-of
```

[26] A grounded expression is an expression or construct is one that does not return any streamed nodes.
[27] See for the definition of climbing section 19.5 of XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#dt-climbing

```
        select="format-date($date, '[MNn] [Do], [Y]', 'de', 'BE', ())" />
</xsl:function>
```

In this example, we would want to pass the attribute node `publish-date` to the function, but we are not allowed because the processor cannot know beforehand that the function will not travel away from that node. To circumvent this, we can simply atomize the attribute node using the `string()` function.

Alternatively, as described in the fourth item of the bulleted list above, we can define the parameter `$date` to be of atomic value, which forces the argument to be atomized prior to it entering the function. Personally, I prefer pre-atomizing it as it is a clearer pattern for other programmers and it does not require them to investigate the signature of the function arguments, which may well be hidden deep in a package[28] [19].

6.4. Rule 4: You can move up the tree, but never down again

The ancestor axis, together with the attributes and properties of the nodes, is the only axis that is kept in memory while performing streaming. Even though streaming is defined to have to process a document in constant memory, the XSL Working Group considered keeping the ancestor axis allowed and of much use for many use-cases. Since in all but a few edge cases[29] have limited depth, the required memory for keeping the ancestor axis around was considered acceptable.

As with the attribute axis, discussed in more depth in the previous section, the parent and ancestor axis are motionless. But other than the attribute axis, it is not possible to consume the parent node in any way and you are not allowed to move it around (which is the same as with attribute nodes). Why it cannot be consumed can best be explained with an example:

```
<xsl:template match="author">
   <xsl:copy-of select="parent::paper" />
</xsl:template>
```

Consider an input XML file that has a structure as follows:

```
<paper>
   <name>Something fantastic</name>
   <publishyear>2012</publishyear>
   <author>John Doe</author>
</paper>
```

[28] Packages are a new feature of XSLT 3.0 and allow component based development and protection of intellectual property by pre-compilation. See for more info the bibliography or XSL Transformations 3.0, section 2.7 Packages and Modules: http://www.w3.org/TR/xslt-30/#packages-and-modules

[29] Theoretically it is possible to have a tree with unlimited depth, but this edge case has not been considered by the working group. In fact, if such a tree exists, it is probably a faulty XML and should be fixed at the source or be processed in another way than through XSL streaming.

Once the template matches `author`, the read pointer is positioned just at the end of the openring tag. Just as with any other streaming situation, this read pointer is not allowed to move back. Even though the processor remembers the parent and ancestor axes, it does not remember all their content, as the content can be of arbitrary size. Hence the `name` and `publishyear` elements have already been forgotten by the time `author` is visited. If the previous example was supposed to be streamable, it would require the processor to remember these nodes. Since it cannot do that, and since as with any other streaming construct it cannot look back, the example is not streamable.

This rule is best remembered as follows: *you can move up the tree, but you can never move back down the tree again.* So `parent::paper[parent::archive]` is allowed, but `parent::paper[name = 'John doe']` is not, as the latter moves up, and then down again (the child-select expression in the predicate).

The same applies to atomizing the parent or ancestor axis. You cannot do that, as an expression such as `parent::paper/string()` would require reading all previous siblings as well, just as in the above example with `xsl:copy-of`.

The only expressions that are allowed to operate on these nodes are expressions that inspect the node without moving away from its head. For instance, the expression `exist(parent::paper)` is allowed, as is `<xsl:if test="parent::paper">`, because both expressions only have to test whether the parent exists, they don't have to consume the element to return true or false.

Likewise, the only path expressions allowed from a parent, ancestor or ancestor-or-self axis are the same upward axes, plus the attribute and namespace axes. For instance, it is allowed to select the union of all ancestoral attributes by doing `<xsl:copy-of select="ancestor-or-self::*/attribute()" />`.

6.5. Rule 5: You cannot store a reference to a node

We have come across this rule already in some of the previous examples. This rule is very easy to master: you cannot store a reference to a streaming node, ever[30].

Remember the XSLT 1.0 times? The times that you required an [4] extension function to re-process a variable containing a result set[31]? When you do streaming in XSLT 3.0, part of that 'nightmare' is back. Since XSLT 2.0 it has become a very common programming paradigm to re-process a temporary result tree by applying the variable with `xsl:apply-templates` or with `xsl:for-each`. This still exists in XSLT 3.0, but when doing streaming, it is not allowed. The Working Group is still

[30] Unless your processor supports streamed references of nodes, but this paper is about guaranteed streamability, not about potential extensions created by implementors.

[31] The particular function referred to here is the infamous `exslt:node-set` function, available in many XSLT 1.0 processors, see http://www.exslt.org/exsl/functions/node-set/exsl.node-set.html, though the list of processors supporting this function is much larger.

discussing possibilities to allow streamed variables, but at its current status, a variable cannot contain a reference to any streamed node.

The reason for this is similar to the reason not to allow `following-sibling::` (see section "What is a downward expression"), it is very hard to define correct rules that allow statically assessable streamability.

Though references to streamed nodes are not allowed, references to non-streamed nodes, to atomic items, to function and map items are allowed. For instance, it is perfectly safe to write the following:

```
<xsl:variable name="names">
    <xsl:for-each select="name">
        <name birthdate="{@year || @month || @day}">
            <xsl:apply-templates
                select="affiliate"
                mode="#current"/>
        </name>
    </xsl:for-each>
</xsl:variable>

<xsl:apply-templates select="$names/name" mode="non-streaming" />
```

The example, assuming it is the only thing inside the sequence constructor of `xsl:template`, creates new elements `name` inside the variable `$names`. Since these are newly created nodes, they are not streamable nodes. In fact, just as in non-streaming XSLT 2.0, you have created a temporary result tree which can be applied again, just like you used to do in XSLT 1.0 with the `exslt:node-set` function.

Note that the streamability assessment would turn out to be different when the immediate child of the `xsl:variable` instruction was `xsl:apply-templates`. The latter returns a sequence of nodes and the sequence returned could, theoretically, be a sequence of streamed nodes. Hence, it is not allowed to do that inside `xsl:variable`, unless we take precautions and make sure that the result is *grounded*. Here we did that by introducing a literal result element, which forces the result of the `xsl:apply-templates` to be copied, as opposed to referenced.

That is not the end of the story of variables and node-sets. In fact, what once was an extension function is XSLT 1.0, has come back as a built-in function in XSLT 3.0, but that is the subject of the next section.

6.5.1. A note on global variables and streaming

While you cannot store references to nodes using normal variables or parameters, the restrictions on global variables are even larger. It is not possible at any rate to refer to the context item from a global variable or parameter if the initial mode is a

streamable mode[32]. The reason behind this is simple: when initiating the global variables, the processor would have to start reading the input stream, then, once it starts with the initial template, it would have to move back to the root node. Since moving back is not allowed, any expression that relies on the context item, either implicitly or explicitly, is disallowed for a global variable or parameter. In fact, it will statically raise error XTSE0545[33].

That does not mean that you cannot use streaming from within a global variable. It is perfectly legal to write something like the following:

```
<xsl:variable name="settings">
    <xsl:stream href="settings.xml">
        <xsl:apply-templates
            select="settings"
            mode="streamable"/>
    </xsl:stream>
</xsl:stream>
```

Such a global variable would then have to read the settings file using streaming. It can be assumed, but is not guaranteed, that a processor caches the result of initiating the variable. But under streaming, a processor might be tight on memory usage and reclaim the memory occupied by global variables, re-evaluating them once they are needed again. It would be a performance vs. memory usage trade-off, but the potential glitch is, that the `xsl:stream` instruction is not stable and if, between invocations, the `settings.xml` file changes, it is possible that on different occasions, different results are returned.

This side effect may not be easily exploitable with global variables because different processors optimize differently, but with stylesheet functions it creates some interesting possibilities that could previously only be achieved by using extension functions. Consider the following example:

```
<xsl:function name="time" caching="no">
    <xsl:stream href="http://time.gov/now.xml">
        <xsl:value-of select="time/@current" />
    </xsl:stream>
</xsl:stream>
```

Given the new `caching` attribute[34]introduced in XSLT 3.0, we hint the processor *not* to remember the result of executing the function. As a result, each time we call the

[32]There is an exception though, you are allowed to use a motionless expression that operates on the document node, such as `base-uri()`, but these are in practice of limited use.

[33]Error XTSE 0545 "It is a static error if there is both (a) a mode definition in the stylesheet that has the effective attribute values streamable="yes" and initial="yes", and (b) a global variable in the stylesheet whose initializing expression is not motionless with respect to its context item, as defined in 19 Streamability.", see XSL Transformations 3.0, section 6.6.1: http://www.w3.org/TR/xslt-30/#err-XTSE0545

[34]See section 10.3.8 on Memoization in the XSL Transformations 3.0 specification: http://www.w3.org/TR/xslt-30/#memoization

function, the `xsl:stream` instruction is evaluated again and a new time value is returned. This can be useful in profiling your stylesheet, for instance, or on long-running stylesheets to add a current time to the output of elements (the function `fn:current-time` is deterministic, which means that, regardless how long your stylesheet runs, it will always return the same time).

At this moment, the `caching` attribute is only a hint. Hopefully implementors take that hint seriously, or assess whether or not the stylesheet function calls an external non-deterministic resource, which *should* be evaluated again on each call.

6.6. Rule 6: Break out of streaming abundantly

Because of the limitations of streaming, it can be hard to write stylesheets that typically require roaming expressions. While in most scenarios it will be possible to write a streaming equivalent to the roaming version you would have written if streaming were not a requirement, it is not always possible, and sometimes, when it is hard, it is better to use a simple technique that I've called *Lazy XML Processing* in a previous paper [3].

Others have called this *Windowed Streaming*, for instance in the earlier mentioned [18]. It is mentioned as a technique used in databases, such as in a paper by Utkarsh Srivastava in 2004 [22], and in the same year, by Lukasz Golab about querying online data streams [12]; it is part of the description of a US Patent from 2006 about processing XML data streams [17], the technique is used in fast hashing and search algorithms meant for large input data. One of the first mentions of this technique was by Richard Karp and Michael Rabin in their 1987 paper for IBM about efficient pattern-matching algorithms [16], where they propose a solution to use windowed searching as opposed to linear searching algorithms. They did not yet use the term *Windowed streaming*, but the basic idea in their paper is the same.

The key to breaking out of streaming is to change your expression from returning streamed nodes, to returning grounded nodes, an atomic value, a map or a function item. We have already seen how to return an atomic value in earlier rules, but you cannot always operate on the atomized version of a node-set. The functions that you can use to create a grounded node-set are `fn:copy-of`[35], which is essentially the equivalent of the XSLT 1.0 `exslt:node-set` function discussed in the previous rule, and `fn:snapshot`[36], which works the same as `fn:copy-of`, except that it retains the ancestor axis.

Consider a typical input XML document that contains a root node and a set of many first children. For instance, a large XML file containing all available books on

[35] See for more information and more elaborate examples `fn:copy-of` in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#func-copy-of

[36] See for more information and more elaborate examples `fn:snapshot` in XSL Transformations 3.0: http://www.w3.org/TR/xslt-30/#func-snapshot [http://www.w3.org/TR/xslt-30/#func-copy-of]

amazon.com. Each individual `book` element easily fits in memory, but not all books at once. If your stylesheet is primarily concerned with the individual book elements, you can create a snapshot of each book and use free-ranging and roaming expressions from then on. For example:

```
<xsl:template match="book" mode="streamable">
    <!-- switching modes to non-streaming is now possible -->
    <xsl:apply-templates select="copy-of(.)" mode="non-streamable" />
</xsl:template>

<xsl:template match="book" mode="non-streamable">
    <!-- use any roaming and free-ranging expressions here -->
</xsl:template> ▶
```

This technique is so simple, that I cannot stress it enough. Look at the structure of your input document and find out how it can be split into smaller chunks that fit into memory perfectly and use the snapshot() or copy-of() functions to split it in usable parts.

Be careful what you copy, however. The moment you copy nodes using `fn:copy-of`, you tell the processor "I know how large this subset is, you can safely copy it in total in memory". Once you start using this function, the requirement to run under constant memory is subject to the maximum extra memory required by the use of the `fn:copy-of` function. As an extreme comparison, consider the following two examples, which are semantically equal:

```
<!-- example 1 -->
<xsl:apply-templates select="copy-of(book)" />

<!-- example 2 -->
<xsl:apply-templates select="book/copy-of()" />
```

The difference is subtle and is often misunderstood. Both examples return copied nodes of the `book` element, and both will result in exactly the same result document. However, the essential difference is that the first example copies all the `book` elements at once and then processes the whole set, and the second example copies one `book` element at the time and processes each element one by one. It is the second example that can be considered *Windowed Streaming* as mentioned in the beginning of this chapter. The first example is how *not* to do windowed streaming. Instead, if the set of `book` elements is large, you just broke out of streaming and read almost the whole document at once: you should have saved yourself the trouble.

When you are in doubt whether windowed streaming can be applied by your processor on your expression, split it up and apply windowed streaming from a matching template, as in the original example in this chapter. That way, you are certain that *that and only that node* will be copied. Once the template goes out of

scope, the processor can garbage collect the memory occupied by your copied node and the overall memory will remain constant.

6.7. Rule 7: Understand streamable patterns

So far we have talked about what constructs are allowed and how you must stick to the rule of a maximum of one downward selection in your expressions. The same applies to patterns, but patterns have slightly more freedom because they are essentially a test on each node that passes through the input stream. As such, you could compare them with an inspection context, where upon visiting a node we only need to know one thing: does it fit the pattern, or not.

Essentially, patterns can be one of two things: they can either be motionless and grounded, or they can be roaming and free-ranging. Because the pattern syntax itself is already limited, you can pretty much use almost every expression with them, provided you take a few precautions[37]:

- A pattern cannot start with a variable reference, nor can it start with `fn:element-with-id`, `fn:doc`, `fn:id` or `fn:key`.
- No top-level predicate may contain any of the functions `fn:position`, `fn:last`, `fn:function-lookup`.
- No top-level predicate may be numeric.
- No top-level predicate may contain a downward, or otherwise consuming expression.
- Nested predicates are allowed to use the functions `fn:position`, `fn:last` and `fn:function-lookup`, but are themselves also required to be motionless.
- Variables are allowed (except the pattern may not start with one), but may not be bound, that is, they may not refer to a `bind-group` or `bind-source` attribute of `xsl:merge-source` or `xsl:for-each-group`.

In practice this means that you can write pretty much anything. For instance, an expression such as `para | section/para` is often not allowed in other places, but is allowed in a pattern. You are also allowed to write more complex patterns, such as `para[@status eq 'edited'][ancestor::section[@editor = 'john']]` or even `text()[matches(., '\d+')]`.

That last expression requires perhaps a bit extra explanation. After all, we saw earlier that consuming a node (the `(.)` expression in this example) was not allowed. But so-called childless nodes are considered a special case. These nodes are `text()`, `attribute()`, `comment()`, `processing-instruction()`, `namespace-node()`. Because the processor *knows* that these nodes cannot possibly have children, it can read the contents completely without worrying that it has to ever look back. Hence, the fol-

[37] These rules follow directly from the rules in XSL Transformations 3.0, section 19.8.9, see: http://www.w3.org/TR/xslt-30/#classifying-patterns

lowing example is fully streamable, even though it looks like both the pattern and the `xsl:value-of` contruct have a consuming expression.

```
<xsl:template
    match="para/comment()[contains('TODO')]"
    mode="streamable">

    <todo by="{ancestor::section/@editor}">
        <xsl:value-of select="." />
    </todo>
</xsl:template>
```

Because patterns cannot themselves contain the preceding, preceding-sibling, the following or the following-sibling axis, the expression prior to the predicate is automatically motionless. Similarly, because the predicate in the pattern itself must be motionless, it cannot possibly contain these axes either. Just as with other expressions, these axes are out of bounds when doing streaming.

6.8. Rule 8: Templates must be grounded

Any template rule or named template must be *grounded*[38], which means, it is not allowed to return streaming nodes. This rule is very similar to variables and parameters not being allowed to return streamed nodes. As a consequence, you cannot use `xsl:sequence` with an expression returning nodes:

```
<xsl:template match="book" mode="streamable">
    <xsl:sequence select="author" />
</xsl:template>
```

While this template contains only one downward experession (see Rule 1), it is not guaranteed streamable. The reason behind this is that the `xsl:sequence` instruction returns references to the nodes it selects, and in turn, the `xsl:template` returns those references as well. Consider you would be allowed to write this, then consider the following example:

```
<xsl:template match="publications" mode="streamable">
    <xsl:variable name="books">
        <xsl:apply-templates select="book" />
    </xsl:variable>

    <xsl:copy-of select="$books" />
</xsl:template>

<xsl:template match="book" mode="streamable">
```

[38]See XSL Transformations 3.0, section 6.6.3: http://www.w3.org/TR/xslt-30/#streamable-templates

```
    <xsl:sequence select="author" />
</xsl:template>
```

If this were allowed to work with streaming, the variable `$books` would contain references to streamed nodes, which, as we know, cannot be retained in memory. Hence, once the `xsl:copy-of` construct is called, the processor would have to look back, up the stream, which is not allowed.

By defining the result of a template to always be grounded, we make the analysis of many streaming situations a lot easier. It is analogous to the the problems we saw when looking at variables not being allowed to contain refernces to streamed nodes.

The solution here is to avoid using `xsl:sequence` as a direct child under `xsl:template`, or to use any construct that returns streamable nodes. Of course, it is still allowed to use `xsl:sequence` with a grounded expression (that is, making sure that the nodes are either copied, or they are atomized). In the above example, the problem is solved with using `xsl:copy-of`, which disconnects the nodes from the streaming process.

6.9. Rule 9: Use motionless filters

In earlier chapters we have already seen what a motionless expression is. In filter expressions, that is, in a predicate, you should always only use motionless expressions, or the result will not be streamable. The same rules apply here as for patterns. Here are a few examples of motionless filters:

- `*[self::para or self::p]`
- `price[@currency[starts-with(., 'EUR')]]`
- `node()[. instance of element()]`
- `parent::foo[ancestor::document/@version eq $glob-version]`
- `*[ancestor::foo >> ancestor::bar]`
- `buildnumber/text()[. = $builds/buildnumber[last()]]`

The last example may perhaps require a bit extra explanation. It assumes a global variable (which, by default, is grounded and motionless, see Rule 5) containing successful buildnumbers and the input document contains all the buildnumbers. You are interested in the most recent successful buildnumber. Similar to the description about patterns (see Rule 7), childless nodes can be consumed in expressions, because there is no navigation away from them. As a result, it is possible to use the predicate with the context item expression, and an otherwise free-ranging expression as the left-hand side of the equal sign, because `$build` is already grounded.

6.10. Rule 10: Master xsl:fork

If you remember my note from Rule 1 about being able to process nodes in any order and how that influences streaming, this last rule offers an alternative to the *one downward selection per construct* rule. By using `xsl:fork`[39], you explicitly tell the processor that from this point on, it should start multiple threads for processing the input stream in parallel. In other words, the stream reader will get multiple read pointers that all have their own starting point, the current node. This process, called *forking* allows multiple downward selections in a single instruction, but is very strictly defined.

Using forking, it becomes easier to split an input stream on disjunct nodes, which could otherwise be very hard to achieve using streaming. For example:

```
<xsl:fork>
    <xsl:sequence>
        <errors>
            <xsl:copy-of select="entry[@type eq 'err']"/>
         </errors>
    </xsl:sequence>
    <xsl:sequence>
        <warnings>
            <xsl:copy-of select="entry[@type eq 'warn']" />
        </warnings>
    </xsl:sequence>
</xsl:fork>
```

In any normal streaming analysis, this would not be allowed, because one construct contains two downward selections, in this case selecting all child `entry` element nodes from an XML log file. By introducing `xsl:fork`, each `xsl:sequence` child construct is allowed to be consuming.

An `xsl:fork` instruction contains solely of `xsl:sequence` children. Each `xsl:sequence` child has a sequence constructor just as a regular `xsl:sequence` instruction, and each of these sequence constructors can have at most one consuming construct, or in the event the `select` attribute of `xsl:sequence` is used, at most one downward selection. However, the `xsl:fork` instruction can contain as many `xsl:sequence` children as you want, with each their own consuming expression, allowing for splitting a source as shown above.

Processors are not *required* to go over the input stream only once. Some stream readers may not allow multiple reading positions, in which case the processor will

[39] See also XSL Transformations 3.0, The `xsl:fork` instruction, section 16.1: http://www.w3.org/TR/xslt-30/#fork-instruction

be forced to go over the stream multiple times. However, as a programmer, you `usually do not have to worry about this[40].

This technique is particularly useful with splitting to multiple result-documents with `xsl:result-document`, as a clear example in the XSLT 3.0 specification shows[41].

7. Follow the arrow

Doing full streaming analysis is a complex task. The previous chapters have shown relatively easy rules that can be followed and that cover most of the common streaming scenarios you will encounter in practice. However, sometimes the rules are not enough, or even after following the rules, you are wondering why it actually is streamable, or not streamable.

The following chapters present a set of flowcharts that make it easier to determine whether a given construct is guaranteed streamable. The rules in the specification can be challenging to read, and these charts may make it easier to do the analysis. It prevents you to have to scroll back and forth through pages of text. Instead, you just have to scroll to a few pages of charts.

At first, the charts may appear overwhelming. But they are set up in such a way that you do not need to go through each and every step for each and every scenario. By visually going over the analysis, it will soon become apparent that many similar scenarios follow the same arrows and after a short while it will get easier and easier to do.

The charts use standard flowchart shapes:

- Oval: start of an (sub) analysis process
- Diamond: decision. Down arrow is always the positive (Yes) answer, right arrow is always the negative (No) answer
- Rectangle: explanation or preparation for a next step, in these charts, most often the collection of operand usages prior to jumping to the general streamability chart
- Rectangle with bars: a process, often elsewhere defined in the charts
- Circle: reference to another flowchart, most commonly, "GS", which means "General Streamability"
- Right square arrow-like symbol: in multi-page flowcharts, means you should continue the same chart at the refered to label
- Callouts: extra information, often normative, about the current step

[40] The exception being that some streams cannot be traversed multiple times, for instance a live twitter feed, unless buffered. Also, an obvious worry can be that multiple passes can have a detrimental effect on performance.

[41] See for the example XSL Transformations section 16.2, Examples of Splitting with Streamed Data: http://www.w3.org/TR/xslt-30/#splitting-examples

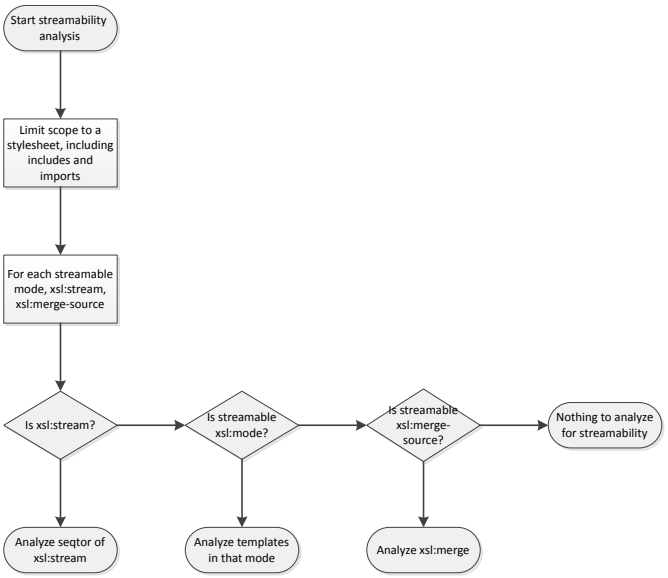
Abbreviations used (it is outsite of the scope of this paper to define each term, please refer to the specification, other online sources or an updated version of this document at http://exselt.net/papers, which will contain more details in this section).

- P: Posture
- S: Sweep
- S': Adjusted sweep
- O or Op: Operand or Operand Role
- OU: Operand Usage
- CP: usually Context Posture, sometimes Combined Posture (in the General Streamability chart)
- PC: Potentially Consuming
- C: Construct
- Fun: Function
- RT: Required Type
- IP: Input Posture
- GS: General Streamability
- Seqtor: Sequence Constructor
- AVT: Attribute Value Template
- TVT: Text Value Template

Most abbreviations are explained on first use in the flowcharts, often in a rectangle shape prior to the steps requiring the use of the new term.

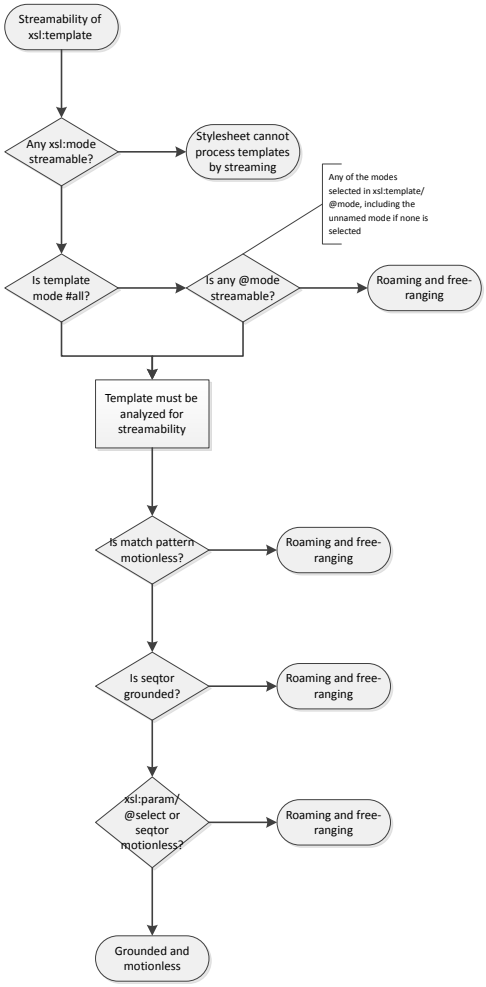
The result of following the flowcharts for a particular construct, such as an expression or an instruction, is a posture, which can be any of *grounded*, *climbing*, *striding*, *crawling* or *roaming* and a sweep, which is any of *motionless*, *consuming*, *free-ranging*. If, at the end of the analysis, the resulting posture is *not* "roaming and free-ranging", your construct is streamable, or potentially streamable. Potentially in the sense that it often depends where and how the construct is used. Only after doing the whole analysis from most outward construct to most inward construct, you can find out whether all the instructions that use streaming are indeed guaranteed streamable.

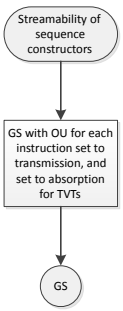
7.1. Start of streaming analysis



7.2. Streaming of templates and sequence constructors

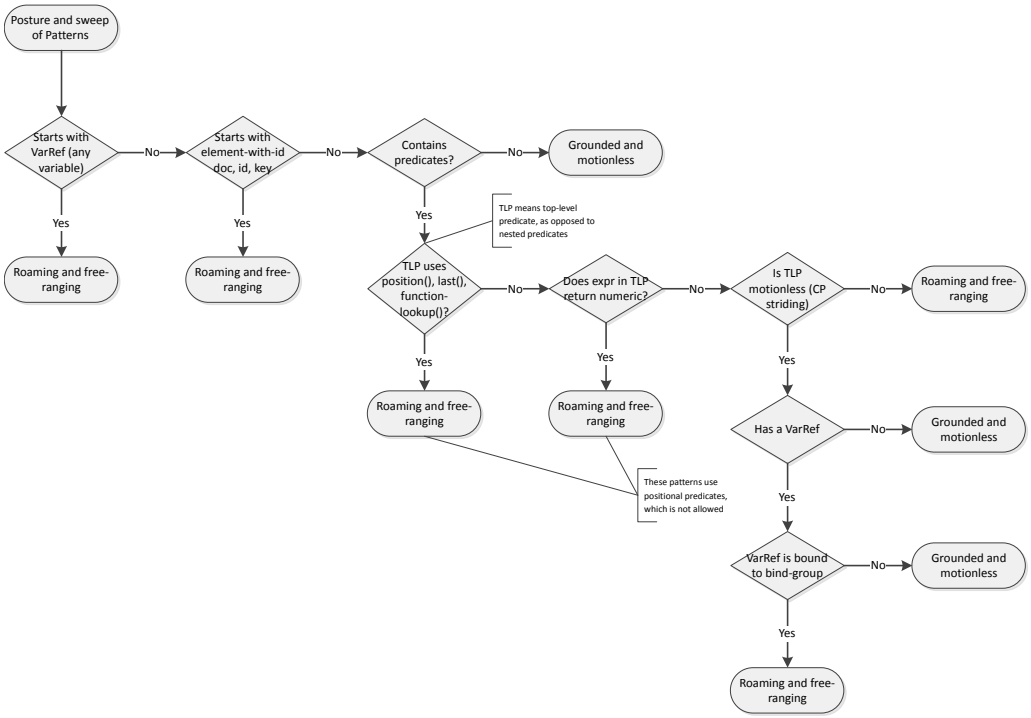
Most streaming analysis starts essentially with either a streaming template or an `xsl:stream` instruction. An `xsl:stream` instruction contains a sequence constructor, as many other instructions in XSLT, and the streamability for both templates and sequence constructors can be determined by using these flowcharts.





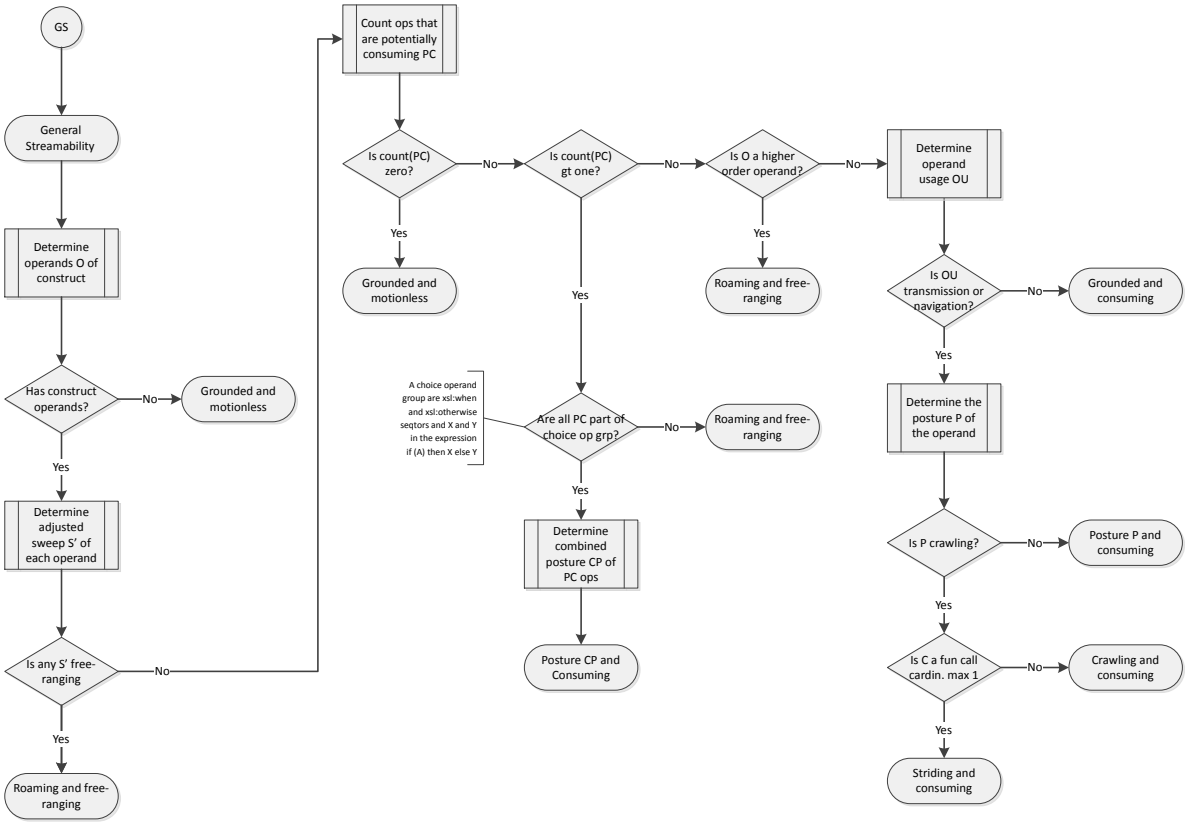
7.3. Streaming of patterns

Patterns are used in matching templates and some other instructions like `xsl:number`. To determine the streamability of a pattern, use this flowchart.



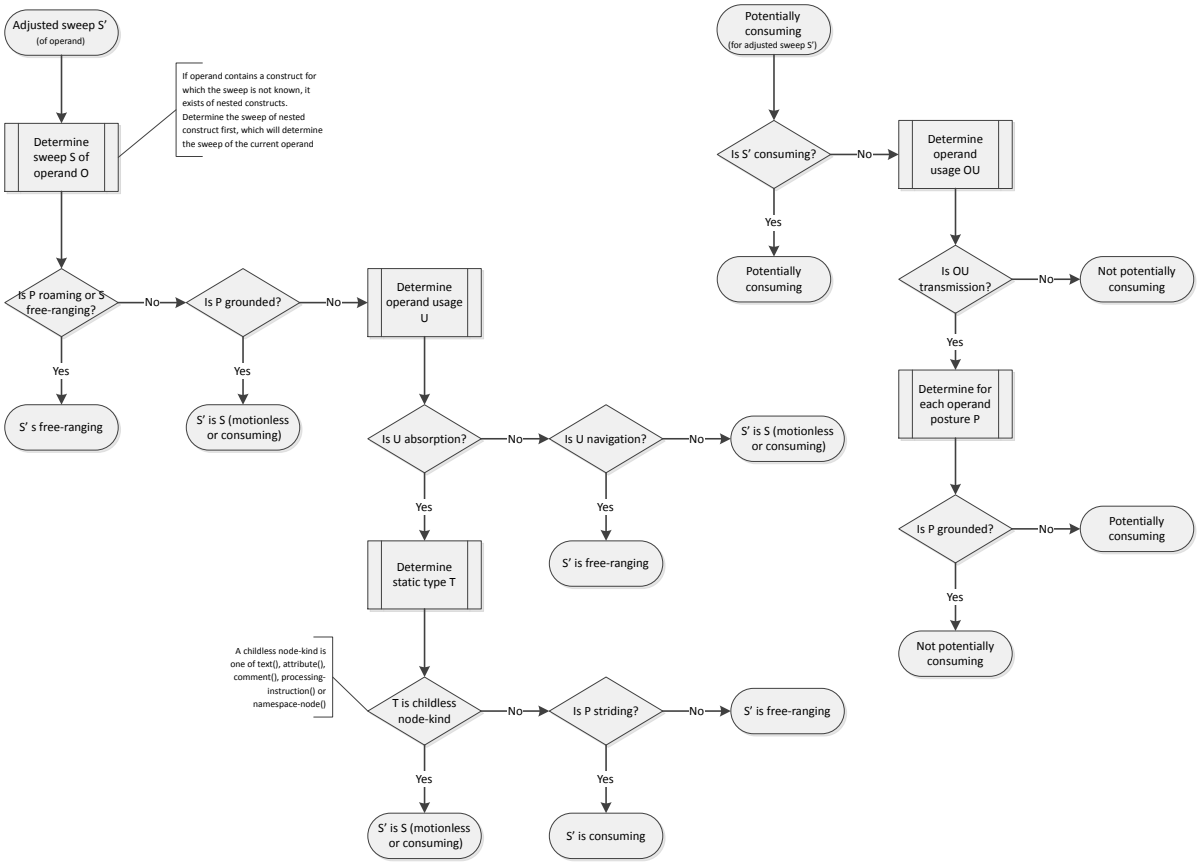
7.4. Flowchats for determining general streamability

Many flowcharts and with a circle and the letters GS, which means that you should apply the general streamability rules layed out in this flowchart, with the operand usage (OU) set to the operand usage as explained in the last rectangle instruction from the refering flowchart.



7.5. Flowcharts for determining adjusted sweep S' and potentially consuming operands

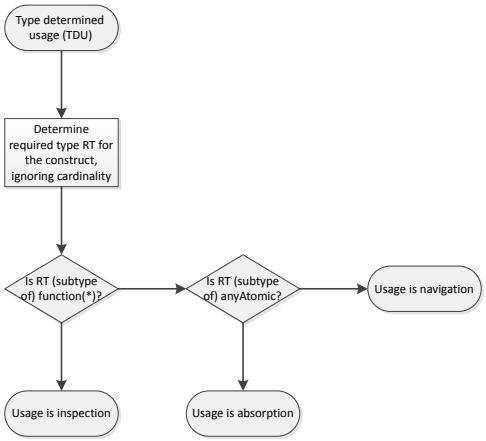
In the General Streamability flowchart, the adjusted sweep of single operands are requested, which can be assessed with this flowchart. The chart on Potentially Consuming is required in combination with the adjusted sweep during the general streamability analysis.

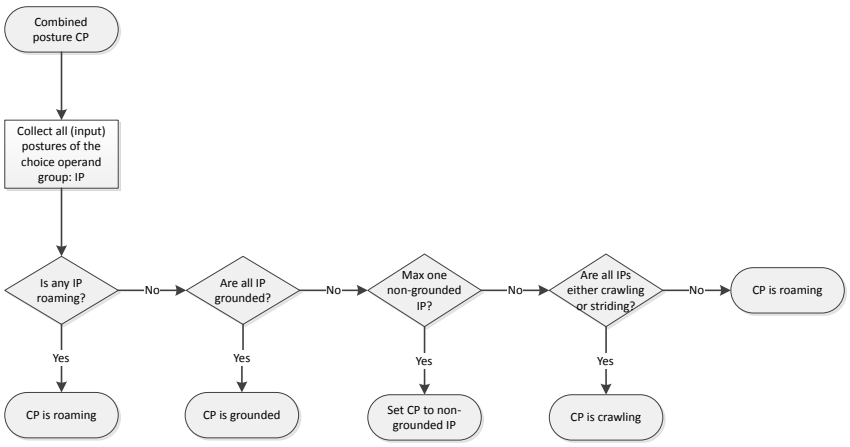


7.6. Flowcharts for determining combined posture and type determined usage

Some charts, such as General Streamability, need to know the combined posture of a choice operand group, which is either the `X` and `Y` in this `if(test) then X else Y` or the combination of `xsl:when` and `xsl:otherwise` in an `xsl:choose` instruction.

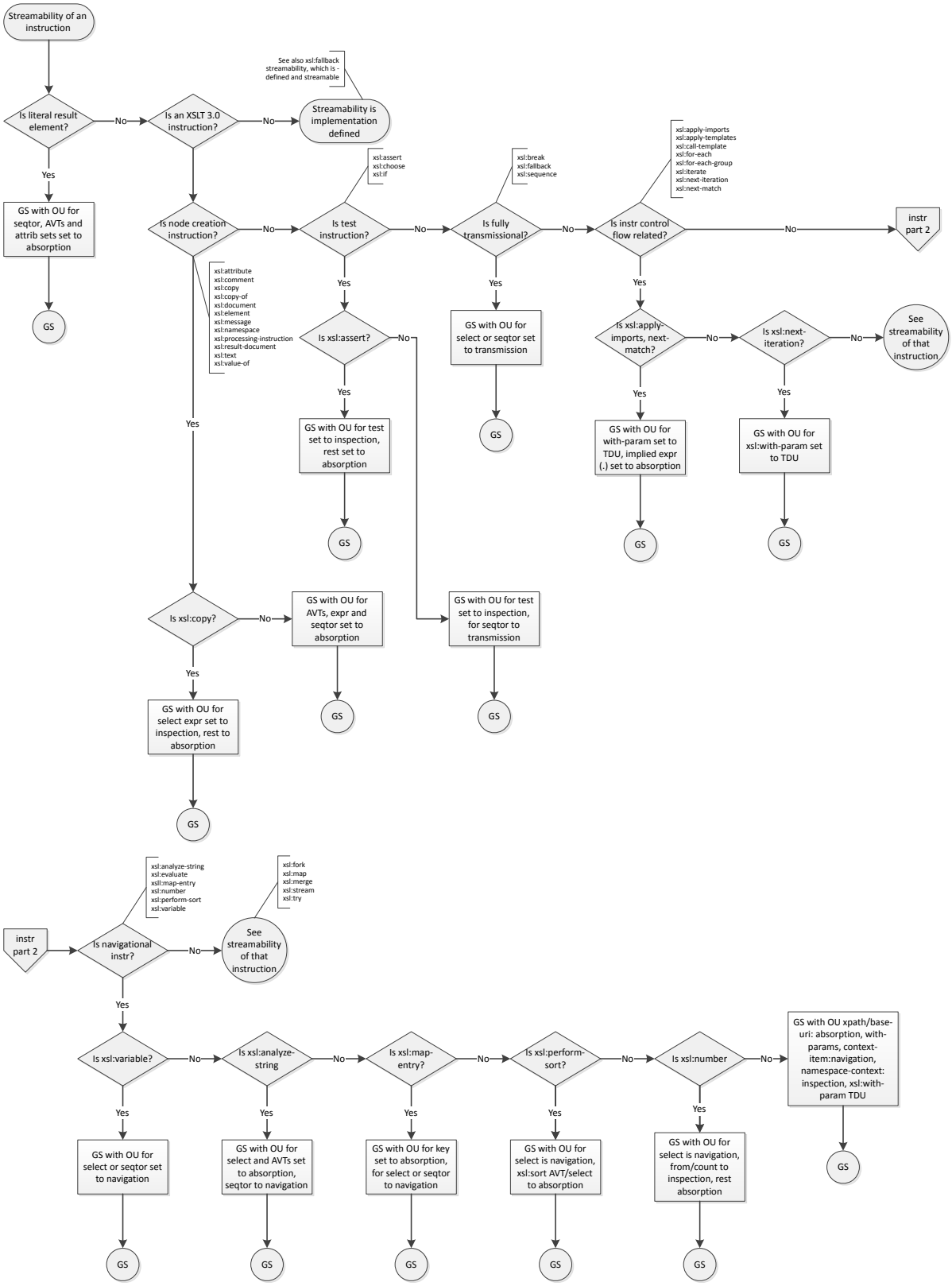
Some operands need to know whether a type is atomic or not through type determined usage (TDU), which can be determined with this brief flowchart.





7.7. Flowcharts for determining streamability of instructions

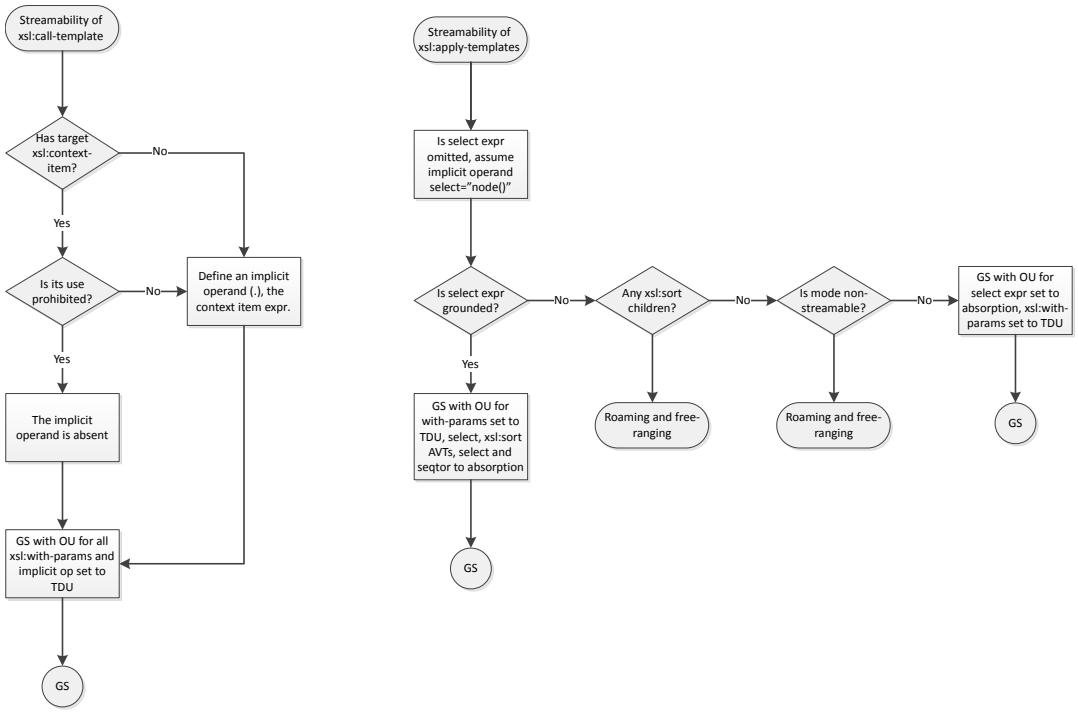
This two-part flowchart can be used to assess the streamability of any XSLT instruction. Many flowchart endpoints refer to the flowcharts of the individual instructions, as it became too challenging to put them all together in one chart.


Streamability of an instruction
Is literal result element?
No
Is an XSLT 3.0 instruction?
No
See also xsl:fallback streamability, which is - defined and streamable
Streamability is implementation defined
Yes
GS with OU for seqtor, AVTs and attrib sets set to absorption
GS
Is node creation instruction?
No
xsl:attribute
xsl:comment
xsl:copy
xsl:copy-of
xsl:document
xsl:element
xsl:message
xsl:namespace
xsl:processing-instruction
xsl:result-document
xsl:text
xsl:value-of
Is test instruction?
No
xsl:assert
xsl:choose
xsl:if
Is fully transmissional?
No
xsl:break
xsl:fallback
xsl:sequence
Is instr control flow related?
No
xsl:apply-imports
xsl:apply-templates
xsl:call-template
xsl:for-each
xsl:for-each-group
xsl:iterate
xsl:next-iteration
xsl:next-match
instr part 2
Yes
Is xsl:assert?
No
Yes
GS with OU for test set to inspection, rest set to absorption
GS
GS with OU for select or seqtor set to transmission
GS
Is xsl:apply-imports, next-match?
No
Yes
GS with OU for with-param set to TDU, implied expr (.) set to absorption
GS
Is xsl:next-iteration?
No
See streamability of that instruction
Yes
GS with OU for xsl:with-param set to TDU
GS
Is xsl:copy?
No
GS with OU for AVTs, expr and seqtor set to absorption
GS
GS with OU for test set to inspection, for seqtor to transmission
GS
Yes
GS with OU for select expr set to inspection, rest to absorption
GS
instr part 2
Is navigational instr?
No
xsl:analyze-string
xsl:evaluate
xsl:map-entry
xsl:number
xsl:perform-sort
xsl:variable
See streamability of that instruction
xsl:fork
xsl:map
xsl:merge
xsl:stream
xsl:try
Yes
Is xsl:variable?
No
Is xsl:analyze-string
No
Is xsl:map-entry?
No
Is xsl:perform-sort?
No
Is xsl:number
No
GS with OU xpath/base-uri: absorption, with-params, context-item:navigation, namespace-context: inspection, xsl:with-param TDU
GS
Yes
GS with OU for select or seqtor set to navigation
GS
GS with OU for select and AVTs set to absorption, seqtor to navigation
GS
GS with OU for key set to absorption, for select or seqtor to navigation
GS
GS with OU for select is navigation, xsl:sort AVT/select to absorption
GS
GS with OU for select is navigation, from/count to inspection, rest absorption
GS

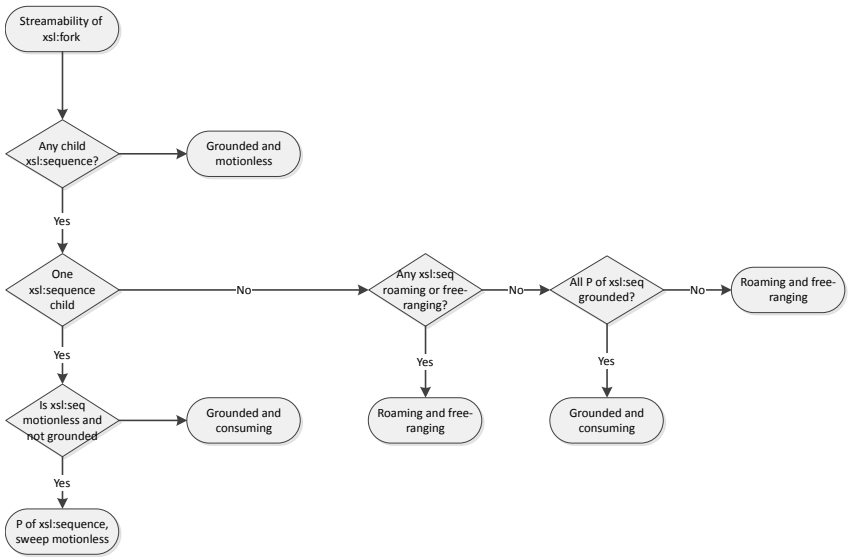
7.8. Flowcharts for determining streamability of selective specific instructions

In the previous flowchart about XSLT instrucions, some end in "see streamability of...", in which case you will find a chart on one of the following pages for that instruction.

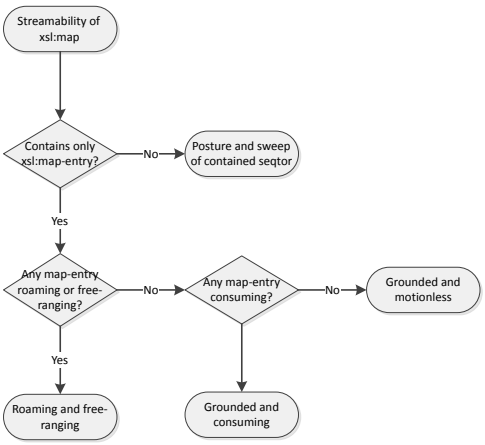
7.8.1. Streamability of xsl:apply-templates and xsl:call-templates

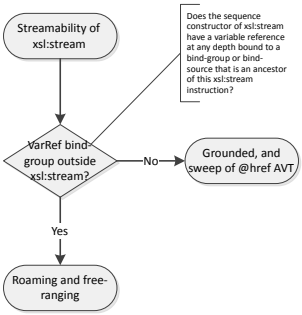


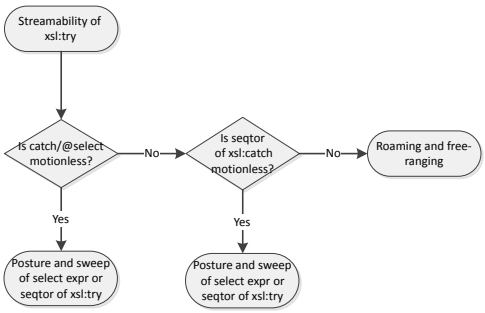
7.8.2. Streamability of xsl:fork

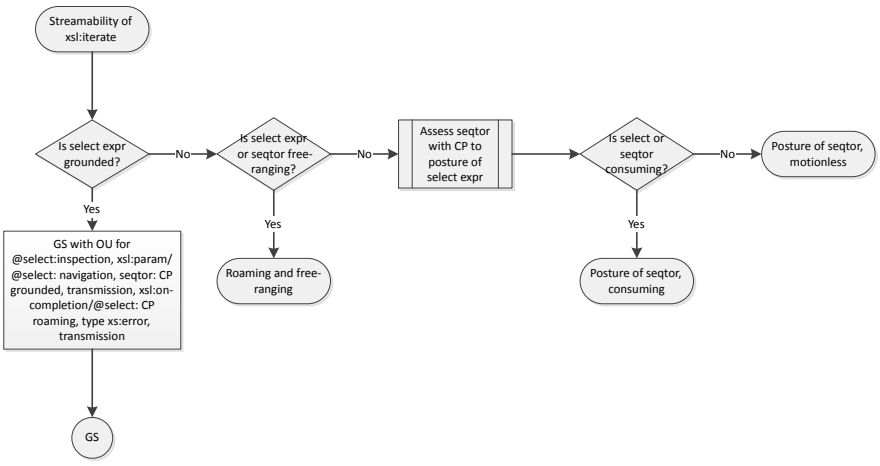


7.8.3. Streamability of xsl:map, xsl:stream, xsl:try and xsl:iterate

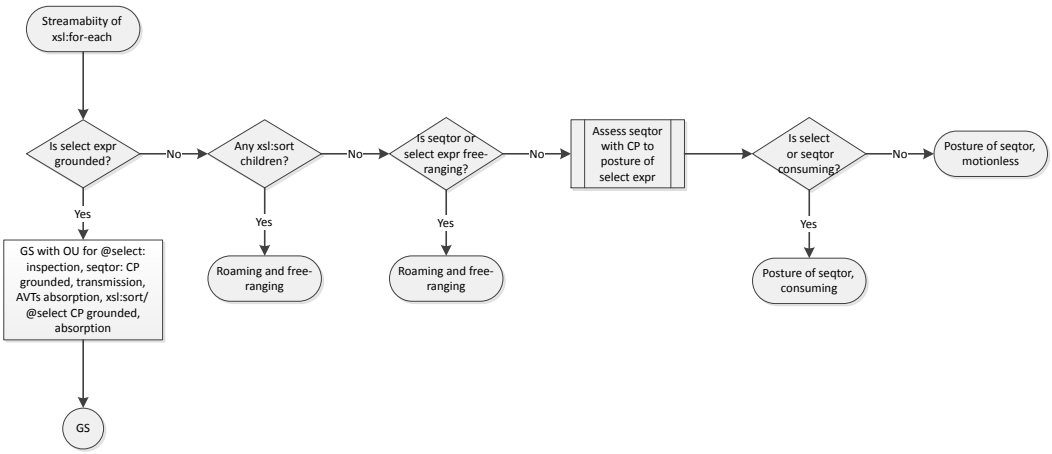


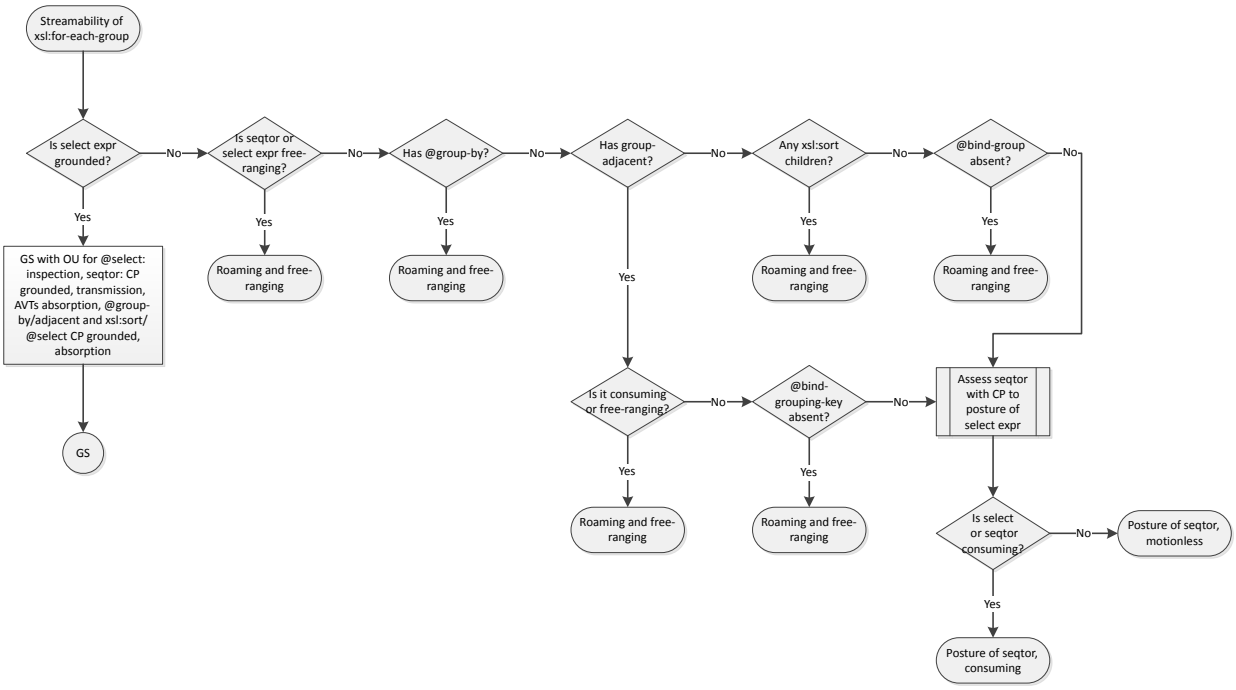






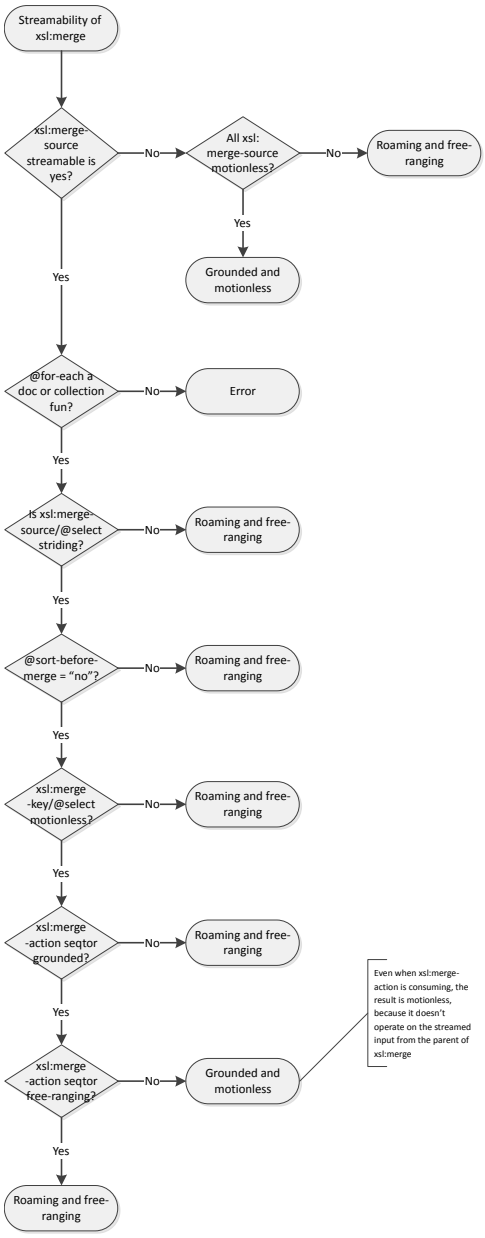
7.8.4. Streamability of xsl:for-each and xsl:for-each-group





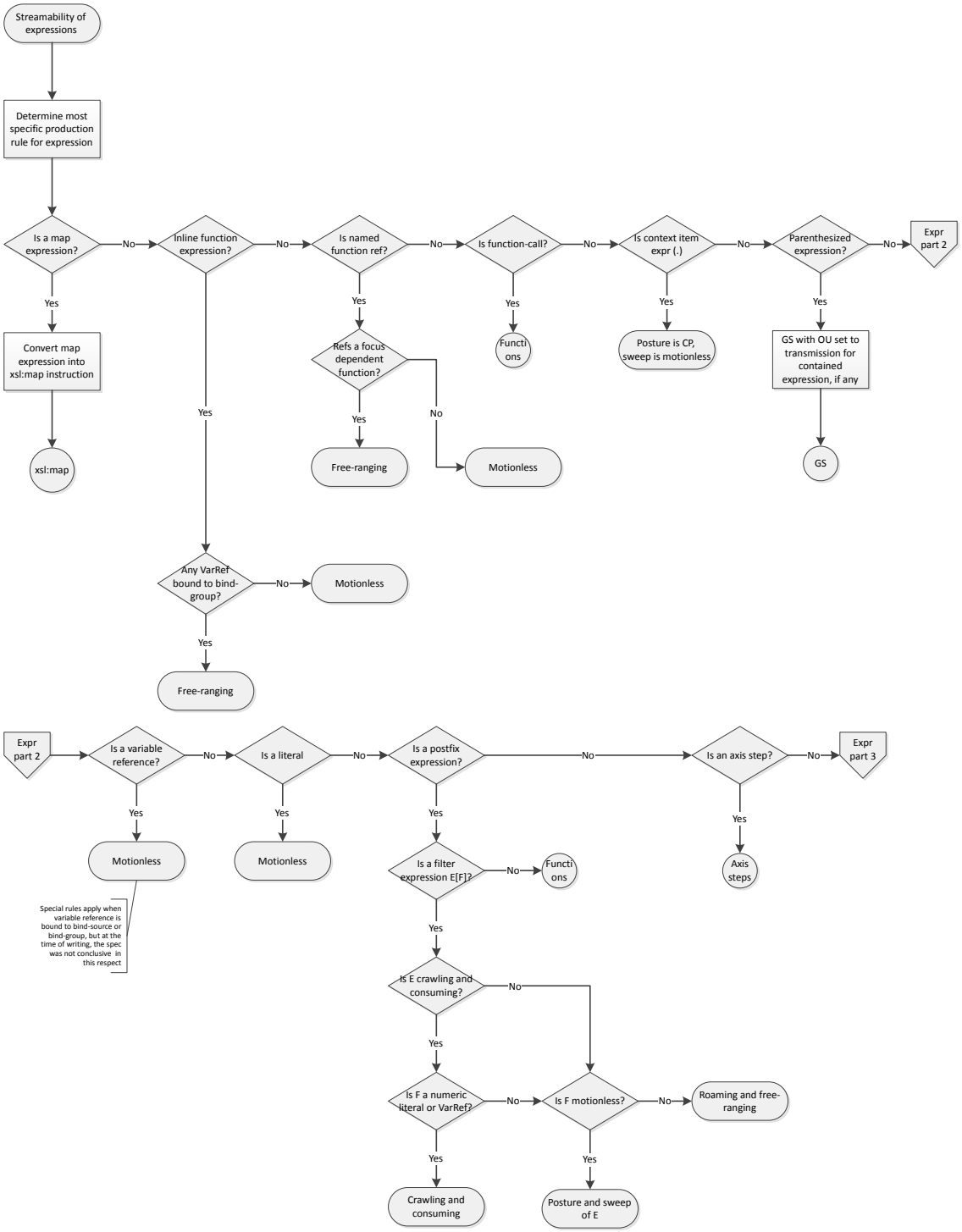
7.8.5. Streamability of xsl:merge

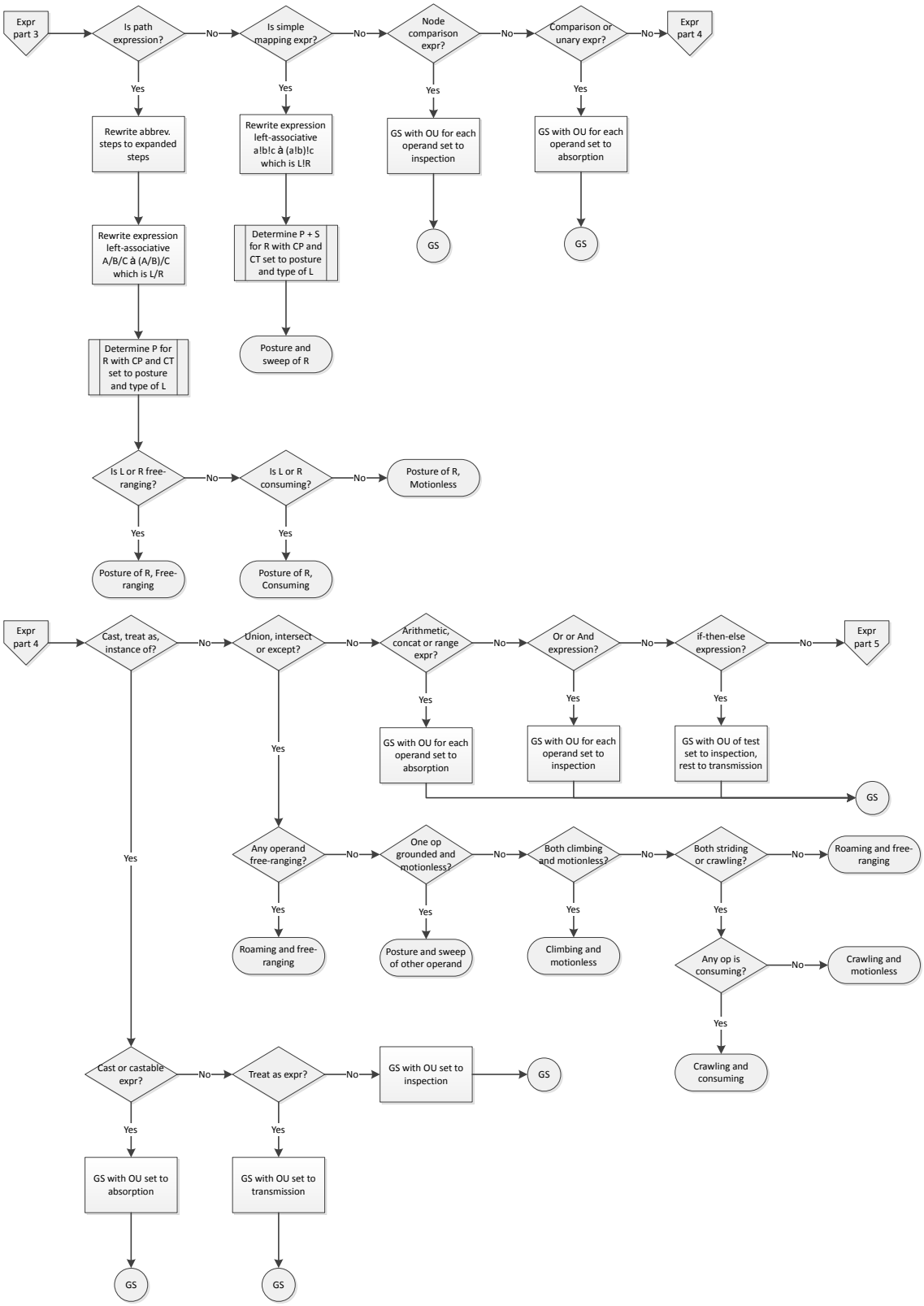
The instruction `xsl:merge` is currently under review for its streamability as a result of a public bug report. It is likely that the instruction will undergo significant changes to allow better streamability analysis.

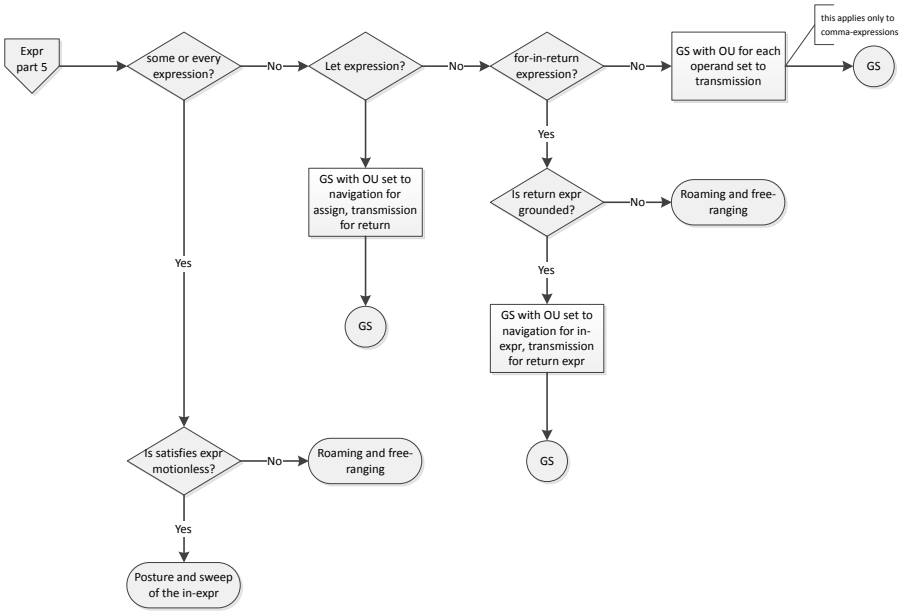


7.9. Flowcharts for determining streamability of expressions

The following five-part flowchart form the core of the expression streamability analysis. Axis expressions have their own flowchart, as do functions, which can be found in the next chapters.

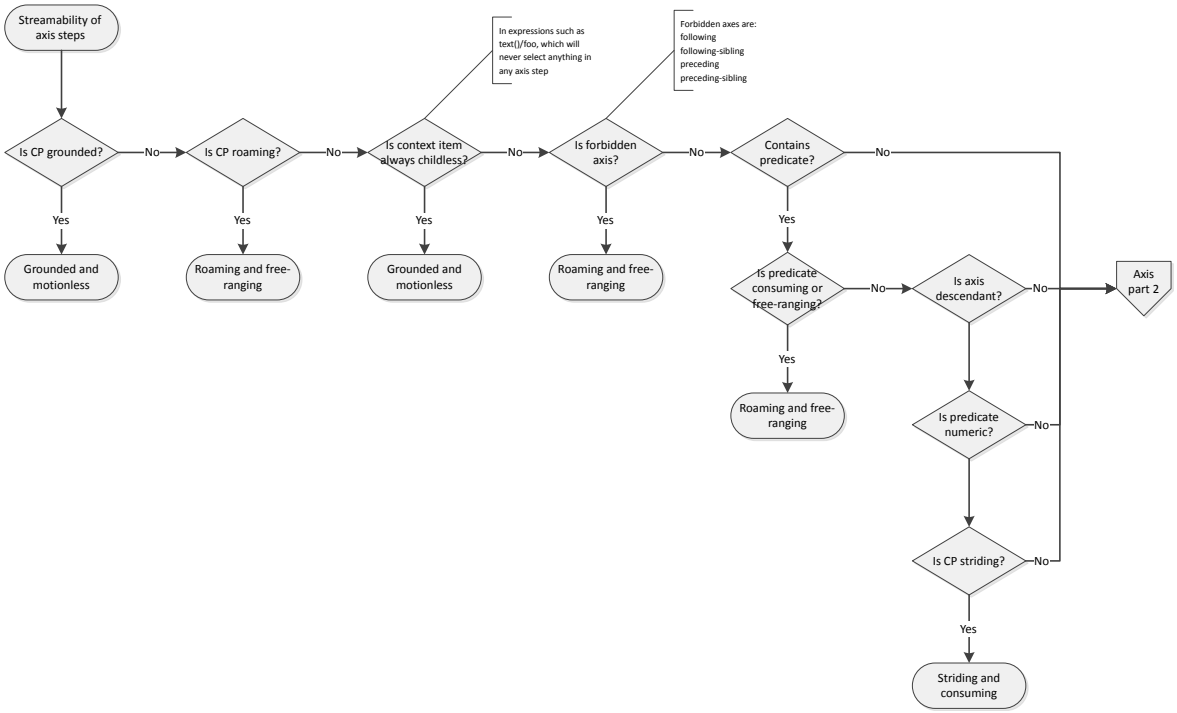

Streamability of expressions
Determine most specific production rule for expression
Is a map expression?
No
Inline function expression?
No
Is named function ref?
No
Is function-call?
No
Is context item expr (.)
No
Parenthesized expression?
No
Expr part 2
Yes
Convert map expression into xsl:map instruction
xsl:map
Yes
Refs a focus dependent function?
Yes
Free-ranging
No
Motionless
Yes
Functions
Yes
Posture is CP, sweep is motionless
Yes
GS with OU set to transmission for contained expression, if any
GS
Yes
Any VarRef bound to bind-group?
No
Motionless
Yes
Free-ranging
Expr part 2
Is a variable reference?
No
Is a literal
No
Is a postfix expression?
No
Is an axis step?
No
Expr part 3
Yes
Motionless
Special rules apply when variable reference is bound to bind-source or bind-group, but at the time of writing, the spec was not conclusive in this respect
Yes
Motionless
Yes
Is a filter expression E[F]?
No
Functions
Yes
Is E crawling and consuming?
No
Yes
Is F a numeric literal or VarRef?
No
Is F motionless?
No
Roaming and free-ranging
Yes
Crawling and consuming
Yes
Posture and sweep of E
Yes
Axis steps

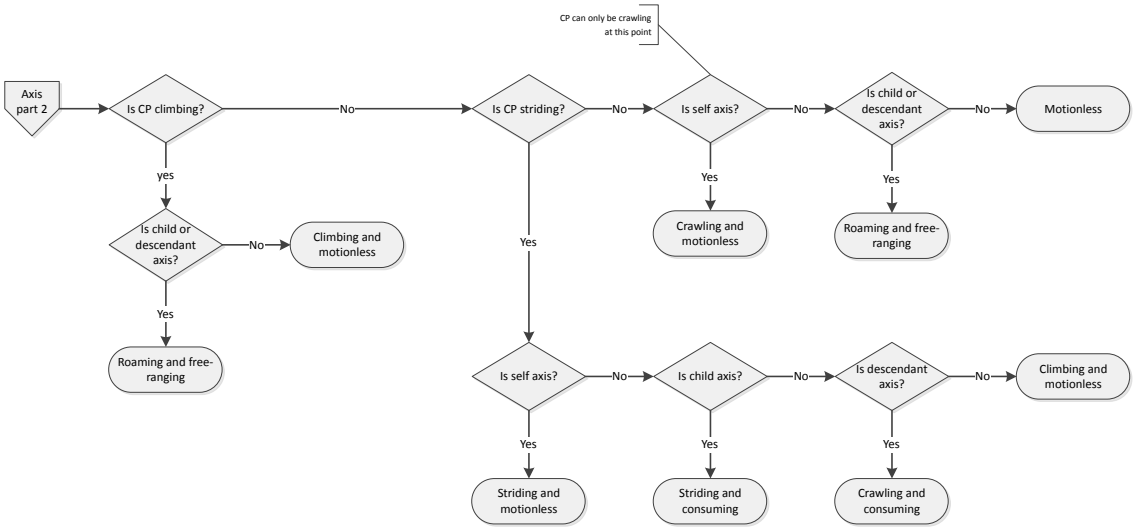

Expr part 3
Is path expression?
No
Is simple mapping expr?
No
Node comparison expr?
No
Comparison or unary expr?
No
Expr part 4
Yes
Rewrite abbrev. steps to expanded steps
Rewrite expression left-associative A/B/C à (A/B)/C which is L/R
Determine P for R with CP and CT set to posture and type of L
Is L or R free-ranging?
No
Is L or R consuming?
No
Posture of R, Motionless
Yes
Posture of R, Free-ranging
Yes
Posture of R, Consuming
Yes
Rewrite expression left-associative a!b!c à (a!b)!c which is L!R
Determine P + S for R with CP and CT set to posture and type of L
Posture and sweep of R
Yes
GS with OU for each operand set to inspection
GS
Yes
GS with OU for each operand set to absorption
GS
Expr part 4
Cast, treat as, instance of?
No
Union, intersect or except?
No
Arithmetic, concat or range expr?
No
Or or And expression?
No
if-then-else expression?
No
Expr part 5
Yes
GS with OU for each operand set to absorption
Yes
GS with OU for each operand set to inspection
Yes
GS with OU of test set to inspection, rest to transmission
GS
Yes
Any operand free-ranging?
No
One op grounded and motionless?
No
Both climbing and motionless?
No
Both striding or crawling?
No
Roaming and free-ranging
Yes
Roaming and free-ranging
Yes
Posture and sweep of other operand
Yes
Climbing and motionless
Yes
Any op is consuming?
No
Crawling and motionless
Yes
Crawling and consuming
Yes
Cast or castable expr?
No
Treat as expr?
No
GS with OU set to inspection
GS
Yes
GS with OU set to absorption
Yes
GS with OU set to transmission
GS
GS



7.10. Flowcharts for determining streamabiity of axis steps

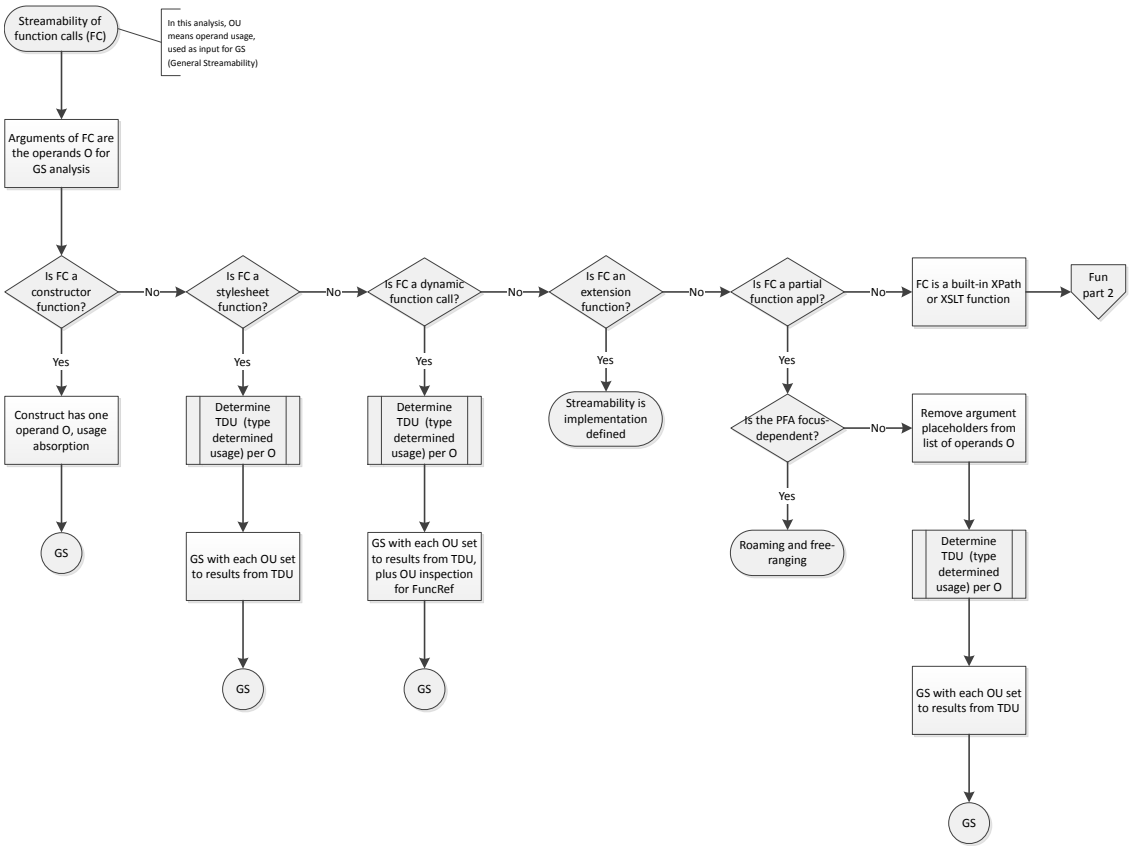
The following two flowcharts can be used for determining the streamability of an axis step, as referenced in the core expression flowcharts.

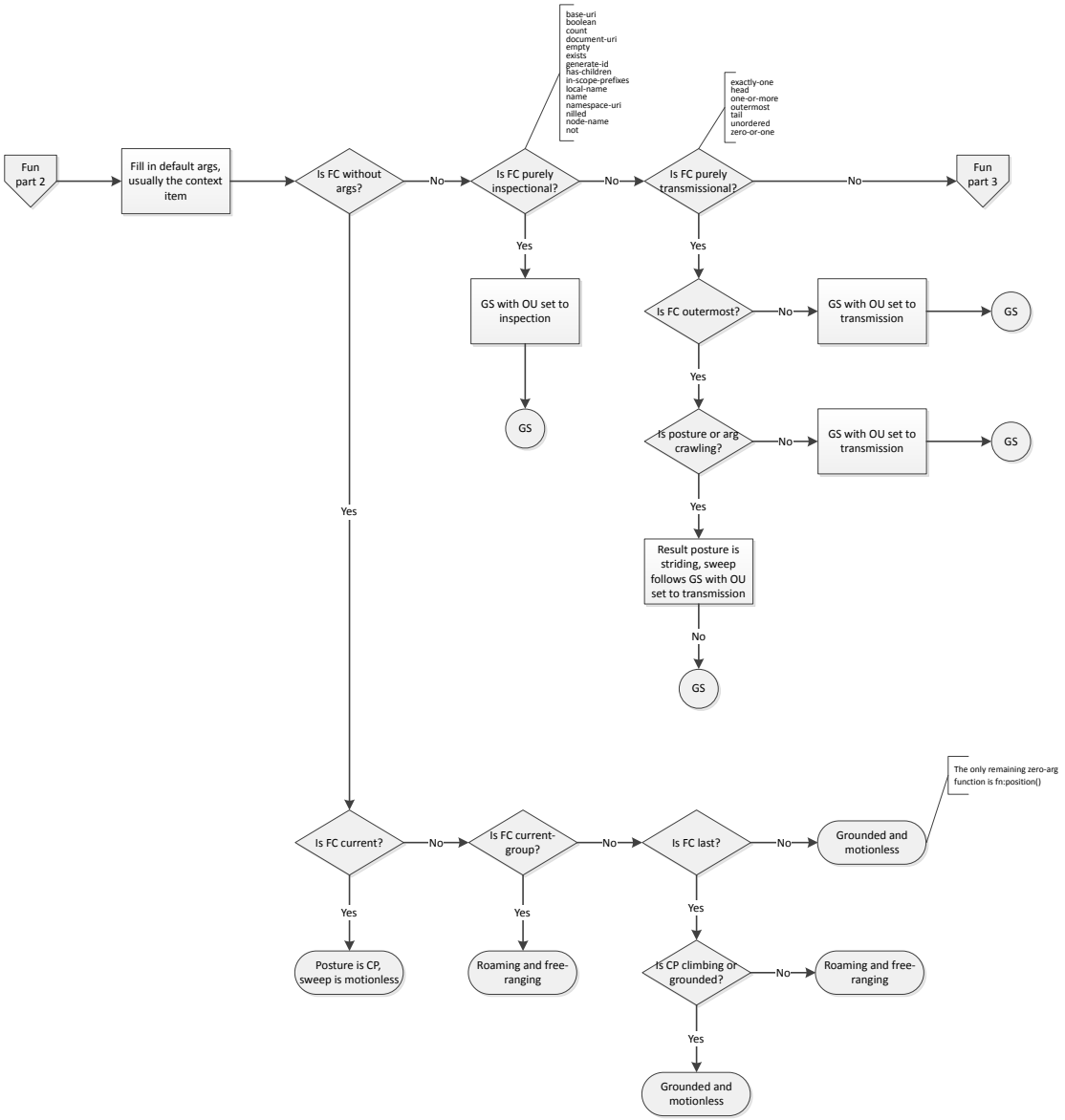


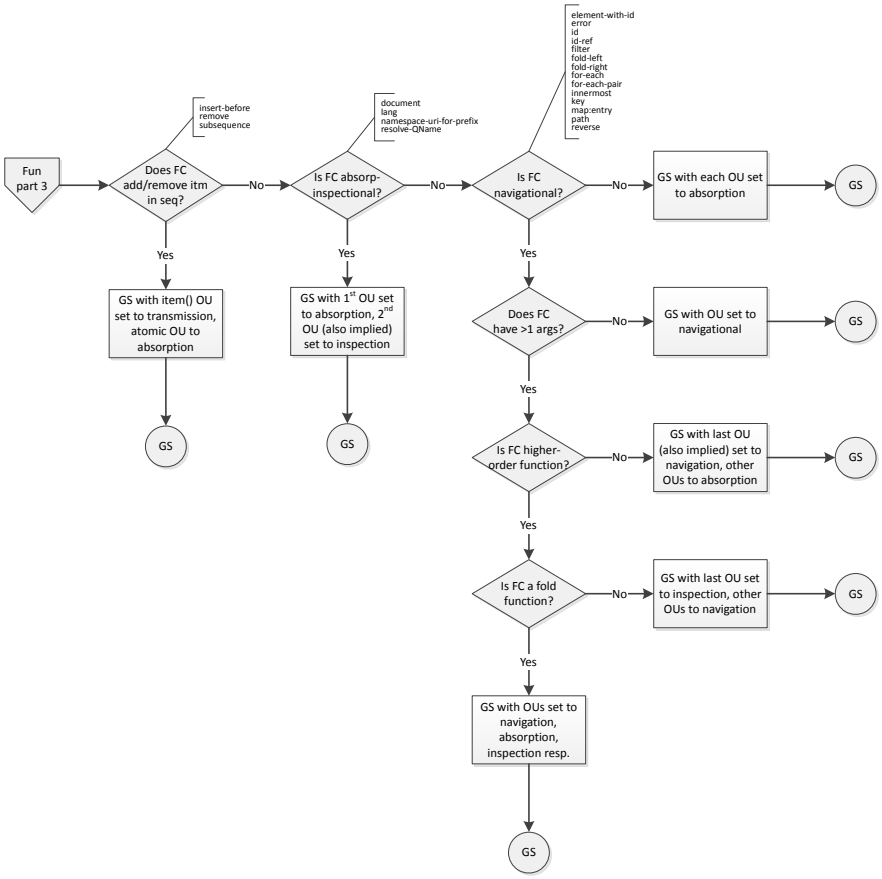


7.11. Flowcharts for determining streamabiity of functions

The following three flowcharts can be used for determining the streamability of a function, when used in an expression or a pattern.




Fun part 2
Fill in default args, usually the context item
Is FC without args?
No
Is FC purely inspectional?
base-uri
boolean
count
document-uri
empty
exists
generate-id
has-children
in-scope-prefixes
local-name
name
namespace-uri
nilled
node-name
not
No
Is FC purely transmissional?
exactly-one
head
one-or-more
outermost
tail
unordered
zero-or-one
No
Fun part 3
Yes
GS with OU set to inspection
GS
Yes
Is FC outermost?
No
GS with OU set to transmission
GS
Yes
Is posture or arg crawling?
No
GS with OU set to transmission
GS
Yes
Result posture is striding, sweep follows GS with OU set to transmission
No
GS
Yes
Is FC current?
No
Is FC current-group?
No
Is FC last?
No
Grounded and motionless
The only remaining zero-arg function is fn:position()
Yes
Posture is CP, sweep is motionless
Yes
Roaming and free-ranging
Yes
Is CP climbing or grounded?
No
Roaming and free-ranging
Yes
Grounded and motionless



Bibliography

[1] *Elevator going down: the story of Muzak*. http://www.redbullmusicacademy.com/magazine/history-of-muzak. Luke Baumgarten. 2012.

[2] *Binary message forms in computer networks*. http://tools.ietf.org/html/rfc31⁇. Daniel Bobrow. 1968.

[3] *Lazy processing of XML in XSLT for big data*. Presented at XML London 2013 http://xmllondon.com/2013/presentations/braaksma/ or direct download: doi:10.14337/XMLLondon13.Braaksma01. Abel Braaksma. 2013.

[4] *Burst mode streaming extension in Saxon*. http://saxonica.com/documentation9.4-demo/html/sourcedocs/streaming/burst-mode-streaming.html. Michael Kay.

[5] *New HOST-HOST protocol*. http://tools.ietf.org/html/rfc33⁇. C. Stephen Carr. 1969.

[6] *EXPath, Standards for Portable XPath Extensions*. http://expath.org/. Collaboration of several authors.

[7] *Exselt, a concurrent streaming processor*. http://exselt.net. Abel Braaksma, Vitaliy Yudenkov, and Eugene Fotin.

[8] *XPath and XQuery Functions and Operators 3.0, latest version*. http://www.w3.org/TR/xpath-functions-30/. Michael Kay.

[9] *XPath and XQuery Functions and Operators 3.0, W3C Proposed Recommendation 22 October 2013*. http://www.w3.org/TR/2013/PR-xpath-functions-30-20131022/. Michael Kay.

[10] *The FORTRAN Automatic Coding System*. http://www.bitsavers.org/pdf/ibm/704/FORTRAN_paper_1957.pdf. J.W. Backus. 1957.

[11] *MPEG: A video compression standard for multimedia applications*. http://www.stanford.edu/class/ee398a/handouts/papers/Gall%20-%20MPEG.pdf. Didier le Gall. 1991.

[12] *Querying Sliding Windows Over Online Data Streams. EDBT 2004 Workshops PhD, DataX, PIM, P2P&DB, and ClustWeb, Heraklion, Crete, Greece, March 14-18, 2004. Revised Selected Papers.* Computer Science Volume 3268, 2005, pages 1-11. http://dx.doi.org/10.1007/978-3-540-30192-9_1. Lukasz Golab and Jennifer Widom. 2004.

[13] *HTTP Client Module, EXPath Candidate Module 9 January 2010*. http://expath.org/spec/http-client. Florent Georges.

[14] *Implementation of the EXPath HTTP Client for Java and Saxon*. https://github.com/fgeorges/expath-http-client-java. Florent Georges.

[15] *Working with web services using the EXPath HTTP client*. https://www.ibm.com/developerworks/library/x-expath/. James R. Fuller.

[16] *Efficient randomized pattern-matching algorithms*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.9502. Richard M. Karp and Michael O. Rabin. 1987.

[17] *Processing XML data stream(s) using continuous queries in a data stream management system*. US Patent. US20080120283. http://www.google.com/patents/US20080120283. Zhen Hua Liu, Shailendra K. Mishra, and Muralidhar Krishnaprasad. 2006.

[18] *A Streaming XSLT Processor, presented at Balisage*. http://www.balisage.net/Proceedings/vol5/html/Kay01/BalisageVol5-Kay01.html. Michael Kay. 2010.

[19] *Stylesheet Modularity in XSLT 3.0, presented at XML Amsterdam*. http://www.xmlamsterdam.com/2013/sessions#xslt3. Michael Kay. 2013.

[20] *Saxon XSLT processor*. http://saxonica.com. Michael Kay.

[21] *Electrical Signaling*. US Patent. US0001641608. http://www.google.com/patents/US1641608[42]. George O. Squier. 1922.

[22] *Proceedings of the Thritieth international conference on Very large databases - Volume 30*. Pages 324 - 335. http://dl.acm.org/citation.cfm?id=1316719. Utkarsh Srivastava and Jennifer Widom. 2004.

[23] *XQuery and XPath Data Model 3.0, latest version*. http://www.w3.org/TR/xpath-datamodel-30/. Norman Walsh, Anders Berglund, and John Snelson.

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

[25] *XML Path Language (XPath) 3.0, W3C Proposed Recommendation 08 January 2013*. http://www.w3.org/TR/2013/PR-xpath-30-20131022/[43]. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[26] *XXProc: An XML Pipeline Language, W3C Recommendation 11 May 2010*. http://www.w3.org/TR/xproc/. Norman Walsh, Alex Milowski, and Henry S. Thompson.

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

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

[42] http://www.w3.org/TR/2013/CR-xpath-datamodel-30-20130108/
[43] http://www.w3.org/TR/2013/CR-xpath-30-20130108/

Streaming in the Saxon XSLT Processor

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

Streaming is a major new feature of the XSLT 3.0 specification, currently a Last Call Working Draft. This paper discusses streaming as defined in the W3C specification, and as implemented in Saxon.[1] *Streaming refers to the ability to transform a document that is too big to fit in memory, which depends on transformation itself being in some sense linear, so that pieces of the output appear in the same order as the pieces of the input on which they depend. This constraint is reflected in the W3C specification by a set of streamability rules that determine statically whether a stylesheet is streamable or not.*

This paper gives a tutorial introduction to the streamability rules and they way they are implemented in Saxon. It then does on to describe the implementation architecture for implementing streaming in the Saxon run-time, by means of push pipelines, and gives rationale for this choice of architecture.

1. Introduction

Even seasoned readers of W3C specifications may find it bewildering to read, in the Last Call draft of XSLT 3.0[7], that "if exactly one operand O of a construct C is potentially consuming, and if the operand usage of O is absorption or inspection, then the posture of C is grounded and the sweep of C is consuming". Welcome to the language of streamability. This talk has two purposes: firstly to give an introduction to these concepts, and secondly to explain how they relate to the challenge of actually building a streaming implementation of XSLT.

Streaming is one of the main planks of XSLT 3.0 (the other is stylesheet modularity). Streaming is rather informally defined in the specification as "a manner of processing in which documents are not represented by a complete tree of nodes [in memory], but rather... as a sequence of events". The definition is deliberately fuzzy, to give maximum scope for implementors to innovate around it. Despite this, the specification gives a very precise definition of a subset of the language that is deemed to be "guaranteed streamable", which means that every processor that claims to

[1] References to Saxon in this paper refer to the current development snapshot, that is, to the state of the code base some time after the release of version 9.5 and some time before version 9.6. There may therefore be no public release of Saxon that corresponds in every respect to the description herein.

implement streaming at all must be capable of streaming this subset (which in turn means that when this subset is used, the processor is expected to be capable of handling indefinitely large source documents.)

Inevitably, in formulating the rules that define this subset, the working group had in mind ideas as to how streaming might be implemented in real processors, and the kind of constraints they might be operating under. Some of the constraints can be formalized, at least in principle: for example, to be streamable, there must be some kind of ordered correspondence between the events representing the source tree and the events representing the result tree. But the WG has not attempted to articulate the constraints in these terms; rather it has used intuitive reasoning to recognize that some functions and operators (such as `min()`, `max()`, and `sum()`) can be evaluated in a forwards pass through the document, and others (such as sorting, or `reverse()`) can not.

The *general streamability rules* that emerge are derived essentially from a process of abstracting these observations into a set of general rules. How do `max()` and `count()` differ, for example? The answer is that `count()` has no problems handling an input sequence that contains overlapping nodes, whereas `max()` cannot handle overlapping nodes without buffering. The streamability of a function like `count()` or `max()` thus depends on two factors: the nature of the supplied argument (does it contain streamed nodes, and if so, can they overlap?) and the way in which the function uses the items supplied as the argument. The first property is called "posture", the second (more intuitively) is called "usage". When the posture is striding, overlapping streamed nodes are allowed, when it is crawling, they are not. From this we get rules that say, for example, that (for an expression to be streamable), if the operand usage is inspection then the posture can be striding or crawling, but if the operand usage is absorption then the posture must be striding.

Streaming in Saxon[6] divides into two parts. The first part is static analysis to determine whether a construct is streamable and to devise the streamed execution plan. This follows the W3C analysis very closely, though Saxon implements some extensions, for example where it is able to take advantage of optimizations such as function and variable inlining. The second part is the actual streamed evaluation at run-time. Streamed execution is in principle possible using either a pull or push approach. The merits of the two approaches were described in [5]. To summarise the conclusions of that paper, the main advantages of a pull approach are (a) the ability to merge two streamed inputs (for constructs such as `deep-equal()`, the union operator, or the new `<xsl:merge>` instruction), and (b) easier coding, because most of the state of the processing can be kept on the programming language stack. By contrast, the advantage of push processing is that input events can be directed to more than one destination, which is essential for constructs such as `<xsl:fork>`. Saxon's streaming implementation is based largely on push processing, because although the implementation is more work, the architecture is more flexible. The

fact that significant components of Saxon have always used push processing (for example, the schema validator and the serializer) is another contributory factor.

The push pipelines used for streamed evaluation in Saxon are interesting because they include a mix of fine-grained events (startElement, endElement), and complete items (including complete trees). The paper will include some examples of how a few simple streamable expressions translate into such hybrid-granularity pipelines, and how the structure of these pipelines relates to the classifications established by the W3C streamability model.

2. Streamability

The static analysis performed by Saxon is modelled very closely on the rules in the W3C specification. The main concern in these rules is to show that

- the body of an `<xsl:stream>` instruction, and
- the body of an `<xsl:template>` whose mode is declared with `streamable="yes"`

are in fact streamable.

2.1. The W3C Streamability Rules

The rules (given in section 19 of the XSLT 3.0 specification) appear complex but once formulated, they are not in fact difficult to implement. Most of the apparent complexity is not in the logic of the rules, but in understanding the abstractions used in the rules, and understanding why the rules work.

The rules start with the idea of modelling a stylesheet (or at least, the parts of it that need to be analysed) as a tree of constructs. *Construct* is our first new technical term: it's a generalisation of an XPath expression, an XSLT instruction, and a few other things that are capable of being evaluated, like sequence constructors and patterns. The children of a construct in the construct tree are called its operands. The result of evaluating a construct is always a value. (Which sounds obvious, but we would have to modify this to handle FLWOR expressions in XQuery, which deliver not values but tuple streams.)

The sweep indicates how much of the input document is needed to evaluate the construct. The values are:

- *Motionless*: the construct either doesn't look at the input document at all, or it only needs to look at the place where the input document is currently positioned. Examples are `2+2`, `name()`, and `@status`. This relies on an assumption that as the input document is read, the system maintains a stack holding the names and attributes of the current node and all its ancestors, and the contents of this stack are always available without moving the input position.

- *Consuming*: the construct needs to read everything between the current start tag and the corresponding end tag. Examples are `string()`, `data()`, `number()`, `<xsl:value-of>`, `<xsl:copy-of>`.
- *Free-ranging*: the construct potentially needs to read outside the slice of the document represented by the current element and its ancestors. Examples are preceding-sibling::x, and xsl:number. Such constructs are never streamable. (But note, this doesn't prevent preceding-sibling::x or xsl:number appearing in a streamable stylesheet; the construct is free-ranging only if it operates on the streamed input document.)

The other property of a construct that affects streamability is a bit harder to visualize, and is referred to as the *posture* of the construct. Posture is concerned with determining whether an expression returns nodes from the streamed input document, and if so, where these nodes come from. There are five values:

- *Grounded*: this means that the expression doesn't return nodes from the streamed input. It either returns atomic values (or function items), or it returns nodes from non-streamed documents only.
- *Striding*: this means that the expression returns a set of nodes from the streamed input document, in document order, and that none of these nodes will contain another node in the result (none is an ancestor or descendant of another). A typical example is an axis expression using the child axis.
- *Crawling*: again, the expression returns a set of nodes from the streamed input document, in document order, but this time some of the nodes may be ancestors or descendants of others. A typical example is an axis expression using the descendant axis.
- *Climbing*: The spec assumes that when an input document is streamed, a stack of information is retained containing details of the names and attributes of all ancestor elements of the element at which the stream is currently positioned. Any expression that accesses ancestor nodes or their attributes from this stack has a posture of climbing. The key thing to remember about climbing expressions is that you can go upwards to ancestors of the current node, but you can't then navigate downwards again, because the children/descendants of these nodes are not retained in memory.
- *Roaming*: This indicates that an expression navigates off to parts of the document that aren't accessible when streaming, such as preceding or following siblings. This always makes the containing expression non-streamable.

Although some constructs have their own special rules, it's worth summarising and explaining the general streamability rules that apply to most instructions and expressions. The rules aim to determine the sweep and posture of a construct. The rules depend on identifying the operands (subexpressions) of a construct; for each operand you potentially need to know:

- its static type (this in fact is not often used)
- the sweep and posture of the operand (which you get by applying the rules recursively)
- the way in which the value of the operand is used, called the operand usage. This is one of the following:
  - *Absorption*: the parent expression makes use of information from the entire subtree rooted at nodes returned by the operand expression. Examples: `string()`, `data()`, `.//descendant::x`
  - *Inspection*: the parent expression makes use of properties of the nodes returned by the operand expression that can be established while positioned at a node's start tag. Examples: `name()`, `base-uri()`, `@status`, `../@status`.
  - *Transmission*: the parent expression returns nodes delivered by the operand expression. Examples: `A|B`, `tail(X)`, filter expressions.
  - *Navigation*: the parent expression performs arbitrary reordering of the returned nodes, or navigates away from them in arbitrary ways. Examples: `reverse()`, `<xsl:number>`.

The general streamability rules start by refining the sweep and usage of the operands by taking additional information into account. Specifically:

- If the type of the operand is a childless node kind, for example `text()`, then usage *absorption* is changed to *inspection*, because the entire subtree of such nodes is a simple property of the node and doesn't involve advancing the input stream.
- If the usage of the operand is *absorption* (for example if the parent expression atomizes the value of the operand), then the sweep of the operand may have to be increased. For example given the expression `contains(., "e")`, the sweep of the first operand is *consuming*, not because "." is intrinsically consuming, but rather because the `contains()` function performs atomization and this involves reading the whole subtree of the context node.

Once the properties of all the operands have been established in this way, the properties of the parent expression can be established:

- If there aren't any operands, the expression is *grounded* and *motionless*. This applies for example to simple literals like `"London"`, and also to the empty sequence `()`. It doesn't apply to axis expressions such as `child::*`, because axis expressions have special streamability rules. The general streamability rules described here are only the default.
- If any operand is non-streamable (technically, if it is *free-ranging* or *roaming*) then the parent expression is also non-streamable.
- If several operands are *consuming*, then in general the parent expression is not streamable (it is free-ranging and roaming). We'll discuss this important rule

below, There are exceptions for conditional expressions, where both branches can be consuming.

- If exactly one operand is *consuming*, then the parent expression will usually have the sweep of that operand. An exception is where the consuming operand is evaluated more than once (consider an expression such as `(1 to 5)!child::x`) in which case the result is not streamable. The posture of the result depends on the operand usage of this operand. If the usage is *transmission* (for example `X[@a = 3]`) then the posture of the result is the same as the posture of the operand. If the usage is *inspection* or *absorption* (for example `name()` or `data()`), then the posture of the result is *grounded*, because the result does not include any streamed nodes.

So there's a general rule that (with a few exceptions), no construct can have two operands that are both consuming. This rule is fairly easy to learn, and it's fairly easy for an implementation to give good diagnostics that explain when it's been violated. it's also fairly easy to understand why it should be true: you can only scan the input file once, and unless you're pretty smart, you can only evaluate one expression while doing so.

The exceptions are cases where the implementation is expected to be smart enough to evaluate both operands during a single pass:

- The `<xsl:fork>` instruction is explicitly introduced to request evaluation of two or more instructions during a single pass. One can imagine this being done by two parallel threads, but in fact it doesn't need true parallelism: as we'll see later in the paper, Saxon implements it simply by passing each parsing event to several expression evaluators in turn.
- Union expressions such as `a|b`, and map expressions such as `map{'a': price, 'b': discount}` can also have multiple consuming operands.
- A rather different case is conditional expressions (`<xsl:choose>`, or XPath `if-then-else`) where both branches can be consuming. This is a bit different because only one of the branches is actually evaluated.

If implementations have to be smart enough to evaluate `<xsl:fork>` and map constructors, then one might reasonably ask why we don't require them to evaluate multiple consuming operands wherever they occur, rather than treating these constructs as a special case. Perhaps some of the reason is pure caution; if there were no constraints at all, the number of parallel evaluations could run completely out of control. This reflects a recognition that forked evaluation has a cost, and indeed, that it's not really pure streaming, because although the input is streamed, the output has to be buffered so that the results of the separate construct evaluations can be assembled in the right order on completion. XSLT has a tradition of not leaving everything to the optimizer but allowing programmers to get involved in some of the key performance trade-offs, and this is an example of this philosophy.

The rules given above (the general streamability rules) apply to most kinds of expression, but they don't apply to the important case of path expressions and axis expressions. For axis expressions, the posture of the result depends on the posture of the context item and the choice of axis, using transition rules like the following:

- striding + child => striding
- striding + parent => climbing
- grounded + any => grounded
- climbing + child => roaming
- crawling + child => roaming

This last rule is one of the trickiest to get used to. The rule in its simplest form can be stated as "if you reached a node via the descendant axis, then you can't select downwards from it".

The reason for this rule is as follows. Suppose you select a sequence of nodes using the descendant axis. Then, in general, this sequence can contain two nodes where the first is an ancestor of the second. Suppose you want to process all the nodes in this sequence in turn. When you evaluate a consuming expression while positioned at the first node (the ancestor), this will move the position in the input stream to the end of that node, by which time you will have moved past the second node (the descendant), which is the next one you want to process.

The way that posture is used in determining the streamability of path expressions gives us another way of thinking about what posture actually means. Suppose that all the navigation in a template is reduced to a simple path, then that path has to match the regular expression `C*D?A*`, where `C` is a child step, `D` is a descendant step, and `A` is an ancestor or attribute step. It turns out that the rules for permitted posture transitions effectively define a finite state automaton that is equivalent to this regular expression; the posture values, with their fanciful names such as striding, crawling, and climbing can be seen as labels for the states in this automaton.

Note that the use of the descendant axis does not have to be explicit to fall foul of this rule. Operations such as taking the string value or typed value of a node, which are used all the time in XSLT programming, implicitly make a downward selection and are therefore not allowed on nodes that were reached via the descendant axis.

The rule disallowing multiple descendant steps is without doubt a great inconvenience. There are a number of workarounds:

- If you know, for example, that `<title>` elements will not be nested, then you can use the function `outermost(//title)` to select those titles that do not contain other titles. This expression, because it always selects nodes with disjoint subtrees, is deemed striding rather than crawling, and therefore allows further downward selection.

- If you only need a single node, you can write this in various ways: `head(//title)`, or `(//title)[1]`, or `zero-or-one(//title)`. Again these expressions cannot return nested nodes, so they are deemed striding rather than crawling.
- Similarly, text nodes are never nested, so the expression `//text()` is also striding.
- If several downward steps occur in a simple path expression such as `//section/title`, the specification says this is to be treated as equivalent to `//title[parent::section]` – that is, it is crawling rather than roaming. The reason here is that path expressions select nodes in document order, so it's always possible to evaluate the entire path in a single scan of the subtree under the current node, making it equivalent to a single use of the descendant axis.

The problem with this rule, as it appears in the W3C spec, is that it is very rigid. A great deal of the time, it prevents you writing constructs that you, with knowledge of the data, know will actually be streamable in practice even though they are not streamable in the worst case. Saxon therefore takes a more pragmatic view here (the spec permits this). Given a construct like the one above, Saxon will attempt an optimistic streamed implementation. If while processing one `<section>` element it encounters another nested `<section>` element, then it will process both of them in parallel during the same pass over the input. Any output produced from the inner, nested `<section>` will be buffered and emitted only when processing of the outer `<section>` is complete. So in the worst case, the process is not fully streamed, but it will still produce the right answer if enough memory is available for the buffered results. In effect, Saxon is doing an implicit `<xsl:fork>`: when it finds that a crawling expression produces two nodes where one contains the other, and there is then a further downward selection from these nodes, then it evaluates these two downward selections in parallel, buffers the results, and assembles the output in the correct order at the end. The beauty of this is that in the common case where elements are not in fact nested (as would typically be the case for `<xsl:value-of select="//title"/>`), no buffering is ever necessary, and the convenience of being able to write the expression in the natural way is delivered without any performance penalty and with no risk of running out of memory.

2.2. Visualising the Streamability Rules

Evaluating the streamability rules by hand for anything but trivial examples is challenging; the detail quickly becomes overwhelming, especially as the rules are highly recursive. This is of course a serious usability problem since stylesheet authors need to know whether they are writing streamable code or not.

With experience, the effect of the rules starts to become more predictable. One quickly develops an eye for coding patterns where the result of applying the rules is immediately obvious. Two of these patterns (expressions with multiple consuming

operands, and downward selection from a node reached using the descendant axis) have already been discussed.

However, for cases where the behaviour of the rules is less obvious, and for the benefit of users who have not yet formed the ability to predict the effect of the rules, Saxonica has developed a tool that allows the construct tree to be visualized, with all the properties of each construct that are relevant to streaming (sweep, posture, usage, type, context item posture, context item type) explicitly displayed.

The tool can be found at `http://dev.saxonica.com/stream`. At the time of writing it does not handle all the rules in the W3C specification, but it handles all the most frequently-encountered ones.

The tool is implemented using Saxon-CE[1] (XSLT logic running client-side in the browser).

2.3. Implementation of the Streamability Rules in Saxon

Saxon internally implements the streamability rules by means of a method `getSweepAndPosture()` on its `Expression` class (which corresponds to what the specification calls a *Construct*). The general streamability rules are defined on the class `Expression` itself, and constructs that have their own special rules override the method as required. The method takes a parameter to indicate whether the evaluation should proceed strictly according to W3C rules, or whether Saxon extensions are permitted. This allows the user to decide whether to take advantage of Saxon extensions or to prefer portability.

Implementation of the rules is not difficult. The expression class provides a method `operands()` which returns the operands of an expression together with their usage; it also provides static type information. So all the input to the W3C rules is readily available. Once the sweep and posture of an expression have been computed, the results are saved in the expression tree to avoid the costs of multiple computation.

Users don't only want to know whether an expression is streamable, they also want to know why not. So the method `getSweepAndPosture()` also takes an (output) parameter called reasons, which on return, if the expression is not streamable, contains messages explaining which rules were violated; these messages are used as the basis for compiler diagnostics.

Saxon performs the streamability analysis after all type-checking and optimization is complete. This creates the possibility that non-streamable code will be rewritten by the optimizer as streamable, or vice-versa.

The first case is not a problem, except for the rather stringent requirement in the W3C specification that an implementation should be capable of distinguishing stylesheets that are "guaranteed streamable" according to the spec, from those that rely on implementation extensions for their streamability. The only way to achieve that with Saxon is to switch optimization off.

Rewriting streamable code as non-streamable would be more of a problem for users. The problem is avoided by ensuring that the optimizer is aware of the need for streaming. In most cases this merely suppresses a rewrite that would otherwise take place, for example the use of indexing to support filter expressions such as `//emp[@id=$id]`.

In a few cases the optimizer deliberately tries to turn a non-streamable expression into one that is streamable. An example is the expression `for $x in //emp return ($x/@name, $x/@salary)`. This is not streamable as written because it is not permitted to bind a variable to a node in a streamed document. However, it can be rewritten as `//emp/(@name, @salary)`, which is indeed streamable.

On other occasions streamability is achieved as an unintended consequence of optimization. For example, the streamability rules don't allow streamed nodes to be bound to variables, passed as arguments to functions, or returned from functions (this is primarily to avoid the need for complex data-flow analysis). The Saxon optimizer will bypass this rule when it does variable and function inlining (replacing a variable reference or function call by the body of the variable or function). For example, a call to the function

```
<xsl:function name="inc">
  <xsl:param name="n"/>
  <xsl:sequence select="$n + 1"/>
</xsl:function>
```

is not streamable according to the W3C rules, simply because it fails to declare the type of its argument (and could therefore be processing a streamed node). After optimization, however, this function call will have been expanded inline, and the expanded code will satisfy all the streamability rules.

3. Run-time execution

While the W3C specification has a lot to say about how a stylesheet is analyzed to classify its constructs as streamable or not streamable, it says nothing at all about how to actually organize evaluation at run-time in a streaming manner.

The architecture of a typical XSLT 1.0 processor [3] is shown in Figure 1. The data flow is from left to right, but the control flow is more complex. In fact there are two control modules: the XML parser reads (pulls) data from a lexical XML input stream and writes (pushes) it to a tree in memory. The XSLT transformer, via its XPath engine, reads (pulls) data from this tree, and then writes (pushes) events down a pipeline which constructs events representing nodes in the result tree, which are in turn pushed to the serializer.

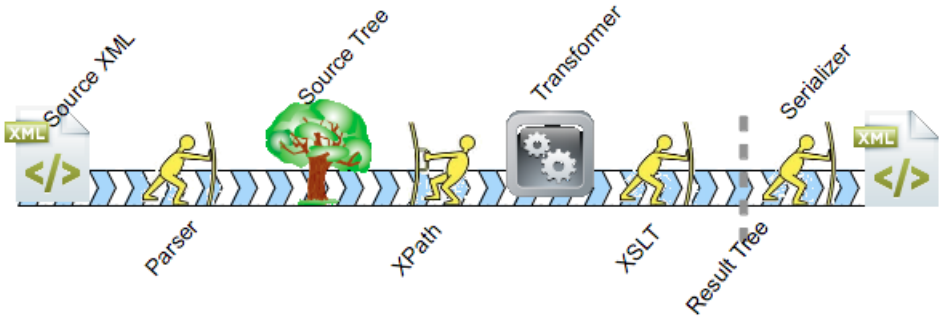


Figure 1. The architecture of a typical XSLT 1.0 processor

Note that the source tree is materialized in memory, but the result tree is not. Evaluation of XSLT instructions that construct nodes, and the serialization of those nodes, operate in a seamless push pipeline.

XSLT 1.0 famously does not allow a stylesheet to create temporary trees and then process them further using XPath; but in practice all 1.0 processors implement the EXSLT `node-set()` extension which circumvents this restriction. A typical XSLT 1.0 processor with the `node-set()` extension operates as shown in Figure 2:

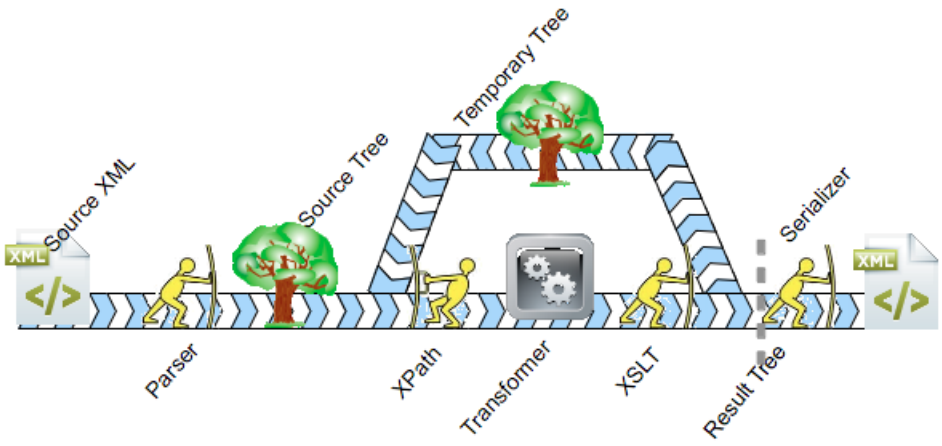


Figure 2. An XSLT 1.0 processor with the node-set() extension

Here variables containing temporary trees are materialized as trees in memory by XSLT instructions operating in push mode, and they are read by XPath expressions operating in pull mode.

XSLT 2.0 adds the possibility of schema validation, which can be applied to source trees, result trees, and also to temporary trees. The places where a schema validator can be invoked are shown with red tick-marks in Figure 3:

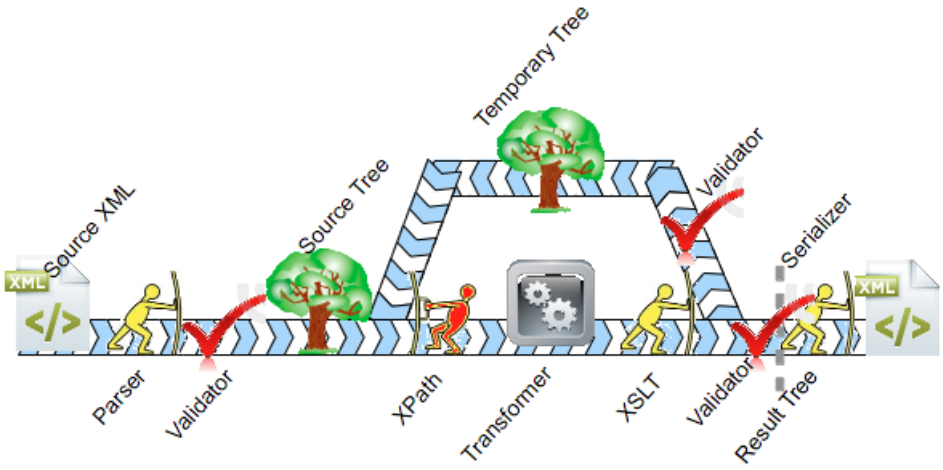


Figure 3. An XSLT 2.0 processor with schema validation

The XML Schema specification was designed to allow validation to be streamed: that is, one can determine schema validity over a stream of events representing the instance document, without needing to materialize the instance document as a tree in memory. Although one could envisage schema processors operating in either pull or push mode, in practice all the ones I know of work in push mode, and it can be seen in this diagram that this is rather convenient because in each case we have added the schema processor to a push pipeline.

In this architecture there are two kinds of pipeline: a push pipeline for the parsing and source validation, a pull pipeline for XPath evaluation, and another push pipeline for result tree consruction, result tree validation, and serialization.

A simple pipeline contains one control module which pulls data from the source end of the pipeline and pushes it to the result end of the pipeline. Data can flow naturally from a pull pipeline to a push pipeline, but the opposite is more difficult. There are essentially two ways to do it. One is to buffer the data into a reservoir from where another pipeline can read it; that needs memory. The other is to have to control modules that operate in some kind of synchrony so that data pushed by one is pulled by the other. This can be achieved by running the two control modules in separate threads under some kind of synchronization control, or with appropriate support from a programming language it can be achieved in a single thread by use of co-routines.

For more detail on these concepts, see [5]. The basic ideas are not at all new. Until the advent of large online disc storage, most data processing was done with magnetic tapes, and a major objective was to perform streamed processing of hierarchic data held in sequential form to transform it to another hierarchic data set also held in sequential form, with minimal use of tapes for holding intermediate data. Michael Jackson built many of the ideas of Jackson structured programming

[2] around these concepts, and the ideas are fully applicable to pipelines of XML transformations today.

Both pull and push pipelines can perform well, but a turbulent pipeline that has to switch between push and pull mode is likely to be less efficient. Some measurements demonstrating this effect can be found in [4].

To eliminate the need for interrupting the transformation pipeline with a reservoir that holds everything in memory, one can envisage a number of possible architectures:

- a single pipeline that operates in push mode from end to end.
- a single pipeline that operates in pull mode from end to end.
- a pipeline that has both push and pull sections, with the push-pull transitions being handled through multithreading (the co-routine alternative can probably be eliminated because of the paucity of modern programming languages that support the concept).

In Saxon's first forays into streaming, the third approach was adopted. This had the advantage that it was least disruptive to the existing architecture of the product; in particular, the XPath engine could continue to operate in pull mode.

In a pull mode XPath engine, evaluation of XPath expressions operates top-down. A parent expression controls the evaluation of its child expressions, typically asking child expressions to deliver their results as a stream of items which can be read as required. There is usually no need for the entire result of a child expression to be stored in memory, because each item can be processed as it becomes available. For example, the `sum()` function might be coded like this:

```
Iterator sum() {
  int total = 0;
  for item i in argument[0].evaluate() {
    total += i;
  };
  return monoIterator(total);
}
```

In general, every XPath construct is implemented by a function that consumes iterators representing the results of its subexpressions, and that itself delivers an iterator over its results. That is, each XPath construct is implemented by a component of a pull pipeline. This design approach is common in the implementation of functional programming languages; it can be seen as a combination of the Interpreter and Iterator design patterns in [Gamma et al].

In particular cases, it is possible to recognize XPath expressions where the tree does not need to be materialized. An example might be `sum(doc('employee.xml')//employee/salary)`. This could be implemented either by having the XPath engine make calls on an XML pull-mode parser as each salary element is required; or, as in the Saxon case, it could be implemented by having a push-mode XML parser

deposit a sequence of salary elements in a cyclic buffer, to be picked up by the XPath engine running in a separate thread. In Saxon this style of processing is implemented using the `saxon:stream()` extension function.

While this approach allows some useful applications to be written, it has many serious limitations. In particular, it does not allow for streaming applications that need to process the input data hierarchically. it's very hard to see how the XSLT 3.0 mechanism for streamable template rules, which perform a top-down hierarchic (but sequential) processing of the source tree, could be implemented using this kind of architecture.

Instead, Saxon is moving inexorably towards the first approach: an end-to-end push pipeline, illustrated by Figure 4.

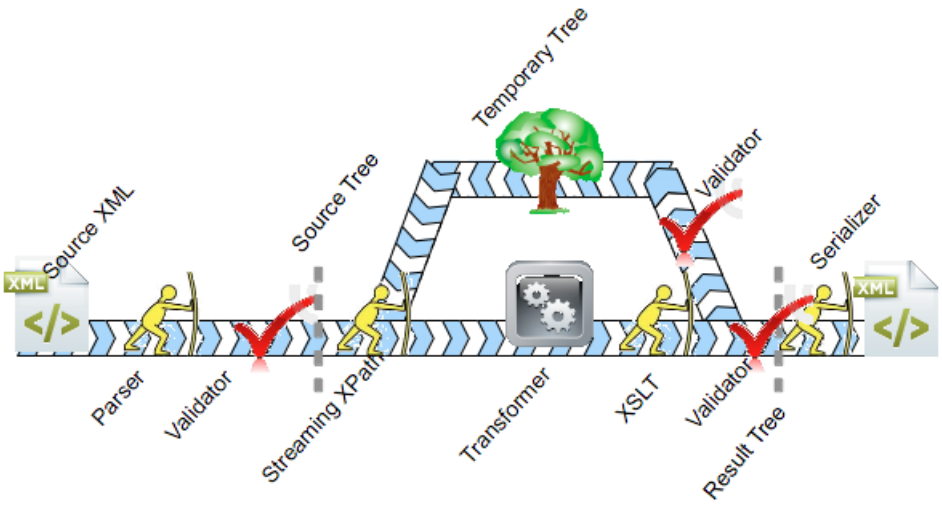


Figure 4. An XSLT 3.0 streamed processor using a pure push pipeline

Many components of Saxon have always been implemented as push pipelines, notably the XSLT instruction engine, node construction, serialization, and schema validation. The serializer alone contains around 30 components which are available to be assembled into a push pipeline based on the serialization options selected; the schema validator also has around 30 components each performing separate tasks. So the only part of the run-time engine which needs to be re-engineered for this streaming architecture is the XPath engine, where the existing pull-mode components need to be replaced by components that work in push mode. The next section of the paper explains how this works.

It's worth noting that where expressions operate on singleton values (for example, arithmetic expressions), the same code can be used in either a pull or push pipeline; the entire input value is available in materialized form, so pipelined evaluation becomes meaningless. By contrast, there are a few expressions (an example is the `insert-before()` function) that have more than one sequence-valued operand, and

where the roles of these operands are not symmetric. In a push pipeline multiple push-based implementations of such an expression are needed, depending on which operand is streamed (the rule that requires at most one operand to be consuming ensures that we can always choose one or the other, and of course we can select which one statically, because we know statically which operand is consuming).

3.1. Example of Push-based Expression Implementation

Perhaps some code would make these ideas more concrete. Here is an example that shows a simplified implementation of the `sum()` function in push mode. (it's simplified by only handling integers, by ignoring the second operand which gives a zero value, and by ignoring conditions such as overflow). Note that `IntegerValue` is a subclass of Item.

```
IntegerValue total;
void open() {
  total = 0;
  getResult().open();
}

void processItem(Item it) {
  total = total.add((IntegerValue)it);
}

void close() {
  getResult().processItem(total);
  getResult().close();
}
```

In push mode, evaluation is bottom-up, so these methods are called by whatever component it is that is evaluating the argument to the `sum()` function. That component is responsible for delivering a sequence of items; each one is delivered by calling `processItem()`, and the sequence is topped and tailed by calls of `open()` and `close()`. The component implementing `sum()` delivers a singleton sequence to the next component in the pipeline (available as `getResult()`), and this too is delivered using a sequence of three calls: `open()`, `processItem()`, and `close()`.

In this example, the things passed from one component to another are complete items. This is always the case for functions that operate on atomic values, which is the case for `sum()` and for a great many other expressions. In fact, more generally it is true whenever the operand is *grounded* or *climbing*. For functions that operate on streamed nodes, however (specifically, *striding* and *crawling* expressions), we don't always want to assemble the items before we can process them. Consider the expression `count(//*)`: the first item to be counted is the outermost element of the document, and we don't want to construct this as an object just so that it can be

counted. For this example, the pipeline needs to operate at a finer level of granularity: it needs to be notified of `startElement` and `endElement` events. The push code for `count()` looks like this:

```
IntegerValue count;
 void open() {
   count = 0;
   getResult().open();
 }

 void startElement(FleetingNode node) {
   count = count.add(1);
 }

 void processItem(Item it) {
   count = count.add(1);
 }

 void close() {
   getResult().processItem(count);
   getResult().close();
 }
```

This implementation can handle both complete items and fine-grained events. Often it will only have to handle one or the other: if the operand is striding (e.g. `child::X`) or crawling(e.g. `descendant::X`) then it will be notified of `startElement` events, while if it is grounded (e.g. `data(X)`) or climbing (e.g. `ancestor::A/@B`) then it will be notified of complete items. Nodes other than element or document nodes are also notified via the `processItem()` method, so with an expression such as `count(//node())` the two methods will both be called, for different kinds of nodes.

The class `FleetingNode` used in the `startElement` call is a representation of a node within a streamed document. It implements Saxon's `NodeInfo` interface, which is the standard way that nodes in trees are represented, but it only supports operations that are permitted when positioned at the start tag in a streamed document (classified in the spec as *inspection* operations). That is, you can call methods such as `name()`, `localName()`, and `baseUri()`; you can determine the type annotation; you can navigate the attribute, ancestor, and namespace axes; but you cannot make a downwards or sideways selection either explicitly by following axes such as `preceding-sibling`, `child`, or `descendant`, or implicitly by getting the string value or typed value. For the `count()` function, of course, we don't need to know any properties of the node, we just need to note its existence by incrementing the tally.

As well as constructs like `count()` that process fine-grained events representing element start and element end, there are also constructs that create new element nodes (for example, the `<xsl:element>` and `<xsl:copy>` instructions). Again, we don't want to materialize these nodes in memory; instead such constructs need to

generate startElement and endElement events which can then filter their way down the output pipeline, usually ending up in the serializer where they can be turned directly into start and end tags. Alternatively, if the streamed output is being captured in a variable, they might end up being passed to a tree builder that constructs a tree in memory, but this will only happen if the tree is actually needed.

3.2. Why not pull?

It would be possible to build a streaming processor that used a uniform pull model throughout, rather than Saxon's push model. Indeed, in some ways it would be easier.

As explained in [5] the primary benefit of a push model is that it allows events to be sent to more than one destination. This is used when implementing the expressions that explicitly allow more than one consuming operand, such as union expressions, `<xsl:fork>`, and map constructors, and it is also used for the Saxon extensions that permit downward selection from nodes reached using the descendant axis.

The other significant reason for using this model for Saxon streaming is that it is already used in significant parts of Saxon such as the serializer and schema validator, and this model therefore permits better re-use of existing code.

The main drawbacks of a push model are (a) that the coding required to implement it is probably more complex, and (b) that it is more difficult to handle constructs that merge two independent streamed inputs. There is only one such construct in XSLT 3.0, the `<xsl:merge>` instruction; Saxon has yet to implement streamed merging, but when it comes, it will probably use multiple threads.

4. Constructing the Push Pipeline

In the previous section we looked at how individual constructs in the expression tree are represented by push-evaluation components feeding events to each other in a bottom-up manner. This is bottom-up in the sense that subexpressions are processed before their parent expressions, and that the code for a subexpression calls the code for its parent expression to supply data, contrasted with top-down evaluation where the control flow is in the opposite direction. The relationship is very much the same as that between a top-down parser and a bottom-up parser, and the process of converting from one form to the other is known in Jackson Structured Programming [2] as *inversion*. Jackson showed that the process of inversion can be automated by a compiler.

It would be nice to think that we could invert Saxon's pull-based implementations of operations like `count()` and `sum()` to push-based implementations by an automated process; unfortunately doing so would require creating something akin to a new Java compiler, so instead we have done the process by hand. However, the assembly of these individual expression implementations into a working push

pipeline is of course fully automated. This process is essentially performing a Jackson inversion of the streamable XSLT templates in the stylesheet, and inversion at this level is greatly simplified because XSLT is a functional language free of side-effects. It is akin to the process of creating a bottom-up parser from a top-down BNF description, which is well-understood technology.

Consider first a simple template rule such as the following:

```
<xsl:template match="employee">
   <e><xsl:value-of select="name"/></e>
 </xsl:template>
```

Saxon will build an expression tree representing this template rule. The expression tree is a little more complex than might be imagined, because it contains nodes representing all the internal operations implied by the semantics of xsl:value-of: specifically a call to `data()` to atomize the `<name>` element; a call to `string-join()` to handle the case where there are multiple `<name>` elements.

Working top-down, the rule that a construct is only permitted one consuming child allows us to construct a path through this tree that contains all the consuming operations from bottom to top: this is called the streaming route, and the expressions on the streaming route provide the raw material for assembling the push pipeline that evaluates the template rule. Expressions on the streaming route may of course have other non-consuming operands; these are evaluated top-down in the usual way, at the point where their values are needed.

At the bottom of the streaming route there is always a pattern, which identifies which nodes the template is interested in (this is not the pattern that the template matches; it is a pattern that matches the nodes which the template reads from the streamed input). In this case the first operation in the push pipeline is to atomize `<name>` elements, so the relevant pattern is `name`. When the template is activated, it constructs a `Watch`; the `Watch` is a combination of the pattern, and the pipeline to be invoked when the `Watch` is matched. This `Watch` is registered with a `WatchManager`, which receives all events emanating from the XML parser, and tests each one against a list of registered `Watch`es, to see who needs to be notified. When the start tag for `<name>` is encountered, the `WatchManager` calls the `startElement()` method for the first component of this pipeline. This component is an atomizer, so it responds to this `startElement()` call by telling the `WatchManager` that it wants to know about all events up to the corresponding `endElement()` call. As these events arrive, it constructs the typed value of the element. The typed value of an element is in general a sequence of atomic values (though in the absence of a schema, this sequence will always be of length one). Each value in this sequence is notified to the next step in the push pipeline by a call on `processItem()`, so this next step (in our example, it represents the implicit `string-join()` operation) sees the sequence of atomic values comprising its streamed input. The `string-join()` operation in our example probably does nothing very interesting, because it's likely that employees only have one name; it

passes this name on to the next operation, which converts the string to a text node as required by the xsl:value-of instruction. The next operation after this in the streaming route is the literal result element that creates the <e> element. This emits events corresponding to the <e> start and end tags as part of its open() and close() calls, while the processItem() call that supplies the text node is passed on unchanged. So the template rule as a whole delivers a sequence of three calls (startElement, text node, endElement) and these become the result of the <xsl:apply-templates> instruction in the calling template rule, to be passed on up that template rule's pipeline.

The situation becomes a little more complex, of course, with template rules that involve loops and conditionals. The logic, however, is very similar to what would happen if all loops and conditionals were translated into apply-templates calls. Saxon doesn't quite go as far as doing that literally (it probably wouldn't be very efficient), but in terms of defining patterns and registering them with the WatchManager, it gives a good picture of what is going on.

As mentioned earlier, there are some constructs like union expressions, <xsl:fork> and <xsl:map> that explicitly allow multiple consuming operands. Saxon handles these by registering with the WatchManager one pattern (and corresponding pipeline) for each consuming operand. The pipelines operate in parallel as matching nodes are encountered, and the final step in each pipeline is to leave the result somewhere (in memory) where it is available to be assembled into the final result of the <xsl:fork> or <xsl:map> instruction.

5. Early exit, and error handling

A feature of functional languages like XPath is that it is not always necessary to evaluate the whole of an operand sequence in order to establish the result of the parent expression. For example, given the expression exists(child::author) it is not necessary to find all the author children; as soon as one is found, the expression can return true.

In a pull model, this is handled naturally by the parent expression iterating over the nodes returned by the operand expression, and simply not reading any further once it knows the answer.

This is less easy to achieve in a push (bottom-up) evaluation model, because the child expression knows nothing of its parent, so it will keep supplying new author elements until it reaches the end of the sequence of children. Of course, the parent expression can simply ignore them, but there are sometimes performance benefits to be gained by avoiding the unnecessary computation.

With streaming, particular gains are possible if the result to the entire transformation can be delivered before the whole input document has been parsed. This is more likely with some XPath or XQuery scenarios than with XSLT itself, but the same principles apply. For example, one can imagine a phase of a publishing pipeline

that is merely interested to read the value of the expression `/article/@version` — that is, an attribute of the outermost element of the document, which can be found very near the start of the file. Delivering this without parsing the rest of the file is potentially a big win.

To achieve this in a push pipeline, Saxon adds a parameter to the `open()` method for each each expression, whose value is a `Terminator` object. If the expression does not need any more input, it can call the `Terminator` object to say so. This enables expressions further up the pipeline to respond by themselves terminating; potentially the `WatchManager` itself is able to recognize that no more input is needed from the XML input stream, and it can then terminate the parse by throwing a (recognizable) exception, which is then caught so the user never knows about it. This does have the side-effect that XML well-formedness errors appearing subsequently in the file will never be detected (something that can be regarded as a bug or a feature).

A related issue is the implementation of the XSLT 3.0 try/catch facility. A conventional top-down implementation of try/catch can take advantage of the exception handling provided by the implementation language, which in Saxon's case is Java. With bottom-up evaluation, however, throwing a Java exception is no use, because the Java call stack is inverted so the exception will never reach the try/catch expression that is watching for it. Instead it is necessary to notify exceptions up the push pipeline in the same way as success results.

As it happens, the XSLT 3.0 try/catch capability is not streamable according to the specification; this is because it requires either output buffering or some kind of rollback capability to ensure that output produced before a failure that is caught does not make it into the result. Saxon however is more liberal here than the specification: it does allow try/catch with streamed input, and buffers the output in case an error occurs. This extension to the streamability rules can be justified on the basis that the output is sometimes much smaller than the input; indeed, in some cases its size is independent of the input, which would make it truly streamable even with buffering.

6. Conclusions

In this paper I have given a brief introduction to the concepts that are used in defining the streamability rules in the XSLT 3.0 specification, and have outlined some of the more important rules that determine whether constructs are or are not considered streamable; I have also explained some of the circumstances in which Saxon achieves streaming even in cases where this is not guaranteed by the specification. I then went on to outline how streaming is implemented in Saxon using an end-to-end push pipeline in which both whole items and fine-grained events can be notified in a bottom-up data flow from called expressions to calling expressions, and I explained some of the consequences of this architecture.

In 2001 [3] I wrote "Perhaps the biggest research challenge is to write an XSLT processor that can operate without building the source tree in memory. Many people would welcome such a development, but it certainly isn't an easy thing to do." I was right: it took a dozen years, but it has now been achieved.

References

[1] Delpratt, O'Neil, and Kay, Michael. Multi-user interaction using client-side XSLT Presented at XML Prague 2013. http://archive.xmlprague.cz/2013/files/xmlprague-2013-proceedings.pdf

[2] Jackson, Michael A. JSP in Perspective. (A retrospective look at Jackson Structured Programming, more accessible to a modern audience than the original publications from the 1970s). SD&M Pioneers' Conference, Bonn, 2001. http://mcs.open.ac.uk/mj665/JSPPers1.pdf

[3] Kay, Michael. Anatomy of an XSLT Processor. Published online by IBM DeveloperWorks https://www.ibm.com/developerworks/library/x-xslt2/

[4] Kay, Michael. Ten Reasons why Saxon XQuery is Fast. Bulletin of the IEEE Technical Committee on Data Engineering. http://sites.computer.org/debull/A08dec/saxonica.pdf

[5] Kay, Michael. You Pull, I'll Push: On the Polarity of Pipelines. Presented at Balisage: The Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3 (2009). doi:10.4242/BalisageVol3.Kay01. http://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html

[6] Saxonica: XSLT and XQuery Processing http://www.saxonica.com/

[7] XSL Transformations (XSLT) Version 3.0. W3C Last Call Working Draft, 12 December 2013. Ed. Michael Kay. http://www.w3.org/TR/xslt-30

XFormsUnit: the Framework to Test Them All

Eric van der Vlist
Dyomedea

Abstract

Current practices to test XForms developments rely on generic web testing frameworks and expose implementation specific details which can change from version to version.

XForms forms can be incredibly complex and they deserve a proper test framework allowing to define tests using XForms paradigms.

This talk presents XFormsUnit, a native XForms test framework.

1. What XForms Unit is

XForms Unit[3] is a web application which combine a test designer and a test runner:

- The test designer is an XForms application developed with Orbeon Form Builder.
- The test runner is implemented as an Orbeon XPL pipeline which can delegate the test execution to a number XForms implementations (Orbeon Forms, betterFORM and XSLTForms are currently supported).

Test suites can be edited with and run from the test designer. Such test suites associate a number of actions and assertions to an XForms document.

The test designer window is divided into three sections with corresponding tool boxes:

- The first section selects the target form on which the tests will be run and the target XForms implementations. Its toolbox enables to download the form and run it with the target implementations.
- The second section is the edition of the test actions and assertions using an embedded XML editor. The edition can be done item by item (each action and assertion being edited in a separated control) or globally (the test suite document is edited in a big control). The associated toolbox includes buttons to save the suite, revert its changes, download or upload it and run it with the different implementations.

[3] http://xformsunit.org/

- The third section is about gathering tests results. With its toolbox you can run the tests with the target implementations and download the results.

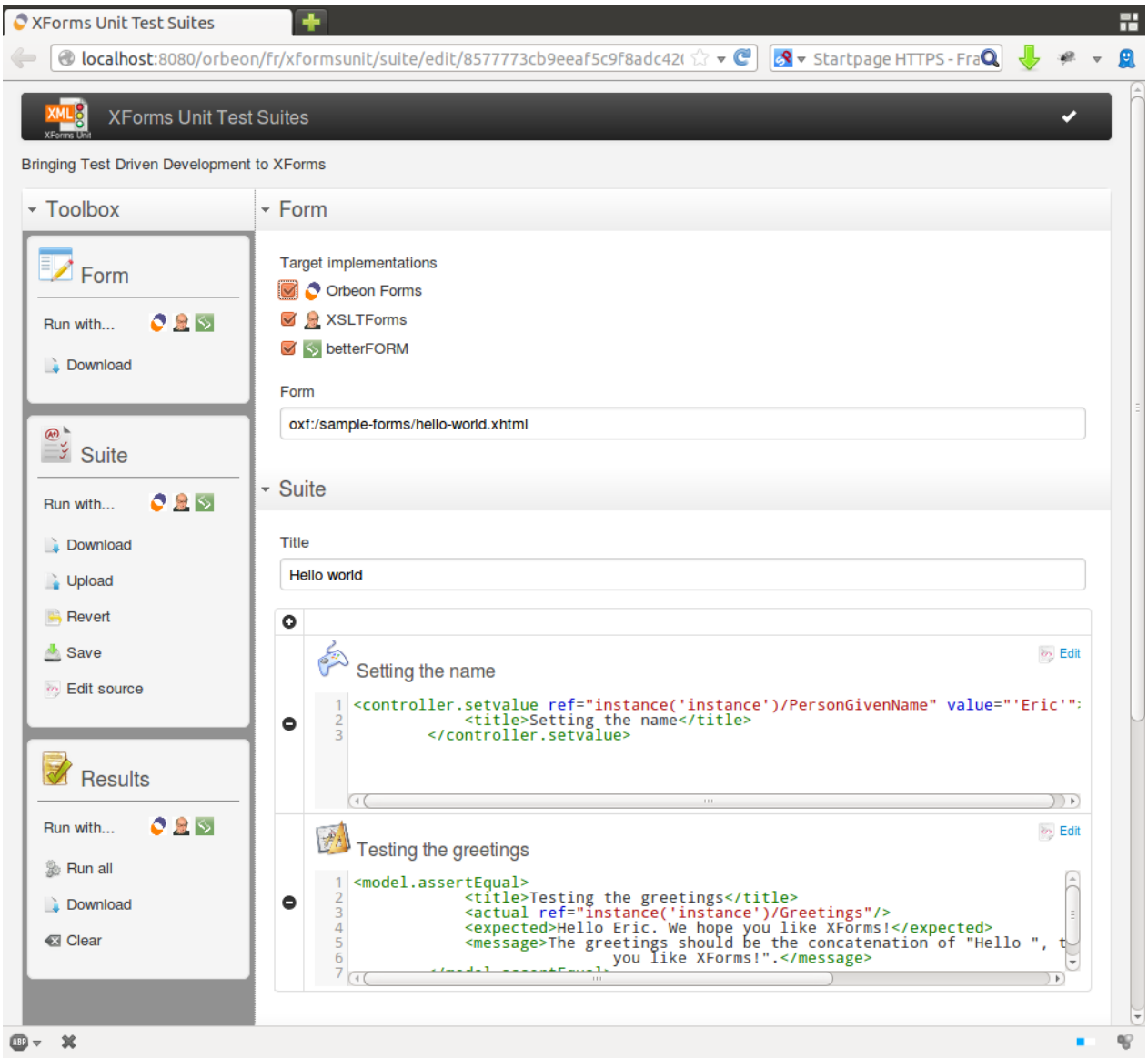


Figure 1. Test builder

The edition being made by editing the XML source of the test suite does expose the XML vocabulary which has been defined to formalize test suites. This vocabulary is designed to be as close as possible from the concepts introduced by XForms. To achieve this, elements are prefixed by the words "controller", "model" and "view" depending on the level with which they interact:

- Elements prefixed by "controller" define ways to act on the controller. These are typically actions, such as `controller.setvalue` which syntaxes are similar to the corresponding XForms actions.

- Elements prefixed by "model" work at the model level. These are assertions, such as `model.assertEqual` which testes that an instance value is equal to an expected value.
- Elements prefixed by "view" work on the view. These are also assertions, such as `view.assertEquals` which testes if a control property (such as visible, enabled, ...) is equal to an expected value.

Note
This vocabulary has been designed[4] and formalized as an Examplotron schema[5].

Under the scene, the most important principle is that these test suites are executed on the target implementations in pure XForms. To that effect, an XSLT transformation is run to add XForms instances and controls which perform the tests defined by the test suite on the XForms read event and write the test results in an instance.

These test results can either be displayed inline in the form itself (this is what happens in the section of the test builder when you run a test suite) or replace the form to be read by the pipeline which runs the test (this is what happens in the third section of the test builder when tests results from different implementations are gathered).

Note
A screen-cast presenting this user interface is available in the latest XForms Unit progress report[6].

2. A native XForms test suite framework

At that point you may wonder why it might be important to have a native test suite framework, after all, neither the XForms Working Group, XForms implementers nor form developers have been waiting XForms Unit to run and define test suites whenever they needed them.

That's true and a common way of testing XForms application is to use a generic purpose web testing tool[7] such as Selenium[8].

The benefits of such tools is that you can test exactly what is displayed in the browser and simulate user actions. It's downside is that the tests are expressed in browser related terms rather than using XForms concepts. To write these tests you

[4] http://xformsunit.org/2013/08/23/a-vocabulary-to-describe-test-suites
[5] http://xformsunit.org/trac/browser/resources/apps/xformsunit/suite.eg
[6] http://xformsunit.org/2013/09/20/progress-report-and-standby/
[7] http://en.wikipedia.org/wiki/List_of_web_testing_tools
[8] http://en.wikipedia.org/wiki/Selenium_%28software%29

need to know how XForms will be transformed into HTML and this transformation depends on the XForms implementation being used and may vary between versions.

By contrast a native XForms test environment does allow to express the tests using XForms concepts such as binds, controls and events.

In practice, if you have a form such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xf="http://www.w3.org/2002/xforms">
  <head>
    <title>Hello World in XForms</title>
    <xf:model id="model">
      <xf:instance id="instance" xmlns="">
        <data>
          <PersonGivenName></PersonGivenName>
          <Greetings></Greetings>
        </data>
      </xf:instance>
      <xf:bind id="greetings" nodeset="/data/Greetings"
        calculate="concat('Hello ', ../PersonGivenName,
                          '. We hope you like XForms!')"/>
    </xf:model>
  </head>
  <body>
    <p>Type your first name in the input box. <br /> If you are running XForms,
      the output should be displayed in the output area.</p>
    <xf:input ref="PersonGivenName" incremental="true">
      <xf:label>Please enter your first name: </xf:label>
    </xf:input>
    <br />
    <xf:output value="Greetings">
      <xf:label>Output: </xf:label>
    </xf:output>
  </body>
</html>
```

You can write the following test suite to test that the calculated value for the greetings element is correct:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="../../suite.rng" type="application/xml"
            schematypens="http://relaxng.org/ns/structure/1.0"?>
<?xml-model href="../../suite.rng" type="application/xml"
            schematypens="http://purl.oclc.org/dsdl/schematron"?>
<suite xmlns:xh="http://www.w3.org/1999/xhtml"
       xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
  <form src="hello-world.xhtml"/>

  <!-- The test cases -->
  <case id="test-greetings">
    <title>Test that greetings are correctly set</title>
    <controller.setvalue
      ref="instance('instance')/PersonGivenName">Eric</controller.setvalue>
    <model.assertEqual>
      <actual ref="instance('instance')/Greetings"/>
      <expected>Hello Eric. We hope you like XForms!</expected>
      <message>The greetings should be the concatenation of "Hello ",
                  the given name and ". We hope you like XForms!".</message>
    </model.assertEqual>
  </case>
</suite>
```

The test runner transforms the form to add actions that check the tests defined in the suite and you can run these tests in a number of different ways on all three XForms implementations.

The tests can be run and displayed in the resulting form itself. A section is then added to the form to display the results before the resulting form. For this example, the resulting form would be:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xf="http://www.w3.org/2002/xforms"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:xh="http://www.w3.org/1999/xhtml"
      xmlns:ev="http://www.w3.org/2001/xml-events"
      xmlns:xfu="http://xformsunit.org/">
  <head>
    <title>Hello World in XForms</title>
    <xf:model id="model">
      <xf:instance id="instance">
        <data xmlns="">
          <PersonGivenName/>
          <Greetings/>
        </data>
      </xf:instance>
      <xf:bind id="greetings" nodeset="/data/Greetings"
             calculate="concat('Hello ', ../PersonGivenName, '. We hope you ▶
like XForms!')"/>
      <!-- Test cases -->
      <xf:instance id="xfu-instance">
        <suite xmlns="">
          <case id="test-greetings">
            <title id="d28e12">Test that greetings are correctly set</title>
            <controller.setvalue ref="instance('instance')/PersonGivenName"
```

```
                                id="d28e15">Eric</controller.setvalue>
            <model.assertEqual id="d28e18" passed="">
              <actual ref="instance('instance')/Greetings" id="d28e20"/>
              <expected id="d28e22">Hello Eric. We hope you like XForms!</▶
expected>
             <message id="d28e25">The greetings should be the concatenation ▶
of "Hello ", the given name and ".
                                  We hope you like XForms!".</message>
            </model.assertEqual>
          </case>
        </suite>
      </xf:instance>
      <xf:dispatch ev:event="xforms-ready" targetid="model"
                   name="xfu-d28e15-action"/>
      <xf:action ev:event="xfu-d28e15-action">
        <xf:recalculate/>
        <xf:refresh/>
        <xf:setvalue
           ref="instance('instance')/PersonGivenName">Eric</xf:setvalue>
        <xf:dispatch targetid="model" name="xfu-d28e18-action"/>
      </xf:action>
      <xf:action ev:event="xfu-d28e18-action">
        <xf:recalculate/>
        <xf:refresh/>
        <xf:setvalue ref="instance('xfu-instance')//*[@id = 'd28e20']"
                     value="instance('instance')/Greetings"/>
       <xf:setvalue ref="instance('xfu-instance')//*[@id = 'd28e18']/@passed"
                    value="(instance('instance')/Greetings) = 'Hello Eric. We ▶
hope you like XForms!'"/>
      </xf:action>
    </xf:model>
  </head>
  <body>
    <xf:group ref="instance('xfu-instance')" id="xfu-group">
      <h3>Test results</h3>
      <dl>
        <xf:repeat nodeset="case">
          <dt>
            <dfn>
              <xf:output ref="@id"/>
            </dfn>
          </dt>
          <dd>
            <ul>
              <xf:repeat nodeset="*[@passed]">
                <li>
```

```
                    <xf:group ref=".[@passed = 'true']">
                      <span>passed</span>
                    </xf:group>
                    <xf:group ref=".[@passed = 'false']">
                      <span>failed</span>
                      <xf:output ref="*:actual|control">
                        <xf:label>Actual :</xf:label>
                      </xf:output>
                      <xf:output ref="*:expected">
                        <xf:label>Expected :</xf:label>
                      </xf:output>
                    </xf:group>
                  </li>
                </xf:repeat>
              </ul>
            </dd>
          </xf:repeat>
        </dl>
      </xf:group>
      <hr/>
      <h3>Form</h3>
      <p>Type your first name in the input box.
        <br/>If you are running XForms,
        the output should be displayed in the output area.
      </p>
      <xf:input ref="PersonGivenName" incremental="true">
        <xf:label>Please enter your first name:</xf:label>
      </xf:input>
      <br/>
      <xf:output value="Greetings">
        <xf:label>Output:</xf:label>
      </xf:output>
    </body>
  </html>
```

3. Limitations

The current version is a merely a proof of concept and has many limitations. A lot of these limitations are teething problems:

- lack of support for multi-model forms
- No other action than `controller.setvalue`
- No other assertions than `model.assertEqual` and `view.assertEqual`
- ...

A more serious limitation is coming from the fact that XForms implementation may require client-server interactions and the situation is different for each implementation:

- XSLTForms is a pure browser side application an requires a browser to run. This is not a problem when a test suite is run from a browser which is the case when we use the test designer. However this is an issue to run the test suite server side, for instance from a continuous integration framework. To work around this issue it is necessary to emulate or run a browser server side.
- On the contrary, Orbeon Forms is a client-server implementation of XForms that has been optimized to apply all the initial actions (basically everything that happens up to the end of the `xforms-ready` event) server side without requiring any client-server interaction. In other words, test suites can be executed server side within an XPL pipeline.
- Like Orbeon Forms, betterFORM is a client-server implementation of XForms but the initial actions are applied after a first client-server exchange. It might be possible to emulate this first exchange to avoid to use a browser but the current version runs the test suites from a browser like it does for XSLTForms with the same consequences.

A last set of limitations are due to XForms itself. XForms has been designed as the response to a demand for "better Web forms with richer interactions" and using it to implement a unit test framework is quite challenging.

In particular, XForms' functions library is quite weak in the domain of introspection, for instance there is no function to determine if a control is visible or relevant.

To implement its `view.assertEqual` assertions, XForms Unit relies on events sent to the controls but XForms events processing has not been defined with this kind of purposes in mind and doesn't really solve the issue in all the cases.

As a result, next versions of XForms Unit will have to rely on extension functions which are not yet available in all three supported implementations.

4. XForms flavours

When I have started the project I was anticipating interoperability issues between these different implementations through incompatible extensions that I would probably need to use sooner or later.

What I had not anticipated was that the lack of interoperability between XForms implementations would be worse than the lack of interoperability between browsers and I would be spending most of my time with interoperability issues between implementations on basic XForms features.

On the web it is impossible to predict which browsers will be used by your visitors and the lack of interoperability between web browsers has a direct and visible

cost on every web development. This is a strong argument to reduce if not eliminate the extensions and incompatibilities between web browsers.

Because of the lack of native implementation of XForms in the browsers, the interoperability between XForms implementations is more like interoperability between SQL databases: most applications are written with a target XForms implementation in mind and each implementation seems eager to add its own extensions without much concern for interoperability.

Since it appears to be almost impossible to write any complex XForms application without using any extension we have now a number of different and incompatible XForms flavours (one per implementation) with their own communities of form developers.

XSLT 3.0 Testbed

Tony Graham
Mentea
<tgraham@mentea.net>

Abstract

https://github.com/MenteaXML/xslt3testbed is a public, medium-sized XSLT 3.0 project where people could try out new XSLT 3.0 features on the transformations to (X)HTML(5) and XSL-FO that are what we do most often and, along the way, maybe come up with new design patterns for doing transformations using the higher-order functions, partial function application, and other goodies that XSLT 3.0 gives us.

There's undoubtedly many things to try out, but a starter list of things to look at includes:

- How will XSLT 3.0 features make it easier to:
  - Customise the output?
  - Modularise stylesheets?
  - Re-use modules between HTML and XSL-FO output?
- Will higher-order functions, anonymous function, partial function application, and/or dynamic XPath evaluation improve upon `xsl:attribute-set`?

The project is ongoing, so the presentation will report on work done to date.

1. Timing

The project started in October 2013, when XSLT 3.0 was sufficiently cooked that it could be used without the language changing drastically before Recommendation but not so hard-baked that it wouldn't be possible for anything the project found making a change to the spec.

The generalisation about comments on W3C specs is that people don't pay much attention to drafts until the spec is in Last Call or Candidate Recommendation stage, by which point it's hard for a WG to make substantive changes.

XSLT 3.0 [1] was released as a Last Call Working Draft on 12 December 2013, so the timing was just about right.

2. JATS

The XSLT 3.0 testbed stylesheets are derived from the XSLT 1.0 "NISO Journal Article Tag Suite (JATS) version 1.0" stylesheets [22] by the National Center for Biotechnology Information [23] at the U.S. National Library of Medicine (NLM) [24].

The JATS preview stylesheets made a good place to start since:

- JATS is in wide use as the medium of exchange, and of archiving, for published scientific journal articles (and more)
- It has the Goldilocks factor of not being too large (it's not as large as DocBook or TEI) nor too small (it's also not a toy)
- There's lots of sample data available
- There's existing XSLT 1.0 stylesheets
- The existing stylesheets are in the public domain
- There's a quick win in just adding XSLT 2.0-isms to the XSLT 1.0 stylesheets

3. Goals

Results to work towards:

- Trial – and make prior art for – different techniques for using XSLT 3.0
- Get an early start for developing the patterns and idioms for using XSLT 3.0 that many of us will be using for years to come
- Standalone XSLT 3.0 xsl:package for transforming XHTML tables to XSL-FO and/or HTML

  xsl:package is new in XSLT 3.0, plus (X)HTML tables are used with many document types, so a stable, reusable module for formatting tables that takes advantage of any new or improved syntax or functions available in XSLT 3.0 and XPath 3.0 will be a good thing

4. Non-goals

Possible results to not work towards or to actively work against:

- The single best way of doing anything

  Since this is a testbed it's okay for it to try different ways of achieving the same result
- The definitive XSLT 3.0 testbed

  This project is on GitHub because it's easy to fork Git projects and develop your own flavour of them. Pull requests are encouraged, but if you want to make your own version of this, that's more than fine, too
- Complete stylesheets for all of JATS

JATS gives the work a focus, but it isn't the focus of the work. Also, the original XSLT 1.0 stylesheets don't cover all of JATS to begin with, so there's no onus for this project to cover all of JATS

5. Results so far

As of 25 January 2014, results include:

- Six new tickets against XSLT 3.0 or XPath 3.0 on the W3C Bugzilla system
  The issues raised have been minor, which is a good sign for XSLT 3.0, but addressing them now is less burden than if the same issues were raised when XSLT 3.0 is either a Proposed Recommendation or already a Recommendation.
- Analysis, below, comparing a map of functions against attribute sets
- One change merged into the original JATS stylesheets [22], with more pull requests to come
- One change merged into Wendell Piez's Oxygen framework for JATS [21], with more under discussion
- A technique for hosting Oxygen add-ons as GitHub project releases [25]

The presentation in February 2014 will be able to report even more results. Developments after the conference will be reported on the GitHub project's wiki [19] and/or my blog [20].

6. Map of functions compared to `xsl:attribute-set`

After comparing using a map of functions that return attributes to using `xsl:attribute-set`, neither is significantly better than the other.

The current input is a JATS [6] document that contains two table-wrap [3][4] that each contain a table. For the sake of the exercise, the table in the second table-wrap has 'style="orange"' [5].

There's currently three branches in the repository, but only two are sufficiently well developed to report on: 'master' uses `xsl:attribute-set`, and 'table-map' uses maps of functions.

Also for the sake of the exercise there's multiple stylesheets in use:

- xhtml-tables-fo3xsl – Base table module ('master': [7]; 'table-map': [8])
- red.xsl – Styles text is the table head as red ('master': [9]; 'table-map': [10])
- blue.xsl – Styles text is the table body as blue ('master': [11]; 'table-map': [12])
- red-blue.xsl – Imports both 'red.xsl' and 'blue.xsl' to achieve a combined effect (or try to) ('master': [13]; 'table-map': [14])
- orange.xsl – Styles table background as orange ('master': [15]; 'table-map': [16])

There's also an Oxygen project file to make it easier to run the different-coloured stylesheets.

The `xsl:attribute-set` approach uses an attribute set named after each table-related element, and the different stylesheets add attribute instructions to the appropriate attribute set. 'red-blue.xsl' takes the convenient approach of just importing 'red.xsl' and 'blue.xsl' (and, coincidentally, manages to import the whole JATS stylesheets twice, but that's the price you pay for convenience) and the `xsl:attribute-set` from the different stylesheets just combine (since at present there's no overlap/conflict to worry about) [1].

For the table that wants to be orange, 'orange.xsl' passes specific attributes in a 'table-attributes' parameter:

```
<xsl:template match="table[@style eq 'orange']">
  <xsl:next-match>
    <xsl:with-param
        name="table-attributes"
        as="attribute()*"
        tunnel="yes">
      <xsl:attribute name="background-color" select="'orange'" />
    </xsl:with-param>
  </xsl:next-match>
</xsl:template>
```

that override attributes defined in the 'table' attribute set:

```
<xsl:template match="table">
  <xsl:param name="table-attributes"
             as="attribute()*"
             tunnel="yes" />

  <fo:table xsl:use-attribute-sets="table fo:table">
    <xsl:sequence select="$table-attributes" />
    <xsl:apply-templates />
  </fo:table>
</xsl:template>
```

For the 'map of functions' approach, the templates for the table-related elements each have a 'table-functions' tunnel parameter that is a map of the functions to use for the appropriate table-related element(s). These override the default functions, which don't do anything. Making a default map is analogous to needing to define empty attribute sets since calling a non-existent attribute set is an error [17], but an alternative would be to only call the function for the current element if it exists in the current `$table-functions` map.

[1] There's also attribute sets corresponding to the FOs that the table elements become, but they have no effect at present.

'red.xsl' and 'blue.xsl' work by passing appropriate maps of functions. 'red-blue.xsl' is the same as for the `xsl:attribute-set` approach, and it doesn't produce red table head text because, with the way that import precedence works, the template for 'thead' in 'red.xsl' is never used.

The template for the table that wants to be orange uses the same mechanism as the general table templates and passes a map containing a function to use for the table element:

```
<xsl:function name="x3tb:orange-table" as="attribute()*">
  <xsl:param name="context" as="element()" />

  <xsl:attribute name="background-color" select="'orange'" />
</xsl:function>

<xsl:template match="table[@style eq 'orange']">
  <xsl:next-match>
    <xsl:with-param
        name="table-functions"
        as="map(xs:string, function(element()) as attribute()*)"
        select="map {
                  'table' := x3tb:orange-table#1
                }"
        tunnel="yes" />
  </xsl:next-match>
</xsl:template>
```

The absence of an XPath-level computed attribute constructor made making the function verbose compared to how you'd make an attribute in XQuery and meant that it could not be an anonymous function.

That function for 'table' overrode the default:

```
<xsl:template match="table">
  <xsl:param
      name="table-functions"
      as="map(xs:string, function(*))?"
      tunnel="yes" />

  <xsl:variable
      name="use-table-functions"
      select="map:new(($default-table-functions, $table-functions))"
      as="map(xs:string, function(*))" />

  <fo:table>
    <xsl:sequence select="$use-table-functions('table')(.)" />
    <xsl:apply-templates />
  </fo:table>
</xsl:template>
```

The alternative to making a XSLT-level function to use the `xsl:attribute` constructor, as was pointed out to me, is to make a custom XSLT function for constructing an attribute:

```
<xsl:function name="x3tb:attribute" as="attribute()*">
  <xsl:param name="name" as="xs:string" />
  <xsl:param name="value" as="xs:string" />

  <xsl:attribute name="{$name}" select="$value" />
</xsl:function>
```

and use that in the anonymous function:

```
<xsl:template match="table[@style eq 'orange']">
  <xsl:next-match>
    <xsl:with-param
        name="table-functions"
        as="map(xs:string, function(element()) as attribute()*)"
        select="map {
                  'table' := function($context as element()) as attribute()* {
                               x3tb:attribute('background-color', 'orange')
                             }
                }"
        tunnel="yes" />
  </xsl:next-match>
</xsl:template>
```

6.1. Discussion

Using a function that takes the context node as a parameter to define attributes is not dissimilar to using `xsl:attribute-set`, since both can evaluate expressions based on the context node and global variables only.

The way that attribute instructions defined with `xsl:attribute-set` can combine across modules can work to your advantage, but there's no way to 'undefine' attribute instructions in attribute sets other than defining attributes with the same name (and, presumably, a neutral value) in a situation that has higher precedence, so the convenience is convenient only up to a point.

Combining the functions that define attributes or the maps of functions that define attributes across modules in a way that 'just works' when you add a new module or override an existing template still requires thought. Currently it can provide more control but is more verbose and less convenient than using attribute sets, and even in its current form may be hard for a novice user to understand compared to using attribute sets.

The 'map of functions' approach could be extended to use functions that take other, additional parameters to further control the processing and/or other maps

of other functions could be used in other places to generate elements. Indeed, the entire table processing could be rewritten to use a map of functions that each 'do' the processing for the context element, but by then you'll have just reimplemented 'typeswitch' [18] in XSLT.

7. Idioms

An idiom is a "manner of expression characteristic of or peculiar to a language" [26]. XSLT 1.0 and XSLT 2.0 each produced their own idioms, and it can be expected that XSLT 3.0 and XPath 3.0 will produce new idioms though we don't yet know what they all will be.

7.1. XSLT 1.0

Muenchian Grouping [27], invented because XSLT 1.0 didn't provide good facilities for grouping, is probably that version's ultimate idiom since the technique became obsolete after the addition of `xsl:for-each-group` in XSLT 2.0.

7.2. XSLT 2.0

Replacement of `xsl:choose` with an XPath `if` expression and, ultimately, a sequence constructor with a predicate are idioms that first became possible in XSLT 2.0. For example, in XSLT 1.0, supplying a default value when an attribute is not present in the source tree requires a `xsl:choose`:

```
<fo:table-cell>
  <xsl:attribute name="text-align">
    <xsl:choose>
      <xsl:when test="@align">
        <xsl:value-of select="@align"/>
      </xsl:when>
      <xsl:otherwise>from-table-column()</xsl:otherwise>
    </xsl:choose>
  </xsl:attribute>
</fo:table-cell>
```

With the addition of the conditional expression in XPath 2.0 [29], this reduces to:

```
<fo:table-cell
    text-align="{if (exists(@align))
                   then @align
                 else 'from-table-column()'}">
```

Yet as people became familiar with using sequences, it has become common to see this idiom:

```
<fo:table-cell text-align="{(@align, 'from-table-column()')[1]}">
```

where the result of "`(@align, 'from-table-column()')[1]`" is the `@align`, if it exists, otherwise it is the string "`from-table-column()`" [30], which will be evaluated by the XSL-FO formatter to return either the value from corresponding property on the table cell's column definition, if the property is specified there, or the property's initial value.

7.3. XSLT 3.0

I expect that people will develop idioms for the common cases of using streaming XSLT, if only as a defense against having to think their way through the terminology and its ramifications each time they want to write a streaming expression. There will be a few tried-and-tested formulations that are all that the majority of people will use with streaming XSLT, but that number can, of course, increase over time as more people become more familiar with using streaming.

The other area ripe for new idioms is higher-order functions and named function references added in XPath 3.0. For example, a recent post by Ihe Onwuka to the 'xquery-talk' mailing list [32] sought a more direct alternative expression for:

```
if (@firstpage eq 0)
  then floor($totalPagesDecimal)
else ceiling($totalPagesDecimal)
```

The first proposed solution [33]:

```
(floor#1,ceiling#1)[@firstpage + 1]($totalPagesDecimal)
```

is elegant but really only workable with schema-aware processing, and at the time of this writing, it's unclear whether there is a general solution using named function references that won't end up longer and harder to understand than the orginal.

8. Stylesheet License

The stylesheets are in the public domain and may be reproduced, published or otherwise used without the permission of Mentea or of the National Library of Medicine (NLM), and neither the GitHub project nor Mentea has any affiliation with NLM.

9. Conclusion

Activity on the XSLT 3.0 testbed to date has already produced worthwhile results. Now is the right time to trial new techniques and new idioms using XSLT 3.0 and XPath 3.0 since they will benefit all users and since any issues found in the specifications are easier to fix now than they will be later when the specifications are W3C Recommendations.

Bibliography

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

[2] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/table-test.xml

[3] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/table-test.xml#L30

[4] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/table-test.xml#L50

[5] http://jats.nlm.nih.gov/articleauthoring/tag-library/1.0/index.html?attr=style

[6] http://jats.nlm.nih.gov/index.html

[7] https://github.com/MenteaXML/xslt3testbed/blob/master/xsl/xhtml-tables-fo3.xsl

[8] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xsl/xhtml-tables-fo3.xsl

[9] https://github.com/MenteaXML/xslt3testbed/blob/master/xml/table-test/red.xsl

[10] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/red.xsl

[11] https://github.com/MenteaXML/xslt3testbed/blob/master/xml/table-test/blue.xsl

[12] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/blue.xsl

[13] https://github.com/MenteaXML/xslt3testbed/blob/master/xml/table-test/red-blue.xsl

[14] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/red-blue.xsl

[15] https://github.com/MenteaXML/xslt3testbed/blob/master/xml/table-test/orange.xsl

[16] https://github.com/MenteaXML/xslt3testbed/blob/table-map/xml/table-test/orange.xsl

[17] http://www.w3.org/TR/xslt-30/#err-XTSE0710

[18] http://www.w3.org/TR/xquery/#id-typeswitch

[19] https://github.com/MenteaXML/xslt3testbed/wiki

[20] https://inasmuch.as/

[21] https://github.com/wendellpiez/oXygenJATSframework

[22] https://github.com/NCBITools/JATSPreviewStylesheets

[23] http://www.ncbi.nlm.nih.gov/

[24] http://www.nlm.nih.gov/

[25] https://inasmuch.as/2013/10/23/oxygen-add-on-hosted-on-github/

[26] http://www.thefreedictionary.com/idiom

[27] http://www.jenitennison.com/xslt/grouping/muenchian.html

[28] http://www.w3.org/TR/xslt#section-Result-Tree-Fragments

[29] http://www.w3.org/TR/xpath20/#id-conditionals

[30] http://www.w3.org/TR/xsl/#d0e5961

[31] http://www.w3.org/TR/xpath-30/#id-named-function-ref

[32] http://x-query.com/pipermail/talk/2014-January/004378.html

[33] http://x-query.com/pipermail/talk/2014-January/004379.html

XML Schema Identity Constraints Revisited

Anne Brüggemann-Klein
Technische Universität München
<brueggem@in.tum.de>

Mustapha Maalej
Technische Universität München
<maalej@in.tum.de>

Marouane Sayih
Technische Universität München
<sayih@in.tum.de>

Abstract

In this paper, we attempt to explain clearly our reading of XML Schema's identity constraint concepts. We illustrate our reading extensively with examples, in the style of a tutorial. We also illustrate usage styles and limitations of identity constraints in XML Schema. Finally, we demonstrate how the limitations that we have identified can be by-passed with assertions as introduced by XPath 2.0 and XML Schema 1.1.

1. Introduction

XML Schema transfers one of the fundamental database concepts into the realm of XML documents, namely identity constraints. Using XML Schema's identity constraint mechanisms, schema developers may impose that specific elements in a document are uniquely identified by some of the data they contain, so that these data serve as key for the elements; they may also demand that other elements refer to such keys, guaranteeing referential integrity.

The contributions of this paper are as follows: Primarily, we attempt to explain clearly our reading of XML Schema's identity constraint concepts. We illustrate our reading extensively with examples, in the style of a tutorial. All examples validate as expected with Saxon. We also illustrate usage styles and limitations of identity constraints in XML Schema. Finally, we demonstrate how the limitations that we have identified can be by-passed with assertions as introduced by XPath 2.0 and XML Schema 1.1.

This investigation was motivated by and is applicable to the second author's PhD work that concerns XForms documents that are generated from an XML Schema and serve as editors for instances of the schema. Since the XForms editors should support identity constraints, it was necessary to understand the concepts and implementation options with XPath.

There has been some critical discussion in the XML community on the wording and semantics of the XML Schema section on identity constraints. In this paper, we hope to clarify some issues and to illustrate benefits and limitations. The general discussion is in terms of an abstract data model for identity constraint constructs; syntax is used only in examples.

2. XML Schema's concepts for identity constraints: key

In XML Schema, an element declaration may contain a key constraint[1] that determines which sub-parts of a conformant element information item should be uniquely determined by which combination of values that each of the sub-parts contains. Each key constraint has a name that is unique within the schema, a selector function `s` and a fields function `f` that is made up of a sequence (`f1`,…,`fn`) of `n` field functions `f1`,…,`fn` for some natural number `n`. An element information item `E` satisfies the key constraint if and only if the following conditions hold:

- The expression `E.s()`[2] evaluates to a sequence of element information items on the `descendent-or-self` axis of `E`. The set of those element information items is called the target set of `E`.
- For each element information item N in the target set of E, the expression `N.fi()` evaluates to an element information item or an attribute information item of an element information item on the descendant-or-self-axis of `N`; in the former case, the element information item `N.fi()` must be declared to be of a simple type, with no subelements allowed. We call the sequence of string[3] values of `N.f()`=(`N.f1()`,…,`N.fn()`) the key sequence of `N`.
- The key sequences of the element information items in the target set of `E` are all different sequences.[4] This condition gives rise to an index table at element information item `E` for the key constraint that collects pairs of key sequences and

[1] Actually, XML Schema introduces two similar concepts called `key` and `unique`. For the sake of brevity, we only discuss `key` in this article.
[2] We are using an object-oriented notational style, so `E.s()` stands for function `s` applied to argument `E`.
[3] XML Schema deals with typed values. We are simplifying to strings in this paper, without any relevant loss of generality. XML Schema also has some condition on nillable elements that we are ignoring here, blending out some subtle points of type definitions.
[4] This is the main uniqueness condition that makes a key a key. The notion of equality of sequences needs to be adapted when items in the sequence are not just strings but typed values.

corresponding element information items in the target set of element `E`. Such an index table has no multiple entries for any one key sequence.

XML Schema's key concept is obviously transfered from relational databases, where a schema may require of a table that each of its rows is uniquely determined by the combination of values of a specific selection of table attributes. The main difference is that elements in XML are more freely structured than tables. Hence, we need to (or may) customize explicitly which sub-parts of the element should be uniquely identified by the key: In databases, automatically all rows in a table must be uniquely identified; in XML Schema, a general selector function defines the set of target elements. Also, in XML Schema we have more flexibility in selecting the fields that make up a unique key sequence: They can be any textual values in (sub-)elements or attributes of (sub-)elements of the "keyed" element, as defined by a general fields function, not just attributes, as in relational databases.

The schema collections.xsd in the appendix illustrates key constraints with progressing complexity. The scenario is a repository of collections of items. There are a number of ways how collections are structured and how items are assigned to collections, as illustrated in Figure 1.

```
<xs:element name="collections">
 <xs:complexType>
  <xs:choice>
   <xs:element ref="collection" minOccurs="1" maxOccurs="unbounded"/>
   <xs:element ref="collectionsDBStyle"/>
   <xs:element ref="collectionXMLStyle"/>
   <xs:element ref="collectionsWithLocalOwners"/>
   <xs:element ref="collectionsWithGlobalOwners"/>
   <xs:element ref="collectionsWithOwnersDBStyle"/>
   <xs:element ref="collectionsWithOwnersHybridStyle"/>
   <xs:element ref="collectionsWithOwnersSensible"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Figure 1. Alternatives how to structure collections.

On the most elementary level, see Figure 2, we have a sequence of `collection` elements, each containing a sequence of `item` elements that are uniquely identified by their attribute `idItem` within that collection container. This constraint is expressed by the key `collectionKey` which is defined within the element declaration of `collection`. Consequently, at the instance level, each element information item has its own index table for the key constraint `collectionKey`, so that key sequences can be re-used in different element scopes. This is illustrated in the XML document sampleCollection.xml, see Figure 3.

```
<xs:element name="collection">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="collectionKey">
  <xs:selector xpath="item"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
</xs:element>
<xs:element name="item" type="itemType"/>
<xs:complexType name="itemType">
 <xs:attribute name="idItem" type="xs:string" use="required"/>
</xs:complexType>
```

Figure 2. The collection structure.

```
<collections xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collection>
        <item idItem="a"/>
        <item idItem="b"/>
        <!-- No duplicate idItem values allowed within collection.
        <item idItem="a"/>
        -->
    </collection>
    <collection>
        <!-- Duplicate idItem values allowed in different collections.-->
        <item idItem="a"/>
        <item idItem="c"/>
        <!-- No duplicate idItem values allowed within collection.
        <item idItem="a"/>
        -->
    </collection>
</collections>
```

Figure 3. The collection instance.

This example uses hierarchical structure that is typical for XML. A database-like approach would use a flat repository of collections and another repository of items that indicate to which collection they belong. The schema collections.xsd, see Figure 4 and Figure 5, illustrates that way of structuring the data with an element `collectionsDBStyle` that may contain sequences of subelements `collectionDBStyle` and `itemInCollection`. Elements `itemInCollection` express to which collection they belong by their attribute `idCollection`, not by the hierarchy within a collection container.

```
<xs:element name="collectionsDBStyle">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="collectionDBStyle" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="itemInCollection" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="collectionDBStyleKey">
  <xs:selector xpath="collectionDBStyle"/>
  <xs:field xpath="@idCollection"/>
 </xs:key>
 <xs:key name="itemInCollectionKey">
  <xs:selector xpath="itemInCollection"/>
  <xs:field xpath="@idCollection"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
 <xs:keyref name="itemRefersToCollection" refer="collectionDBStyleKey"> [3 lines]
</xs:element>
```

Figure 4. The collectionsDBStyle outer structure.

```
<xs:element name="itemInCollection">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="itemType">
    <xs:attribute name="idCollection" type="xs:string" use="required"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
<xs:element name="collectionDBStyle">
 <xs:complexType>
  <xs:attribute name="idCollection" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
```

Figure 5. The collectionsDBStyle inner structure.

The key constraint `collectionDBStyleKey` expresses that elements `collecionDBStyle` must be uniquely indentified by their attributes `idCollection`. The key constraint `itemInCollection` expresses that elements `itemInCollection` are uniquely identified by the combination of their `idCollection` and `idItem` attributes.

In addition, the schema expresses referential integrity with the key reference `itemRefersToCollection`, a feature we explain in the next section.

The XML document sampleCollectionDBStyle.xml, see Figure 6 illustrates the database style of structuring the data on the instance level.

```
<collectionsDBStyle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionDBStyle idCollection="x"/>
    <collectionDBStyle idCollection="y"/>
    <!-- No duplicate idCollection values allowed.
    <collectionDBStyle idCollection="y"/>
    -->
    <itemInCollection idCollection="x" idItem="a"/>
    <itemInCollection idCollection="x" idItem="b"/>
    <!-- Duplicate idItem values allowed,
         as long as idCollection values are different. -->
    <itemInCollection idCollection="y" idItem="a"/>
    <itemInCollection idCollection="y" idItem="c"/>
    <!-- No duplicate idCollection/idItem values allowed.
    <itemInCollection idCollection="y" idItem="a"/>
    -->
    <!-- Referenced collection must exist (referential integrity).
    <itemInCollection idCollection="z" idItem="a"/>
    -->
</collectionsDBStyle>
```

Figure 6. The collectionsDBStyle instance.

Finally, with the element `collectionXMLStyle`, we further explore the use of key constraints for hierarchical structures by adding `collectionXMLStyle` recursively, see Figure 7.

```
<xs:element name="collectionXMLStyle">
 <xs:complexType>
  <xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:element ref="collectionXMLStyle"/>
   <xs:element ref="item"/>
  </xs:choice>
  <xs:attribute name="idCollection" type="xs:string" use="required"/>
 </xs:complexType>
 <xs:key name="collectionXMLStyleKey">
  <xs:selector xpath="collectionXMLStyle"/>
  <xs:field xpath="@idCollection"/>
 </xs:key>
 <!-- [13 lines]
</xs:element>
```

Figure 7. The collectionXMLStyle structure.

Each element `collectionXMLStyle` establishes its own scope for key constraints on unique collection and item IDs. The relationships between containing and contained elements is expressed implicitly, through hierarchy, not through explicit references.

Alternatively, we can demand globally unique IDs for collections and items in this scenario by activating the keys `globalCollectionXMLStyle` and `globalItemKey`.

Instantiation of recursive structures is illustrated by sampleCollectionXMLStyle.xml, see Figure 8.

```
<collectionXMLStyle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd" idCollection="x">
    <collectionXMLStyle idCollection="z">
        <collectionXMLStyle idCollection="x">
            <item idItem="a"/>
            <item idItem="b"/>
            <collectionXMLStyle idCollection="z">
                <collectionXMLStyle idCollection="x">
                    <item idItem="a"/>
                    <item idItem="b"/>
                </collectionXMLStyle>
                <collectionXMLStyle idCollection="y">
                    <item idItem="a"/>
                    <item idItem="c"/>
                </collectionXMLStyle>
            </collectionXMLStyle>
        </collectionXMLStyle>
        <collectionXMLStyle idCollection="y"> [3 lines]
    </collectionXMLStyle>
</collectionXMLStyle>
```

Figure 8. The collectionXMLStyle instance.

3. Propagating index tables upwards

Element information items that carry a key constraint make their index tables available for referencing, just as keys in one table of a relational database can be used as foreign keys in another table to refer to table rows.

How to design a reference concept that utilizes XML Schema key constraints? The case is less clear-cut than the definition of the key-foreign key concept in relational databases, for reasons of hierarchy: In XML documents, element information items that carry identity constraints may be buried in the document hierarchy and they may be recursively nested.

So what does it mean that such an element information item makes it index table available for referencing? The XML Schema concept is that index tables propagate upwards in the element hierarchy and that each element that gets an index table that way may use it for referencing, as explained in the next section. There is only one problem with that approach: Entries in the index tables of two sibling element informations might be in conflict, as might be entries in the index tables of a parent and child element information item. in conflict. Two entries consisting of a key sequence and an element information item are in conflict if the two key sequences are

equal but the two element information items are not. XML Schema defines two conflict resolution rules for index table propagation:

- If two or more sibling element information items have conflicting entries for some key sequence in their index table, none of the entries is included in the parent element information item's index table: Competing children cancel each other out.
- If a parent and a child element information item have conflicting entries for some key sequence in their index table, the parent's index table keeps its entry and the child's entry is discarded at the parent level: Parents dominate children.

There are a number of implications:

First, a key can only be used for referencing at an element information item at which it is defined or above in the element information item hierarchy.

Second, entries in index tables that propagate upwards in the element information item hierarchy may become unavailable for referencing, due to conflict resolution rules.

Finally, at an element information item `E`, all entries that arise directly from a key at `E`, are available for referencing. Since parents dominate children, they are never removed from that element information item's index table for the key, although they might be cancelled out further up in the hierarchy.

We illustrate these effects after discussing XML Schema's key reference constructs.

4. XML Schema's concepts for identity constraints: Referencing keys

In XML Schema, an element declaration may also contain a key reference constraint. Just like a key constraint, a key reference constraint has a name that is unique within the schema, a selector function `s` and a field function `f`=(`f1`,…,`fn`) of width `n` for some natural number `n`. In addition, it has the name of the key constraint that it refers to. The fields functions of the key reference constraint and of the referenced key constraint must have identical width. An element information item `E` satisfies the key reference constraint if and only if there is an index table for the referenced key constraint at E that has, for each element information item `N` in the target set `E.s()`, an entry in the index table for the key sequence `N.f()`. The referenced index table includes any entries that originate directly at `E`, plus any entries that propagate upwards from below `E` and survive conflict resolution. The semantics of this condition is that the element information item `N` refers to the node that appears in the index table for the key sequence `N.f()`. Hence, reference key constraints guarantee referential integrity.

Let us now look at examples that add owner information to our collections scenario in a number of ways.

On the most elementary level, see Figure 9, generalizing the `collection` elements, we have elements `collectionWithLocalOwners` that contain `item` elements and `owner`

elements, the latter listing the `item` elements that this owner owns. As before, `item` elements within the collection are uniquely identified by their attribute `idItem`, as expressed by key constraint `uniqueItems`. In addition, each `item` element can be owned by at most one `owner`, as expressed by key constraint `uniqueOwnership`. Finally, each `item` that has an `owner` must be listed within the collection, as expressed by key reference constraint `ownedItem2ItemInCollection`. Conversely, we could also postulate that each item in a collection must be owned by someone, via a second key reference constraint. We position key and key reference constraints at the element declaration for `collectionWithLocalOwners`. Hence, items listed in a collection and ownership information for these items live completely within each collection, without interfering with items and owner in parallel collections. We illustrate with the XML instance sampleCollectionsWithLocalOwners.xml, see Figure 10.

```
<xs:element name="collectionWithLocalOwners">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="owner" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="uniqueItems">
  <xs:selector xpath="item"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
 <xs:key name="uniqueOwnership">
  <xs:selector xpath="owner/item"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
 <xs:keyref name="ownedItem2itemInCollection" refer="uniqueItems">
  <xs:selector xpath="owner/item"/>
  <xs:field xpath="@idItem"/>
 </xs:keyref>
</xs:element>
```

Figure 9. The collectionWithLocalOwners structure.

```
<collections xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionsWithLocalOwners>
        <collectionWithLocalOwners>
            <item idItem="a"/>
            <item idItem="b"/>
            <owner>
                <item idItem="a"/>
            </owner>
            <owner>
                <!-- Each item within a collection can only be owned once. [2 lir
            </owner>
        </collectionWithLocalOwners>
        <collectionWithLocalOwners>
            <item idItem="a"/>
            <item idItem="c"/>
            <owner>
                <item idItem="a"/>
                <item idItem="c"/>
            </owner>
        </collectionWithLocalOwners>
    </collectionsWithLocalOwners>
</collections>
```

Figure 10. The collectionsWithLocalOwners instance.

Now we reorganize the data slightly, pulling ownership information from the collection and moving it up one level, into a new container element `collectionsWithGlobalOwners`, see Figure 11. In tandem, we also move the unique ownership constraint and the key reference constraint that expresses integrity of references from owned items to items in a collection up. The problem with this structure is that `item` attributes `idItem` that are locally unique within collections are not unique at the level of the container element `collectionsWithGlobalOwners`. The conflict resolutions rules for the upwards propagation of index tables imply, that on the instance level items with conflicting `idItem` attributes cannot be referenced by non-local owners. This phenomenon is illustrated in the XML instance sample-CollectionsWithGlobalOwners.xml see Figure 12.

```
<xs:element name="collectionsWithGlobalOwners">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="collection" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="owner" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="uniqueGlobalOwnership">
  <xs:selector xpath="owner/item"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
 <xs:keyref name="globalOwnedItem2itemInCollection" refer="collectionKey">
  <xs:selector xpath="owner/item"/>
  <xs:field xpath="@idItem"/>
 </xs:keyref>
</xs:element>
```

Figure 11. The collectionsWithGlobalOwners structure.

```
<collections xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionsWithGlobalOwners>
        <collection>
            <!-- This element cannot be referenced by owner,
                 due to conflict resolution rules. -->
            <item idItem="a"/>
            <item idItem="b"/>
        </collection>
        <collection>
            <!-- Duplicate idItem values allowed in different collections.
                 But: This element can not be referenced by owner. -->
            <item idItem="a"/>
            <item idItem="c"/>
        </collection>
        <owner>
            <!-- Reference cannot be resolved. <item idItem="a"/> -->
        </owner>
        <owner>
            <item idItem="c"/>
        </owner>
    </collectionsWithGlobalOwners>
</collections>
```

Figure 12. The collectionsWithGlobalOwners instance.

The previous example demonstrates that key reference constraints are not as fully adapted to the hierarchical nature of XML documents as are key constraints. Before delving further into causes and potential solutions, we show one straight-forward use of key reference constraints in a database-like scenario with flat, non-hierarch-

ical structures in XML instance sampleCollectionsWithOwnersDBStyle.xml, see Figure 13 and Figure 14.

```
<xs:element name="collectionsWithOwnersDBStyle">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="collectionsDBStyle"/>
   <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="uniqueOwnershipDBStyle">
  <xs:selector xpath="ownerDBStyle/itemInCollection"/>
  <xs:field xpath="@idCollection"/>
  <xs:field xpath="@idItem"/>
 </xs:key>
 <xs:keyref name="ownedItem2ItemInCollectionDBStyle"
  refer="itemInCollectionKey">
  <xs:selector xpath="ownerDBStyle/itemInCollection"/>
  <xs:field xpath="@idCollection"/>
  <xs:field xpath="@idItem"/>
 </xs:keyref>
</xs:element>
```

Figure 13. The collectionsWithOwnersDBStyle structure.

```
<collectionsWithOwnersDBStyle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionsDBStyle>
        <collectionDBStyle idCollection="x"/>
        <collectionDBStyle idCollection="y"/>
        <itemInCollection idCollection="x" idItem="a"/>
        <itemInCollection idCollection="x" idItem="b"/>
        <!-- Duplicate idItem values allowed in different collections -->
        <itemInCollection idCollection="y" idItem="a"/>
        <itemInCollection idCollection="y" idItem="c"/>
        <!-- Referenced collection must exist (referential integrity)
        <itemInCollection idCollection="z" idItem="a"/>
        -->
    </collectionsDBStyle>
    <ownerDBStyle>
        <itemInCollection idCollection="x" idItem="a"/>
    </ownerDBStyle>
    <ownerDBStyle>
        <itemInCollection idCollection="y" idItem="a"/>
        <!-- Violation of referential integrity: owned item  must exist
        <itemInCollection idCollection="z" idItem="a"/> -->
    </ownerDBStyle>
</collectionsWithOwnersDBStyle>
```

Figure 14. The collectionsWithOwnersDBStyle instance.

Let us now explore how we can reconcile key reference constraints with true hierarchies. How can we fix the container element `collectionsWithGlobalOwners`, referencing into locally defined keys? First of all, methodically, it is necessary to give unique IDs to collections. Then we need a method to refer to an item within a specific collection. Theoretically, this could be achieved in to possible ways:

- First, by having a global key constraint that states that items are uniquely defined by their own attribute `idItem` and by the attribute `idCollection` of the collection to which the item belongs. Unfortunately, such a key constraint cannot be defined in XML Schema, because field functions may only point to information within the element that is to be "keyed".
- Second, by keeping the local key constraint that guarantees locally unique attributes `idItem` within each collection and combining it with a two-part, "chaining" key reference constraint that references the collection with one component and the item within that collection with the second component. Evidently, XML Schema provides no such mechanism that locates a local index table with one component and an entry in that local index table with another component. We feel that such a chaining key reference mechanism would be the perfect complement to local key constraints, fully adapting not only key constraints but also key reference constraints to the hierarchical nature of XML documents.

We have seen that XML Schema's key reference mechanism, as it is currently defined, is too weak to express referential integrity with locally defined key constraints. We can solve this type of problems, however, with another mechanism that was introduced in XML Schema 1.1, namely assertions. We illustrate this with element `collectionsWithOwnersSensible` and the corresponding XML instance in sample-CollectionsWithOwnersSensible.xml, see Figure 15 and Figure 16.

```
<xs:element name="collectionsWithOwnersSensible">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="collectionWithID" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="idCollection" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:key name="uniqueItemsInCollection"> [3 lines]
   </xs:element>
   <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:assert
   test="every $ownerItem in ownerDBStyle/itemInCollection satisfies
   (some $collection in collectionWithID satisfies
   ($ownerItem/@idCollection=$collection/@idCollection and
   (some $item in $collection/item satisfies $ownerItem/@idItem=$item/@idItem)))"
  />
 </xs:complexType>
 <xs:key name="uniqueCollections"> [3 lines]
 <xs:key name="uniqueOwnershipSensible"> [4 lines]
</xs:element>
```

Figure 15. The collectionsWithOwnersSensible structure.

```
<collections xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionsWithOwnersSensible>
        <collectionWithID idCollection="x">
            <item idItem="a"/>
            <item idItem="b"/>
        </collectionWithID>
        <collectionWithID idCollection="y">
            <item idItem="a"/>
            <item idItem="c"/>
        </collectionWithID>
        <ownerDBStyle>
            <itemInCollection idCollection="x" idItem="a"/>
        </ownerDBStyle>
        <ownerDBStyle>
            <itemInCollection idCollection="y" idItem="c"/>
        </ownerDBStyle>
    </collectionsWithOwnersSensible>
</collections>
```

Figure 16. The collectionsWithOwnersSensible instance.

The XML Schema recommendation provides a solution to our problem that works but, unfortunately, introduces redundancy, namely to repeat the collection id on each item within the collection. This is a cross between the database-style solution and the hierarchical XML approach that accommodates the shortcomings of key and key reference constraints with respect to hierarchical data. We demonstrate this solution with the container element `collectionsWithOwnersHybridStyle` see Figure 17. We use the assertion mechanism of XML Schema 1.1 to guarantee that redundant information is consistent. An XML instance for the hybrid solution can be found in sampleCollectionsWithOwnersHybridStyle.xml, see Figure 18.

```
<xs:element name="collectionsWithOwnersHybridStyle">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="collectionHybridStyle" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element ref="itemInCollection" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="idCollection" type="xs:string" use="required"/>
     <xs:assert
      test="every $id in itemInCollection/@idCollection satisfies
        $id=@idCollection"
     />
    </xs:complexType>
   </xs:element>
   <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="uniqueItemsHybridStyle"> [4 lines]
 <xs:key name="uniqueCollectionsHybridStyle"> [3 lines]
 <xs:key name="uniqueOwnershipHybridStyle"> [4 lines]
 <xs:keyref name="ownedItems2ItemsInCollectionHybridStyle" [5 lines]
</xs:element>
```

Figure 17. The collectionsWithOwnersHybridStyle structure.

```
<collectionsWithOwnersHybridStyle xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
    xsi:noNamespaceSchemaLocation="collections.xsd">
    <collectionHybridStyle idCollection="x">
        <itemInCollection idCollection="x" idItem="a"/>
        <itemInCollection idCollection="x" idItem="b"/>
    </collectionHybridStyle>
    <collectionHybridStyle idCollection="y">
        <itemInCollection idCollection="y" idItem="a"/>
        <itemInCollection idCollection="y" idItem="c"/>
        <!-- Attributes idCollection in itemInCollection and in the parent
             collectionHybridStyle must be consistent.
        <itemInCollection idCollection="x" idItem="c"/> -->
    </collectionHybridStyle>
    <ownerDBStyle>
        <itemInCollection idCollection="x" idItem="a"/>
        <itemInCollection idCollection="y" idItem="a"/>
        <!-- Referenced collection must exist (referential integrity).
        <itemInCollection idCollection="z" idItem="a"/> -->
        <!-- Referenced collection must exist (referential integrity).
        <itemInCollection idCollection="z" idItem="a"/>
        -->
    </ownerDBStyle>
</collectionsWithOwnersHybridStyle>
```

Figure 18. The collectionsWithOwnersHybridStyle instance.

Unfortunately, the hybrid solution works for one level of collections only. It breaks down for recursive structures. We still have to verify that we can solve the problem with assertions even for the recursive case.

5. Conclusions and further work

XML Schema transfers one of the fundamental database concepts into the realm of XML documents, namely identity constraints, introducing the concepts of key constraint and key reference constraint. XML Schema allows index tables for key constraints that have local element scope, thus adapting the key-constraint concept successfully to the hierarchical nature of XML documents. However, adaptation is incomplete because locally defined key constraints are not fully available for referencing. Some kind of key chaining is missing.

We have demonstrated how assertions, that have been introduced with XML Schema 1.1, can be used as a substitute for key references in scenarios that involve hierarchy. With assertions, the key reference constraints are essentially expressed with XPath expressions.

Although XML Schema identity constraints are much more powerful than the ID/IDREF concepts that orginate from XML DTD, they don't seem to be used much in practice. We intend to present an overview regarding current practice at the 2014

XML Prague conference. We also hope that we have clarified some issues in connection with XML Schema's identity constraints.

This investigation was motivated by the second author's PhD thesis that targets XForms documents that are generated from an XML Schema and serve as fully functional editors for instances of the schema. Naturally, these editors should support identity constraints, guaranteeing uniqueness and referential integrity dynamically, while the document is edited. This can be achieved within XForms, when identity constraints are expressed as XPath expressions. We have demonstrated for the case of key reference constraints, how this can be accomplished with XPath 2.0. A more extensive discussion is included in Mustapha Maalej's PhD thesis.

6. Acknowledgement

Eric van der Vlist in personal communication pointed out to us that XML Schema's concepts of identity constraints are better adapted to database-like flat structures than to truely hierarchical structures, providing a valuable entry point into this investigation. Thank you!

Bibliography

[1] Mustapha Maalej: *Generieren von XML-Editoren in XForms aus XML Schema.* Ph.D. Thesis, TU München, 2014. In preparation.

[2] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn: *XML Schema Part 1: Structures Second Edition.* W3C Recommendation, W3C, October 2004, *http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.*

[3] Eric van der Vlist: *XML Schema.* O'Reilly, Kindle Edition, 2011.

[4] Priscilla Walmsley: *Definite XML Schema.* Prentice Hall, 2nd edition, 2012.

A. Appendix: Listing

This paper comes with an XSD file, as listed below completely. Screenshots of the schema and instances are embedded into the paper.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Items in a collection have unique IDs.
There are a number of ways how collections may be structured.
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="collections">
    <xs:complexType>
```

```
    <xs:choice>
      <xs:element ref="collection" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="collectionsDBStyle"/>
      <xs:element ref="collectionXMLStyle"/>
      <xs:element ref="collectionsWithLocalOwners"/>
      <xs:element ref="collectionsWithGlobalOwners"/>
      <xs:element ref="collectionsWithOwnersDBStyle"/>
      <xs:element ref="collectionsWithOwnersHybridStyle"/>
      <xs:element ref="collectionsWithOwnersSensible"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="collection">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="collectionKey">
    <xs:selector xpath="item"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
</xs:element>
<xs:element name="item" type="itemType"/>
<xs:complexType name="itemType">
  <xs:attribute name="idItem" type="xs:string" use="required"/>
</xs:complexType>
<xs:element name="collectionsDBStyle">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="collectionDBStyle" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element ref="itemInCollection" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="collectionDBStyleKey">
    <xs:selector xpath="collectionDBStyle"/>
    <xs:field xpath="@idCollection"/>
  </xs:key>
  <xs:key name="itemInCollectionKey">
    <xs:selector xpath="itemInCollection"/>
    <xs:field xpath="@idCollection"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
  <xs:keyref name="itemRefersToCollection" refer="collectionDBStyleKey">
    <xs:selector xpath="itemInCollection"/>
    <xs:field xpath="@idCollection"/>
```

```
  </xs:keyref>
</xs:element>
<xs:element name="itemInCollection">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="itemType">
        <xs:attribute name="idCollection" type="xs:string" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="collectionDBStyle">
  <xs:complexType>
    <xs:attribute name="idCollection" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="collectionXMLStyle">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="collectionXMLStyle"/>
      <xs:element ref="item"/>
    </xs:choice>
    <xs:attribute name="idCollection" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:key name="collectionXMLStyleKey">
    <xs:selector xpath="collectionXMLStyle"/>
    <xs:field xpath="@idCollection"/>
  </xs:key>
</xs:element>
<xs:element name="collectionsWithLocalOwners">
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="collectionWithLocalOwners"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="collectionWithLocalOwners">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="owner" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="uniqueItems">
    <xs:selector xpath="item"/>
    <xs:field xpath="@idItem"/>
```

```
  </xs:key>
  <xs:key name="uniqueOwnership">
    <xs:selector xpath="owner/item"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
  <xs:keyref name="ownedItem2itemInCollection" refer="uniqueItems">
    <xs:selector xpath="owner/item"/>
    <xs:field xpath="@idItem"/>
  </xs:keyref>
</xs:element>
<xs:element name="owner">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="collectionsWithGlobalOwners">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="collection" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="owner" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="uniqueGlobalOwnership">
    <xs:selector xpath="owner/item"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
  <xs:keyref name="globalOwnedItem2itemInCollection" refer="collectionKey">
    <xs:selector xpath="owner/item"/>
    <xs:field xpath="@idItem"/>
  </xs:keyref>
</xs:element>
<xs:element name="collectionsWithOwnersDBStyle">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="collectionsDBStyle"/>
      <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="uniqueOwnershipDBStyle">
    <xs:selector xpath="ownerDBStyle/itemInCollection"/>
    <xs:field xpath="@idCollection"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
  <xs:keyref name="ownedItem2ItemInCollectionDBStyle"
```

```
    refer="itemInCollectionKey">
    <xs:selector xpath="ownerDBStyle/itemInCollection"/>
    <xs:field xpath="@idCollection"/>
    <xs:field xpath="@idItem"/>
  </xs:keyref>
</xs:element>
<xs:element name="ownerDBStyle">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="itemInCollection" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="collectionsWithOwnersHybridStyle">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="collectionHybridStyle"
        minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="itemInCollection"
              minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
         <xs:attribute name="idCollection" type="xs:string" use="required"/>
          <xs:assert
            test="every $id in itemInCollection/@idCollection satisfies
            $id=@idCollection"
          />
        </xs:complexType>
      </xs:element>
      <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="uniqueItemsHybridStyle">
    <xs:selector xpath="collectionHybridStyle/itemInCollection"/>
    <xs:field xpath="@idCollection"/>
    <xs:field xpath="@idItem"/>
  </xs:key>
  <xs:key name="uniqueCollectionsHybridStyle">
    <xs:selector xpath="collectionHybridStyle"/>
    <xs:field xpath="@idCollection"/>
  </xs:key>
  <xs:key name="uniqueOwnershipHybridStyle">
    <xs:selector xpath="ownerDBStyle/itemInCollection"/>
    <xs:field xpath="@idCollection"/>
    <xs:field xpath="@idItem"/>
```

```
    </xs:key>
    <xs:keyref name="ownedItems2ItemsInCollectionHybridStyle"
      refer="uniqueItemsHybridStyle">
      <xs:selector xpath="ownerDBStyle/itemInCollection"/>
      <xs:field xpath="@idCollection"/>
      <xs:field xpath="@idItem"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="collectionsWithOwnersSensible">
    <xs:complexType>
      <xs:sequence>
       <xs:element name="collectionWithID" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
           <xs:attribute name="idCollection" type="xs:string" use="required"/>
          </xs:complexType>
          <xs:key name="uniqueItemsInCollection">
            <xs:selector xpath="item"/>
            <xs:field xpath="@idItem"/>
          </xs:key>
        </xs:element>
        <xs:element ref="ownerDBStyle" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:assert
        test="every $ownerItem in ownerDBStyle/itemInCollection satisfies
        (some $collection in collectionWithID satisfies
        ($ownerItem/@idCollection=$collection/@idCollection and
        (some $item in $collection/item satisfies $ownerItem/@idItem=$item/▶
@idItem)))"
      />
    </xs:complexType>
    <xs:key name="uniqueCollections">
      <xs:selector xpath="collectionWithID"/>
      <xs:field xpath="@idCollection"/>
    </xs:key>
    <xs:key name="uniqueOwnershipSensible">
      <xs:selector xpath="ownerDBStyle/itemInCollection"/>
      <xs:field xpath="@idCollection"/>
      <xs:field xpath="@idItem"/>
    </xs:key>
  </xs:element>
</xs:schema>
```

Data and Documents, Together Again

RDF-in-XML for simple and flexible data management

Charles Greer
MarkLogic Corporation
<cgreer@marklogic.com>

Abstract

The practice of embedding RDF triples in XML documents proves a surprisingly useful paradigm for data stores that combine structured and unstructured data.

In this paper I consider well-known features of an XML document-oriented database, and mix those with RDF data and SPARQL queries. On the one hand, XML documents are well-suited for encoding human-readable text and markup. On the other hand, RDF is an) emergent de facto standard for structured, typed, and distributed data. These two worlds are conceptually quite distinct; RDF data has no inherent interaction with the concept of the document boundary. But it turns out that the document boundary can scope RDF access; the interaction between RDF data and their enclosing documents can help solve problems around structured and unstructured data together in the same database management system.

In this paper I explore a few aspects in which RDF and documents are complementary when used together. First, I will consider the hybridization of query and mixed-content search. Since we can now mix data and text content freely, the lines between search and query blur in favor of a kind of information retrieval based on both relevance and exactitude. Second, I'll take a look different kinds of RDF-in-XML documents. Some examples of document-based RDF use cases include a simple (and naive) method for maintaining rule-based inference state machines, and data binding objects to XML within a greater RDF context.

Document databases are mature and provide many capabilities that are missing from native RDF triple stores. We can help people leverage structured data simply by overlaying that structured data on top of an XML document-oriented substrate, and at the same time have providing continuity to legacy applications already using documents. Storing RDF in XML document databases opens them up to a wide new range of capabilities, as the global indexing and querying of data in the XML database becomes more interconnected and randomly accessible when indexed as RDF.

Keywords: XML, RDF, Data Architecture

1. Introduction

I've come to believe (and it is a belief) that SPARQL will replace SQL as query language of choice for the post-relational generation of databases. But saying that so an XML conference begs further explanation. I'm telling this story at an XML conference because XML documents happen to provide an easy-to-understand and flexible transport mechanism and serialization container for managing RDF data. Furthermore, XML databases can function as RDF triple stores, with wide-ranging implications.

How is RDF-in-XML different from simply using XML to encode data directly? Well, the main distinction in the context of this paper is that data in an RDF form, although contained and managed in a document, is not bounded whatsoever by that document with regard to other RDF data. The interface to RDF data remains the same regardless of how documents are managed; the documents provide the context and scope in which one accesses the structured data therein, without affecting interfaces to those data.

This RDF-in-XML approach assumes that documents can contain arbitrary RDF data, intermingled with, derived from, or invisible to the XML context. The documents in the database contain RDF triples as first-class citizens; they are incorporated seamlessly into the graphs accessed by a SPARQL query processor. However, these documents can also contain markup, text, and be managed as a single human-readable atom in the corpus. A collection of such documents provides the data architecture to bridge search and exact queries, including inferencing, in ways hitherto either unknown or very awkward.

Even for large datasets, a document-oriented database provides a single simple stratum of the data model at which to manage updates and dependencies among sets of triples.

> **Note**
> This paper uses MarkLogic for its examples. Indeed, there's really no other database that combines these features at this point, but my hope is that this paradigm's usefulness transcends the particular implementation that supports it now.

2. The document model, with triples as special guest.

Here is an XML document that is used in MarkLogic to contain RDF triples. I'll refer to this structure throughout the paper, simply to bore you with these simple facts about an accordion:

Figure 1. The Saltarelle

1. "Charles's Saltarelle is a Button Box (diatonic accordion)
2. It has 2.5 rows of buttons.
3. It is in the key of C
4. It is (also) in the key of G.
5. "Key of G" is is known as (labeled with) "Key of G"

Here is the XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<mydocument>
  <sem:triples xmlns:sem="http://marklogic.com/semantics">
    <sem:triple>
      <sem:subject>http://example.org/charlesSaltarelle</sem:subject>
      <sem:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
sem:predicate>
      <sem:object>http://example.org/ButtonBox</sem:object>
    </sem:triple>
    <sem:triple>
      <sem:subject>http://example.org/charlesSaltarelle</sem:subject>
      <sem:predicate>http://example.org/numberOfRows</sem:predicate>
      <sem:object
         datatype="http://www.w3.org/2001/XMLSchema#float">2.5</sem:object>
    </sem:triple>
```

```
    <sem:triple>
      <sem:subject>http://example.org/charlesSaltarelle</sem:subject>
      <sem:predicate>http://example.org/key</sem:predicate>
      <sem:object>http://example.org/keys/C</sem:object>
    </sem:triple>
    <sem:triple>
      <sem:subject>http://example.org/charlesSaltarelle</sem:subject>
      <sem:predicate>http://example.org/key</sem:predicate>
      <sem:object>http://example.org/keys/G</sem:object>
    </sem:triple>
    <sem:triple>
      <sem:subject>http://example.org/keys/G</sem:subject>
     <sem:predicate>http://www.w3.org/2000/01/rdf-schema#label</sem:predicate>
      <sem:object>Key of G</sem:object>
    </sem:triple>
  </sem:triples>
      <para>This is arbitrary mixed content that perhaps, just <i>perhaps</i>,
          has something to do with the data included.  I could explain, for
         example, that a cherry-wood finish is a veneer of cherry over a balsa
          wood frame.  I could say random and unpleasant things.</para>
</mydocument>
```

This particular use of XML for representing triples is not a standard, but that's not relevant for the ideas in this paper. The important thing about how XML encodes RDF is that (in MarkLogic) such XML elements are all indexed triples, accessible to the SPARQL query engine immediately upon ingestion. I can manage such a document like any other XML with XQuery:

```
xdmp:document-insert("/accordion-triples.xml", <mydocument>...</mydocument>)
```

You can query the triples via a SPARQL endpoint or as a string argument to the XQuery `sem:sparql($sparql)` function:

```
select ?s ?o where { ?s <http://example.org/key> ?o }

?s                                          ?o
<http://example.org/charlesSaltarelle> <http://example.org/keys/C>
<http://example.org/charlesSaltarelle> <http://example.org/keys/G>
```

MarkLogic provides low-level access to the values in the triple index as well. There is a primitive function simply to extract matching patterns from the triple index. The following will return triples from the index that have `"http://example.org/charlesSaltarelle"` in subject position, and any data at all in predicate or object position:

```
cts:triples(sem:iri("http://example.org/charlesSaltarelle"), (), ())

@prefix xs: <http://www.w3.org/2001/XMLSchema#> .
```

```
@prefix : <http://example.org/>
:charlesSaltarelle
    a :ButtonBox ;
    :key :keys/C , :keys/G ;
    :numberOfRows "2.5"^^xs:float .
```

So that's how to access embedded RDF. But despite this document-agnostic access, the backing XML is still a first-class citizen within the database, and you can use the same document-oriented queries that have been supported with XQuery for years. The point of the query that follows is not to return triples, but rather to find documents which contain the triples specified by the query. This kind of query bridges the worlds of RDF and XML by using the structured data embeded within a document to qualify it for search filters:

```
cts:search(collection(),
  cts:triple-range-query(sem:iri("http://example.org/charlesSaltarelle"),
                         (), ()))

(returns the entire document as quoted above)
```

This query returns all of the *XML documents* which contain embedded triples that have the given subject.

3. Mixing triple queries and document searches

So you've now are the building blocks of a hybrid application. SPARQL and XQuery access to the triple index complements how you can use documents to manage the blocks of RDF that go together and should be accompanied by mixed text. Now I'll enumerate some of the use cases that might use RDF-in-XML to mixed search and SPARQL.

3.1. Document security to change access to structured data.

First, one might use implicit restrictions on document access to qualify the data returned by a SPARQL query. Embedding RDF in documents allows a straightforward and well-known security overlay across the entire triple store.

If I insert documents such as the one above with a certain set of permissions, then SPARQL queries will respect that document's permissions. Thus queries automatically respect access restrictions, orthogonally to the queries themselves.

It's hard to imagine data about my accordions being sensitive to security considerations, so let's imagine we have instead a database that contains information about diseases and also records for individual patients and their diagnoses. The metadata about diseases can be embedded within documents with minimal security, since they contain public information. Documents about individual patients, however, would be restricted in order to protect their privacy.

```
Document 1 contains this triple:

@prefix : <http://example.org/health/> .
:familial_hypercholesterolemia a :HereditaryDisease .

Document 2 contains this triple, with restricted permissions

@prefix : <http://example.org/health/> .
:patientJohnDoe :diagnosedWith :familial_hypercholesterolemia .
```

The following query would reveal information about patients only to clients with sufficient access privileges. Anonymous users could hit the same database but not have access to individual cases.

```
prefix : <http://example.org/health/>
select ?patient ?disease
where
{ ?disease a :HereditaryDisease
  OPTIONAL { ?patient :diagnosedWith ?disease }
}

public's result:

?disease                          ?patient
:familial_hypercholesterolemia    [null]

doctor's result:
?disease                           ?patient
:familial_hypercholesterolemia     :patientJohnDoe
```

3.2. Document search to constrain query to subset of corpus.

We can also select documents explicitly, using familiar search methods. One way to do this might be to arrange documents within a directory structure in the database. Each directory represents on way to partition the available data so as to

Documents are stored at a URI, and directories are implicit given the structure of the document URI.

Say an accordion company has three product lines [8]. Maybe the data for each accordion in the production line is stored at a URI that corresponds to the product line:

```
/products/2014/centreville
/products/2014/sterling
/products/2014/parismelrose
```

We can use these directory structures to limit queries just to the documents contained within each directory. So, to get prices and descriptions for all products in the "Centre Ville" line, we use this combination of SPARQL and XQuery in MarkLogic:

```
sem:sparql('
  select ?desc ?price
  where
  {?product a :Product ;
            :hasPrice ?price ;
            :hasDescription ?desc }
  ', (), cts:directory-query("/products/2014/centreville"))
```

Out of respect for RDF purists, I'm bound to mention that this kind of semantics could of course be encoded directly within the RDF data model. Keeping the document concerns separate from the structured data model however encourages query reuse, by manipulating the context in which a single query operates.

From this `cts:directory-query` example, one can extrapolate. Word searches can also be used as qualifying queries to determine the dataset upon which SPARQL queries run. Next, we'll consider how documents provide not just a partitioning strategy, but also a natural unit of data management, a way to keep updates limited to a set of triples that naturally go together.

3.3. Document updates to maintain data state.

In the history of native RDF triple stores, users found that maintaining triples without some external context for them is really impractical. N-Quads [5] were introduced for just this reason; a fourth IRI (in addition to the three in the "triple") was added to provide the grouping context for a set of triples. Documents that contain triples provide this same kind of management layer to data encoded in RDF.

This idea doesn't take much to illustrate; it just means that I can model objects as RDF, and package them as documents, and with that combination of model and packaging, I get an alternate to named graphs as atomic unit of update. Document-oriented databases already understand this approach to managing data.

One might wonder how to get standard RDF serializations in and out of a document store. It's as simple as using an input and output transform on a REST endpoint. The following XQuery module provides a function to parse RDF in turtle format and return MarkLogic's internal format for triples, as an XML document:

```
xquery version "1.0-ml";

module namespace ingest = "http://marklogic.com/rest-api/transform/ingest";

import module namespace sem = "http://marklogic.com/semantics"
    at "/MarkLogic/semantics.xqy";
```

```
declare function ingest:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node())
as document-node() {
    let $turtle := $content/node()
    let $triples := sem:rdf-parse($turtle, "turtle")
    return document {
        element sem:triples {
            $triples
        }
    }
};
```

MarkLogic's REST API can be used in combination with such transform modules to enable RESTful CRUD for RDF documents. If you install the above transform:

```
curl -X PUT \\
     -Hcontent-type:application/xquery \\
     -d@'transform.xqy' \\
      http://rest-host:port/v1/config/transform/rdfparse
```

Then you can use it on the document CRUD endpoint to transform turtle documents into MarkLogic XML.

```
curl -X PUT \\
     -Hcontent-type:text/turtle \\
     -d@'turtle.ttl' \\
     http://rest-host:port/v1/documents?uri=docuri.xml&ingest=rdfparse
```

Note

The only difference between using this method and the graph protocol endpoint included with MarkLogic REST API is that the triples ingested with this method will be guaranteed to reside together within a single document, while the graph protocol implementation chooses another method for managing documents.

4. My content accompanies data I can query.

You've seen that it's possible to mix RDF and documents according to the document-oriented functionality you want out of the database. This section considers the varying relationship that this structured data has with the document envelope, and the consequences for search and query that emerge from various embedding strategies.

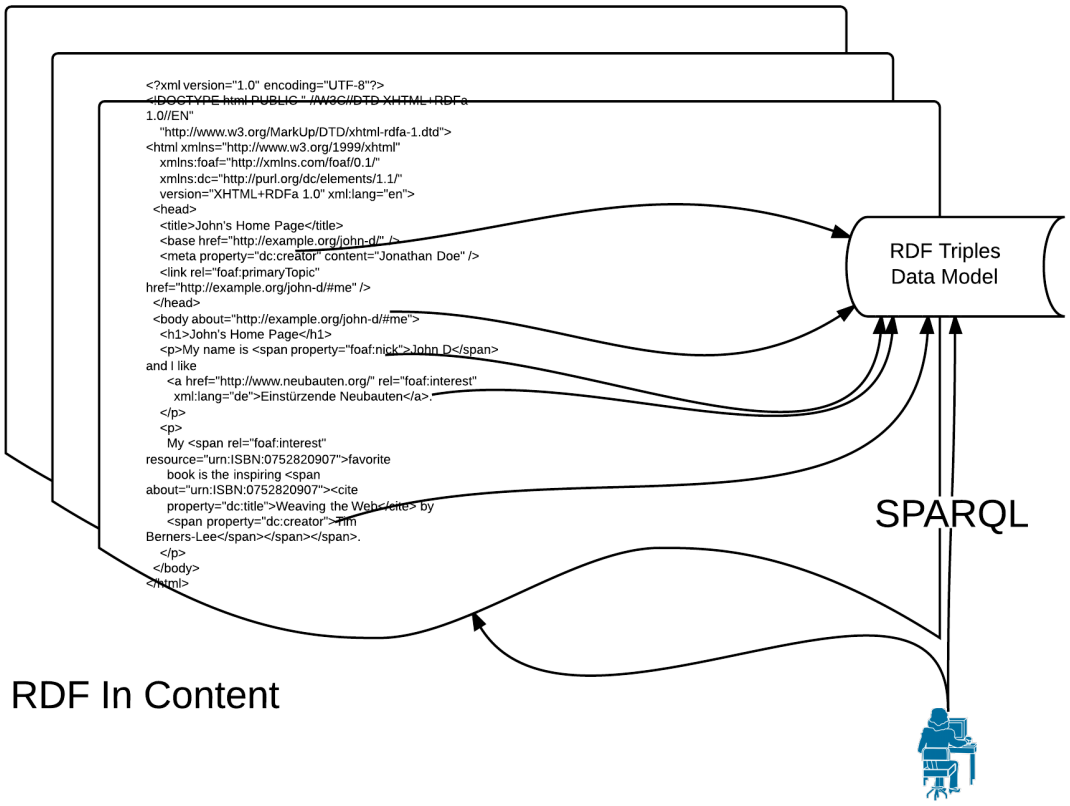
4.1. RDF Data Embedded in (or extracted from) Content

While MarkLogic currently supports triples only in the internal XML format, embedded RDF already exists in a number of forms. The most well known method in use today is RDFa [6]. Others include schema.org [9]markup, microformats [4], and to some extent JSON-LD [3].

The idea behind each of these is that markup contains not just text for reading, but also has implicit data in it. RDFa annotates these data in order to make it explicit. An RDFa parser can traverse an HTML or XML document and recognize triples within it.

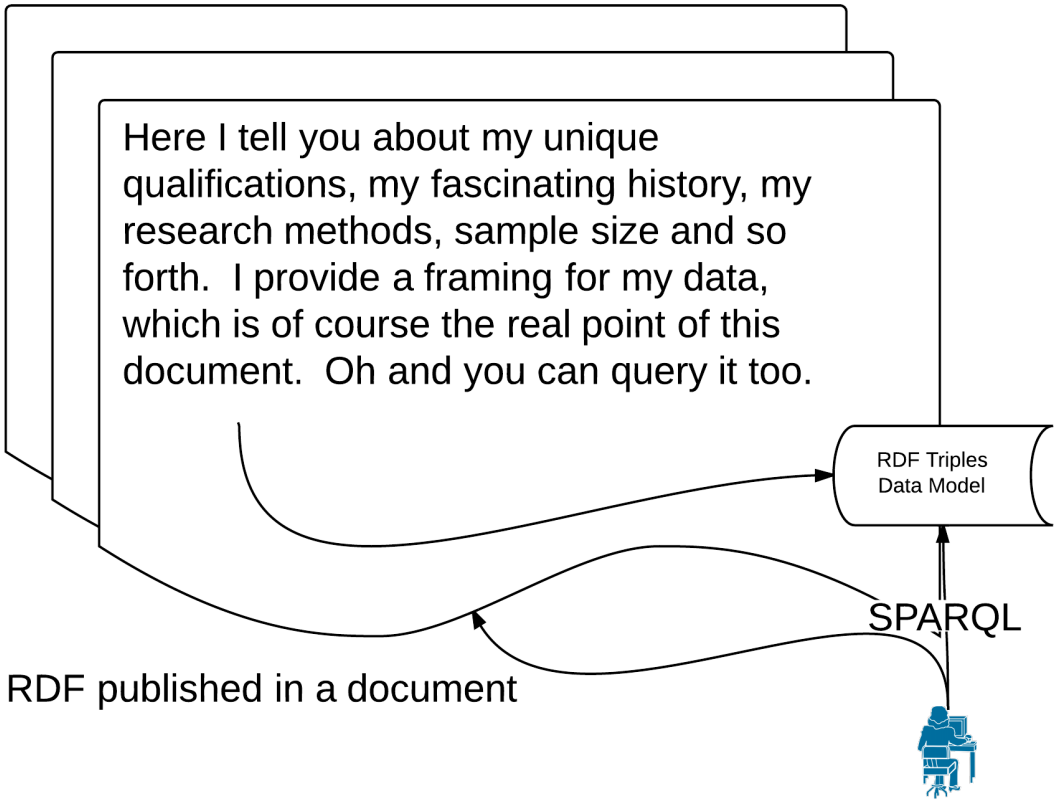
As an example, a document that contains form data is very useful as a data entry template or display for reading; annotations that are not of interest to a human reader can tell the database how that data can be queried directly with SPARQL. A form document, for example, might be marked up with not only values, but the names and referents of individual form elements.

This kind of RDF is distinguishable from the ones in subsequent sections because it actually interleaves the data elements within mixed content; the data is embedded within a template for human consumption.



4.2. Content describes the data.

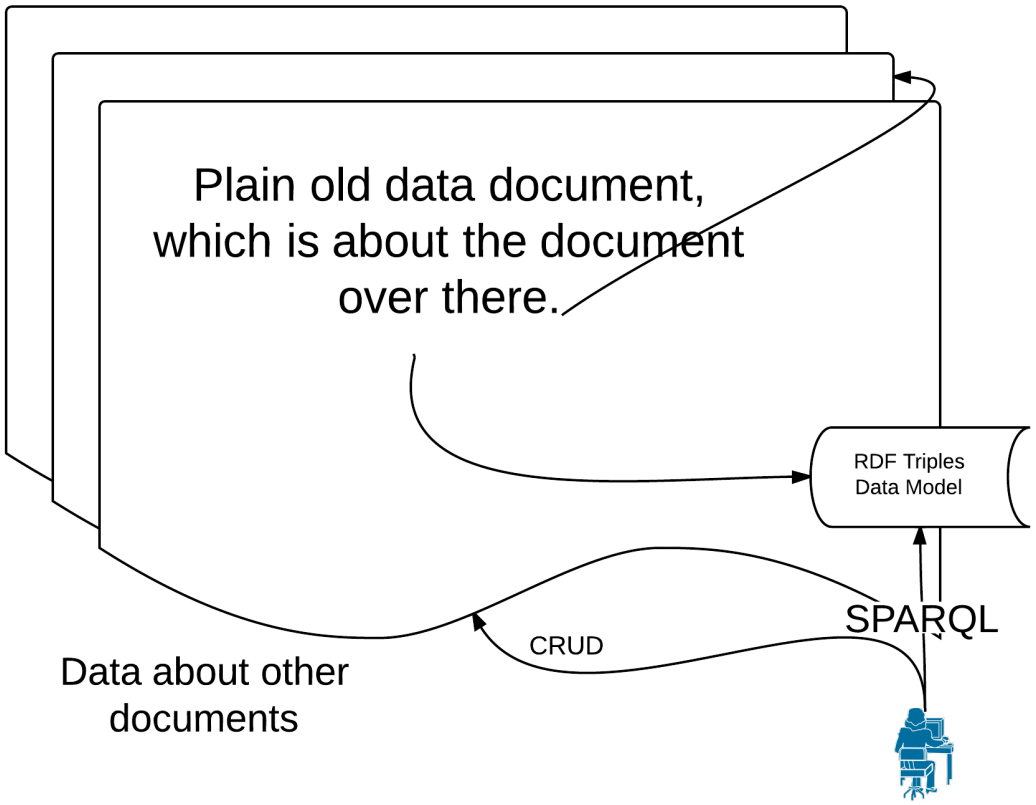
Let's flip the preceding scenario on its head. Sometimes a dataset is the primary published artifact. Say, for example, I wanted to publish a database about all the various kinds of accordions in use. This data would be all but useless without some amount of accompanying documentation. I might wish to write the whole narrative concerning my data, and also include that data within the same document such that it's clear they are always to travel together. This is a perfect strategy for simplified provenance of data, and also for circulation of published data with docs.



4.3. Document contains statistics about other data.

Often one processes RDF graphs, say, using CONSTRUCT transformations or reasoners. The processing from such operations is often stored in a named graph separate from the originals, so that it can be invalidated or reprocessed if dependencies upstream change. Rather than naming the results of such an operation and managing them at the application tier, it's may be simpler to use existing document-oriented facilities to manage the dependencies among datasets, rather than to manage graph names in an application tier.

One useful application of this idea would be to store documents that contain entity annotations. The original document and thesaurus are two documents, and the annotation metadata could be a third, with a clear dependency graph in case either the document or thesaurus change.



4.4. Commonalities

All of these cases have in common the ability to pool all structured data into one query space, while maintaining dependencies and relationships among documents and their associated data clear and easy-to-manage. The separation of document boundaries from data boundaries makes essentially a kind of pluggable RDF data store, and a wide variety of ways to leverage it.

5. An example – Rule-based inference

So how might we use RDF-in-XML? Here's a naive method for constructing and managing rule-based inferences. I'm assuming the same kind of document-embedded RDF as above in section 2. By using the XML document as an input to an inferencing toolchain, we can easily manage the relationship between "given" triples, those

which as asserted by a data model, and "inferred" ones, which have been generated from the given triples using generative rules.

Inference operations in RDF, devil-in-details complexity aside, can be thought of as update triggers. The idea is to match existing RDF triples, and generate some more triples from them. You implement these rules with SPARQL CONSTRUCT queries. Combined with XQuery update support, we can manage inferences. Here's a fragment of RDF that extends the accordion model further:

```
:charlesSaltarelle a :ButtonBox ;
                   :finish :cherryWoodFinish .

 :ButtonBox a owl:Class ;
          rdfs:label "Button Box"@en .
:Melodeon a owl:Class ;
          rdfs:label "Melodeon"@en ;
          rdfs:label "Melodeon"@cs .
```

A "button box" is a kind of accordion, which is known in many places as "Melodeon." If for some reason I had included a `:Melodeon` class in my data, I could assert that :Melodeon and :ButtonBox are identical by including the following statement in my dataset:

```
:ButtonBox owl:equivalentClass :Melodeon .
```

You can implement owl:equivalentClass by asserting thatWhile a specially-made OWL reasoner could take this assertion and generate the required inferences, you can also do it "by hand" with a CONSTRUCT QUERY:

```
"For every class1 and class2 that are equivalent,
     objects of type class1 are also of type class2,
     and vice-versa"

CONSTRUCT {
    ?x a ?c2 .
}
WHERE {
    ?c1 owl:equivalentClass ?c2 .
    ?x a ?c1 .
}[7]
```

The output of this query is a single triple:

```
:charlesSaltarelle a :Melodeon
```

I can store the output of the CONSTRUCT query in a new document. Since I have XQuery as a programming language, doing so is quite simple:

```
xdmp:document-insert("/charles-saltarelle-inferences.xml", ▶
<sem:triples>{sem:sparql('THE QUERY ABOVE')}</sem:triples>)
```

Actually, there's one more step to making this a fully-functioning reasoning scenario. Since I want this result to be tied directly to an input document, I'll filter the input query as well, in order to tie a particular document to a single inferred one:

```
xdmp:document-insert("/charles-saltarelle-inferences.xml",
     <sem:triples>{
          sem:sparql('THE QUERY ABOVE',
          (),
          cts:document-query("/charles-saltarelle-asserted.xml"))}
     </sem:triples>)
```

Imagine now that this XQuery is the code executed for an update trigger. A trigger firing on update to the original document will regenerate inferences, and keep the model up-to-date, within a transaction.

6. Another Example: Data pipelines

There are well-known methods to describe pipelines and state machines over documents. If your documents are RDF-in-XML, then such methods can be used to manage structured data. The document boundary makes it simple(r) to maintain the multitude of documents that appear in a processing pipeline.

Here's a common scenario that I've been considering, cribbed from the "Dynamic Semantic Publishing" scenario popularized by Jon O'Donovan and Jem Rayfield [1]. Let's say I have these triples in my database:

```
:ButtonBox a owl:Class ;
    rdfs:subClassOf :Accordion ;
    :tags ("Button Box", "Diatonic Accordion", "Melodeon") .
:CajunBox a owl:Class ;
    rdfs:subClassOf :ButtonBox ;
    rdfs:label "Cajun Box" ;
    :tags ("Cajun Accordion" "Single Row") .
:ThreeRow rdfs:subClassOf :ButtonBox ;
    :tags ("Tex Mex Accordion", "Three-row button box") .
```

I can use a classification scheme such as this one to provide entity tagging for textual documents. First step is to have the input document available, say this one:

```
<text>
  The Cajun Accordion is still played, manufactured and taught
  throughout Louisiana.
</text>
```

I can use the tags from the RDF as search terms. I search for them in input documents and store information about matches, along with ontology information, in a new document. So searching for tags from the classification, I'll be able to tag this docu-

ment with `:CajunBox`. These triples, stored in a new document, can provide match information to the SPARQL engine:

```
:match1 a :Match ;
    :startToken 10;
    :endToken 24 ;
    :entity :CajunBox ;
    :classes ( :ButtonBox, :Accordion ) .
```

Since I'm using a document-oriented database, it's simple to track the dependencies between the original text and the tag document.

7. Databinding into Structured Context

A last example to conclude. We can mix RDF data and typical XML data binding scenarios. Java Objects (POJOs) are often used for a domain model. An instance of this Java class:

```
@OWLClass
public class ButtonBox {
 long id;
 String brand;
 int numberOfButtons;

...  getters and setters ...
}
```

Transforms trivially using JAXB [2] (or XStream [10]) into the following XML:

```
<ButtonBox>
    <id>1</id>
    <brand>Saltarelle</brand>
    <numberOfButtons>26</numberOfButtons>
</ButtonBox>
```

While it would be possible to bind this Java object to RDF instead, XML data binding is nearly ubiquitous, and perhaps less storage-intensive.

The intent of the `OWLClass` attribute on the class is to signal a data-access-layer to insert a triple along with the XML, resulting in this document:

```
<ButtonBox>
    <sem:triple>
       <sem:subject>http://example/org/charlesSaltarelle</sem:subject>
       <sem:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
sem:predicate>
       <sem:object>http://example.org/ButtonBox</sem:object>
    </sem:triple>
    <id>1</id>
    <brand>Saltarelle</brand>
```

```
    <numberOfButtons>26</numberOfButtons>
</ButtonBox>
```

If I happen to have a database full of such documents, one can serialize them as part of a SPARQL result set. Thus the Java objects can themselves be part of a larger RDF ecosystem. Presuming that the URIs of such objects are available as triples, this query:

```
select ?uri (fn:doc(?uri) as ?pojo)
where
{
  ?uri a :ButtonBox
}
```

would return all of the ButtonBox objects in the database in the following kind of format, which could be marshalled back into Java Objects:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="uri"/>
    <variable name="doc"/>
  </head>
  <results>
    <result>
      <binding name="uri">
        <uri>http://example/org/charlesSaltarelle</uri>
      </binding>
      <binding name="doc" type="XMLLiteral">
        <ButtonBox xmlns="">
          <sem:triple>
            <sem:subject>http://example/org/charlesSaltarelle</sem:subject>
            <sem:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
sem:predicate>
            <sem:object>http://example.org/ButtonBox</sem:object>
          </sem:triple>
          <id>1</id>
          <brand>Saltarelle</brand>
          <numberOfButtons>26</numberOfButtons>
        </ButtonBox>
      </binding>
    </result>
    ...
  </results>
</sparql>
```

This method provides a way to construct scalable data-access layers in SPARQL, while keeping marshalling and unmarshalling as expected from enterprise Java applications.

8. Concluding

Documents in XML are among the more expressive data models. This expressiveness allows documents to embed and/or encode simpler structures, such as JSON or RDF. When a database knows what embedded RDF looks like in XML -- it can index, query and retrieve it. This new access to RDF within documents combines two very powerful modeling techniques, and when used together, they appear to tear down data management boundaries.

Bibliography

[1] Jem Rayfield, "Sports Refresh: Dynamic Semantic Publishing", 2012 http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html

[2] http://www.oracle.com/technetwork/articles/javase/index-140168.html

[3] http://json-ld.org/

[4] http://microformats.org/

[5] Richard Cyganiak, et al. "N-Quads: Extending N-Triples with Context". http://sw.deri.org/2008/07/n-quads/

[6] Shane McCarron; et al. RDFa Core 1.1 Syntax and processing rules for embedding RDF through attributes. 07 June 2012. W3C Recommendation. http://www.w3.org/TR/2012/REC-rdfa-core-20120607/

[7] Holger Knublauch RDFS Plus as a subset of OWL RL in SPARQL Rules http://composing-the-semantic-web.blogspot.com/2010/09/rdfs-plus-as-subset-of-owl-rl-in-sparql.html

[8] Saltarelle Accordions, Company website. http://www.saltarelle.com

[9] http://schema.org

[10] XStream http://xstream.codehaus.org/

Scientific Computing in the Open Web Platform

R. Alexander Milowski
ILCC, School of Informatics, University of Edinburgh
<alex@milowski.com>

Henry S. Thompson
ILCC, School of Informatics, University of Edinburgh
<ht@inf.ed.ac.uk>

Abstract

Publishing and using scientific data on the Web is difficult; size and data formats thwarts its use within the browser. Yet, the Open Web Platform provides a basis for many forms of computing and communication and so we look to the principles of Web Architecture to help enable scientific data on the Web. Through a combination of these principles and the use of RDFa annotation technologies, we describe a methodology for publishing data and show how it can be computed upon within the Web browser as a platform for scientific computing.

1. Science and the Open Web Platform

Publishing, accessing, and processing scientific data on the Web is much harder than publishing other content on the Web. In this paper we advance the claim that this can and should be fixed, by a judicious combination of existing Web technologies and some core principles of Web Architecture [1]. We'll examine the ability of the existing Web to enable scientific data to become a first-class constituent.

The *Open Web Platform* (OWP) [2] is a "platform for innovation, consolidation and cost efficiencies" focused on those things happen within or intersect the actions of the Web browser. This platform is defined by both the shared behavior expected by the publisher and users of content and services--a type of contract readable by developer and authors alike. The collection of individual recommendations (standards documents), technologies, practical algorithms, APIs, vocabularies, and their interactions make this a cohesive and motivating platform for business and consumers alike.

Consumers use this platform to access the Ordinary Web; the typical Web pages, whether mapped from database content or hand-authored by individuals, organized by content-management systems or on an *ad hoc* basis, hosted by Web servers, and

possibly discovered and/or disseminated by search engines or social networks. Time has shown that operating on the Ordinary Web has become increasingly easy. Tools and technologies have advanced to enable the ordinary person to publish information on their blogs, as comments or reviews, or to build their own Web sites.

It is by using these relationships between the written word, their context in the document's markup, and the linked structure of the Web, that search engines and other systems derive knowledge from the Web. These systems are able to apply pipelines of processing to extract meaning from the context of the markup and the native language to build massive Deep Web databases [3] to use for applications of search, relevance, or mapping. These Deep Web services, like mapping services that expose mash-ups of the Web and GIS data, wouldn't be able to exist unless they were embedded in and harvested from information from the Ordinary Web.

The result is that by participating in the Web in a seemingly minor way, the ordinary user is creating value for the Web as a whole. That is, there is a huge economy of scale in allowing people to publish information on the Web in simple ways that is then easily harvested by some set of criteria. The information published is informal, often unreliable, possibly incorrect, but, taken as a whole, very valuable.

But for scientific endeavors, the story degrades: data is inaccessible due to formats, size, resource constraints, and limited discovery. Deep Web services do not really exist for scientific data as Web crawlers tend to ignore data they do not understand. As a final insult, data that is accessible is usually in formats incompatible with the Web browser as a platform.

2. Scientific Data Sets

A survey of the geospatial data contained in the US Government repository at `data.gov` as shown in Figure 1 reveals that a majority, possibly more than two thirds, of the data is or contains tabular data. Similar forays into other areas of scientific data sets reveal a propensity for the use tabular data formats. While the complexity of specific collections of data may go beyond simple tables of data, at a particular level of granularity, data is often tables of measurements of observed phenomenon.

For example, the International Virtual Observatory Alliance (IVOA) [4] develops standards to enable astronomers to exchange information directly across the Web and, while their data sets contain “images” in various formats and different instrument measurements, their basic method of data exchange is a XML vocabulary for tabular data called VOTable [5]. The various services they provide are oriented around the table as a unit of information and we've used their model as inspiration for this generalized approach to scientific data.

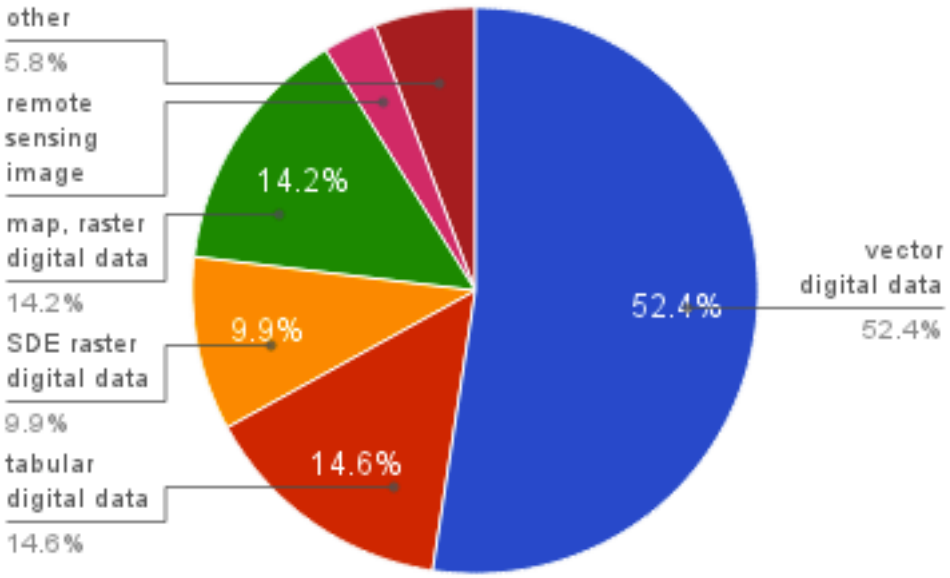


Figure 1. Geospatial Data by Type

3. Changing the Paradigm

We want to come back to the idea of the OWP as a mechanism for computing and aggregation and enable real scientific computations to take place in real time. To accomplish this, we need:

1. A methodology for publishing scientific data sets onto the Web so that they are accessible.
2. A model for processing data within the browser.
3. The APIs necessary to support this within the OWP.

First we must address the problem of sharing potentially large data sets with precise semantics in "small enough" portions that the Web browser can process the data. Any reasonable sized scientific data set can potentially overload the browser--especially if it is time-series data. As such, data handling needs markup, annotations, resource structures, and naming rules that all work together to allow the browser platform to navigate the data set.

Within the browser, processing the data cannot happen all at once or otherwise the efforts of (1) would be in vain. A processing model such as map/reduce needs to be applied so that a computational process can be enacted without overloading the browser or host.

Finally, accessing data cannot be *ad hoc* and so additional APIs within the browser may be necessarily. Specifically, if RDFa annotations are used, then some kind of RDFa API will be necessary. Which APIs are necessary is a direct consequence of the choices made in (1).

4. The PAN Methodology

We start with three general problems of using the Web for scientific data:

1. Data sets are typically too large to be processed by the typical Open Web Platform (OWP) implementation as one large Web resource.
2. HTML table markup lacks the constructs to convey all the information coded within typical tabular data sets.
3. A naming strategy must be developed so that information is usable on the Web such that it can be both identified and easily retrievable by a common mechanism (e.g. over HTTP GET requests).

In solving these problems, it is essential that we return to the principles of the Web where naming (URIs) is used to access the data set, common formats are used, and the size of the representation returned when the URI is accessed work together to meet the needs of a user on the Web. We want to avoid the pitfalls of other attempts to disseminate scientific data where large packaged archives (e.g. compressed tar files) of data files in a variety of formats are distributed and meant for offline processing. Instead, we want to expose this data in "Web sized" portions that are usable within the OWP via a new methodology.

The basic tenets of the PAN methodology are:

1. Partition the data set along properties inherent in the data (e.g. time, geospatial coordinates, etc.) into reasonable sized subsets suitable to Web applications.
2. Annotate the data according to some ontology and encode in a common syntax (HTML) using RDFa.
3. Name each data partition with a unique URI using a consistent naming scheme that can be traced back to your partitioning scheme from (1).

In the ontology, shown in Figure 2, there are classes for the basic structures: `DataCollection`, `DataSet`, `Partition`, and `DataView`. At the core of the ontology are data sets that have partitions that contain items typed as some kind of `DataView` instance. These are currently limited to tabular data (`Table`) or an labeled table (`LabeledTable`) in the PAN ontology but other item types are possible and allowed (e.g. non-tabular data representations). A `Table` instance is used to describe typical tabular data and a `LabeledTable` is a matrix of a single kind of data with labeled rows and columns.

The main complexity comes in navigating between the data set and its partitions. While there is a property called `partition` on the `DataSet` class that contains the subject URI of a partition, it is impractical to just enumerate all the partitions. In fact, in certain cases, this may be nearly impossible due to very large or countably infinite number of partitions.

Instead, partitions are discovered by following links within their Web representations as shown in Figure 3. A user can navigate from summaries to partitions and

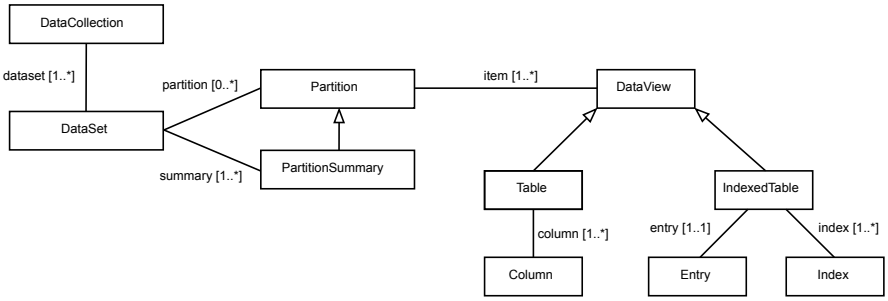


Figure 2. Pantabular Ontology Class Structure

then from partition to partition by links typed by relationship and other properties. As links are traversed and resources are processed, new data set partitions are discovered and expanding the knowledge the application has of the available data.

A typical discovery process starts by examining the `summary` property of a data set to find the URI of the partition summary. The resource returned provides a summary of available partitions (e.g. a `LabeledTable` data item) with links to each of these partitions. For example, a partition summary of a set of weather reports might contain a set of counts of weather reports within the last hour with links to partitions defined by specific quadrangles.

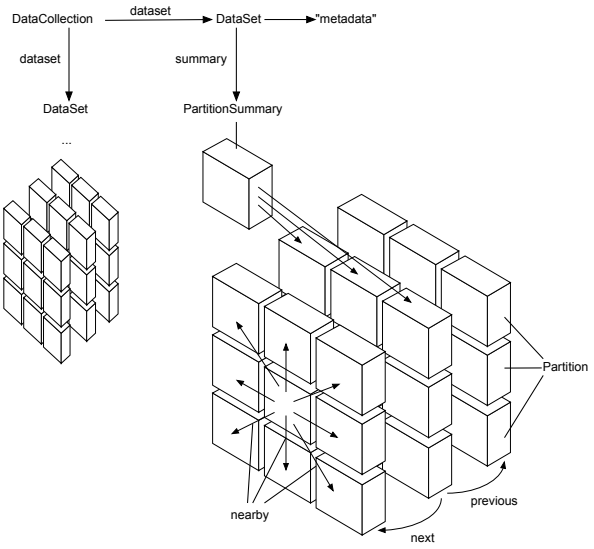


Figure 3. Pantabular Resource Structure

Once an application navigates from the summary to a particular partition, the partition can contain certain properties such as `nearby`, `next`, or `previous`. These properties link to related partitions with some adjacent property range. For example, "next" might be the following time period (future) while "nearby" might be an adjacent quadrangle in the same time period.

By properly providing a starting summary and links within each partition, a whole data set can be enumerated by just examining annotation graph and the links contained within it. The property dimensions (e.g. time) along which the link extends should be available within the current partition to allow applications to make decisions about whether a new partition resource should be retrieved. By doing so, an application can test whether the currently visited set of partitions satisfy a particular query by its inclusion in a particular property range and possibly avoid addition partition retrievals.

When a data set is multidimensional, it may be partitioned by multiple property ranges at the same time as shown in Figure 4. Each partition is shown as blocks in the diagram and can be addressed by a specific range for each of the partitioned properties. For example, in the diagram the basic facets are latitude, longitude, and observation time and each block has a specific range for each of them.

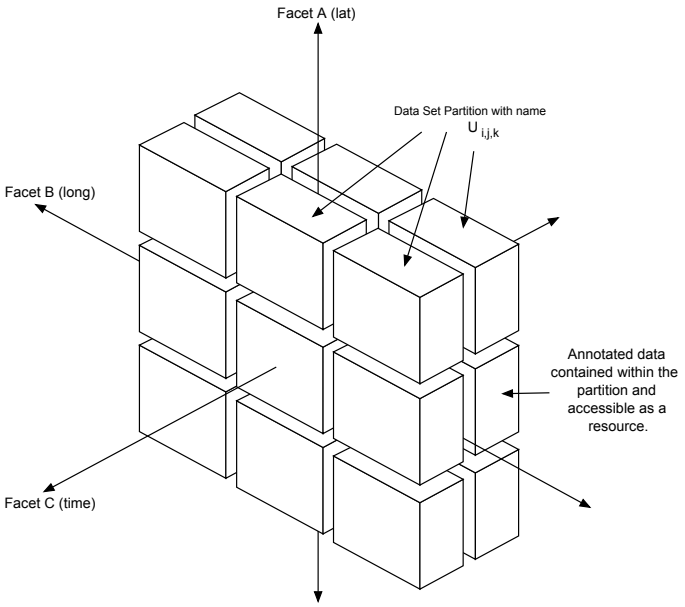


Figure 4. PAN Methodology Partitioning

While any data set is likely to contain a finite amount of data, it should be noted that the set of partitions can be open-ended (infinite). That is, particular properties (e.g. observation time) whose value space is infinite lend themselves to an possibly infinite number of partitions. At any point in time, the actual set of partitions that contain data is likely to be finite and the data accessible for that particular partition is stable, but the addressable space of partitions is infinite.

```
<table typeof="Table">
<thead>
<tr>
...
  <th property="column" typeof="Column">
     <span property="title">Temperature</span>
     <span property="property" resource="w:airTemperature"/>
     <span property="valueSpace" typeof="ValueDescription">
        (°<span property="symbol">C</span>)
        <span property="datatype" resource="xsd:double"/>
      <span property="quantity" resource="quantity:ThermodynamicTemperature"/>
        <span property="unit" resource="unit:DegreeCelsius"/>
     </span>
  </th>
...
</tr>
</thead>
<tbody>
   <tr>
...
     <td>22.2</td>
...
   </tr>
```

Figure 5. Data Partition Table Markup

The actual data of a data set is accessed by accessing partitions. Within each partition (blocks in the diagram) is the subset of data (rows) from the data set that has properties values in the property ranges. A consuming application accesses that data by a fixed URI assigned via some encoding rules for translating the property ranges into a URI. Accessing that URI returns a representation that contains the data set subset in a common syntax such as HTML with RDFa annotations.

Crucial to the exchange of data, each data partition of tabular data is encoded as an HTML table with RDFa annotations for each column header. Each row of data is just a sequence of table cells containing plain literals (see Figure 5). An application can interpret the table cell by using the associated annotations from the column header. This technique minimizes the markup and annotation necessary for tabular data without loss of specificity but possibly increases the processing by applications that expect everything as RDFa properties.

5. Mesonet.info Example

One such data set is weather observations from the *Citizen Weather Observation Program* (CWOP) [6], which is a loosely associated network of automated weather stations hosted by citizens, local governments, and businesses. These weather stations

provide their data through a peer-to-peer network that communicates over the APRS-IS protocol [7]. This line-oriented protocol can be received from servers that aggregate the feeds of weather and position reports from all the various weather stations and an example is shown in Figure 6.

The APRS feed contains coded weather reports accord to the US National Weather Service NWS APRS standard [8] that also contain date/time and location information. To make this format easier to handle, the feed is turned into XML and stored into a MarkLogic database [9] via XProc [10]. The system receives more than 74,000 weather reports per hour from more than 10,000 weather stations unevenly distributed throughout the world. On average, over 53.6 million weather reports per month generates more than 12GB of XML data.

```
DW3904>APRS,TCPXX*,qAX,CWOP:@090158z5132.18N/00043.53W_061/▶
000g001t030r000p000P000h87b10389L000.DsVP
CW1604>APRS,TCPXX*,qAX,CWOP:@090158z4444.70N/06531.17W_204/▶
004g009t027r000p000P000h80b10204.DsVP
DW6741>APRS,TCPXX*,qAX,CWOP:@090158z3749.55N/08000.08W_296/▶
005g...t036r...p...P008h74b10188.DsVP
DW6916>APRS,TCPXX*,qAX,CWOP:@090158z4310.23N/10818.40W_238/▶
001g002t027r000p000P000h58b10189.DsVP
DW6011>APRS,TCPXX*,qAX,CWOP:@090158z4307.07N/08756.60W_261/▶
002g006t028r000p000P000h55b10249.DsVP
```

Figure 6. Example APRS Feed

As a scientific data set, it has two distinct qualities: geospatial orientation and measurement of observed phenomena. For the weather measurements, there are number of different properties that are necessary to define detail the data so that, for example, "temperature" means "air temperature, over land, measured at a particular elevation." As such, the data set provides more than sufficient examples of the complexity of exchanging data with precisely defined semantics.

Layered over MarkLogic is a multi-tiered system at `mesonet.info` whose architecture is shown in Figure 7 that both stores the data set and provides it via the PAN Methodology. On the left side of the diagram the APRS feed is turned into XML and stored into MarkLogic via an XProc pipeline. The data is then indexed in various ways and made accessible via PAN-enabled Web resources at various URIs via XProc again.

For example, the quadrangle summaries are available at the URI path template `/data/q/{size}/` where the *size* variable is the size of the quadrangle in degrees. Each partition of data is available at the path template `/data/q/{size}/n/{seq}` where the *seq* variable is the *sequence number* of the quadrangle. Both of these URI paths allow at optional start time to be appended; if it is omitted, the agent is redirected to the most current time period.

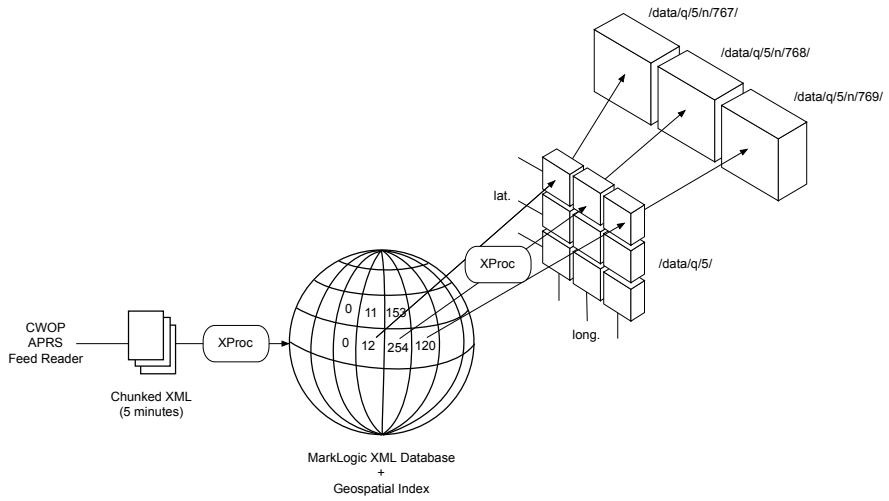


Figure 7. mesonet.info Architecture

A user can navigate from the summary pages, which show the weather report counts for each quadrangle for a specific time period, to specific quadrangles for that same time period. Each quadrangle is just a simple link to the quadrangle data page. As the data partition page is also a regular Web page, it can include useful visualization as shown in Figure 8 where the page demonstrates a dual purpose as both data and a visualization of the weather.

These data access methods implement the PAN methodology and its encoding of the weather in HTML with RDFa annotations. They provide the ability to navigate the CWOP data set both in terms of quadrangles (geospatial) and by time period. The summary provides a simple way to get a snapshot of how much data is available for all the quadrangles for a given time period. Meanwhile, the specific quadrangle data pages provide access to the weather reports.

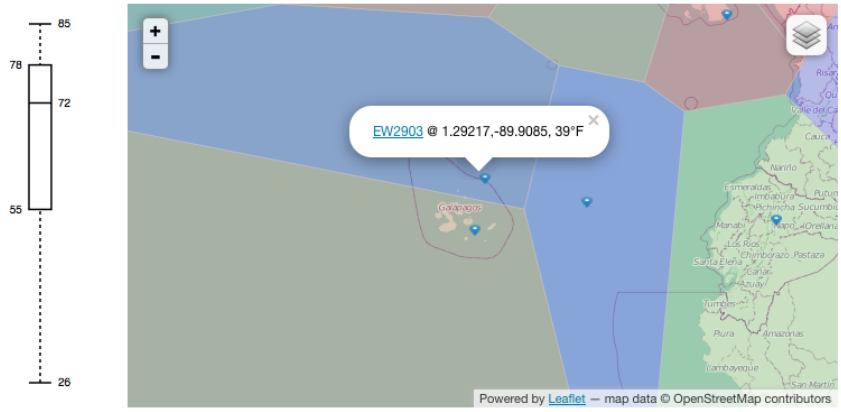
6. Computing on the Open Web Platform

6.1. Data Access Methods

An application task that requires access to data for particular time periods and geospatial regions must first:

1. Choose a reference quadrangle size.
2. Calculate the number of intersecting quadrangles needed for the task's geospatial region.
3. Calculate the number of time period partitions needed for the task's total time period.

While calculating the number of time period partitions necessary is a straight-forward calculation, it might seem like computing the necessary quadrangles is complex.



31 Weather Reports from Received from 2013-11-27T20:30:00Z to 2013-11-27T21:00:00Z (PT30M) within Quadrangle #28 [18 -108 -18 -108 -18 -72 18 -72]

17 18 19
Previous period PT30M @ 2013-11-27T20:00:00Z 27 28 29 Next period PT30M @ 2013-11-27T21:00:00Z
37 38 39

Station	Lattitude	Longitude	Received At	Wind Direction (°)	Wind Speed (km/h)	Wind Gust (km/h)	Temperature (°F)	Temperature (°C)	Humidity (%)	Pressure (Pa)	Ra la ho (c
DW1029	17.24617	-88.77033	2013-11-27T20:46:54Z	313	19.31		72	22.2	71	101740	
DW4785	-0.35817	-78.49567	2013-11-27T20:36:37Z	275	1.61		65	18.3	64	100150	0

Figure 8. Quadrangle Data Partition Page

Fortunately, all that is required is to compute a bounding box and then compute the *sequence number* for each corner of the box. The corners give the extreme values for the sequence numbers. As sequence numbers simply enumerate quadrangles over the whole reference ellipsoid (i.e. the earth's surface), the quadrangles that cover the bounding box can just be listed by counting from a given reference quadrangle in the upper right and the using the number of quadrangles that tile the circumference (which is constant) to move to the next row. An example of this process is shown in Figure 9.

Given the naming choices implemented for `mesonet.info`, requesting the data for a polygon becomes a simple process of generating a sequence of URIs from the *sequence numbers* found using the *bounding box algorithm*. Each URI can be directly constructed from the set of sequence numbers allowing all the data can be retrieved in serial or parallel in as much as the OWP platform implementation allows.

This gives us an efficient algorithm for retrieving data over both a geospatial region and time period as shown in Figure 10. Rather than computing the exact number of time partitions and quadrangles necessary as a complete cross product, an application can backtrack from the end of the time period. Requests are made

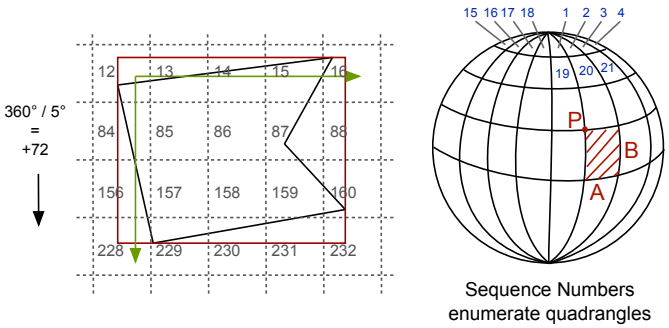


Figure 9. Bounding Box Algorithm for Sequence Numbers

for just the quadrangle data partitions that contain the end of the time period for each quadrangle identified by the *Bounding Box Algorithm*. From the annotation graph for each retrieved quadrangle data partition, the previous link is selected and if the time period for that newly discovered partition is within the desired range, the link is queued for retrieval. This process continues until all the necessary partitions are visited.

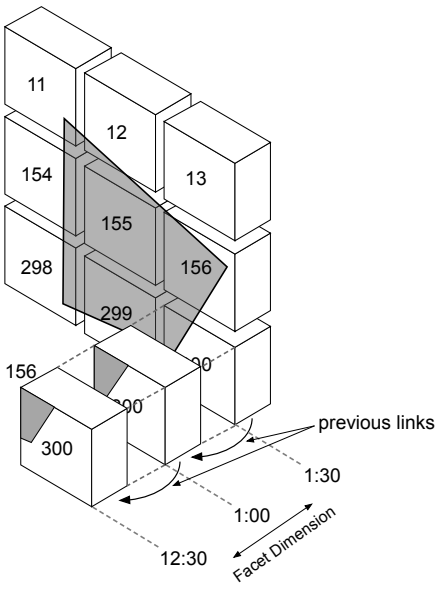


Figure 10. Backtracking Algorithm

6.2. Data Navigation via APIs

There are number of APIs to help navigate both HTML and RDFa. First, the DOM has long defined interfaces to HTML elements [11] and defines specific interfaces to HTML tables. Every table row and cell can easily be enumerated via a script via a matrix-like API (`rows` and `cells` properties). The advantage being that any row or column spans, etc. are computed by the browser before the API is presented to the consuming script.

More importantly, there is a document-oriented API for RDFa [12] that is published as a W3C Note and defined by the W3C Working Group that published RDFa 1.1. This API provides the ability to access the RDFa annotations directly from the document and to navigate back and forth between the document and specific properties. The graph can be accessed and processed directly or in parts as constructs are found within the document.

A conforming RDFa 1.1 processor and the RDFa API was implemented, along with some extensions, in Green Turtle [13] as an RDFa processor for browsers. While browsers may provide native RDFa implementations within the OWP in the future, existing Websites can enable RDFa by simply including the Green Turtle script. Other processing environments may have similar capabilities and also enable RDFa via Green Turtle or other implementations.

A typical application interaction starts with accessing a data partition's data using the API and is shown in Figure 11. While more complex partitions may contain many items, `mesonet.info` only has one and so finding the table of data within the Web page is simple. In the example, the script (1) finds the *container element*, (2) uses the element returned to get a *list of subject URIs*, and (3) uses the subject URI to find the *table element*. As such, the script navigates from the document into the annotation graph and then back to the document.

```
// (1) Find the element that holds the partition
var datasets = document.getElementsByType("pan:Partition");

// (2) Use the subject to find the partition's item subjects
var items = document.data.getValues(datasets[0].data.id,"pan:item");

// (3) Access the first item (a table)
var table = document.getElementsBySubject(items[0])[0];
```

Figure 11. Accessing a Table via the API

Once the tabular data is located, an application has a direct reference to the table element containing the data and can enumerate the table rows and cells. To understand which columns contain the desired data, the script must first locate the columns by their annotations. For example, locating the first air temperature column is shown in Figure 12.

```
var columns = document.data.getValues(table.data.id,"pan:column");
var column = null;  // A variable to hold the subject URI.

for (var i=0; !column && i<columns.length; i++) {
  // Find the column labeled with the air temperature property
  if (document.data.getValues(columns[i],"pan:property")
        .indexOf("http://mesonet.info/airTemperature")>=0) {
     column = columns[i];
  }
}

// Find the index by finding the column element by subject URI.
var index = document.getElementsBySubject(column)[0].cellIndex;
```

Figure 12. Finding Air Temperature

As the API currently lacks more complex query capabilities, finding joint values such as "air temperature whose unit is Celsius" is more complicated. The PAN ontology attaches the unit to the `valueSpace` property's subject and so the iteration must navigate an additional set of properties (e.g. `unit` or `quantity`) in the annotation graph. The resulting script is only slightly more complicated.

In both examples, the table column can be located in each row by using the `cellIndex` property. This property provides the array index of the table cell within each `rows` property array value. An application can now simply enumerate the table rows, skipping the column definitions, and pick out the specific column of data by accessing the value via `table.rows[...].cells[index]`.

The result is that an application has a wide range of abilities to consume data within the table. From the column definition, the datatype and other properties of the value space can be extracted and dynamic interpretation of the textual value can be applied. On the other end, once a column is recognized by label, the cell values can be used directly as textual content without conversion into a specific datatype. This flexibility enables the ability to scale up or down in terms of its complexity for using cell values.

6.3. Computing on the OWP

As data sets expressed via the PAN methodology can be accessed and traversed within the OWP using a combination of RDFa and HTML APIs, the immediate question that arises is what can be accomplished with that ability. The browser within the OWP has become a surprisingly solid platform for deploying applications with a myriad of advanced features and capabilities. The idea of directly using the OWP and computing against data is captivating. The question remains as to how far this platform can be practically pushed before other methods are more tractable.

6.3.1. Map / Reduce

Given the structure that results in regularly partitioning data sets, it is straight forward to define a map / reduce algorithm for use over data sets. For geospatial data, computing over geospatial regions is also enhanced by the use of quadrangles and sequence numbers within the PAN methodology. The process becomes a task of mapping user functions over Web resources (data partitions) and then apply the reduction.

We start with the input of a geospatial region G , a time period $T(t_s, t_e)$ and a set of facet types F (i.e. the URIs of types of columns - temperature, etc.). The process produces an output O is as follows:

1. Let $R = \varnothing$.
2. Calculate the quadrangle sequence numbers S from the region G .
3. For each sequence number in S , generate a URI for the latest partition which contains t_e and add to the queue Q .
4. While Q is not empty:
   a. Remove U from Q and request a representation W of U .
   b. Locate and harvest the columns $C(F) \in R$ from W via the RDFa annotations and unmarshall as necessary into an array of data D.
   c. Apply the user's map function: $\text{map}(D) \rightarrow M_D$ and add the result to R
   d. Locate a link to the previous partition. If t_e is after the time period start for the partition, add the URI to Q .
5. Apply the user's reduce function: $\text{reduce}(R) \rightarrow O$.

An example of a map / reduce process has been implemented in about 450 lines of JavaScript [14] code (uncompressed) without including the RDFa implementation that may not be provided directly by the browser. The implementation has been limited to rectangular regions to simplify generation of the starting quadrangle sequence numbers and testing of membership of returned data locations. The user only needs to provide the region, time period, columns they are interested in processing, and their map / reduce functions. A simple use of this is shown in Figure 13for calculating average temperature.

```
// Calculate Average Temperature

var mr = new MapReduce();

// The date/time as of now.
var endDateTime = new Date();
// The date/time as of one hour ago.
var startDateTime = new Date(endDateTime.getTime()-60*60*1000);

mr.init("http://www.mesonet.info/");
mr.columns.push({ uri: "http://mesonet.info/airTemperature",
                  unit: "http://qudt.org/vocab/unit#DegreeFahrenheit" });

// Calculates the average per quadrangle data partition
mr.mapper = function(data) {
   var total = 0;
   var count = 0;
   for (var i=0; i<data.length; i++) {
      total += data[i][0];
      count++;
   }
   return total / count;
}

// Calculates the average over all quadrangles
mr.reducer = function(data) {
  var total = 0;
  for (var i=0; i<data.length; i++) {
     total += data[i];
  }
  return total / data.length;
}
mr.apply(
   [38,-123,37,-122],         // geospatial region
   startDateTime,endDateTime, // time period
   2.5                        // quadrangle size
);
```

Figure 13. Using Map / Reduce to Calculate Average Temperature

6.3.2. Barnes Interpolation

Barnes Interpolation [15] is the interpolation of data points from a set of measurements across a two-dimensional surface that was originally developed specifically for weather forecasting. It produces the renderings of temperature, wind, pressure, etc. that are commonly expected (e.g. a colored gradient) from discrete measurements

such as those from weather stations. As the interpolation is typically for a geospatial region over time, this process is a perfect further application the Map / Reduce implementation from the previous section.

The interpolation process starts with a weighted average and then calculates a refinement using all the grid points:

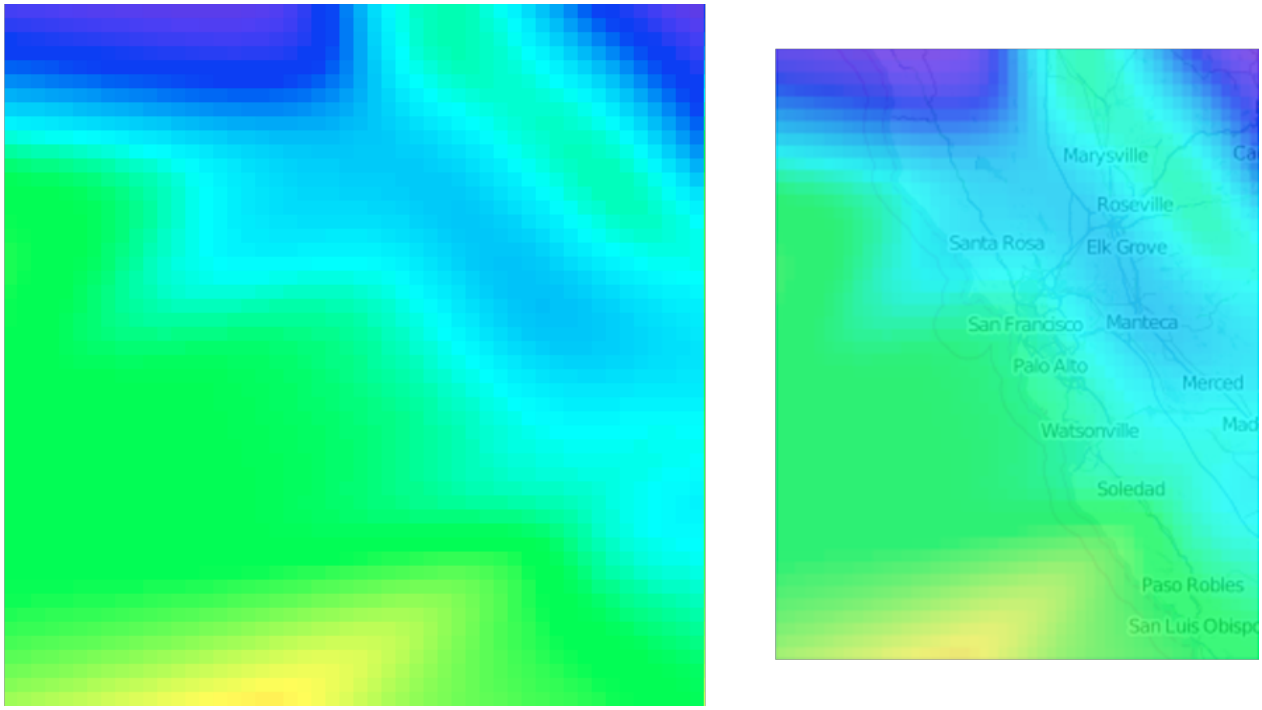
$$g_k^0(x,\ y)\ = \frac{\sum_i^n w_i o_i}{\sum_i^n w_i},\quad g_k^1(x,\ y) = g_k^0(x,\ y) + \frac{\sum_i^n w_i\left(o_i - g_i^0(x,\ y)\right)}{\sum_i^n w_i},\quad \ldots\ ,$$
$$g_k^{n+1}(x,\ y) = g_k^n(x,\ y) + \frac{\sum_i^n w_i\left(o_i - g_i^n(x,\ y)\right)}{\sum_i^n w_i} \tag{1}$$

where:

- weight for each observed value is $w_i = \exp\left(\frac{-d_i^2}{L^2 C}\right)$,
- o_i is the observed value,
- d_i is the distance from the grid point to observation point (e.g. in kilometers),
- L is the length scale relative to the observed phenomena (e.g. 111.3km) and C is a convergence factor (e.g 1 for the initial pass and 0.3 afterward) [16].

The OWP provides all the basics elements for computing and visualizing an interpolation surface for weather data. As an experiment, an implementation of Barnes Interpolation was developed as a script that takes as input a rectangular geospatial region, a time period, and a quadrangle size. The Map / Reduce process described previously first calculates averages over the necessary quadrangle data partitions to produce the interpolation surface after first calculating an average temperature for each station within the geospatial region.

An example visualization of the output of the process is shown in Figure 14 for the region defined by the two locations (40°,-125°) and (35°,-120°) for an half hour of data where on the left is the raw colored grid and on the right is an overlay of a map. To access the necessary data with 2.5° quadrangles and 30 minute durations, 9 quadrangles (2975, 2976, 2977, 3119, 3120, 3121, 3263, 3264, 3265) are used where 9 to 18 partitions are retrieved over the Web depending on the time period chosen. The example took 2.5 seconds to render of which 1.7 seconds was the map / reduce process over the data and 870 milliseconds was the interpolation and rendering. Accessing the data over the Web required 66% of the processing time where for each request that time was spent on receiving the response (query, formulation, and transport). These transport times may vary and the example was computed over a network with relatively high latency which can add several seconds to the measured time (total times vary from 2.5 seconds to over 4 seconds).



A warm winter morning (2014-01-17 8:00:00 PST) in the San Francisco Bay Area.

Figure 14. Example Barnes Interpolation for the San Francisco Bay Area

7. Conclusion

One cannot underestimate the value of "view source" in the development of the Web. The ability to extend this to both scientific data and its use within computations allows enables the "copy and modify" model that has allow good constructs to go viral on the Web. This only works if the OWP platform can become a direct participant in scientific endeavors.

The PAN Methodology was born from a desire to address this for both the professional and citizen scientist while providing access to any interested party. The use of the OWP as a basis allows the methodology to scale down to very small data sets and this enables both medium and small scale science to operate on the Web at a lower cost with increased network effects. At the same time, compatibility is maintained by allowing applications to crawl and harvest data via the annotations.

Current "open data" trends advocates exposing data on the "Semantic Web" and for providing a complex stack of semantic-enabled (triple aware) technologies. These trends ignore the need to enable data usage for simple consumers and the PAN methodology provides a useful middle ground between complex data representations or services and simple expressions of partitioned tabular data. By doing so, PAN enables the OWP platform as an active participant; all that is required is a Web browser for real work to be accomplished.

Bibliography

[1] *Architecture of the World Wide Web, Volume One*, 2004-12, W3C Ian Jacobs and Norman Walsh http://www.w3.org/TR/webarch/

[2] *The future of applications: W3C TAG perspectives*, Henry S. Thompson, School of Informatics,University of Edinburgh, 2011-03-28, *W3C Technical Architecture Group* http://www.w3.org/2001/tag/doc/IAB_Prague_2011_slides.html

[3] *Deep Web*, 2012-05, Wikipedia http://en.wikipedia.org/wiki/Deep_web

[4] *International Virtual Observatory Alliance* http://www.ivoa.net/

[5] *VOTable Format Description, Version 1.2*, 2009-11-30, IVOA, Francois Ochsenbein, Roy Williams http://www.ivoa.net/documents/VOTable/20091130/

[6] *Citizen Weather Observation Program* http://www.wxqa.com/

[7] *Automatic Packet Reporting System-Internet Service*, Bob Bruninga http://www.aprs-is.net/

[8] *Automatic Position Reporting System; APRS Protocol Reference, Protocol Version 1.0*, Ian Wade, 2000-08-29 http://www.aprs.org/doc/APRS101.PDF

[9] *MarkLogic* http://developer.marklogic.com/inside-marklogic

[10] *XProc: An XML Pipeline Language*, Norman Walsh, Alex Milowski, and Henry S. Thompson, W3C, 2010-05-11 http://www.w3.org/TR/xproc/

[11] *Document Object Model (DOM) Level 2 HTML Specification; version 1.0*, Johnny Stenback, Philippe Le Hégaret, and Arnaud Le Hors, W3C, 2003-01-09 http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/

[12] *RDFa API*, Nathan Rixham, Mark Birbeck, and Ivan Herman, W3C, 2012-07-5 http://www.w3.org/TR/rdfa-api/

[13] *Green Turtle*, R. Alexander Milowski https://code.google.com/p/green-turtle/

[14] *ECMAScript Language Specification, 5th Edition*, ECMA International, 2011-06 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[15] *A Technique for Maximizing Details in Numerical Weather Map Analysis* Stanley L. Barnes *Journal of Applied Meteorology*, American Meteorological Society, vol 3, issue 4, pp. 396-409, 1964-08-01 http://dx.doi.org/10.1175/1520-0450(1964)003<0396:ATFMDI>2.0.CO;2

[16] *Barnes Analysis for Surface Interpolation*, Martin Davis http://lin-ear-th-inking.blogspot.com/2012/02/barnes-analysis-for-surface.html

RESTful API Description Language (RADL)

Hypermedia-driven API design

Jonathan Robie
<jonathan.robie@emc.com>

Rémon Sinnema
<remon.sinnema@emc.com>

Erik Wilde
<erik.wilde@emc.com>

Abstract

In a REST API, the server provides options to a client in the form of hypermedia links in documents, and the main thing a client needs to know is how to locate and use these links in order to use the API. The main job of a REST API description is to provide this information to the client in the context of media type descriptions. Unfortunately, most REST service description languages and design methodologies focus on other concerns instead.

RESTful API Description Language (RADL) is an XML vocabulary for describing Hypermedia-driven RESTful APIs. The APIs it describes may use any media type, in XML, JSON, HTML, or any other format. The structure of a RADL description is based on media types, including the documents associated with a media type, links found in these documents, and the interfaces associated with these links.

RADL can be used as a specification language or as run-time metadata to describe a service.

This is an article that will be presented at XML Prague 2014, based on a pre-release version of RADL. The most recent version of the RADL specification is always available at http://github.com/restful-api-description-language[1].

Keywords: REST, XML, authoring, metadata

1. Introduction

Web APIs are critical to the business strategy of many companies, and vital to the way users use information on the Web. Companies like eBay, Amazon, Salesforce,

[1] https://github.com/restful-api-description-language

and Google provide valuable services via Web APIs, and developers are using these APIs together with other data sources to create new kinds of applications that run on a variety of devices and environments. Some analysts are now writing about the API Economy, and even people who have never programmed may know why Web APIs are important.

Web service providers may have little knowledge of the clients that use them, but they need to ensure that these clients can continue to run as they evolve their services. Web services may need to support large numbers of users, so they must be scalable. This means that Web API providers must design their APIs to support evolution and scalability in distributed systems. This is not easy.

Fortunately, this is precisely the problem that REST (Representational State Transfer) was designed to solve. Unfortunately, designing and documenting RESTful APIs is still too much of a black art, causing many difficulties for the average developer. An XML vocabulary that supports the design process is extremely helpful for designing a RESTful API and for teaching RESTful API design, and is well suited to writing documentation.

In a REST API, the server provides options to a client in the form of hypermedia links in documents, and the main thing a client needs to know is how to locate and use these links in order to use the API. The main job of a REST API description is to provide this information to the client in the context of media type descriptions. This is the main focus of RADL. Unfortunately, most REST service description languages and design methodologies focus on other concerns instead, and most APIs that claim to be REST APIs ignore the principle that REST APIs must be hypertext-driven, publishing lists of URIs and conventions for using those URIs instead. Roy Fielding, the inventor of REST, has frequently pointed out that systems like these are not RESTful, and lead to tightly coupled systems.

RESTful API Description Language (RADL) is an XML vocabulary created to support RESTful Web API design. RADL is hypertext-driven, designed to support tooling and design methdologies for purist REST, and to make it easier to teach REST design. RADL can be used to create both documentation and runtime metadata, transforming to formats like HTML and JSON to suit various environments. The design of RADL is focused on the media type, in keeping with this famous quote by Roy Fielding REST APIs must be hypertext-driven:

> *Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type.*

Unlike most other description languages for REST APIs, RADL is hypertext-driven. The structure of RADL is driven by media types. To support implementation, RADL also provides resources, which associate interfaces with URI patterns. Resource descriptions are for implementation only, and are not generally provided to clients. RADL also provides support for authentication, which is orthogonal to the REST service per se.

2. The REST Architectural Style

REST Is an Architectural Style that is formally defined by the following constraints:

1. Client/Server
2. Stateless
3. Cache
4. Uniform interface
5. Layered system
6. Code-on-demand (optional)

The client/server constraint demands that we divide our system in multiple components, since monolithic applications cannot be made to scale. Each component can be further divided as needed. For instance, the client can be a browser and the server a web application. We can then further divide the web application into a web server for the CPU-intensive processing and a database server for I/O-intensive processing. Distributing capabilities over multiple components gives us the opportunity to give each the resources it needs.

The layered system constraint additionally requires that the client cannot see beyond its immediate server. This allows us to change our server landscape without breaking the client. We could, for example, insert proxy servers to aid with scaling.

We can further improve scalability by moving some of the processing from the server to the client. The stateless constraint makes us move data to the client so that the server need not maintain application state. Here, we must distinguish two different kinds of state: the server will maintain resource state, so that interesting data is available from multiple clients, but the client is responsible for application state. The client knows what goal it wants to achieve and can easily remember where along the path to that goal it is.

The code-on-demand constraint additionally pushes processing to the client. We see this when web servers send JavaScript to the browser, which then executes it. This optional constraint is very common on the Web, but less common in RESTful APIs.

The cache constraint also helps with scalability. The server indicates how long data that it sends is valid, so that the client does not have to keep asking for it. If clients do so anyway, we can insert caching proxy servers to keep our servers available for useful work.

The uniform interface constraint is the one that really distinguishes REST from other approaches. Anybody who has played with Legos knows the power of a universal interface. In REST over HTTP, we capture all actions with the standard HTTP methods.

3. REST APIs must be hypertext-driven

The previous section describes the REST architectural style, this section describes REST APIs. A REST client uses a service much as a human user uses a set of web pages. The client starts out by retrieving an initial document from a known URI. Each document represents a client application state, and the links in the document represent the choices available to the client in that state. The server provides options to a REST client in the form of hypermedia links that it places in these documents, just as it provides options to a human user using links. The client can use these links to proceed to another state, just as a human user might click on a link.

But a REST client can only do this if it knows how to interpret the documents it receives, and how to locate and use the links that it encounters. The main job of a REST API description is to provide this information to the client.

Roy Fielding described this in REST APIs must be hypertext-driven, where he says that REST APIs must be hypertext-driven and describes how a client uses this kind of API:

> *A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]*

In the same post, Fielding says that a REST API description must explain these things in the context of the media type for the document.

> *Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type.*

Internet media types are used in many different ways on the Web. Email clients use media types to identify the format of attached files, web browsers use them to determine how to display or process HTML, images, video, audio, scripts, and other types of data appropriately. Search engines and feed readers use media types to correctly index or distribute information. Most Web programmers are familiar with media types like HTML (`text/html`), JPEG (`image/jpeg`), Atom (`application/atom+xml`, JavaScript (`application/javascript`), JSON (`application/json`), XML (`application/xml`), MP4 (`audio/mp4`), etc. Each of these media types is associated with a specification that describes the media type's format and the semantics asso-

ciated with the format. The specification for a media type is called a media type description.

For services that use structured information, the media type is generally described in terms of structured formats like XML or JSON. They can be generic, like XML, or tailored to a specific use, like XACML (`application/xacml+xml`) for access control. They can even be specific to a single service, such as Documentum (`application/vnd.emc.documentum+xml`). However, media types can also be unstructured, like JPEG (`image/jpeg`). A media type may define a single kind of document or more than one kind of document. The Atom Syndication Format defines feed and entry documents, for example. A media type used in a complex custom RESTful service might define many more kinds of documents.

In a REST API, the media type description identifies the documents associated with a media type, links found in these documents, and the interfaces associated with these links. The interface for a link includes the HTTP methods that can be applied to that link, together with any URI parameters, headers, and request bodies used when making these requests. These are the things that a REST client must understand in order to use a REST API, and their interpretation depends on the documents in which they are found, so most of the documentation for a REST API belongs in these media type descriptions.

Unfortunately, most APIs that claim to be RESTful completely ignore this paradigm, as Cory House points out in How RESTful is Your API?.

> *Here's the unicorn. Virtually none of today's APIs honor this. To prevent tight coupling between the client and the service, truly RESTful APIs provide a discovery based API. Each call provides a reference to related calls. This allows the API to be highly evolvable because it avoids creating a coupling between the client and the server. This aspect is nearly universally ignored by today's popular APIs, as made evident by the common pattern of publishing a list of URIs.*

Most tools that claim to support REST actually use a tightly-coupled approach that documents a static list of URIs and the interface associated with each URI. The information provided at runtime rarely includes descriptions of the links available in the documents that a server provides to a client, so clients do not have the information they need to discover and use these links. As a result, most Web APIs borrow bits and pieces from Roy Fielding's REST vision, but violate key aspects such as HTTP verb semantics, correct use of HTTP status codes, use of media types that identify the semantics of documents that are served, discovery of links in hypermedia, and loose coupling. Some have started to use the term "Pragmatic REST" to describe these non-REST APIs, implying that REST is not practical.

We believe that REST is both practical and simple, and the benefits of loose coupling are important for scalability and for supporting clients as servers evolve. However, we also believe that the REST community has not done well at teaching REST design and providing the tooling and run-time metadata that developers

need. RADL provides a model of a REST API that is useful for teaching purposes and an XML format that is useful for generating client documentation, server specifications, and run-time metadata. We hope that RADL will be used for tools and frameworks that offer the same level of functionality currently available for "pragmatic REST", making it easier for developers to create services that are truly RESTful.

4. RADL - a hypertext-driven REST API description

RADL is a description for hypertext-driven REST APIs, as described in the previous section. Most of the information in a RADL description tells a client how to interpret the documents it receives, and how to locate and use the links that it encounters. This information is provided in media type descriptions. To support well-known media types that are described by specifications, RADL also allows a reference to human-readable documentation instead[2]. A media type can be defined as an extension to an existing media type.

In addition to media types, RADL describes resources, which are needed for implementation but are not part of the client API. Once complete media type descriptions have been done, the resource description for a service is simple and small, associating URI formats with interfaces that are defined in the media type descriptions.

RADL also supports authentication, which is needed by both the client and the server implementation, but is orthogonal to the API per se.

RADL allows the entire service description to be specified in one description The information needed for client documentation, server stub generation, runtime metadata, and testing can then be extracted from the complete description as needed.

The following template illustrates the overall structure of RADL (a RELAX-NG schema can be found in Appendix A):

```
<service name="Outline" xmlns="http://identifiers.emc.com/vocab/radl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/restful-api-description-language/▶
RADL/tree/master/schema"
    radl.xsd">

  <start document-ref="">
    <!-- optional: describes the document associated with the initial URI -->
  </start>

  <link-relations>
    <!-- Link relations used by all media types -->
  </link-relations>
```

[2]If you need run-time metadata for well known media types, you need a RADL description that provides this information, which cannot be automatically extracted from a HTMl document.

```
  <media-types>
    <!-- Media types used by all resources -->
    <media-type>
      <documents>
        <!-- The kinds of documents defined by this media type -->
      </documents>
      <interfaces>
        <!-- The interfaces for links defined by this media type -->
      </interfaces>
    </media-type>
  </media-types>

  <resources>
    <!-- Resources that make up the service -->
    <!-- Resources implement interfaces of media types -->
  </resources>

</service>
```

The document element is service. It contains elements that model the main concepts we discussed in the previous section: link relations, media-types, and resources. Media types contain documents that describe the data format and interfaces that describe the semantics.

Now let us provide a concrete example, taken from chapter 5 of the book RESTful Web Services. This chapter deals with the read-only aspects of a service that provides information about places. To get an overview of the service we are describing, let us look at the client API documentation for this interface, generated from the RADL description that we describe in the rest of this section.

This example illustrates some important aspects of a RADL description:

- A service is defined primarily by its media types.
- A media type is defined primarily by its documents.
- A document is defined primarily by the links it contains.
- A link is defined by its link relation[3] and the requests and responses supported by the link.
- In the descriptions of links, each link relation [4] / HTTP method pair identifies a request and the associated response.
- A request is is defined by the conventions associated with applying the HTTP method to the given link, such as URI parameters, HTTP headers, and documents used in the request body (for PUT, POST, and PATCH requests).

[3] A link type can also be used, if a link does not have a link relation.
[4] or link type

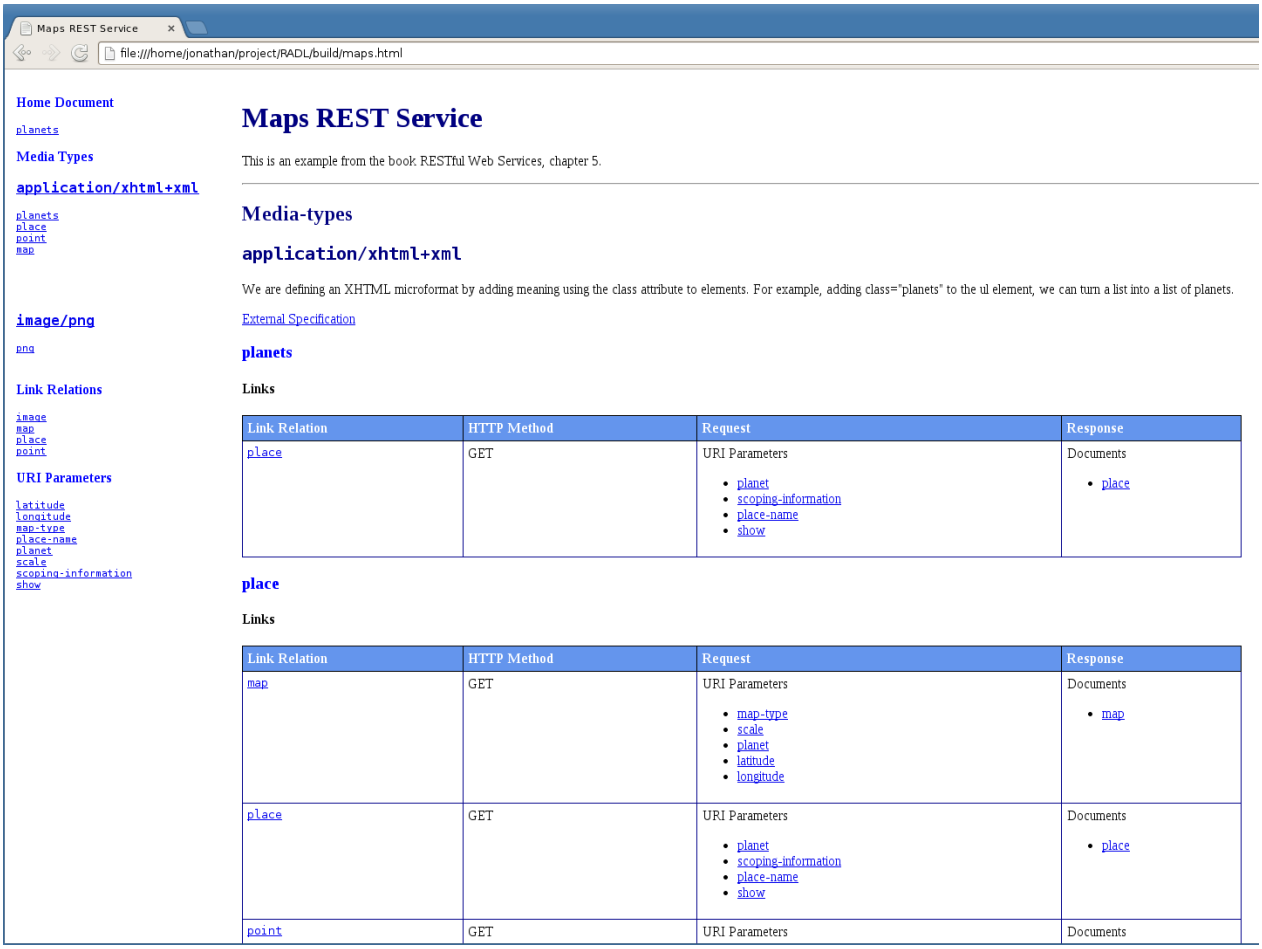


Figure 1. HTML client documentation generated from a RADL description

- A response is defined by an HTTP method, the kinds of documents that can be returned in the response, and any headers or status codes that need to be documented for the response.

Now we will see how these things are represented in RADL markup. Here are the first few lines of the RADL description:

```
<service name="Maps" xmlns="http://identifiers.emc.com/vocab/radl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://identifiers.emc.com/vocab/radl radl.xsd"
    xmlns:html="http://www.w3.org/1999/xhtml/">

  <documentation>
    This is an example based on chapter 5 of the book
    <ref uri="http://shop.oreilly.com/product/9780596529260.do">RESTful
      Web Services</ref>.
  </documentation>

  <start interface-ref="int-planets" />
```

This fragment shows how we can add documentation to RADL. We support different documentation modules. Our TechPubs department uses DocBook, for instance, but in this paper we will only use HTML. To keep the RADL document well-formed XML, we will use XHTML and we declare its namespace. We can use all of HTML's elements and additionally we can use the `ref` element to refer to things. We added `ref`so that we could refer to RADL concepts, like link relations, and more easily process those references. Working with only generic `a` elements would be awkward.

The fragment also contains a `start` element, an optional element that identifies the interface associated with the initial URI for the service. Note that the initial URI is not part of the RADL description - the same service can be deployed from many different locations, and is not part of the API per se.

Here are the link relations we use in this service:

```
<link-relations>
  <link-relation id="rel-place" name="place">
    <documentation>
      The target resource is a place related to the current resource.
    </documentation>
  </link-relation>
  <link-relation id="rel-point" name="point">
    <documentation>
      The target resource is a point on a planet related to
      the current resource.
    </documentation>
  </link-relation>
  <link-relation id="rel-map" name="map">
    <documentation>
      The target resource is a map related to the current resource.
    </documentation>
  </link-relation>
  <link-relation id="rel-image" name="image">
    <documentation>
      The target resource is an image related to the current resource.
    </documentation>
  </link-relation>
</link-relations>
```

This example uses only link relations that were invented for this service, but you could add more generic ones, like `self` as well.

There is no defined model for naming link relations. All the generic ones like `self` use simple names, and they are registered by IANA in a flat namespace. However, to avoid name collisions, you may want to use URIs for your own specialized link relations. For instance, EMC is moving towards using URIs of the form `http://identifiers.emc.com/linkrel/<name>`.

The next section defines the media types. The first media type used in this service is PNG (`image/png`). It is used for displaying images of maps of places. This media type defines only one kind of document and one interface. Since this chapter of the book deals with a read-only service, all you can do with the images is retrieve them through the GET method.

```
<media-types>
  <media-type id="med-png" name="image/png">
    <description type="html"
                 href="http://www.iana.org/assignments/media-types/image/png"/>
    <documents>
      <document id="doc-png" name="png"/>
    </documents>
    <interfaces>
      <interface id="int-image" name="image">
        <methods>
          <method name="GET">
            <request>
            </request>
            <response>
              <document ref="doc-png"/>
            </response>
          </method>
        </methods>
      </interface>
    </interfaces>
  </media-type>
  !!! SNIP !!!
```

The next media type defines most of the documents used in this service, together wtih their semantics. It is based on XHTML.

```
  <media-type id="med-planets" name="application/xhtml+xml">
    <documentation>
      We define an <ref uri="http://www.w3.org/TR/xhtml11/">XHTML</ref>
      <html:em>microformat</html:em> by adding
      meaning using the <html:code>class</html:code> attribute to elements.
      For example, adding <html:code>class="planets"</html:code> to the
      <html:code>ul</html:code> element, we can
      turn a generic list into a list of planets.
    </documentation>
    <description type="html" href="http://tools.ietf.org/html/rfc3236"/>
    <documents>
      <document id="doc-planets" name="planets">
        <links>
          <link link-relation-ref="rel-place" interface-ref="int-place">
            <documentation>
```

```
          Links of this type are found by looking for
          <html:code>a</html:code> elements with
          <html:code>class="place"</html:code>. Additionally, you can
          find search links to places via the
          <html:code>form</html:code> element with
          <html:code>id="searchPlace"</html:code>.
        </documentation>
      </link>
    </links>
  </document>
  <document id="doc-place" name="place">
    <links>
      <link link-relation-ref="rel-map" interface-ref="int-map">
        <documentation>
          Links of this type are found by looking for
          <html:code>a</html:code> elements with
          <html:code>class="map"</html:code>.
        </documentation>
      </link>
      <link link-relation-ref="rel-point" interface-ref="int-point">
        <documentation>
          Links of this type are found by looking for
          <html:code>a</html:code> elements with different values for
          the <html:code>class</html:code> attribute, like
          <html:code>coordinates</html:code>,
          <html:code>map_nav</html:code>, <html:code>zoom_in</html:code>,
          and <html:code>zoom_out</html:code>.
        </documentation>
      </link>
      <link link-relation-ref="rel-place" interface-ref="int-place">
        <documentation>
          Links of this type are found by looking for
          <html:code>a</html:code> elements with
          <html:code>class="place"</html:code>. Additionally, you can
          find search links to places via the
          <html:code>form</html:code> element with
          <html:code>id="searchPlace"</html:code>.
        </documentation>
      </link>
    </links>
  </document>
  <document id="doc-point" name="point">
    <links>
      <link link-relation-ref="rel-place" interface-ref="int-place">
        <documentation>
          Links of this type are found by looking for
```

```
            <html:code>a</html:code> elements with
            <html:code>class="place"</html:code>. Additionally,
            you can find search links to places via the
            <html:code>form</html:code> element with
            <html:code>id="searchPlace"</html:code>.
          </documentation>
        </link>
        <link link-relation-ref="rel-point" interface-ref="int-point">
          <documentation>
            Links of this type are found by looking for
            <html:code>a</html:code> elements with different values for
            the <html:code>class</html:code> attribute, like
            <html:code>coordinates</html:code>,
            <html:code>map_nav</html:code>, <html:code>zoom_in</html:code>,
            and <html:code>zoom_out</html:code>.
          </documentation>
        </link>
      </links>
    </document>
    <document id="doc-map" name="map">
      <links>
        <link link-relation-ref="rel-image" interface-ref="int-image">
          <documentation>
             Links of this type are found by looking for
             <html:code>img</html:code> elements with
             <html:code>class="map"</html:code>.
          </documentation>
        </link>
        <link link-relation-ref="rel-map" interface-ref="int-map">
          <documentation>
            Links of this type are found by looking for
            <html:code>a</html:code> elements with
            <html:code>class="map"</html:code>.
          </documentation>
        </link>
      </links>
    </document>
  </documents>
  <interfaces>
    <interface id="int-planets" name="planets">
      <methods>
        <method name="GET">
          <response>
            <document ref="doc-planets"/>
          </response>
        </method>
```

```
        </methods>
      </interface>

      <interface id="int-place" name="place">
        <methods>
          <method name="GET">
            <response>
              <document ref="doc-place"/>
            </response>
          </method>
        </methods>
      </interface>

      <interface id="int-point" name="point">
        <methods>
          <method name="GET">
            <response>
              <document ref="doc-point"/>
            </response>
          </method>
        </methods>
      </interface>

      <interface id="int-map" name="map">
        <methods>
          <method name="GET">
            <response>
              <document ref="doc-map"/>
            </response>
          </method>
        </methods>
      </interface>
    </interfaces>
  </media-type>
</media-types>
```

This media type defines several kinds of documents and interfaces. It defines how to get from one document to another via links that are typed using link relations. It also defines how these documents can be processed using interfaces.

In the above definitions, the author could have defined its own media type, say `application/vnd.oreilly.maps+xml`, but decided against that, and extends the `application/xhtml+xml` media type instead. The HTML part of the media type makes it easy to consume the service with generic clients like web browsers. The X part of the media type also makes it possible to consume the service with a special-purpose client. For those types of clients, it would probably have been easier to use a new vocabulary, but then you would loose the browsers. Every service has to

weigh the pros and cons. In the services that EMC provides, general purpose clients can provide very little value, so we create new media types like `application/vnd.emc.documentum+xml` and build special purpose clients instead.

The final section of the RADL document contains the implementation details: the resources. These are not part of the client API, but they are important for servers that implement the API. A resource associates one or more interfaces with a URI or URI Template. URI Templates. [5] Resources implement interfaces. In this example the correspondence is 1:1, so that's not very interesting, but there are cases where it makes sense to let a resource implement more than one interface.

```
  <resources>
    <resource id="res-planets" name="planets">
      <location uri="/"/>
      <interface ref="int-planets"/>
    </resource>

    <resource id="res-place" name="place">
      <location uri-template="/{planet}/[{scoping-information}/▶
][{place-name}]{?show}"/>
      <interface ref="int-place"/>
    </resource>

    <resource id="res-point" name="point">
      <location uri-template="/{planet}/{latitude},{longitude}"/>
      <interface ref="int-point"/>
    </resource>

    <resource id="res-map" name="map">
      <location uri-template="/{map-type}{scale}/{planet}/▶
{latitude},{longitude}"/>
      <interface ref="int-map"/>
    </resource>

    <resource id="res-image" name="image">
      <location uri-template="/{map-type}{scale}/{planet}/images/▶
{latitude},{longitude}.png"/>
      <interface ref="int-image"/>
    </resource>
  </resources>

</service>
```

[5] URI Templates allow URIs to be created using variables, which can be supplied by the client in the REST API. They are not used in this example.

This short example does not show how to handle things like URI parameters, HTTP headers and status codes, or authentication. All of these are possible in RADL.

5. Using RADL Descriptions

The following are some of the major uses we envision for RADL descriptions. We have experience supporting the following uses in our commercial REST APIs using a precursor of RADL:

- Providing a clear, complete model of REST APIs for teaching purposes, whether or not RADL descriptions are actually used as part of the teaching.
- Generating client documentation using HTML or DocBook.
- Providing a standard representation of a service to make it easier to review and to mentor REST API design. We have successfully taught RESTful design to two teams with RSDL using this approach.
- Generating test clients to ensure that a running instance correctly implements the specified interfaces. We have limited experience with this using RSDL.

We do not yet have implementation experience with the following, but we believe they are plausible uses of RADL.

- Providing a clear, complete model of REST APIs for tools that generate RADL descriptions for those who prefer not to author in XML.
- Comparing RADL descriptions to REST annotations to identify discrepancies in those aspects of the API that are actually described by the annotations.
- Definining formats to provide API descriptions for those aspects not described by conventional REST annotations so that complete run-time metadata can be generated in a manner that is always consistent with the current server implementation.
- Providing run-time metadata that can be generated dynamically to allow multiple services to be combined, modify the service based on available permissions or preferences, etc. This metadata may be provided in either XML, JSON, or HTML. For instance, a service might provide run time metadata via the `about` link relation from JSON Home Documents or XML Home Documents.

A. RADL Schema

The RADL schema is a RELAX-NG compact notation schema that allows documentation to be embedded using a separate documentation schema. This appendix shows sample documentation schemas for XHTML and DocBook, then shows the schema for RADL per se.

This schema is current as of the time of writing. For the latest version, see the RADL github repository at https://github.com/restful-api-description-language.

B. Schema for embedded XHTML

```
namespace ns1 = "http://www.w3.org/1999/xhtml/"

documentation = element documentation { inline?, doc-title?, html }
inline = attribute inline { "true" | "false" }
doc-title = element title { text }
html = html-content*
html-content = html-element | text | ref
html-element = element ns1:* { html-attribute*, html-content* }
html-attribute = attribute * { text? }
```

C. Schema for embedded DocBook

```
namespace docbook = "http://docbook.org/ns/docbook"

documentation = element documentation { inline?, doc-title?, docbook }

inline = attribute inline { ( "true" | "false" ) }
doc-title = element title { text }

docbook =  docbook-content*
docbook-content = (docbook-element | text | ref)
docbook-element = element docbook:* { docbook-attribute*, docbook-content* }
docbook-attribute = attribute * { text? }
```

D. RADL schema

```
default namespace radl = "http://identifiers.emc.com/radl"

## Copyright 2014, EMC Corporation
##
## Licensed under the Apache License, Version 2.0 (the "License");
## you may not use this file except in compliance with the License.
## You may obtain a copy of the License at
##
##    http://www.apache.org/licenses/LICENSE-2.0
##
## Unless required by applicable law or agreed to in writing, software
## distributed under the License is distributed on an "AS IS" BASIS,
## WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
## See the License for the specific language governing permissions and
## limitations under the License.


start = service
include "ref.rnc"
include "documentation.rnc"
service =
  element service {
    id?,
    name,
    documentation?,
    service-start?,
    link-relations?,
    link-types?,
    service-conventions?,
    media-types?,
    resources?,
    authentication?
  }
# Generic definitions
id = attribute id { xsd:ID }
idref = attribute ref { xsd:IDREF }
title = element title { text }
name = attribute name { text }
href = attribute href { xsd:anyURI }
ref-attribute = attribute ref { xsd:IDREF }
foreign-element =
  element * - radl:* { any-attribute*, (foreign-element* | text)* }
any-attribute = attribute * { text? }
public = attribute public { "true" }
status = implementation-status?, design-status?
implementation-status =
  attribute implementation-status {
    "future" | "assigned" | "poc" | "partial" | "complete" | "passed"
  }
design-status =
  attribute design-status {
    "future" | "assigned" | "poc" | "partial" | "complete" | "approved"
  }
service-start = element start { href?, document-ref, identity-provider-ref? }
identity-provider-ref = attribute identity-provider-ref { xsd:IDREF }
link-relations =
  element link-relations { documentation?, link-relation* }
link-relation =
  element link-relation {
```

```
    documentation?, id, status, link-relation-name, href?
  }
link-relation-name = attribute name { xsd:anyURI }
link-relation-ref = attribute link-relation-ref { xsd:IDREF }
link-types = element link-types { documentation?, link-type* }
link-type =
  element link-type { documentation?, id, status, name?, href?, path? }
path = attribute path { xsd:string }
service-conventions = element conventions { documentation?, headers?, ▶
uri-parameters?, status-codes? }
media-types = element media-types { documentation?, media-type* }
media-type =
  element media-type {
    id?,
    (href
     | (media-type-extends?,
        name,
        documentation?,
        description*,
        documents?,
        media-type-conventions?,
        interfaces?))
  }
media-type-ref = attribute media-type-ref { xsd:IDREF }
media-type-extends = attribute extends { xsd:anyURI }
description = element description { type, href, documentation? }
type =
  attribute type {
    "rnc" | "rng" | "xsd" | "JSONSchema" | "sedola" | "text" | "html"
  }
documents = element documents { document* }
document =
  element document {
    (id?,
     extends?,
     name,
     documentation?,
     properties?,
     links?,
     document*)
    | ref-attribute
  }
document-ref =
  element document {
    attribute ref { xsd:IDREF },
    documentation?
```

```
  }
document-refs =
  element documents {
     document-ref*
  }
media-type-conventions = element conventions { documentation?, headers?, ▶
uri-parameters?, status-codes? }
properties = element properties { documentation?, property* }
property = element property { id?, name, documentation? }
links = element links { documentation?, link* }
link =
  element link {
    (link-relation-ref | link-type-ref),
    interface-ref,
    status?,
    documentation?
  }
link-type-ref = attribute link-type-ref { xsd:IDREF }
interfaces =
  element interfaces {
    interface-conventions?,
    interface*
  }
uri-parameters =
  element uri-parameters { documentation?, uri-parameter* }
uri-parameter =
  element uri-parameter {
    id?, name, documentation, datatype, value-range?, default-value?
  }
uri-parameter-ref = attribute uri-parameter-ref { xsd:IDREF }
interface-conventions = element conventions { documentation?, headers?, ▶
uri-parameters?, status-codes? }
interface = element interface { headers?, id?, name, methods }
interface-ref = attribute interface-ref { xsd:IDREF }
headers = element headers { header* }
header = element header { id?, name, header-type, documentation? }
header-type =
  attribute type { "request" | "response" | "general" | "entity" }
methods = element methods { method* }
method =
  element method { id?, method-name, status?, request?, response? }
method-name = attribute name { http-method }
http-method =
  "GET"
  | "PUT"
  | "HEAD"
```

```
  | "POST"
  | "DELETE"
  | "TRACE"
  | "OPTIONS"
  | "CONNECT"
  | "PATCH"
request =
  element request {
    documentation?, request-uri-parameters?, header-refs?, document-refs?
  }
request-uri-parameters =
  element uri-parameters { request-uri-parameter* }
request-uri-parameter =
  element uri-parameter {
    documentation?, id?, name?, request-uri-parameter-ref
  }
request-uri-parameter-ref = attribute ref { xsd:IDREF }
header-refs = element header-refs { documentation?, header-ref* }
header-ref = element header-ref { ref }
response =
  element response {
    documentation?, response-status-codes?, header-refs?, document-refs?
  }
response-status-codes =
  element status-codes {
    element status-code { ref }*
  }
status-codes = element status-codes { documentation?, status-code* }
status-code = element status { code, id, documentation?, http-problem? }
status-code-ref = attribute ref { xsd:IDREF }
code = attribute code { HTTP-status-enum }
HTTP-status-enum =
  "100"
  | "101"
  | "102"
  | "200"
  | "201"
  | "203"
  | "204"
  | "205"
  | "206"
  | "207"
  | "208"
  | "301"
  | "302"
  | "303"
```

```
| "304"
| "305"
| "306"
| "307"
| "308"
| "400"
| "401"
| "402"
| "403"
| "404"
| "405"
| "406"
| "407"
| "408"
| "409"
| "410"
| "411"
| "412"
| "413"
| "414"
| "415"
| "416"
| "417"
| "418"
| "420"
| "422"
| "423"
| "424"
| "425"
| "426"
| "428"
| "429"
| "431"
| "444"
| "449"
| "450"
| "451"
| "494"
| "495"
| "496"
| "497"
| "499"
| "500"
| "501"
| "502"
| "503"
```

```
    | "504"
    | "505"
    | "506"
    | "507"
    | "508"
    | "509"
    | "510"
    | "511"
    | "598"
    | "599"
http-problem =
    element problem { problemType, title, detail, supportId, more }
problemType = element problemType { xsd:anyURI }
detail = element detail { text }
supportId = element supportId { xsd:anyURI }
more = element more { foreign-element* }
resources = element resources { id?, documentation?, resource* }
resource =
    element resource {
      documentation?,
      id,
      name,
      identity-provider-ref?,
      public?,
      status?,
      extends?,
      location?,
      resource-interface*
    }
resource-ref = attribute resource-ref { xsd:IDREF }
extends = attribute extends { xsd:QName }
location = element location { documentation?, (uri | uri-template) }
uri-template = attribute uri-template { text }
resource-interface = element interface { ref-attribute }
authentication =
    element authentication { authentication-conventions?, mechanism*, ▶
identity-provider? }
authentication-conventions = element conventions { documentation?, headers?, ▶
status-codes? }
mechanism =
    element mechanism {
      id?, name, authentication-type, documentation?, scheme*
    }
mechanism-ref = attribute mechanism-ref { xsd:IDREF }
identity-provider = element identity-provider { id, mechanism-ref }
authentication-type = attribute authentication-type { text }
```

```
scheme = element scheme { id?, name, documentation?, scheme-parameter* }
scheme-parameter = element parameter { id?, name, documentation? }
datatype =
  attribute datatype {
    "string"
    | "boolean"
    | "decimal"
    | "float"
    | "double"
    | "duration"
    | "dateTime"
    | "time"
    | "date"
    | "hexBinary"
    | "base64Binary"
    | "anyURI"
    | "integer"
    | "language"
    | "ID"
    | "IDREF"
    | "integer"
    | "long"
    | "short"
    | "byte"
  }
value-range = element value-range { text }
default-value = element default { text }
uri = attribute uri { xsd:anyURI }
```

E. Complete Maps Example

Here is the complete RADL description for the example used in the text of this article.

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="radl2html.xsl"?>
<service xmlns:html="http://www.w3.org/1999/xhtml/" xmlns="http://identifiers.emc.com/radl"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="Maps">

  <documentation> This is an example from the book RESTful Web Services, chapter 5. </documentation>

  <start interface-ref="int-planets" />

  <link-relations>
    <link-relation id="rel-place" name="place">
      <documentation> The target resource is a related place. Links of this type are found by
        looking for <html:code>a</html:code> elements with <html:code>class="place"</html:code>.
        Additionally, you can find search links to places via the <html:code>form</html:code>
        element with <html:code>id="searchPlace"</html:code>. </documentation>
    </link-relation>
    <link-relation id="rel-point" name="point">
```

```
    <documentation> The target resource is a related point on a planet. Links of this type are
      found by looking for <html:code>a</html:code> elements with different values for the
        <html:code>class</html:code> attribute, like <html:code>coordinates</html:code>,
        <html:code>map_nav</html:code>, <html:code>zoom_in</html:code>, and
        <html:code>zoom_out</html:code>. </documentation>
  </link-relation>
  <link-relation id="rel-map" name="map">
    <documentation> The target resource is a map related to the current resource. Links of this
      type are found by looking for <html:code>a</html:code> elements with
        <html:code>class="map"</html:code>. </documentation>
  </link-relation>
  <link-relation id="rel-image" name="image">
   <documentation> The target resource is an image related to the current resource. Links of this
      type are found by looking for <html:code>img</html:code> elements with
        <html:code>class="map"</html:code>. </documentation>
  </link-relation>
</link-relations>

<media-types>
  <media-type id="med-planets" name="planets" extends="application/xhtml+xml">
    <documentation> We are defining an XHTML <html:em>microformat</html:em> by adding meaning
    using the <html:code>class</html:code> attribute to elements. For example, adding
    <html:code>class="planets"</html:code> to the <html:code>ul</html:code> element, we can
    turn a list into a list of planets. </documentation>
    <description type="html" href="http://tools.ietf.org/html/rfc3236"/>
    <documents>
      <document id="doc-planets" name="planets">
        <links>
          <link link-relation-ref="rel-place" interface-ref="int-place"/>
        </links>
      </document>
      <document id="doc-place" name="place">
        <links>
          <link link-relation-ref="rel-map" interface-ref="int-map"/>
          <link link-relation-ref="rel-point" interface-ref="int-point"/>
          <link link-relation-ref="rel-place" interface-ref="int-place"/>
        </links>
      </document>
      <document id="doc-point" name="point">
        <links>
          <link link-relation-ref="rel-place" interface-ref="int-place"/>
          <link link-relation-ref="rel-point" interface-ref="int-point"/>
        </links>
      </document>
      <document id="doc-map" name="map">
        <links>
          <link link-relation-ref="rel-image" interface-ref="int-image"/>
          <link link-relation-ref="rel-map" interface-ref="int-map"/>
        </links>
      </document>
    </documents>
    <interfaces>
      <conventions>
        <uri-parameters>
          <uri-parameter id="par-planet" name="planet" datatype="string">
            <documentation> Human friendly name of a planet, like <html:code>Earth</html:code>.
            </documentation>
          </uri-parameter>
```

```
            <uri-parameter id="par-place-name" name="place-name" datatype="string">
              <documentation> Human friendly name of a place, like <html:code>Mount%20Rushmore</▶
html:code>.
              </documentation>
            </uri-parameter>
          <uri-parameter id="par-scoping-information" name="scoping-information" datatype="string">
            <documentation> A hierarchy of <ref uri-parameter="par-place-name">place names</ref> ▶
like
              <html:code>/USA/New%20England/Maine/</html:code>. </documentation>
            </uri-parameter>
            <uri-parameter id="par-map-type" name="map-type" datatype="string">
            <documentation> The type of map, like <html:code>satellite</html:code>. </documentation>
            </uri-parameter>
            <uri-parameter id="par-scale" name="scale" datatype="string">
             <documentation> Dot followed by an integer, like <html:code>.1</html:code>. A bigger ▶
number
              indicates more details. </documentation>
            </uri-parameter>
            <uri-parameter id="par-show" name="show" datatype="string">
              <documentation> Things to search for near a given place, like <html:code>diners</▶
html:code>.
              </documentation>
            </uri-parameter>
            <uri-parameter id="par-latitude" name="latitude" datatype="float">
              <documentation> Latitude on a planet, like <html:code>24.9195</html:code>. </▶
documentation>
            </uri-parameter>
            <uri-parameter id="par-longitude" name="longitude" datatype="float">
              <documentation> Longitude on a planet, like <html:code>17.821</html:code>. </▶
documentation>
            </uri-parameter>
          </uri-parameters>
        </conventions>

        <interface id="int-planets" name="planets">
          <methods>
            <method name="GET">
              <response>
                <document ref="doc-planets"/>
              </response>
            </method>
          </methods>
        </interface>

        <interface id="int-place" name="place">
          <methods>
            <method name="GET">
              <request>
                <uri-parameters>
                  <uri-parameter name="planet" ref="par-planet"/>
                  <uri-parameter name="scoping-information" ref="par-scoping-information"/>
                  <uri-parameter name="place-name" ref="par-place-name"/>
                  <uri-parameter name="show" ref="par-show"/>
                </uri-parameters>
              </request>
              <response>
                <document ref="doc-place"/>
              </response>
```

```
        </method>
      </methods>
    </interface>

    <interface id="int-point" name="point">
      <methods>
        <method name="GET">
          <request>
            <uri-parameters>
              <uri-parameter name="planet" ref="par-planet"/>
              <uri-parameter name="latitude" ref="par-latitude"/>
              <uri-parameter name="longitude" ref="par-longitude"/>
            </uri-parameters>
          </request>
          <response>
            <document ref="doc-point"/>
          </response>
        </method>
      </methods>
    </interface>

    <interface id="int-map" name="map">
      <methods>
        <method name="GET">
          <request>
            <uri-parameters>
              <uri-parameter name="map-type" ref="par-map-type"/>
              <uri-parameter name="scale" ref="par-scale"/>
              <uri-parameter name="planet" ref="par-planet"/>
              <uri-parameter name="latitude" ref="par-latitude"/>
              <uri-parameter name="longitude" ref="par-longitude"/>
            </uri-parameters>
          </request>
          <response>
            <document ref="doc-map"/>
          </response>
        </method>
      </methods>
    </interface>
  </interfaces>
</media-type>

<media-type id="med-png" name="image/png">
  <description type="html" href="http://www.iana.org/assignments/media-types/image/png"/>
  <documents>
    <document id="doc-png" name="png"/>
  </documents>
  <interfaces>
    <interface id="int-image" name="image">
      <methods>
        <method name="GET">
          <request>
            <uri-parameters>
              <uri-parameter name="map-type" ref="par-map-type"/>
              <uri-parameter name="scale" ref="par-scale"/>
              <uri-parameter name="planet" ref="par-planet"/>
              <uri-parameter name="latitude" ref="par-latitude"/>
              <uri-parameter name="longitude" ref="par-longitude"/>
```

```
              </uri-parameters>
            </request>
            <response>
              <document ref="doc-png"/>
            </response>
          </method>
        </methods>
      </interface>
    </interfaces>

  </media-type>

</media-types>

<resources>

  <resource id="res-planets" name="planets">
    <location uri="/"/>
    <interface ref="int-planets"/>
  </resource>

  <resource id="res-place" name="place">
    <location uri-template="/{planet}/[{scoping-information}/][{place-name}]{?show}"/>
    <interface ref="int-place"/>
  </resource>

  <resource id="res-point" name="point">
    <location uri-template="/{planet}/{latitude},{longitude}"/>
    <interface ref="int-point"/>
  </resource>

  <resource id="res-map" name="map">
    <location uri-template="/{map-type}{scale}/{planet}/{latitude},{longitude}"/>
    <interface ref="int-map"/>
  </resource>

  <resource id="res-image" name="image">
    <location uri-template="/{map-type}{scale}/{planet}/images/{latitude},{longitude}.png"/>
    <interface ref="int-image"/>
  </resource>
</resources>

</service>
```

Bibliography

[1] Marc Hadley, Sun Microsystems. *Web Application Description Language*, W3C Member Submission 31 August 2009. http://www.w3.org/Submission/wadl/.

[2] Joe Gregorio, Google; Roy Fielding, Adobe; Marc Hadley, MITRE; Mark Nottingham, Rackspace; David Orchard, Salesforce.com. *URI Template*, IETF RFC 6570, March 2012. http://tools.ietf.org/html/rfc6570

[3] Mark Nottingham, Rackspace. *Home Documents for HTTP APIs*, May 8, 2013. http://www.ietf.org/id/draft-nottingham-json-home-03.txt

[4] Erik Wilde, EMC. *Home Documents for HTTP Services: XML Syntax*, June 11, 2013. http://www.ietf.org/id/draft-wilde-home-xml-01.txt

[5] N. Freed, Oracle; J. Klensin; T. Hansen, AT&T Laboratories. *Media Type Specifications and Registration Procedures*, IETF RFC 6838, January 2013. http://tools.ietf.org/html/rfc6838

[6] N. Freed, J. Klensin, J. Postel. *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*, IETF RFC 2048, November 1996. http://tools.ietf.org/html/rfc2048

[7] Bill Burke. *To WADL or not to WADL*, blog post, May 21, 2009. http://bill.burkecentral.com/2009/05/21/to-wadl-or-not-to-wadl/.

[8] Roy Fielding. *REST APIs must be hypertext-driven*, blog post, Mon 20 Oct 2008. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven/.

[9] *Home Documents for HTTP APIs*, http://tools.ietf.org/html/draft-nottingham-json-home-02. http://tools.ietf.org/html/draft-nottingham-json-home-02

[10] *Problem Details for HTTP APIs*, http://datatracker.ietf.org/doc/draft-nottingham-http-problem/. http://datatracker.ietf.org/doc/draft-nottingham-http-problem/

[11] *XML Media Types*, IETF RFC 3023, MURATA Makoto (FAMILY Given), Simon St.Laurent, Daniel Kohn. http://tools.ietf.org/html/rfc3023

[12] *Media Type Specifications and Registration Procedures*, IETF RFC 4288, Ned Freed, John C. Klensin. http://tools.ietf.org/html/rfc4288

[13] *Additional Media Type Structured Syntax Suffixes*, IETF RFC 5830, Tony Hansen, Alexey Melnikov. http://tools.ietf.org/html/rfc4288

[14] Aristotle Pagaltzis. *Does REST need a service description language?*, blog post, May 27, 2007. http://plasmasturm.org/log/460/.

[15] Cory House. *How RESTful is your API?*, blog post, August 26, 2012. http://www.bitnative.com/2012/08/26/how-restful-is-your-api/.

[16] Martin Fowler. *Richardson Maturity Model: steps toward the glory of REST*, blog post, 18 March 2010. http://martinfowler.com/articles/richardsonMaturityModel.html.

[17] Dare Obasanjo. *What's Wrong with WADL?*, blog post, June 4, 2007. http://www.25hoursaday.com/weblog/2007/06/04/WhatsWrongWithWADL.aspx

[18] Jim Webber, Savas Parastatidis and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media; 1 edition (September 24, 2010). ISBN-13: 978-0596805821.

[19] Leonard Richardson, Sam Ruby *RESTful Web Services*. O'Reilly Media; Dec 17, 2008f. ISBN-13: 978-0596554606.

[20] Erik Wilde. *Service Documentation Language* https://github.com/dret/sedola/

[21] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures, PhD Dissertation Thesis*, University of California, Irvine © 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[22] RESTful Service Description Language (RSDL). Michael Pasternak, Red Hat. Available at http://www.ovirt.org/RSDL.

[23] Jonathan Robie, Rob Cavicchio, Rémon Sinnema and Erik Wilde. *RESTful Service Description Language (RSDL): Describing RESTful Services Without Tight Coupling.* 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.Robie01. Available at http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html.

[24] Swagger. Available at http://developers.helloreverb.com/swagger/, https://github.com/wordnik/swagger-ui.

[25] RAML (RESTful API Modeling Language. Available at raml.org.

[26] I/O Docs (Mashery). Available at https://github.com/mashery/iodocs.

XML Authoring On Mobile Devices

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

Abstract

Not too long ago XML-born content was not present in a mobile-friendly form on mobile devices. Now, many of the XML frameworks like DocBook, DITA and TEI provide output formats that are tuned to be used on mobile devices. These are either different electronic book formats (EPUB, Kindle) or different mobile-friendly web formats.

Many people find XML authoring difficult on computers, let alone mobile devices. However, due to the constantly increasing number of mobile devices, that made people create mobile-friendly output formats from XML documents, there is clearly a need to provide also direct access to authoring XML content on these devices.

I would like to explore the options for providing XML authoring on mobile devices and describe our current work and the technology choices we made to create an authoring solution for mobile devices. Trying to enable people to create XML documents on mobile devices is a very exciting, mainly because the user interaction is completely different on a mobile device: different screen resolutions, different interaction methods (touch, swipe, pinch), etc. See how we imagined XML authoring on an Android phone or on iPad! How about editing XML on a smart TV? Leverage speech recognition/dictation and handwriting recognition technologies that are available on mobile devices to enable completely new ways of interacting with XML documents!

Keywords: XML, authoring, mobile, review, user experience

1. Introduction

When an XML-based solution is implemented the lowest impedance for communicating between different processing steps is to use XML, so people try to use XML in as many places as possible, but there are usually a few processes where using XML is not always easy. One of these processes is the review of XML content. Another process is the contribution of initial content from people that are not familiar with XML.

A traditional review process will convert the XML information to an output format, usually PDF, and have reviewers annotate that PDF with comments, then

align the PDF with the XML documents that generated it to identify the places in the XML source the comments refer to and manually act on those comments to make the corresponding changes to the XML documents.

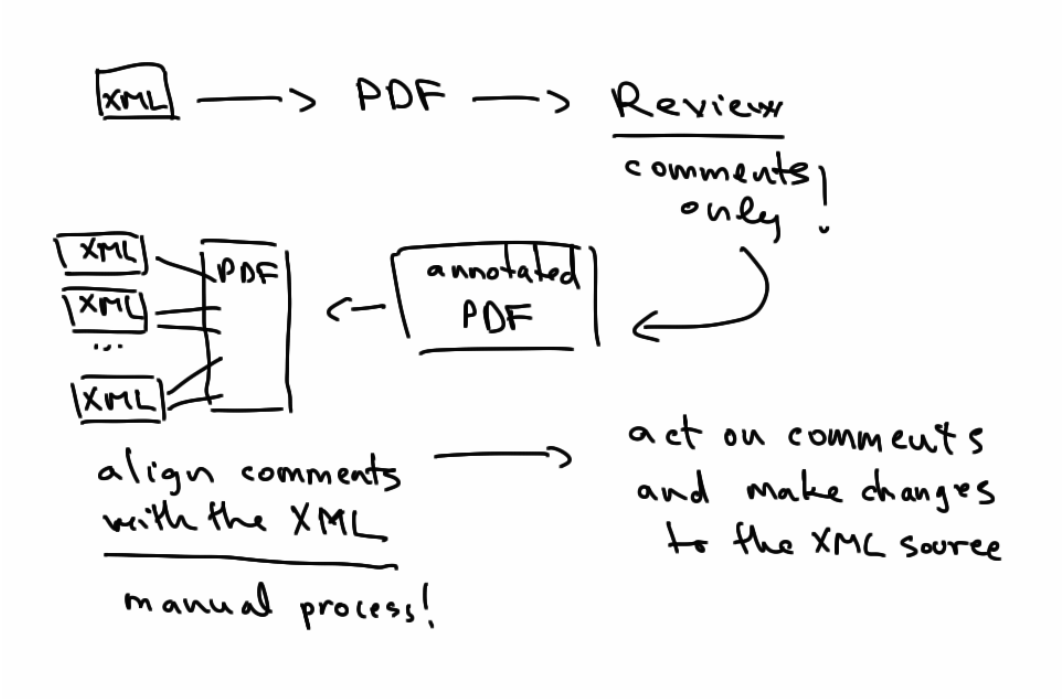


Figure 1. Traditional review process

This process has many steps and some of them are not automated so they not only consume time but errors can appear at different stages. Many of the issues can be solved by adopting a direct XML review process, where users can annotate directly on the XML content, and add not only comments but also make changes to the document that will be considered proposed changes. Thus, responding to a comment by identifying the XML source the comment refers to and then updating the document as described in the comment can be replaced with an simple action to accept a proposed change to the document.

Contributing initial content is very much linked to the tools the users already know and the devices he has access to. Thus initial content is contributed in whatever format the users can use and then converted to XML to be able to enter the XML-based solution. Usually people use Word and there is a conversion process that tries to get from Word to XML. We can cut the conversion cost if we are able to get this initial content in XML form.

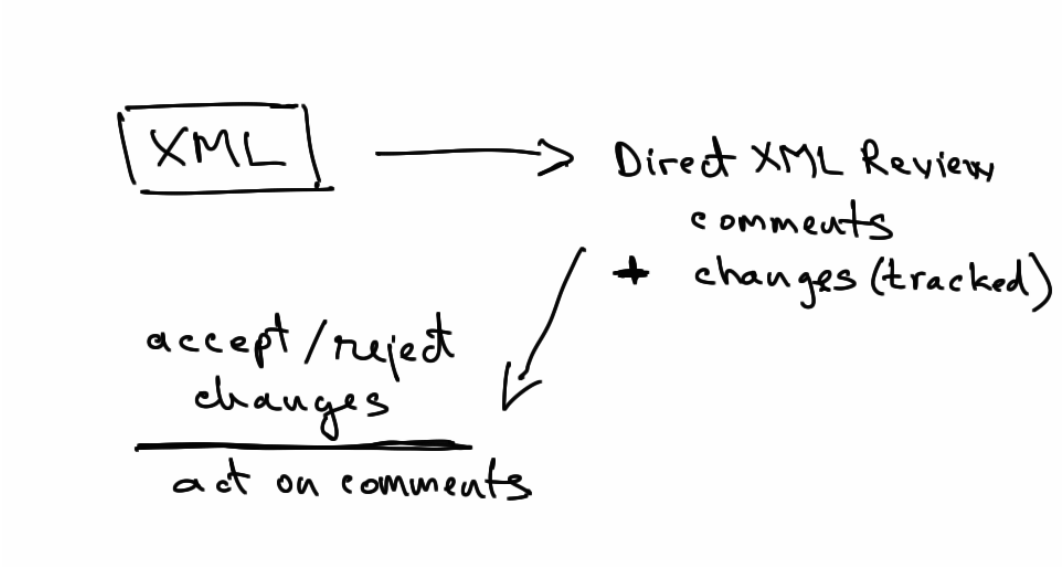


Figure 2. Direct XML review process

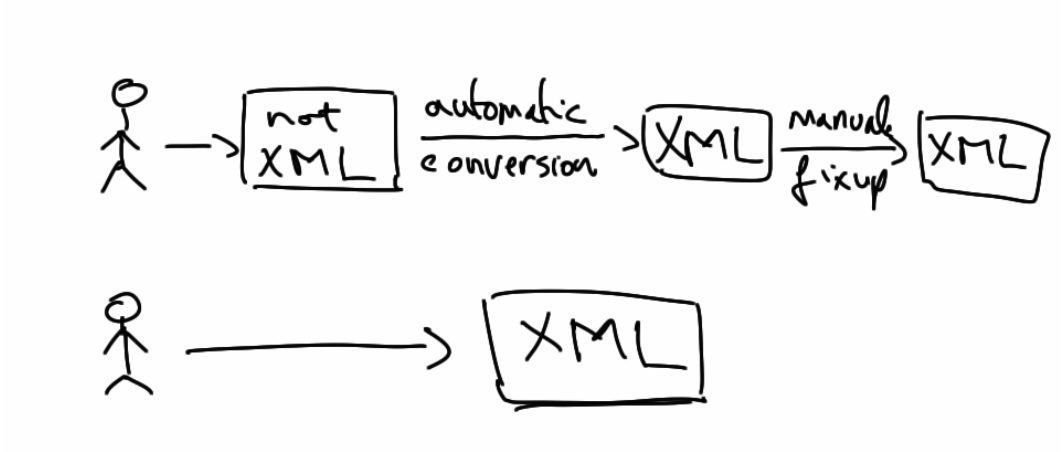


Figure 3. Non XML data conversion to XML vs XML first

When you move to an XML-based solution it is important to be able to cut costs on the review process and to implement an XML-first system, where people can contribute initial data directly in XML. We tried to address these problems and we currently provide solutions for both the review process and for creating an XML-first solution. However, the current solution requires the use of a laptop or a desktop computer.

The people that perform reviews or the ones that contribute initial content are in general external to the department that deals with the XML-based solution, so it is difficult to control the resources available to these people. The increasing use of mobile devices during the last years and the projections for next years show that mobile devices are not something we can ignore (mobile devices are expected to exceed the number of desktops this year) and the only device some of those people

have may be a mobile one. So, if we want to be able to cut the costs and the complexity of processes similar to the ones described, we need to be able to provide at least direct XML review and simplified XML authoring on mobile devices.

2. Technology choices

Once we decided to start building a tool for XML authoring on mobile devices the next step was to decide on what technologies that will be based on. The first decision was if it was to be a native application or a web application.

From our experience with oXygen we found that it is great to be able to support multiple platforms with the same code - oXygen being built in Java works on any platform that provides a JVM. So one problem with a native solution was that we had to build a different application for each mobile platform while a web application will allow us to reuse the same code for all devices, as long as they support the required web technology (HTML5 and JavaScript). A web application has also other advantages over a native application like immediate update, no app-store interference and an important one - the fact that it will work also on desktops. A disadvantage will be that the access to device specific functionality will not be possible but we it should be possible to use a hybrid application if such functionality will be critical in the future.

Another decision point was on how much processing should be done on the client and how much on the server. Targeting mobile devices we wanted to have as little as possible processing on the client, in order not to drain the device battery. This factor and the fact that we already have in Java many of the components needed for XML authoring made us decide to prefer the server side processing to the client processing and keep the current oXygen on the server and have only the display part on the client side, like a remote display, thus reusing almost all of the existing components and technology stack.

We experimented with different approaches for a rendering XML in the web application, including:

1. Placing XML directly inside an HTML document and render it though the same CSS that we use now
2. Using the Canvas to display the rendered XML document, similar to how it is done inside oXygen, using a CSS parser that will provide the rendering styles for each element
3. Render/convert the XML as/to HTML and convert the CSS used for XML to match the converted HTML format

In the first case we hit limitations in the browser support for CSS that made it impossible to use this approach. For example browsers do not support the CSS `attr/2` function as specified in CSS3, where along with the first parameter that specifies the attribute name you can specify also a second parameter that represents the at-

tribute value type. This is used in oXygen to specify that an attribute value is a URI and it should be a link.

In the second case we implemented all the rendering primitives that are used in oXygen (we have a Graphics interface that is used for rendering the XML documents and the methods from this interface were implemented also based on the HTML5 Canvas) but then we needed also the CSS parser, the layout engine, caret management, etc. which were not easy tasks.

In the 3rd approach we converted the XML document to HTML5 and then we modified the CSS that matched on XML to match on the converted HTML5 structure to obtain the same rendering as the XML+CSS that we currently use. This allows us to use the browser editing support for HTML to modify the document content.

For mobile interaction we use JQuery mobile due to existing experience - we use this also for the mobile-friendly WebHelp transformations that we provide for DITA and DocBook. However, other frameworks may be used as we plan to support multiple templates for the user interface.

3. Web application architecture

Here it is a diagram showing the current architecture showing how the oXygen existing support is reused on the server side:

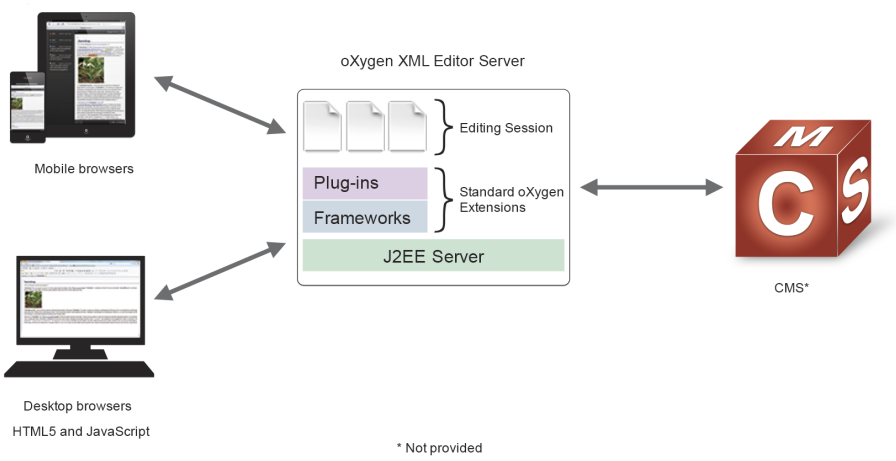


Figure 4. oXygen web application architecture

Three components can be identified:

- oXygen on the server
- The HTML+JavaScript that render the document on the client side

- Content storage that can be in the form of a CMS

oXygen on the server is a Java servlet that encapsulates the Java-based oXygen to provide the visual editing support. It reuses the same customizations created for the oXygen desktop that come in the form or frameworks and plugins. This allows for example to reuse the editing support for DITA, DocBook, etc. as well as plugins that provide access to remote repositories like the CMS connector plugins.

The server part will generate HTML5+JavaScript for an XML file that when rendered will provide the view for that XML document. The generated HTML5 content keeps XML related information in `data-*` attributes. The CSS that matched on XML is automatically converted to match on the generated HTML5 and its `data-*` attributes that encode the XML information. The conversion from XML to HTML5 uses mainly `div` elements but sometimes it also takes advantage of specific HTML elements, like the table element for example.

4. Samples

We will demo XML reviewing functionality, using custom XML interfaces and full XML editing. Here you can see some screen-shots taken on iPad:

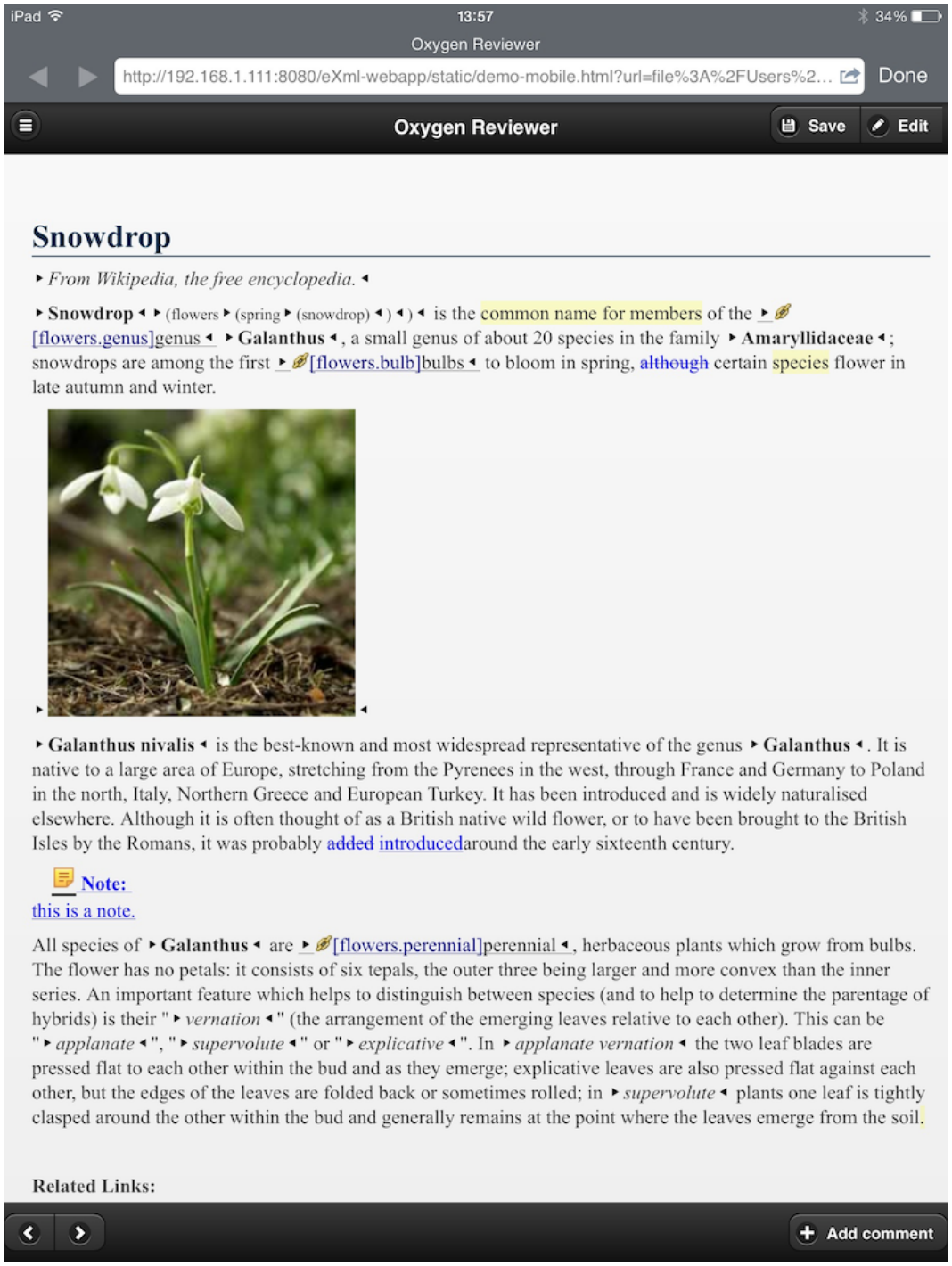


Figure 5. A DITA topic

Note the highlights that represent areas with associated comments as well as the added and deleted content styled with underline and strikeout decorations.

Figure 6. A DocBook article

Here we have a DocBook article rendered though CSS with different structure like images and lists. Note again the comment highlights and the decorations for changed content.

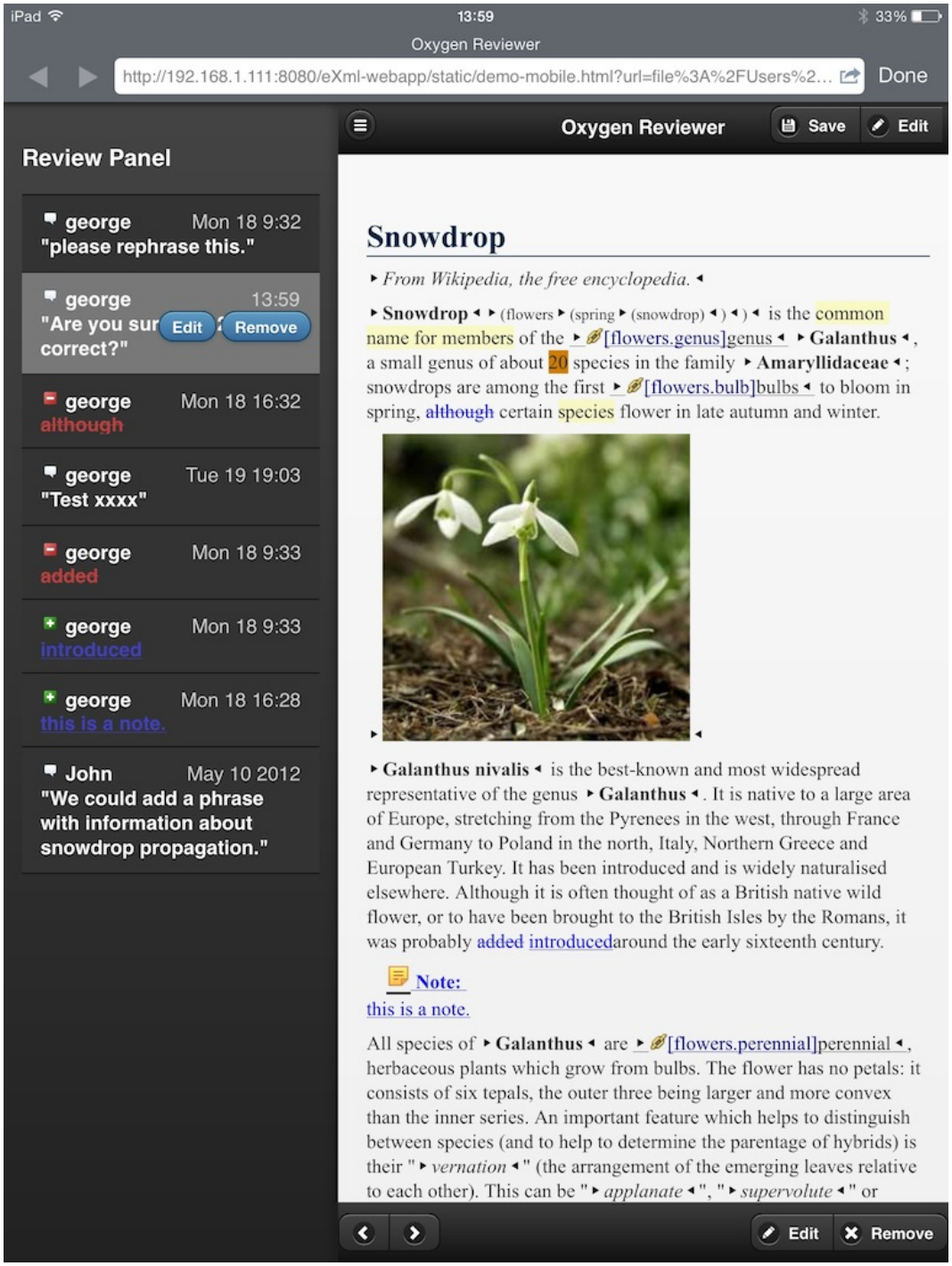


Figure 7. Review Panel showing all review comments and changes

You can swipe right on the editing area to make the Review Panel visible. When you swipe over a review entry the available actions are displayed so you can easily act on a review to edit or remove a comment, accept or reject a change. Swipe left to hide the Review Panel and return to the editor.

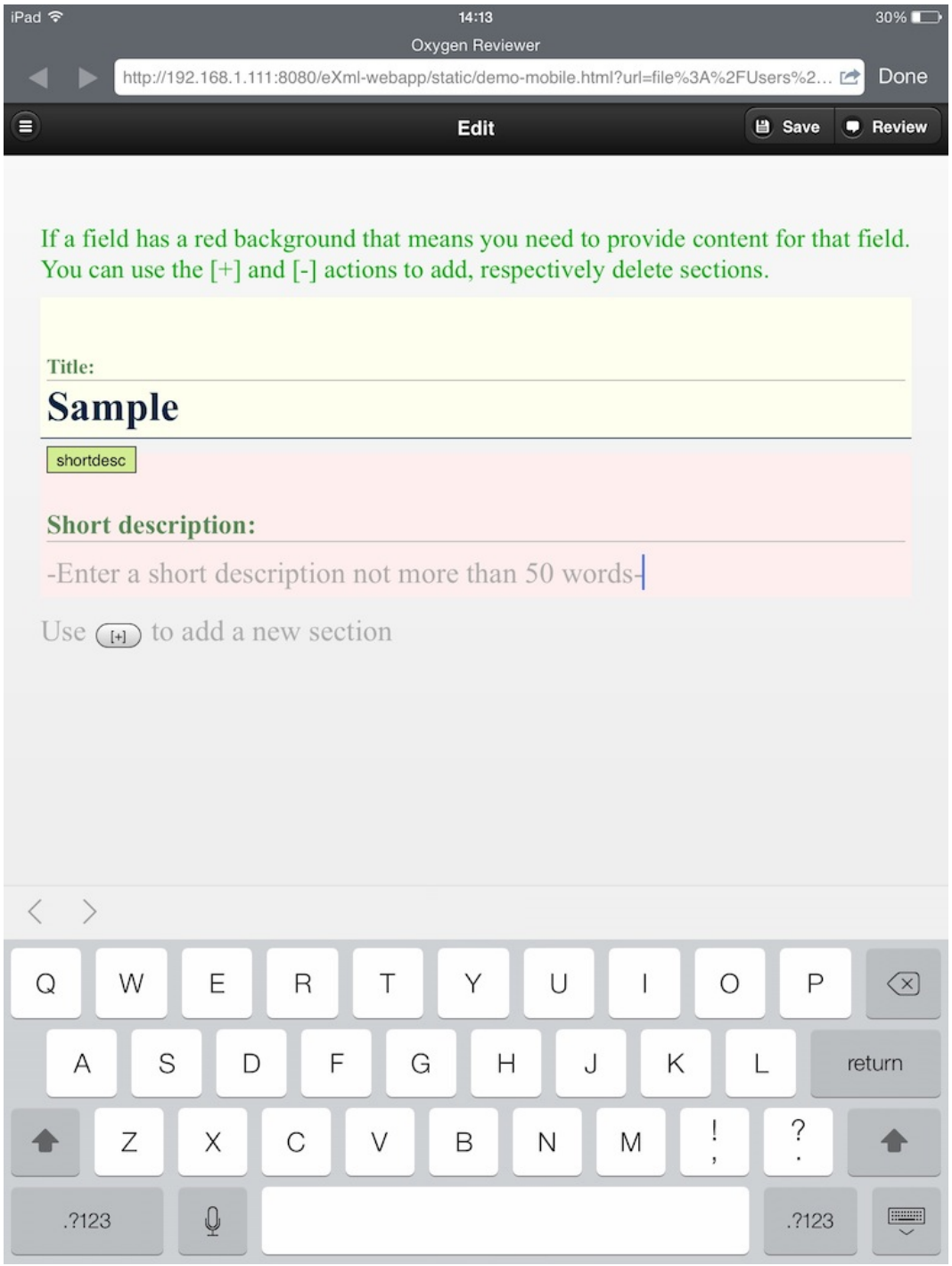


Figure 8. A custom XML editing interface

Note the inline action marked with "[+]" that can be used to add a new section to the document.

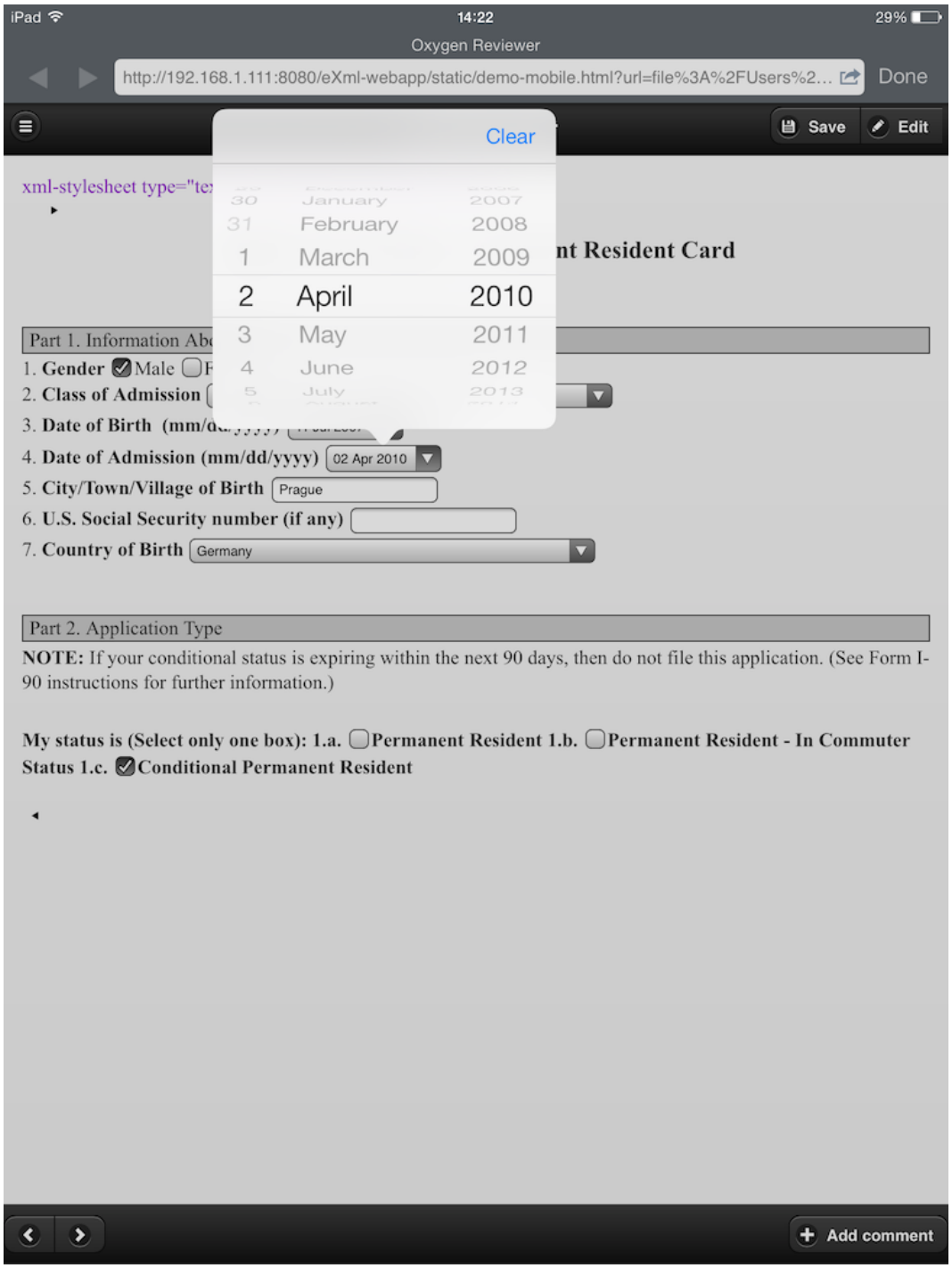


Figure 9. Enter a date value using the standard iPad date picker

Different form controls can be used to build custom interfaces that will provide access to text and attribute values, thus making the editing simpler and removing the need

to train users. Each form control will use the native support on each platform, thus the user will have the same editing experience he is already used to on that device.

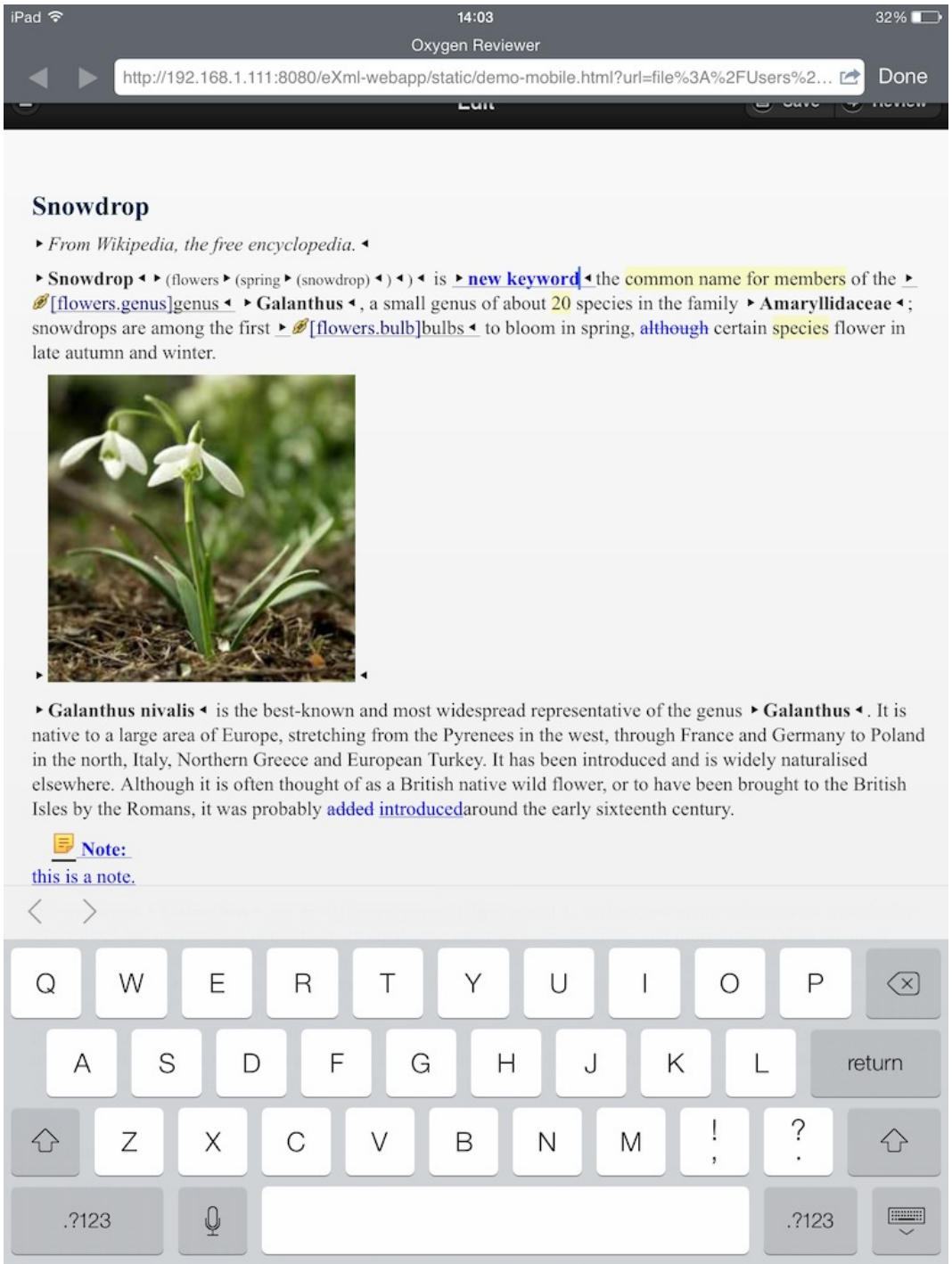


Figure 10. Text editing with changes recorded as tracked-changes

Here you can see the editing mode, where we have the keyboard show up and the document contains a caret. The changes in this case are recorded as tracked changes.

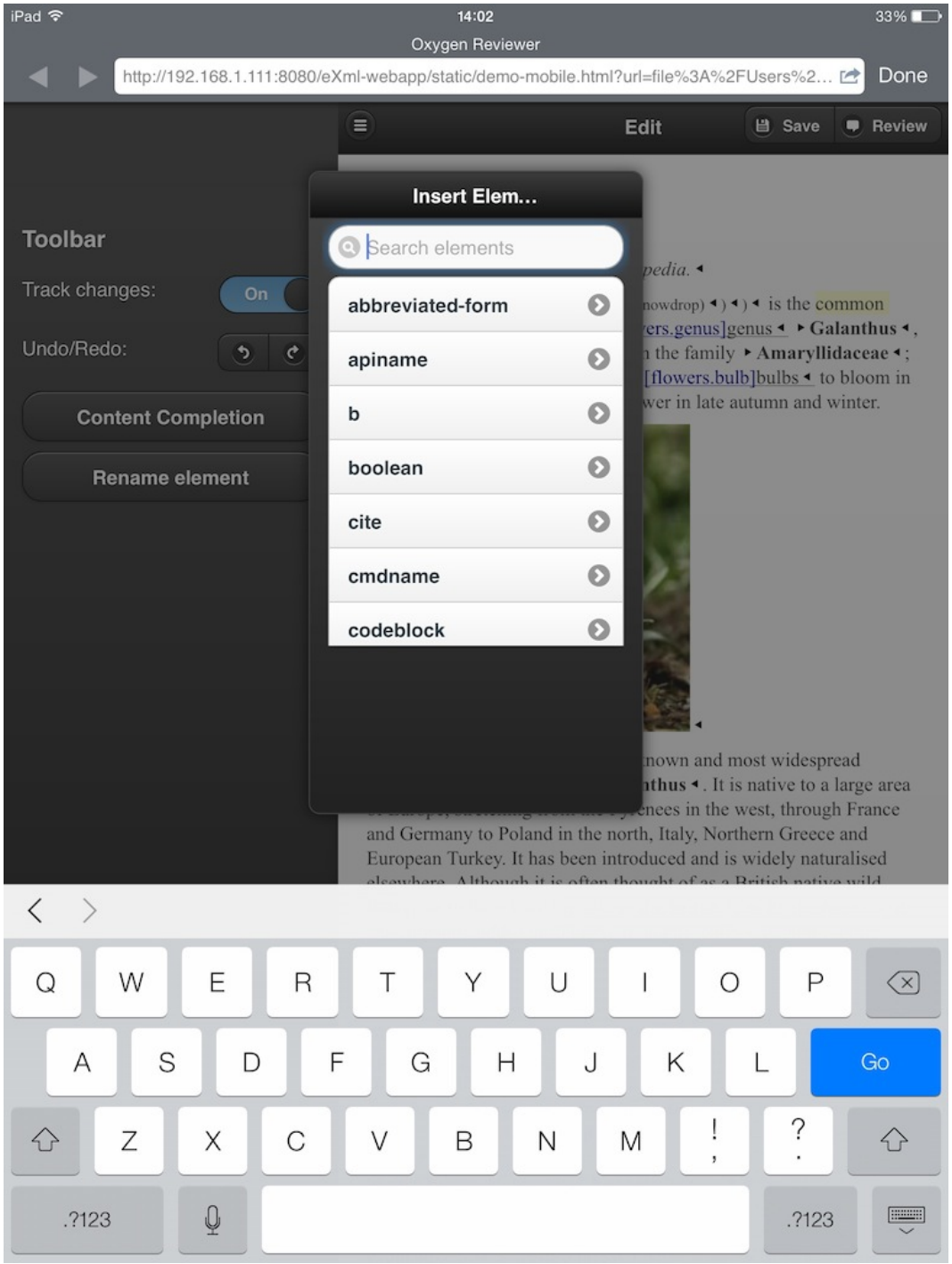


Figure 11. Inserting markup

You can insert markup either with the dedicated action or by pressing the enter key in the virtual keyboard. That will show a popup with valid element names where you can filter to see only the elements that match, then select one to insert in the document.

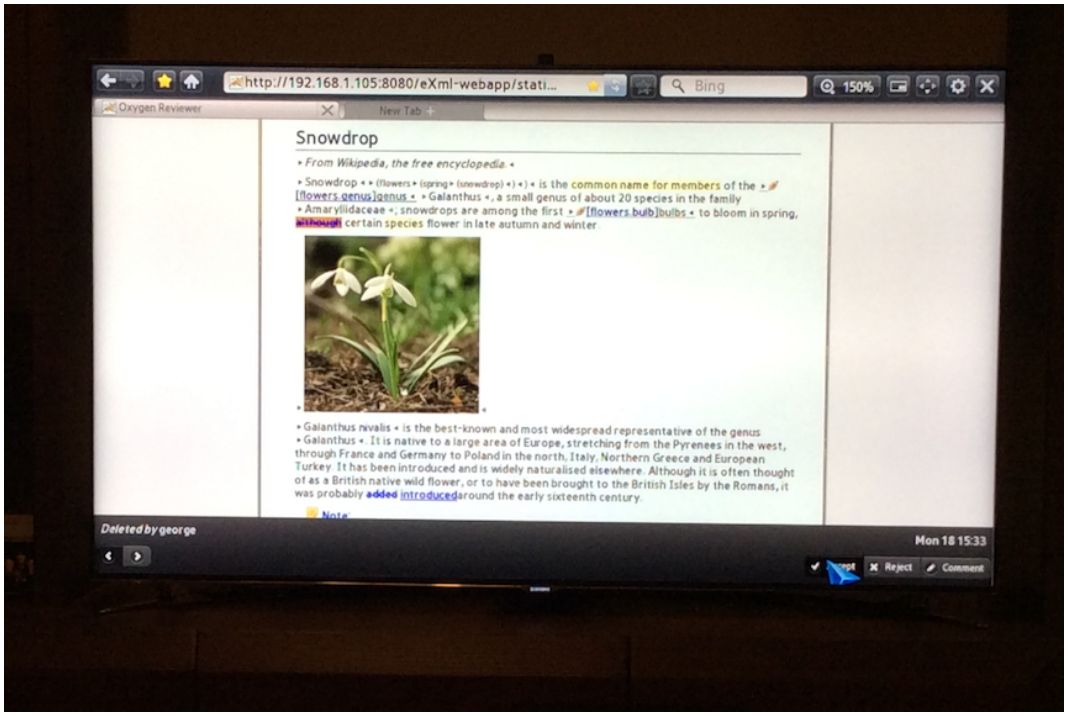


Figure 12. A DITA topic on a smart TV

The web editing platform works on any device supporting HTML5 and JavaScript, in this case we have it running on a smart TV.

5. Conclusions

XML editing on mobile devices can solve some real use-cases where people that are not XML-aware can contribute XML content using their preferred or available device at that moment. Creating a customized user interface using form controls bind to attribute values and inline actions reduce the training sometimes to zero - probably this is the way further, putting more effort on the developer to customize the user interface so that users will not have to think in terms of XML concepts but focus on the information they want to record. There is a lot of exploration to come up with the best possible user interface that takes advantage of specific input methods and interaction patterns from mobile devices and this is just the start.

More generally, the web editing support for XML makes it available on any device, not only on mobile devices and it will be interesting to see if we can get different other applications based on XML like an XML-based blogging system or an XML-based wiki-like system.

A MathML Progress Report

Autumn Cuellar
Design Science, Inc.
<autumnc@dessci.com>

Abstract

In the early days of HTML, math was a heavy topic of conversation within the HTML Working Group. The World Wide Web, after all, was built by scientists for scientists, and math resides at the heart of science. Displaying math on the Web was a tricky problem, however. Math is not an image and should not be treated as an image. Math is text and should be an inherent part of the document along with the paragraph text in the document, but the special formatting required was beyond the capabilities of browsers at the time. The problem was more than the HTML WG was equipped to handle. Thus, the Math Working Group was formed to tackle the challenge of a math markup language not only for display of equations on the Web but for a standard format for mathematics to be used within any mathematical and scientific communication. The MathML 1.0 specification became a W3C Recommendation in 1998. This paper will discuss the progress of MathML since.

The MathML language has undergone two major revisions since the initial MathML 1.0 specification. The latest revision, MathML 3.0, was finalized in October 2010. For the latest version, the Math Working Group carefully considered the needs of various groups with a stake in math communication. For example, support for better control of automatic linebreaking/line wrapping was added for the publishing community, who wanted rendering engines to be able to automatically break an equation extending beyond a set column or page width. MathML 3.0 also includes improved features for specifying elementary math notation and new support for international math. Though no standard is ever really complete, MathML has reached maturity with the latest specification.

Equations are rarely standalone objects. MathML is most useful when used in conjunction with a doctype that is larger in scope, and lately the standard has been gaining steam as a worthwhile format for encoding mathematics within wider standards. On the data side, scientific markup languages such as CellML and Systems Biology Markup Language (SBML) rely on MathML to contain the mathematics of the stored models. On the document side, MathML has been adopted by a range of XML standards from DAISY and NIMAS on the accessibility front to the Journal Article Tag Set (JATS) for use in scientific journal articles to use in DITA, which is used primarily

for technical documentation. But perhaps the most significant milestone for MathML has been its recent inclusion in the HTML5 and EPUB 3 standards.

Now that MathML is nearly ubiquitous as a standard, what about tool support? Support for the MathML standards can be found in a range of applications, including authoring systems, computer algebra and other scientific computation systems, and reading systems. Nevertheless, a couple of challenges remain in this area. One is that where most want to see equations is in their browser and ebook systems, but support for MathML is lagging in both browsers and EPUB e-readers. One reason for this is that the makers of these systems can now depend on MathJax, an open-source Javascript library for rendering MathML in browsers. MathJax is a useful short-term solution, but it is insufficient for a number of reasons. The other remaining challenge is that conversion of documents in legacy formats can be difficult.

MathML has come a long way since its early days. The language has been steadily evolving over the past 15 years and has reached a healthy maturity in its latest version. The wider standards communities have come to recognize the value that MathML adds as a means of communicating mathematical and scientific information and have responded by including MathML where needed. The next step in the evolution of MathML is the continued development of tool support, especially native rendering in browsers and ereading systems and conversion tools for legacy formats.

1. Introduction

MathML is the XML standard for encoding mathematical information. It's maintained by W3C, the same organization that maintains the XML and HTML standards. MathML was conceived in the mid-1990's during a discussion of requirements for HTML 3. HTML, the language behind the World Wide Web (a network built by scientists for scientists) at this time had no way to represent mathematics -- a failing that many felt needed to be addressed. Mathematical information, after all, is as inherently significant to a document as paragraph text. The gains that could be made by encoding the mathematics as a natural part of the document rather than capturing the math as images were recognized early.

However, the challenges for representing math on the web seemed too complex to be tackled in an update to HTML. The Math working group was formally chartered in 1997 to develop a math markup language not only for display of equations on the Web but to provide a standard format for mathematics to be used within any electronic mathematical and scientific communication. [1]

The first MathML specification became a W3C recommendation in 1998. This paper reviews the progress of the MathML markup language since the MathML 1.0 recommendation, its adoption within larger standards, and the status of software support for this mathematical language.

2. The MathML standard

Early on, the Math working group recognized that a significant challenge in providing a standard metadata for mathematical content existed due to the symbolic structure of mathematical notation. In certain situations, the notation of an expression must be very precise to correctly convey the meaning of the expression. At the same time, due to the symbolic nature of mathematics, one notation can sometimes be used to express different ideas. If the logic is not present to distinguish whether an expression of x(y) indicates x is a function of y or two variables being multiplied, for a simple example, electronic communication of such an expression is meaningless.

To overcome potential ambiguity while enabling content publishers to have precise control over math notation, the Math working group broke the MathML language out into categories consisting of content elements, presentation elements, and interface elements. Content MathML specifically captures the content or semantics of the expression while Presentation MathML focuses on the layout of the math. The interface markup is used to mix and link the two so that one might specify both the unambiguous intent of the expression and the rules for the display of the equation.

The MathML 1 specification, when it became a W3C Recommendation in 1998, set a solid foundation for digital communication of math. Five years later, in 2003, MathML 2 was unveiled to address requirements overlooked in the initial specification. For instance, in version 1, multi-line equations could only be expressed through the use of tables. MathML 2 also favors certain features of Cascading Stylesheets (CSS), which had grown more popular for applying styles to content, over its former attributes for specifying key properties.

MathML 3 was finalized in late 2010. With MathML 3, MathML shows a new maturity by taking into consideration the needs of various communities with a more specialized interest in mathematics. For example, the publishing industry is better served with improved support for automatic line breaking: for long equations that may extend past the width of pages or columns, content providers can now specify both a maximum width (so that the equation automatically wraps) as well as how to indent or align subsequent lines. For the education community, MathML 3 adds new features to support elementary math notation, such as stacked operations including addition and substraction, long division, and repeating decimals. MathML 3 also became a truly international standard by adding support for right-to-left languages and for notations that vary from region to region.

Most standards are ever-evolving, and I imagine the same will continue to be true of MathML. Since the MathML 3 specification became a recommedation, there have already been two revisions to correct errata in the specification. However, MathML can now be considered a mature standard; the major requirements of the different stakeholders have been met. What remains are minor questions, such as

whether styles could be applied to individual characters within a token and whether certain operation attributes should be applied to token elements instead.

3. Adoption of MathML within other standards

MathML was never intended to be a standalone language or doctype. Mathematics are often just a single component of larger documents and models. The power of MathML as a communication standard has been made manifest by its uniform inclusion into other standards of broader scope.

Content MathML is particularly suited for use in scientific fields and has been successfully incorporated into a number of scientific markup languages. CellML[1], a language for describing biological models at the level of the cell, uses Content MathML for the mathematical components of the model. Systems Biology Markup Language (SBML[2]) is another biological modeling markup language used for describing biological processes such as metabolism and cell signaling. SBML also relies on Content MathML. PhysML[3] is a markup language for physics that extends OMDoc (Open Mathematical Documents), a markup language giving context to a range of mathematics. OMDoc also requires formulae to be encoded in Content MathML if using MathML.

Presentation MathML has seen broader acceptance as a standard for use within document markup languages. DocBook[4] and DITA[5] are two popular document XML languages often used for technical documentation, though with a wider possible range of application. Both offer support for STEM fields with the inclusion of MathML. The Journal Article Tag Set (JATS[6]), which, as its name suggests, is used for journal articles, has also for some time provided support for MathML.

The accessibility community has widely embraced Presentation MathML. The level of detail that MathML provides for formulae makes it an ideal candidate for use with Accessible Technology (AT), allowing for highlighting, navigation, and conversion to spoken text and braille. Math is a fairly visual language, but the fine-grained control that AT software has over mathematics encoded in MathML gives audiences with learning or vision disabilities tools for comprehending complex expressions. For this reason, DAISY[7], a standard for digital talking books; NIMAS[8],

[1] http://www.cellml.org/
[2] http://sbml.org/Main_Page
[3] https://trac.omdoc.org/OMDoc/wiki/PhysML
[4] http://www.docbook.org/
[5] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=dita
[6] http://jats.nlm.nih.gov/
[7] http://www.daisy.org/publishers
[8] http://aim.cast.org/learn/policy/federal/what_is_nimas#.Ut2VMBDnaUk

a textbook standard; and PDF/UA[9], a specification for tagging PDF for "universal access", all require math to be expressed as MathML.

However, arguably the most significant standards' development for STEM fields has been the recent inclusion of Presentation MathML in HTML5[10] and EPUB 3[11]. Mathematics did not make the cut for inclusion into early versions of HTML due to the challenges of rendering a complicated and varied language. However, through MathML these challenges have been overcome. Thus, HTML5, which as of December 2012 is a W3C Candidate Recommendation, now includes MathML.

HTML5 has been praised for its inclusion of different content types. In previous versions, different types of media (such as video, audio, math, and even images) were treated as external objects, many of which required plug-ins to the browser for the visitor to experience. The benefit of including media in the HTML is that browsers will consistently and correctly display the content without requiring external software. This means, theoretically, a given page with its included media will display the same on any platform or device.[2]

EPUB, the open standard for e-books, is maintained by the International Digital Publishing Forum (IDPF). In version 2 of the EPUB standard, the content of the e-book could be expressed in either of two varieties: DAISY or XHTML. DAISY, as previously mentioned, is an independently managed standard for specifying Digital Talking Books, mostly used for accessibility purposes. Because the DAISY standard included MathML for specifying mathematical content, one could create a valid EPUB file with MathML.

The DAISY version of EPUB was not widely used. In the most recent version of the EPUB standard, IDPF eliminated the DAISY variant, based the XHTML variant on HTML5, and endorsed inclusion of MathML as an important aspect of e-books. Like HTML5, EPUB 3 has been lauded for taking digital content to the next level with its support for media.

4. Tool support of MathML

As any computer language is useless without adequate tool support, while MathML has gained popularity amongst other standards for communicating mathematical information, it has also been receiving growing support in a wide variety of applications: computer algebra systems, graphing applications, calculators, modeling software, assessment creation systems, educational whiteboards, etc.

Creating MathML has never been easier. Microsoft has shipped software called the Math Input Panel[12] with its Windows operating system since version 7 was re-

[9] http://www.aiim.org/Research-and-Publications/Standards/Committees/PDFUA
[10] http://www.w3.org/TR/html5/
[11] http://idpf.org/epub/30
[12] http://windows.microsoft.com/en-us/windows7/use-math-input-panel-to-write-and-correct-math-equations

leased in 2009. The Math Input Panel can convert handwritten math to MathML with a fair amount of accuracy. Alternatives for other platforms are available through Enventra's MoboMath[13] and Vision Objects' MyScript[14] applications.

Many computation systems such as Maple[15] and Mathematica[16] will import and export MathML, sometimes giving users a wide array of options over the form the MathML should take. For example, Mathematica allows export of a MathML string on its own, generation of a full XHTML + MathML document, and the choice between Presentation and Content MathML, among other options.

For document editing, most XML editing and documentation authoring systems, including FrameMaker[17], <oXygen/>[18], XMetaL[19], and Flare[20], now include a MathML editor or the option to add one as a plug-in. Content management systems are now also starting to add support for equations by adding web-based MathML editors to their systems. MathML editing components, such as Design Science's MathFlow Components[21] and WIRIS Editor[22], have given application and website developers a straightforward mode of entry into MathML creation by providing simple APIs.

On the rendering side, one of the most popular rendering engines is MathJax[23], an open source Javascript library for rendering MathML in modern browsers. MathJax can either convert the MathML to an HTML/CSS layout or to SVG, either of which are formats that browsers can display effectively. A complicated issue with displaying mathematics on unknown systems is in regards to fonts. Standard desktop fonts do not support a wide range of mathematical characters. MathJax gets around this issue with a font algorithm that checks first for desktop fonts, secondly for an appropriate web font, and if none of the above will do, it can fall back to glyphs for certain characters.

Other popular rendering engines include JEuclid[24], another open source solution, and Design Science's MathFlow Composers[25]. JEuclid and the MathFlow Composers can both be used for converting MathML to various image formats, and both of these components have been included in a large number of composition pipelines,

[13] http://enventra.com/products/mobomath/overview.htm
[14] http://www.visionobjects.com/en/myscript/about-myscript/
[15] http://www.maplesoft.com/products/maple/
[16] http://www.wolfram.com/mathematica/
[17] http://www.adobe.com/products/framemaker.html
[18] http://www.oxygenxml.com/
[19] http://xmetal.com/
[20] http://www.madcapsoftware.com/products/flare/
[21] http://www.dessci.com/en/products/mathflow/mf_components.htm
[22] http://www.wiris.com/en/editor
[23] www.mathjax.org
[24] http://jeuclid.sourceforge.net/
[25] http://www.dessci.com/en/products/mathflow/mf_components.htm

allowing for conversion of XML+MathML to a variety of other formats including Word documents, PDF, and HTML Help files.

Tool support for MathML is growing at a steady pace, but this author can identify a couple of areas for which there are a need for improvements. The first is in browser and e-reader support for MathML. HTML 5 adds MathML functionality to the standard. However, of the five major browsers (Chrome, Firefox, Internet Explorer, Opera, and Safari), only two, Firefox and Safari, currently provide native MathML rendering, and both of these only support a subset of the MathML specification. MathJax provides a short-term solution, but even the consortium that manages MathJax campaigns for native MathML implementations by the major browser vendors. When MathML is treated by the browser as a native element of the page (as the rest of the elements are), there is a lot to gain, such as the MathML being modifiable and queryable as a regular part of the DOM, being able to fully apply CSS to the MathML, and faster rendering of the MathML, to name a few. Browser vendors have their reasons for being reluctant to add support for MathML. Some worry about code bloat and some worry about security vulnerabilities, but most are also using MathJax's existence as an excuse to put off the MathML implementations. [3]

E-reader systems are often built on browser technology. When MathML is rendered natively within browsers, MathML support by e-readers will soon follow. This was the case with Apple's iBooks, which was able to leverage the work done in WebKit for the Safari browser to enable readers to display MathML in iBook files.

Another area requiring development is conversion of legacy formats to MathML. Some equations exist in SGML/XML documents under a proprietary math format such as Adobe Framemaker math. In such cases, there is little support for conversion from the propietary format to the standard MathML. Framemaker users with legacy documents have only two choices: to develop their own conversion system or to hire a conversion services company such as Stilo International[26] or Data Conversion Laboratory[27].

Other equations exist as part of a large $\LaTeX$ library. A number of projects have attempted to convert $\LaTeX$ to XML+MathML or to HTML+MathML with varying degrees of success. The difficulty here is the long tail of $\LaTeX$ macros in use. The $\LaTeX$ macros which have made the language so popular as a choice for publishing scientific documents have also impeded its conversion to XML.

Finally, PDF documents with equations in them are notoriously difficult to quickly bring into XML+MathML format. For plain text PDF documents, Optical Character Recognition (OCR) software has had a decent success rate, but the special formatting of math makes it an altogether different beast. Only the InftyReader[28]

[26] http://www.stilo.com/
[27] http://www.dclab.com/
[28] http://www.inftyproject.org/en/

OCR application is known to attempt to convert the mathematics in PDF documents into MathML.

5. Conclusion

MathML has come a long way since its early days. The language itself has undergone two major revisions and has reached a point of stability with features supporting international audiences and those with a range of interest in mathematics. MathML has received widespread acceptance from other standards communities as evidenced by its inclusion into other standards, including those used for biological models, physics models, and various document types. Finally, software support for MathML has grown at a steady pace. It is now quite easy to create or generate MathML and to display it. However, a need for browsers and e-readers to render MathML natively still exists, and organizations with legacy content continue having difficulties converting their content to XML+MathML.

Bibliography

[1] Patrick Ion. Robert Miner. 7 July 1999. *Mathematical Markup Language (MathML™) 1.01 Specification*. W3C.

[2] Rossi Fernandes. 15 March 2012. *What HTML5 means to you*[29]. Tech2.com India.

[3] Shankland Stephen. 5 November 2013. *Google subtracts MathML from Chrome, and anger multiplies*[30]. CNET.

[29] http://tech2.in.com/features/general/what-html5-means-to-you/290302

[30] http://news.cnet.com/8301-1023_3-57610854-93/google-subtracts-mathml-from-chrome-and-anger-multiplies/

Finalising a (small) Standard

John Lumley
jwL Research, Saxonica
<john@jwlresearch.com>

Abstract

This paper discusses issues and lessons that arose during the finalisation of a standard (library) for XSLT/XPath/XQuery extension functions to manipulate binary data. This process took place during 2013 in the EXPath community, through shared (mailing-list) commenting, specification redrafting, implementation experimentation and test suite development. The purpose, form and specification of the library (which isn't technically difficult) are described briefly. Lessons and suggestions arising from the development are presented in four broad categories: establishing policies, concurrent implementation and application, using tools and declarative approaches, and pragmatic issues. None of these lessons are new, but bear reinforcement. This work was performed under the auspices of the EXPath community and was funded by Saxonica Ltd.

1. Standards – let's have plenty

Since its inception in the mid 90s, the world of XML has been governed by standards. Originally attempting to regularise the extension of web pages, XML was developed as a meta-syntax for markup, aimed at using a strict tree-based representation of propertied element nodes containing sub-trees or text nodes. Very soon work started on developing XSL as a formatted and paginated alternative to HTML for documents with professional appearance. As we're all well aware, the two aspects of XSL, formatting and variable document generation, split into two orthogonal standards - XSL-FO and XSLT. The latter developed, with XPath (and associated XQuery), into a full declarative XML-transformational language. In most recent versions XSLT/XPath/XQuery have become full functional programming languages, with XML trees as central data type.

Significant work under the auspices of W3C has developed and finalised these standards for XSLT/XPath/XQuery[1] through three major versions over the past 15 years. In each case a very comprehensive specification has been developed, reviewed, criticised and modified in cycles that are typically 3 years long and involve a few

[1]Whilst there are differences, for the rest of this paper, unless stated otherwise, the term XSLT or XPath is used to refer to any of the three.

dozen contributors. Examples, test cases and experimental/operational implementations are all used to develop and finalise the specifications, which is often followed by a further 3 years of polishing.

Once a standard (version) has been finalised, a degree of stability should then encourage developers of both implementations and applications to build and support software, without the language's syntax or semantics altering. Of course it is exactly such full-scale use of a language that could expose shortcomings or new features that are needed to increase utility. The art of developing standards is to anticipate as much as is needed to get a useful and robust set of features that i) is adequate for significant application use but ii) not too complex for implementation or application.

In the case of XSLT, the first version (1.0) was developed very quickly, concentrated on defining a model for transforming XML through pull-based selection (using XPath to select from sub-trees of the input document) or push-based case generation, using pattern-matching templates. A minimal necessary set of functions and instructions was also defined (e.g. `count()`, `translate()`, `xsl:number`) to support necessary computation.

Whilst the functionality of XSLT 1.0 was sufficient for many initial application purposes, additional requirements appeared slowly. Some of these could be satisfied by interesting but somewhat complex coding techniques (such as 'Muenchian grouping'); others would require an extension to the language itself. In the case of XSLT it had been anticipated that extension functions or even extension instructions could be added to an implementation, to provide additional functionality. These could be very application specific (`my:jpg.size($uri)`) or somewhat more general (`math:cosec()`) and implementations were encouraged to provide support mechanisms for such extensions.

Over time a few common libraries of such (XPath) extension functions (but not XSLT instructions) were developed, by mechanisms described later, in topics such as mathematical functions, or even something as language-central as being able to reuse generated sub-trees in XSLT. Gradually such libraries increased the firmness of their specification and influenced some of the additional requirements for subsequent versions of the main standard. A specific example is the incorporation into XPath3.0 of mathematical (`math:pow()`...) and transcendental (`math:sin()`...) functions, which had been developed originally by the EXSLT group[4]) as a library for XSLT1.0 operating on values of type `xs:double`.

1.1. Community spirit

Large standards usually grow under rigorous and well-controlled frameworks, such as W3C, but these additions often arise from some small group of enthusiasts identifying common ground and interest and collaborating on an informal basis. These developments are usually a 'community effort' with a group of (world-dis-

tributed) volunteers who propose, define, refine, criticise and revise some 'specification' document, whilst discussing and developing implementations and tests. The tests are usually built as large sets of small test cases, and become a key part of proving the specification, especially when multiple implementations are checked against them. But whilst there are varying degrees of formality in the process of development of such specifications, all developments have some aspect of being a social process, subject to personalities, biasses and individual interests.

These efforts sometimes succeed, sometimes they peter out, sometimes they stall, or the effort is abandoned and a new direction chosen. The degree of formality of such a standards effort can be variable – from simple discussions and documents, right up to near-W3C levels of rigour. As mentioned previously, the EXSLT group was active and influential in the period 2001-6 exploring features like mathematical functions, dates and times and regular expressions. Several of these made their way into the larger standards in subsequent years, though many still exist in a form of limbo – half-finished, partially supported by one or two implementations, and used in very few applications.

The EXPath community[1] is such a group, attempting to develop suites of rigorous extensions for the XPath/XQuery/XSLT world. It started in 2010 and has perhaps a dozen or so active contributors. Early efforts include specifications for file, http client, packaging and ZIP manipulation. The intention was to create such suites to standards of rigour approaching those demanded by W3C for its full specifications, whilst having a more agile platform for developing new functionality. In particular the specification documents it publishes use the same format and organisation as those of W3C[2]. However, until very recently none of the specifications had stabilised enough to warrant being labelled 'Version 1.0', despite one of them (File[3]) having been used extensively by a number of developers over the past few years.

2. Fiddling with bits – Binary Module

In the summer of 2013 the author was invited by Saxonica to work on EXPath's Binary module[2], intended to support binary and bit-level manipulation of data within XSLT/XPath/XQuery[3]. An initial draft of the module specification had been created by Jirka Kosek (University of Economics, Prague) in the spring of 2013. Over the next four months, the specification was revised several times, discussion on features and criticisms were fielded through a mailing list, test suites were developed

[2]It operates under the W3C Community Final Specification Agreement [http://www.w3.org/community/about/agreements/final/].

[3]The W3C Working Group developing XSLT had agreed such features would be useful, but were out of scope for XSLT3.0

and tested. Eventually a specification (whose rigour and structure is based on those of W3C) was finalised in early December as 'Version 1.0'.

The module eventually ended up with a library of 26 functions, in four broad classes – generating binary constants, basic operations of breaking, joining, extending and searching, encoding and decoding text and numeric values and finally standard bitwise operations.

A simple use case was finding the size of a JPEG image. Many independent instances of such an extension function will have already been written in Java, but with the facilities from the Binary module this could be written directly in XSLT:

```
<xsl:variable name="binary"
  select="file:read-binary(@href)" as="xs:base64Binary"/>
<xsl:variable name="location"
  select="bin:find($binary,0,bin:hex('FFC0'))"/>
<size width="{bin:unpack-unsigned-integer($binary,
                $location+5,2,'most-significant-first')}"
      height="{bin:unpack-unsigned-integer($binary,
                $location+7,2,'most-significant-first')}"/>

      => <size width="377" height="327"/>
```

A JPEG image consists of a series of *segments*, starting with a marker consisting of `0xFF` followed by a single byte type identifier, which is never `0x00`. (`0xFF` in data is byte-stuffed with a trailing `0x00`, so two-byte sequences starting with `0xFF` followed by non-null always indicate start of segment.) Identifier `0xC0` denotes a *Start Of Frame* segment and contains the size, number of (colour) components and subsampling type of the image, all with defined byte lengths.

We need the binary of the JPEG image, which in this case we've read from a file using `file:read-binary()` from the EXPath File module, but could have been web-uploaded as base64 data. Our method is to i) find the offset location of the Start Of Frame segment and then ii) decode the byte-positioned values for the width and height at that location. This needs just two functions: `bin:find()`, which searches for the first occurrence of a contiguous byte sequence inside another and `bin:unpack-unsigned-integer()`, which returns an integer of specified length from a range of bytes in the input.

Even the most evangelistic wouldn't class this module as requiring 'rocket science' skills to implement in an XPath processor – the Saxon implementation is a single Java class with about 1000 (sparse) source lines[4]. The interest is not what this module is really about, but how its specification is developed to consensus, what lessons might be learned from the experience and what appear to be effective decision-making processes.

[4]Saxon already contained a class to represent items of `xs:base64Binary` type, with data storage and (de-)serialisation machinery.

Whilst the author had been working in software research for many many years, and been a very heavy user of XSLT (including building large extension functions[5]) in the field of document engineering for much of the last decade, he had not been involved in the W3C or related standards activities. Being asked to work not just on the implementation (inside a very well established software product, *viz* Saxon), but also helping drive the specification itself to a 'standard' position would undoubtedly be educational. And so it proved.

The initial work covered three main areas:

- Studying the current specification and producing a modified draft in the light of previous discussions, further comments from below and evident points of consensus. This led to the publication of a revised draft at the end of July 2013.
- Instigating further discussion in the EXPath community to push the specification forward. This involved summarising what appeared to be the main issues that had been discussed beforehand and those that were apparent from my reading. For example it was uncertain which of two different binary types, `xs:hexBinary` or `xs:base64Binary` would be supported or both[6]. These discussions were carried out entirely through the group's mailing list.
- Building a skeletal Java extension class to use with Saxon (including gaining familiarity with the company's build environment), creating a few sample test applications, getting it all running and generating early test-sets and exercising and testing them.

This was the bulk of the continuous work on the project, taking 10 days of effort spread over a calendar month. However it was not the end of the affair - two more intermediate drafts were published over the next three months, until convergence to a version 1.0 recommendation followed at the beginning of December 2013. During this process of refinement, which of course took place in bursts of frantic activity and oh-so-silent lulls and involved ~100 mailing-list postings as well as private correspondence, there were several issues to be resolved. Four of these were significant:

- What was the main binary type to be used?
- 'Endianness' for numeric (de-)coding, including what the default should be.
- Overhauling the error code naming.
- Issues arising from access to the 'end' of data, and behaviour when arguments are empty sequences.

[5] SVG-PDF converters, text-block paragraph wrappers that preserve tree isomorphism, constraint-based layout resolvers – that sort of thing.

[6] `xs:base64Binary` was chosen – it is more efficient in serialisation, can be cast to and from `xs:hexBinary` cheaply, as both are usually implemented as wrapper classes around the primary byte sequence data, and was the *de facto* binary type used in the File module.

The following sections discuss lessons from that work, in four broad areas – concurrent implementation and application, establishing policies, using tools and declarative approaches, and pragmatic issues.

3. Prove the specification – do it all together

A specification is a document intended to be read and understood by a human. But it also needs to be unambiguous, with little doubt in what it is defining, since later on applications will rely on processors behaving very closely to that specification indeed. And to check such lack of ambiguity, a specification needs to be *proven* – tested that it describes what was intended, and not have nasty surprises lurking for the unwary, be they either those implementing processors that meet the specification ("That operation is $O(n^4)$"), or those writing applications ("There's no way to check that condition without triggering an error.").

A specification that was purely mathematical might of course be susceptible to being proven mathematically, but the practical and readable specifications we deal with are not quite that rigorous. So how do we carry out such proof while the specification is being developed? Unsurprisingly I suggest techniques that these days might be referred to as *agile*: i) use the computer early and often, ii) find some medium-sized examples, both to illustrate and test and iii) build specification, implementation, examples and tests concurrently.

All the parts of the development – discussion, specification, implementation, tests and applications – are related and whilst there are consequential dependencies, some of these are circular and refining. Thus there are distinct advantages if you can close the loops quickly, which requires progress on all almost fronts simultaneously.

3.1. Don't just think – use a computer

You can attempt to build a standard purely by thought, but it really helps if you have an *idiot savant* to assist and check your validity. When you have to explain all the rules to someone who will follow them slavishly *to the letter* then you will certainly get useful feedback. Luckily we have such an assistant, though we have to transcribe the rules we wish to verify into utterances in some form of programming language they understand, and we might conceivably make errors[7] in such transcription.

A specification *editor* really should be developing a working implementation in parallel with the specification. Not only does that give them early indications of the complexity and size of the problems being tackled, but it also helps gain understanding of what the proposed functions really mean, and with suitable examples,

[7] Otherwise known as *bugs*

whether there are significant shortfalls in functionality. Corner cases in input data quickly reveal issues either in the implementation ('null pointer exception' !) or which are unaddressed within the specification.

3.2. The power of the medium-sized example

During the development of the specification many small examples will be suggested, and often used as notes within the specification document itself. Usually these are to illustrate typical anticipated usage of the feature being discussed. For example:

```
bin:shift(bin:hex("000001"), 17) => bin:hex("020000")
```

appears in the Binary specification where a 'long' bit-shift is demonstrated – actually this showed clearly that shifts that moved significantly across byte boundaries (`$shift > 8`) were supported, partly in response to a query from another implementer[8]. Often these simple examples are added to the test suites.

But such examples don't really show *why* the features being described in the specification are useful and how they might be applied to solve useful problems. For this we need examples that *combine* several different features. Obviously the original suggesters of the standard usually have their own ideas for large applications, but to be successful in showing the initial reader how the standard features work together, a few medium-sized examples can be very helpful.

For the Binary module, one of these medium-sized examples involved coding and decoding long ASN.1 integers[9]:

```
<xsl:function name="bin:int-octets" as="xs:integer*">
  <xsl:param name="value" as="xs:integer"/>
  <xsl:sequence select="
     if($value ne 0)
     then (bin:int-octets($value idiv 256),$value mod 256)
     else ()"/>
 </xsl:function>

<xsl:function name="bin:encode-ASN-integer" as="xs:base64Binary">
  <xsl:param name="int" as="xs:integer"/>
  <xsl:variable name="octets" select="bin:int-octets($int)"/>
  <xsl:variable name="length-octets" select="
    let $l := count($octets) return (
      if($l le 127) then $l
      else (let $lo := bin:int-octets($l)
            return (128+count($lo),$lo)))"/>
  <xsl:sequence select="bin:from-octets((2,$length-octets,$octets))"/>
```

[8] Needed for example in a PDP11 emulator written in XSLT.

[9] Used in telecommunications, where numbers of arbitrary length are minimally encoded – but the integers can be so long (e.g. cryptokeys) that even their byte-length values need variable-length encoding.

```
</xsl:function>

<xsl:function name="bin:decode-ASN-integer" as="xs:integer">
  <xsl:param name="in" as="xs:base64Binary"/>
  <xsl:sequence select="
    let $lo := bin:unpack-unsigned-integer($in,1,1,'BE')
    return (
      if($lo le 127)
      then bin:unpack-unsigned-integer($in,2,$lo,'BE')
      else (let $lo2 := $lo - 128,
                $lo3 := bin:unpack-unsigned-integer($in,2,$lo2,'BE')
            return bin:unpack-unsigned-integer($in,2+$lo2,$lo3,'BE')))"
      />
</xsl:function>
```

which has results:

```
bin:encode-ASN-integer(0) => "AgA="
bin:encode-ASN-integer(1234) => "AgIE0g=="
bin:encode-ASN-integer(12345678912345678912345678912345678 9)
  => "Ag8XxuPAMviQRa10ZoQEXxU="
bin:encode-ASN-integer(123456789.. 900 digits... 123456789)
  => "AoIBdgaTo....EBF8V"
bin:decode-ASN-integer(xs:base64Binary("AgA=")) => 0
bin:decode-ASN-integer(xs:base64Binary("AgIE0g==")) => 1234
bin:encode-ASN-integer(xs:base64Binary("Ag8XxuPAMviQRa10ZoQEXxU="))
  => 12345678912345678912345678912345678 9
bin:encode-ASN-integer(xs:base64Binary("AoIBdgaTo....EBF8V"))
  => 123456789.. 900 digits... 123456789
```

This example not only exercised a number functions collectively, it also had the unexpected benefit of testing issues of scale. Early tests showed that small integers were handled correctly (the ASN.1 coding results could be checked by hand for numbers a few digits long...). But the ASN.1 integer was designed to handle numbers of arbitrary size and by using encode/decode combinations we could explore larger usage. The example at 40 digits worked, showing that `BigInt` integer values were no problem, but the next at 900 digits was seriously large-scale, and would involve a variable-length data length field[10]. It simply worked.

The essential requirements on such examples are that they i) are sufficiently complex that compound and possibly non-obvious combination of features are required, ii) involve a subject that should be clear enough to the average reader, iii) small enough that the reader can mentally walk through the example and understand

[10]The 900-digit number requires 503 octets to encode, needing two bytes to encode the octet length, which needs a byte to describe *its* length.

its operation[11] and iv) have clear examples of invocation and result. Needless to say, the example results contained in the specification should be generated *by machine*, and in addition these examples test the ability of an implementation to combine successfully results from several parts of the specification.

In retrospect I feel that a slightly larger example would have been helpful – one that exploited a few more of the library features in combination. The author had worked on parts of a simple SVG-PDF generator (about 100 lines of XSLT), that might have been useful as an appendix in the specification.

4. Fix policy early (and not too often)

Most libraries have a degree of regularity about them – certain types of function have similar signatures, behave (and fail) in similar ways and might be expected to share common semantics about aspects of their behaviour. Some of these can be classed as policies that the library (and indeed other libraries and standards that are of related forms) might impose on its members. Four particular issues which could be considered to relate to policy arose during the course of the development: function names, error handling (and naming), null or empty arguments and access to the 'edge' of binary sequences, and versioning and future-proofing.

Such policies should also take into account other similar policies from the execution environment. For example error codes could have used the prefix-numeric style of XSLT, but in this case a different coherent style from EXPath was used.

Getting these policies explosed and discussed early increases the awareness of the community to the importance and consequences of these common decisions and reduces the risk of large-scale and fundamental changes having to be made across the specification at some late stage.

4.1. The naming of parts

Developers invoke facilities such as functions by *names*. For the library such a name is some form of index that identifies which feature is requested – any set of enumerations would suffice (`fn1()`, `fn2()`...). But for us humans these names should be meaningful, making the general nature of the function being requested clear. They should also be relatively concise, to aid both reading and writing. And if a function has *strong* similarities to another function in a well-known parallel domain, then some similar name might be attractive.

As an example, the first draft contained a function `bin:binary-subsequence()`, which returned a section of some input binary data. In the final specification this was now called `bin:part()`. The `binary-` section was dropped as the `bin:` already

[11] The reader is assumed i) fully competent in the language and ii) cognisant of common paradigms, e.g. recursion and higher-order constructs.

implied it was concerned with binary data. `subsequence` implied strong similarity with the XPath function of similar name, which wasn't quite correct – the `substring()` function had better parallels in the meaning, but '`subbinary`' didn't quite ring true. Hence `bin:part()`.

Often several functions have generally similar behaviour, effectively with some parametric or type variation. The issue then is whether a single function with control parameter(s) should be specified, or several with differing names. We chose to define `bin:binary()`, `bin:octal()` and `bin:hex()` rather than a single `bin:constant($in as xs:string,$base as xs:integer)`, because i) meaning is clearer, ii) there's no need to check whether the base is supported[12]and iii) if a developer does need to choose programmatically between 2, 4, or 8, she can use `xsl:choose`.

4.2. Errors, and how to live with them

Any useful program element can be invoked under erroneous conditions. Some errors might be considered warnings, where graceful degradation is possible. Some errors are comparatively trivial and a default result can be chosen. Some errors are seriously fatal.

Some errors will be generic and likely to occur in a number of functions, such as indexed access outside the range of some binary data. Others will be very specific to a very small number of functions, such as the types of decoding errors that can be encountered in creating strings from binary forms. To be effective error codes used should be specific enough help trap different types of failure for appropriate types of response (e.g. fatal termination, fallback, warning...) and collected into common cases amongst the functions.

It really helps if the error codes are meaningful to the reader. Luckily during the development, a change from prefix-numeric to textual codes was suggested for the specifications of the EXPath community. Hence `err:BINA0006`[13] changed to `bin:octet-out-of-range` as the error raised when a value beyond 8 bits was being used as an octet.

But too many codes can swamp the definition or make error trapping in an application unduly cumbersome. The author probably proposed too many fine-grain error codes such as `bin:index-before-start` and `bin:index-after-end` which were subsequently rationalised into the single `bin:index-out-of-range`. XSLT 3.0's `try/catch` supports providing more detailed information (such as what was the index and data size) through bindings to the `$err:*` variables.

[12]Base 9 is unlikely, base 1 might *just* be credible. Even octal was questioned, though critics hadn't built a PDP11 emulator in XSLT.
[13]A naming protocol deriving originally from W3C/QT tests

Some argue that when a function is being initially defined error cases should be outlined concurrently with functionality. The declarative function catalog (see Section 5.1) encourages that, by including a list of errors raised for each function definition. [Currently the error codes and descriptions are included as a narrative list within the specification body and cross-referred from the function definitions. It might be more coherent to define those codes and their descriptions in the declarative body of the catalog. Then the error codes themselves can be consulted by other tools, for example to check that all error codes have been exercised within a test-suite.]

4.3. Arguments 'on the edge'

Just as the author thought the specification was completed, questions were raised about some edge cases in accessing the ends of data, or in cases with 'empty' data. The subsequent discussions (which involved a little frustration!) overturned some of the already defined, implemented and tested error behaviour in a number of functions. An example situation was with the function:

```
bin:insert-before($in as xs:base64Binary?,
                  $offset as xs:integer,
                  $extra as xs:base64Binary?) as xs:base64Binary?
```

whose functional summary was simply:

> *The `bin:insert-before` function inserts additional binary data at a given point in other binary data.*

This seems comparatively simple but some of its edge cases, and similar situations in related functions, caused extensive rework until quite late in the specification. Normal use of the function is straightforward – concatenate the first `$index` bytes of `$in` with all the bytes of `$extra`, then followed by the remainder of `$in` after the `$index`th byte. However, the problems come when these arguments are not so accommodating:

- `$in` is empty – this can occur in two ways: it could have no binary data (similar to `xs:string('')`) or the argument could be an empty sequence, i.e. the XPath equivalent of null.
- `$extra` is similarly empty.
- `$index` points outside any binary data of `$in`.

The behaviour already defined was very conservative – any access outside the strict limits of the data of `$in` raised an error - even if the index pointed to just before or just after the data of `$in`. Parallels with the function `fn:insert-before()` were not terribly helpful, as its behaviour was liberal, and dated from earlier versions of XSLT where error management wasn't available. Such erroneous conditions on an index were allowed to degrade gracefully, defaulting to pointing to the appropriate

end. After much discussion the behaviour was modified to accommodate 'just on the edge' values for the index and similarly on related functions, However by this time the rework required was not trivial: extensive changes had to be made in three places at once – the functional signature definitions, the working implementations and the by-now-extensive test suites.

This is a case where deciding general principles early in the process (and perhaps deliberately describing them in the specification) would have i) started discussion about these issues early, ii) fixed significant parts of the error model before code was written for it and iii) avoided late and repeated rework.

4.4. Future-proofing – which version am I?

Again, very late in the development process, some 'nice-to-have' features (such as decoding *sequences* of numbers) were proposed, but then agreed to be 'postponed to version 1.1'. The issue arose about how we could i) define and examine which version of the specification was supported and ii) how other future-proofing and backwards-compatibility would be approached. Such mechanisms (required for example with fallback options and conditional compilation in the target language) require something beyond the strict extension library (e.g. adding cases to the response of `system-function()`). In this case pragmatic considerations and perhaps some fatigue, postponed such version identification to a later version[14]!

5. The declarative tool-user

As software engineers we live with Wirth's maxim: "Algorithms + Data Structures = Programs", in that the design of suitable data structures can make algorithms necessary to meet program goals more efficient, robust and flexible. A similar sentiment can be employed in this case – by designing declarative structures for some parts of the specification (as opposed to narrative sections) and employing modest tools to process these structures, we can increase the robustness and most importantly the *flexibility* of the specification considerably. Some forms of late change can require nothing more than alteration to a declaration and automatic reprocessing. To put it another way:

Copy and Paste is not necessarily your friend

5.1. Anything to declare? – Plenty!

XSLT is at heart a declarative, rather than an imperative, language. We're encouraged to define statements (in a tree form) about *what* is true, rather than intricate formulae to compute such information. Features such as tables, maps, archetypical (tree) data

[14] An interesting example of 'self-non-reference' ?

structures and the like can be used comparatively easily to define useful relationships and can be consulted with XPath and small fragments of XSLT code. They also encourage definition *in one place only*.

For the Binary module there was a 'function catalog', using the format employed for the standard XPath built-in function library[15]. This contains a declarative list of functions, detailing their names, signatures, semantic summary, detailed rules of behaviour, error conditions, examples and notes. An example entry was:

```
<fos:function name="insert-before" prefix="bin">
  <fos:signatures>
    <fos:proto name="insert-before" return-type="xs:base64Binary?">
      <fos:arg name="in" type="xs:base64Binary?"/>
      <fos:arg name="offset" type="xs:integer"/>
      <fos:arg name="extra" type="xs:base64Binary?"/>
    </fos:proto>
  </fos:signatures>
  <fos:summary>
    <p>The <code>bin:insert-before</code> function inserts additional
      binary data at a given point in other binary data.</p>
  </fos:summary>
  <fos:rules>
    <p>Returns binary data consisting sequentially of the data from
     <code>$in</code> up to and including the <code>$offset - 1</code>
     octet, followed by all the data from <code>$extra</code>,
     and then the  remaining data from <code>$in</code>.</p>
    <p>The <code>$offset</code> is zero based.</p>
    <p>The value of <code>$offset</code>
      <rfc2119>must</rfc2119> be a non-negative integer.</p>
    <p>If the value of <code>$in</code> is the empty sequence,
     the function returns an empty sequence.</p>
    <p>If the value of <code>$extra</code> is the empty sequence,
     the function returns <code>$in</code>.</p>
    <p>If <code>$offset eq 0</code> the result is the binary
     concatenation of <code>$extra</code> and <code>$in</code>,
     i.e. equivalent to <code>bin:join(($extra,$in))</code>.</p>
  </fos:rules>
  <fos:errors>
    <p><bibref ref="error.indexOutOfRange"/> is raised if
      <code>$offset</code> is negative or <code>$offset</code> is
      larger than the size of the binary data of <code>$in</code>.</p>
  </fos:errors>
  <fos:notes>
```

[15]The File module, with a genesis some 2 years earlier than Binary, contains the function definitions as narrative text (albeit of regular form) in the specification itself. Fortunately a 70-line XLST program can attempt some reasonable reverse-engineering.

```
      <p>Note that when <code>$offset gt 0 and $offset lt
        bin:size($in)</code> the function is equivalent to:</p>
      <eg>bin:join((bin:part($in,0,$offset - 1),
                    $extra,bin:part($in,$offset)))</eg>
    </fos:notes>
  </fos:function>
```

The specification generation tools (XSLT stylesheets from W3C, with minor additions) can generate a function definition section from such a declaration, which can be requested from the main specification via a processing instruction (`<?function bin:insert-before?>`). Thus for example the groupings and order of such functions in the final document (or even whether the function is to be presented at all) is separated from the actual definition of the function itself.

Whilst this declaration was originally written for use in the specification, its utility is potentially much wider. It can provide data for an online reference (as Saxon's documentation does) or auto-hinting in editors. Simple XSLT tools can collect signatures, or sets of error codes, or generate empty templates for test sets. It can even be referenced (as a signature) from another specification that suggests the use of the given function for some compound purpose.

5.2. Tools help you rework

Despite all the measures outlined earlier, there will always be some rework necessary. With a bit of forethought and some very modest tools, the effort required for such reward can be minimised and the flexibility of the development components (spec., tests, implementation) increased. Here is a very simple example from the test suite whose final format will be QT3:

```
<expand name="binary-to-octets"
         function-name="to-octets" prefix="bin">
  <created by="John Lumley" on="2013-07-18"/>
  <environment ref="binary"/>
  <test-case>
    <description>Octets from a zero-length binary</description>
    <test> $FUNCTION(xs:base64Binary("")) </test>
    <result>
      <assert-empty/>
    </result>
  </test-case>
  <test-case>
    <description>Generate octets from a 4-length</description>
    <created by="Jirka Kosek" on="2013-10-06"/>
    <test> $FUNCTION($man.base) </test>
    <result>
      <all-of>
```

```
          <assert-type>xs:integer*</assert-type>
          <assert-deep-eq>(77,97,110)</assert-deep-eq>
        </all-of>
      </result>
    </test-case>
    ...
  </expand>
```

This will end up being expanded into test cases in QT3 format such as:

```
<test-case name="binary-to-octets-002">
  <environment ref="binary"/>
  <description>Octets from a zero-length binary</description>
  <created by="Jirka Kosek" on="2013-10-06"/>
  <test> bin:to-octets($man.base) </test>
  <result>
    <all-of>
      <assert-type>xs:integer*</assert-type>
      <assert-deep-eq>(77,97,110)</assert-deep-eq>
    </all-of>
  </result>
</test-case>
```

The substitutions involved are extremely trivial and perhaps a good re-factoring editor could make the changes. But the flexibility shown here is i) the actual name for the function can be altered in just one place, ii) common elements within the tests (environment reference, base test name...) are defined *just once*. Code necessary to achieve the expansion is simple - as all data is an XML tree, XPath accessors such as `$common[not(name() = current()/*/name())]` will select all those elements in `$common` (the common elements of the `expand` parent) that are not overridden in the specific test-case.

There are many other cases where simple tools operating on declarative descriptions can increase flexibility. In some cases it can even be worthwhile developing a generic macro processor to assist, such as one that supports buried XSLT pull-trees (e.g. `for-each select="2,4,8">.....`)

6. Be realistic – we haven't got all day

We all like our creations to be useful. We also like them to be elegant, long-lived and peer-respected. Paymasters like them to be robust, high-performing, patentable and *cheap*. Some of these goals are invariably in conflict. We only have finite time and resources to refine the specification, and the most effective feedback, actual use, will only appear when real implementations are available. So we need to consider priorities and focus early effort where absolutely necessary (e.g. firming core functions), or which will be cost-effective in the medium term (interpreting declar-

ative representations). But some requirements may have a fundamental conflict with other features of the application environment, such as the purity of the target language.

6.1. Pragmatism vs Purity

Programming languages vary in their theoretical purity from *ad hoc* affairs, such as BASIC or Perl, through to languages that are really frameworks of mathematical declarations and theorems. Extensions in the forms described in this paper can lead to tensions with the underlying semantics of the base language, especially with those of higher theoretical purity. In the case of the binary module and XSLT/XQuery, as the functions are totally pure (i.e. have no side-effects at all), these extensions do not compromise the functional nature of the underlying language.

But we don't have to stray far to find such tension appearing even in something this innocuous. The Binary module provides no facilities to read or write binary data to or from files – for this it relies on other libraries, most notably the EXPath sibling File module[3] which provides three functions, `file:read-binary()`, `file:write-binary()` and `file:append-binary()`. The last two of course *do* have side-effects – it's their sole purpose, to write a file in the outside world.

Now we get to the nub of the tension in this case - suppose we have a program that is creating a PDF file from fragments of SVG[16] by generating sections of binary data for each graphics child of an `svg:svg` element and then appending each result into an output document:

```
<xsl:template match="svg:path" mode="create-pdf" as="xs:base64Binary">...

<xsl:template match="svg:rect" mode="create-pdf" as="xs:base64Binary">...
...
<xsl:variable name="pdf" as="xs:base64Binary*">
   <xsl:apply-templates select="svg:svg/*" mode="create-pdf"/>
</xsl:variable>

...
  <xsl:sequence select="
     for $p in $pdf.parts return file:append-binary($uri,$p)"/>
```

The `file:append-binary()` is used here to accumulate a result by parts within a single file. But there is no requirement that the `for` expression evaluates for each of its iterative values in temporal sequence (as long as the order of the result, which in this case for each is an empty sequence, is preserved). So theoretically the order of pieces in the output document (and hence the apparent draw order in the resulting PDF) may not follow that of the input `svg:*` graphics pieces. This is a case where

[16] There would be more indexing required, but it illustrates the point.

there has to be considerable (higher-order?) early discussion on the approach to take[17].

6.2. Focus on the core

Comments such as "could we have a pattern-directed number-decoder?" are a two-edged sword. On the positive side it shows enough of an interest from a potential user of the library to make such a suggestion. But on the other hand it deflects focus from the core issue – defining the fundamental functions that *must* be extensions, and getting them right. These core functions operate within an environment that has complete computational capability, so a useful approach is to sketch out how such a 'bell-and-whistle' could be written as an XPath/XQuery/XSLT package using the already-defined minimally-required functions[18].

But such minimalism should be tempered with common sense. (In the extreme for the Binary module only two core functions are absolutely necessary: `bin:to-octets()` and `bin:from-octets()` – everything else can be computed in XPath around sequences of bytes, but performance and readability would plummet.) So simple convenience functions that may make code much clearer and will evidently cost little to implement when other fundamental functions are supported should be tolerated. For example, under non-error conditions, the function `bin:pad-left($in,$pad-length,$pad-octet)` is equivalent to:

```
bin:join((bin:from-octets((1 to $pad-length) ! $pad-octet), $in))
```

This is a pretty simple substitution, but given i) that *pad-left* has a clearer meaning and, more importantly, ii) all the machinery for adding byte sequences together is required in any implementation of `bin:join($parts as xs:base64Binary*)`, then the support cost for `bin:pad-left()` will be minimal. Performance will be enhanced too, avoiding the iteration across the repeated padding octets.

6.3. It doesn't *have* to be perfect

This might sound like an anathema; after all we're supposed to be building robust standards. But the real objective is to get a *useful* specification and standard – one that is going to be *used* and then perhaps further developed after substantial experience from application developers. As such there will be times when 'enough is enough', and further bickering over small details may create a great deal of unnecessary delay and frustration. In the Binary module, after the publication of the fourth draft and provided all the necessary core functions were sound, further fea-

[17] Oh dear - here we go trying to understand *monads* again.....
[18] Preferably enlist the help of the original suggester in defining and testing such a package.

tures or alternate 'edge behaviours' could always be emulated by XSLT scripts by those keen enough[19].

7. Conclusion

This paper has taken a stroll through the finalisation of a small specification/standard, trying to extract some useful lessons. What are the most valuable to the author?

- Establish as much policy as possible early on and get it written down near the head of the specification.
- Declare as much as you reasonably can and use the computer to generate therefrom.
- Get several medium-sized examples and expose them to discussion and execution.
- Build a skeletal implementation from the start and use it to run examples, and early tests.
- Build XSLT/XPath/XQuery emulations of some suggested function, using core functionality and use it for discussion, experimentation and definition.

7.1. Acknowledgements

The author must thank several individuals in the EXPath community. Jirka Kosek started the whole thing off, originally proposing the Binary module, drawing up the first draft and commenting on subsequent re-drafts. Florent Georges helped steer the whole EXPath community and gave the author much help on dealing with its systems and tools. Many others contributed in mailing-list constructive criticism, but Christian Grün should be singled out for the depth of his contributions and being amongst the first to build an implementation to the specification, complete with a separate test suite. Finally Michael Kay needs thanks for giving me the challenge of getting a standardised binary library ready for Saxon!

7.2. *Quo vadis?*

Implementations of the Binary module are in the process of publication and will hopefully find application use. From there will undoubtedly flow issues (all minor of course!) and perhaps some suggestions for 'Version 1.1'.

"No peace for the wicked" – the author has another module (on manipulating files in archive format) to be finalised within the EXPath framework.....

[19] Packaging in XSLT3.0 will make such functional extension much easier.

References

[1] *EXPath (: Collaboratively Defining Open Standards for Portable XPath Extensions :)*. http://expath.org

[2] *Binary Module 1.0*. John Lumley. Jirka Kosek. EXPath Community Group. http://expath.org/spec/binary

[3] *File Module*. Christian Grün. Matthias Brantner. EXPath Community Group. http://expath.org/spec/file

[4] *EXSLT*. EXSLT. http://exslt.org

Publishing in Style with XML

Or, Why It's Not XSL-FO

Liam Quin
W3C
<liam@w3.org>

Abstract

This paper reviews the status of CSS for producing books, both in print and on screen, discusses W3C strategy and CSS Working Group practice for moving CSS forward, and indicates some major areas of CSS strengths and weaknesses compared to XSL-FO.

Keywords: XML, XSL, publishing, CSS

1. Introduction

The primary standard way to format an XML document has for over a decade been to transform the document (typically using XSLT) into a format-specific vocabulary called the Extensible Style Sheet Language Formatting Objects, XSL-FO and then to use an XSL rendering engine to produce PDF, RTF or other formats, primarily for print.

There are several XSL-FO rendering engines, both proprietary and open source. XSL-FO usage has been increasing over the past three to five years, at least as measured by support forum activity and conference attendance, and this increase appears to be largely as a result of publishers moving to the XML single-source multiple-output way of working: the publishers need to produce both printed books and ebooks.

At the same time that usage of software based on the XSL-FO specification has been increased, energy to enhance the specification dissipated. Rendering systems based on XSL-FO continue to be developed and evolve, but W3C in 2013 ceased formal development of the XSL-FO specification because of a lack of participation. This state of affairs might seem strange, but may indicate that a competing technology has started to come of age.

This paper gives an overview of the use of Cascading Stylesheets (CSS) to format XML documents for print, describes some commonly encountered difficult areas, and discusses W3C's position and what work is being done to help CSS take on the work that XSL-FO had been doing.

2. Not Your Grandma's CSS

Many people in the XML community (some of whom were previously, or are still, people in the SGML community) first met CSS many years ago, perhaps when the authors had the idea that if the stylesheet specified red text and the user preferred blue, the colour used should actually be purple. In 1995 CSS was intended as a very simple mechanism for styling text in a Web browser. By 1998 it had become more sophisticated and powerful, although the authors still seemed not to understand much about basic typography. But it was enough that most of XSL-FO is defined in terms of CSS properties applied to text.

If your knowledge of CSS predates 2010 or so, you should be prepared for something very different today.

3. Not Your Aunt Tillie's CSS

Although CSS was designed for simplicity, CSS for print is not by any useful stretch of the imagination simple. The specifications themselves (there are more than sixty documents that form CSS today) are complex and interact in ways that are difficult to understand. Production CSS stylesheets tend to be long and cumbersome, and can be difficult to write and to debug. hundreds or even thousands of lines of CSS is not an unheard-of size. One reviewer of a draft of this paper had 370 CSS style rules containing 986 separate properties and said "most of that is probably not used," but that was for relatively simple books and not (for example) aircraft manuals. Aunt Tillie, a hypothetical figure used in the Open Source community to represent a user who is not particularly computer-literate, would have a hard time learning CSS for print.

If you are considering moving to CSS from XSL-FO, you should be aware that although the CSS page model is simpler, there is more overall complexity. It is easier to hire staff who know enough to be dangerous with CSS than with XSL-FO but much harder to hire experts in producing print with CSS. You need to hire someone who can learn, who can debug problems, and who can listen to the document designers, and not expect to be up and running in a single day (or week).

There are a great many resources for learning CSS for Web design. There are some resources for learning CSS for EPUB 3, and there is at least one book in preparation for people producing books from CSS, but today there are more resources for learning XSL-FO. Unfortunately, the XSL-FO specification, although very clear and well-written, is complex and difficult to understand,. People today, often more familiar with HTML, JavaScript and CSS than with XML or XSL, find CSS more inviting to learn. Even the mythical Aunt Tillie won't argue with motivation.

4. Moving from XSL-FO: The Models

An XSL-FO document has two main parts, which can be (and often are) interleaved: the definitions, including "page masters," and the actual text. The text is an XML document in which every element is in the XSL-FO vocabulary and has explicit CSS styles: fo:inline font-weight="bold".

In CSS the document is in XHTML (you can apply CSS to arbitrary XML in principle but in practice it had better not be too arbitrary, and sticking to XHTML puts you back in documented, charted waters); the stylesheet is in a syntax inspired by languages like C and AWK, languages which also inspired Java and JavaScript and feel comfortable and familiar to most programmers today.

5. Sample Difficulties

This section describes some difficulties one might encounter when moving from an XSL-FO perspective on formatting to using CSS-based products. Although such a list cannot be exhaustive it can be illustrative. The examples have been chosen to try to show how different ways of thinking are needed, how different classes of problem emerge, and how some XSL-FO functionality is not yet available.

6. Pages

6.1. Defining And Numbering Pages

XSL-FO has a "page master" model in which you define a template to be instantiated for each page; the template contains "regions" into which the content will flow. CSS does not have page templates. There are page rules, but these are very limited. Example 1 shows how a simple page rule can define a paper size.

Example 1. Defining a Page

```
@page {
    width: 21cm;
    height: 29.7cm;
    /* can also just say A4 or letter,
     * but the list of possible names is short */
 }
```

6.2. Areas Within a page

A page in CSS has sixteen named areas surrounding it that are used for running headers and footers on the top and sides. This is more than XSL-FO's simple page

master, but the formatting of these page areas is not precisely defined and it is not currently possible to have multiple flows on the same page.

The CSS page regions have names like top-left and bottom-right. Example 2 shows one way to number pages. The "page" counter is automatically incremented at the start of each page, and does not need to be declared; user-defined counters are also available, and the sanity of the stylesheet writer depends on declaring those counters and initializing them to zero. The example also gives a simple example of "inheritance": the page size is defined in a general rule for every page and then specialized rules for left (verso) and right (recto) pages are defined which are said to "inherit" properties from the general page rule.

Example 2. Page Areas

```
@page {
  width: 21cm;
  height: 29.7cm;
}
@page:left {
    @top-left {
        content: "Page " counter(page);
    }
}
@page:right {
    @top-right {
        content: "Page " counter(page);
    }
}
```

As with XSL-FO, page definition proliferation is a problem, since you end up with left, right, blank and first variants of page definitions for prelims, table of contents, introduction, chapter, appendix, foldout, and so on and on. Use of a pre-processor can mitigate this, but whereas XSL-FO is designed for use with XSLT, there is no standard preprocessor for CSS, and most or all of the existing ones seem to be aimed primarily at on-screen scrolling use rather than print or paged media use. This situation is changing.

The author of this paper has experimented with CSS preprocessors written in XSLT, but they are rather specialized tools.

A common difficulty with page numbering in CSS is that when an element switches to a new page rule or resets a chapter or page counter the footer on the page with the transition might have the new page number but in the old (previous section's) format. A way around this seems to be to nest elements: put an HTML div element around each section element to separate the page break from the page style transition.

6.3. Static Page Content

In XSL-FO if you want some text to appear on every page you put it in the page master. In CSS you can't do this: the text has to come from an element in the document. You can supply a background image for each page using a stylesheet, but you often need more.

Currently the solution is to take an element from the document and arrange for it to have fixed positioning, which is taken (arbitrarily) to mean that it should appear on every page until the content of its containing element has been exhausted.

You can also use the content of an element in a running header or footer, but today if you do that the element is deleted from the main content, so if it's a title that should appear in the document you have to duplicate it.

Finally, you can save the content of an element in a string and use that in a running header or footer; in that case, arbitrarily, the element is not deleted from the main flow, but of course contained markup is not carried through into the running header.

7. After the first page

The lack of page templates (currently) in CSS means that formatting has to be associated with elements in the document. To some extent that's also true in XSL-FO, and the clearest comparison is thus between the HTML document and the FO document, even though it was not the intention of the XSL Working Group that FO documents be shared or authored directly. In this comparison we see that items such as a table of contents would in the XSL-FO case be generated (often by XSLT) but in the HTML case would have to be part of the document. The most common answer here is to generate the XHTML with XSLT from an XHTML or other XML master document. The formatting examples in this section, then, assume that either you're willing to put format-specific details into your XHTML or you're generating the XHTML from another document. An emerging possibility is using JavaScript to generate a table of contents; JavaScript support in commercial print-quality CSS formatters is still very new and experimental at the time of writing, however.

7.1. Tables of Contents

It might seem reasonable that different presentations of a document have different tables of contents. A book for study might have a fully detailed table of contents, with annotations about each section, and a book intended for casual reading (or a cheaper edition saving money by having fewer pages) might have a simpler table of contents.

Example 3. XHTML for Table of Contents

```
<ul class="toc">
  <li><a href="#chapter1">Chapter 1</a></li>
  <li><a href="#chapter2">Chapter 2</a></li>
</ul>
```

Markup for a table of contents typically treats each entry in the table of contents as a list item in an HTML list; CSS is then used to disable the list behaviour of the bullets and indents; CSS can be used to insert dot leaders for an old-fashioned (or corporate) appearance, although there is no straight-forward way to control the spacing or size of the leaders.

Example 4. CSS for Table of Contents

```
ul.toc {
  list-style: none;
  display: block; /* as opposed to list */
}

ul.toc li {
  list-style: none;
  display: block; /* as opposed to list-item */
}

ul.toc li a {
  /* don't want blue underlined links */
  color: black;
  text-decoration: none;
}

ul.toc lia a:after {
  /* after the "a" element comes the dot leader and page */
  content: " " leader(dotted) " " target-counter(attr(href, url), page);
}
```

This idea of changing an HTML list to be something else vaguely list-like is very widespread in the HTML world, and, remarkably, is not considered tag abuse. Some people might say that a table of contents is an "ordered list" although in an interactive environment a reader might want to see it sorted by chapter title rather than by page. A table of contents is a presentation of a set of relationships.

7.2. Units and Expressions

In XSL-FO every value is really an expression, so to allow for a 2pt border you can write `width="3in - 2pt"`. With CSS that's a syntax error because values are not expressions.

Luckily, CSS has added `calc()`, a function that takes a simple expression language so you can now write calc(3in - 2pt); watch that you need spaces around the subtraction sign, as in XPath.

7.3. Selectors and Specific Gravity

Once you have more than a few hundred lines of CSS you are almost certain to run into a common problem: you add a new style rule and an old one stops working, or the new rule is ignored, or it doesn't do what you expected.

The reason is that CSS chooses the *most specific* selector expression when determining which style rule to apply. Worse, *multiple* style rules can be combined automatically by the CSS renderer to format a single element. The intent was two-fold: to support a sort of object-oriented inheritance that makes style sheets smaller and easier to manage (as shown in the @page example), and also to allow end users to write their own CSS to override aspects of remote Web sites they don't like. The attempt largely failed on both counts. Most users do not write their own style sheets (Web browsers are in the process of removing the menu items to give users access to adding their own styles, as it's so rarely used). The inheritance turns out to be confusing enough that people are more likely these days to use a CSS preprocessor to manage it, although it can still be helpful when used with care.

If inheritance in CSS was not all it was hyped to be, no matter: many technologies have been over-hyped. But we are left now with the problem that adding a new stylesheet rule can affect others in ways that are hard to predict. Unfortunately there is no way to override specificness of a selector, nothing like `priority="6"` with XSLT.

People working on CSS for print have communicated to me that working out exactly which style rules are being applied and why takes up a significant amount of their time. it may therefore be worth noting that the most effective debugging tool the author has found for this situation is to open the document in the Chromium Web browser (or any Webkit-based browser) and to use the Inspect Element feature; the resulting window shows which CSS selectors and rules contributed to the styling of the element instance in question. Although Chromium does not at the time of writing support paged media CSS properties, it does still show them in its debugger. The Firefox debugger is less useful in this regard at the time of writing, but the important point is to try opening the HTML file, with its CSS for print, in a Web browser, and, even though it won't look good, use the Web debugging tools.

7.4. Baseline Positioning and Boxes

Neither XSL-FO nor CSS guarantee exact positioning of text baselines. You can position a *box*, but the position of the baseline relative to the edge of the box depends on the actual font used, and that's up to the formatter.

For print use, we can usually control the font that is being used and, perhaps through trial and error, determined the distance from the baseline of the text to the bottom (or top) of the CSS content box. On the Web you can't do that: a user might have Web fonts turned off, or might have specified their own fonts. For the purpose of this paper, we are considering primarily printed applications and formatting to PDF, so fonts are a known quantity. However, CSS still cannot reliably align text in two adjacent columns of text by baseline, resulting in a distracting untidiness at the foot of the columns and, potentially, show-through in print.

7.5. Indexes

An index at the back of the book (the correct English plural is indexes; indices are constructs found in mathematics) is an important tool for readers. Established conventions help readers use an index effectively. Formatting in index requires collapsing ranges of page numbers, something that cannot be done until after pagination.

CSS does not currently have sufficient mechanisms for making such an index; XSL-FO has fairly sophisticated support for it. There have been some experiments in proprietary tools such as PrinceXML, and we may expect to see advances in this area soon.

7.6. Tables

Although HTML tables could in principle repeat headers and footers on each page, in practice the support is vestigial; XSL-FO tables are much more robust than HTML tables today. CSS also does not treat table notes differently from regular footnotes, a requirement in much military and engineering publication work.

8. Why so Glum?

When people move from one technology to another there's a learning curve; new things can seem difficult and require unfamiliar thought patterns. This paper has given some examples to try and illustrate how an XSL-FO user will need to change the way they think about formatting, at least a little, to work with CSS. But there are also strengths in CSS.

8.1. Sharing Styles

Since many print publications today must also be made available as electronic books and often Web sites, being able to share the formatting can considerably reduce costs and time to market. Even where there are differences in the way certain elements (such as images or large tables) are handled, there is likely to be a large commonality of style in everything from cross references through to superscripts. Web sites and most newer digital book formats use HTML and CSS, so using HTML and CSS for the printed book makes sense in such an environment.

It should be noted that the author of this paper is not advocating the use of HTML for archival content or for working with complex publications. An XML format that models the information directly should be used where possible; some publishers are also using a highly constrained HTML vocabulary such as HTMLBook, with added metadata, alhough it remains to be seen how well this approach will stand up to the needs of archivists.

8.2. Sharing Style Rules

The inheritance feature of CSS mentioned earlier can be difficult to work with if you are not familiar with CSS, and can cause surprises even years later, but it can also help to support significant reductions in complexity. As with many programming tools, CSS cascading and HTML class attributes should be used with some care and understanding; they can be more powerful than XSLT attribute sets and XSL-FO inheritance because of the way individual properties can be overridden all the way up the inheritance chain, but in that sense "goto" is more powerful than "while". CSS inheritance is powerful and useful when used well.

8.3. Rapid Development

The ability to preview results in a Web browser, and to use the Element Inspector to *change* style properties temporarily, is very powerful. Although similar environments have existed for SGML and XML stylesheet systems in the past they are not widely available today.

8.4. The In crowd

There can be no doubt that CSS is attracting many more resources and developers than XSL-FO today. Use the right tool for the job, but if either tool will do, use the one that's going to be maintained into the future.

8.5. Communicating with the Young and Hip

Web designers, wither young or old, must through necessity have at least a working knowledge of CSS in both its strengths and its weaknesses. Since it would be hard to graduate any sort of graphic design course today without at least *some* exposure to Web design, using CSS means you have a common language with the designer, and also means you are likely to be able to do the things the designer wants and that you won't be asked to do things you can't do.

9. W3C and CSS for Publishing

The W3C staff have taken an initiative to try to get more people from the world of professional publishing involved with the CSS Working Group, and any other relevant areas. To this end a new Publishing Activity has been formed, and a Digital Publishing Interest Group has started, co-chaired by Marcus Gyling from IDPF, the organization responsible for the EPUB specification widely used in digital books.

9.1. The W3C Publishing Activity

The W3C Publishing Activity was launched officially in the Summer of 2013. It provides a focus for W3C's work with the professional publishing community, and will also be where academic publishing and other publishing-related topics are discussed.

9.2. The Digital Publishing Interest Group

Within the Digital Publishing Activity, the Digital Publishing Interest Group is a W3C Member-only group gathering requirements for standardization. Several publishers are participating, as well as industry experts, digital book vendors, accessibility and internationalization experts and others. They are working actively on a document describing Western (Latin-script) typographic requirements from a professional digital publishing perspective, so that the CSS Working Group, and any other Working Groups as appropriate, can have a reference that's focused on providing the information they need. The scope is roughly comparable to the Japanese Layout Requirement Document that has already been produced.

The Interest Group also has a close relationship with IDPF, the organization responsible for the EPUB specification widely used in digital book readers.

9.3. Working With CSS

The Digital Publishing Interest Group includes expertise in a wide range of areas, and is working closely with the CSS Working Group to suggest, emphasize and

prioritize features; the Generated Content and Paged Media CSS draft now has a new editor who is also active in the Interest Group, and that document is moving forward rapidly.

The CSS Working Group itself has been very receptive to the input from the publishing community (both for ebooks and for print); this is a major step forward in itself, and bodes well for the future of CSS in publishing.

10. Conclusions

Cascading Style Sheets have come a long way in the past few years, and when used with a suitable print formatter are now a credible way to produce printed content. CSS is still along way behind XSL-FO for print, although of course it is a long way ahead for Web development and interactive work, and does have many features not present in XSL-FO. Work is ongoing at W3C to bring CSS to the level of XSL-FO and beyond for print work. Today, most Web browsers do not have good support for "paged media" and separate rendering engines are needed for good quality (or acceptable) print; whether this will change remains to be seen.

Formatting from XML

Tony Graham
Mentea
<tgraham@mentea.net>

Abstract

Formatting from XML is in freefall. On one hand, XSL-FO standardisation quietly died even as XSL-FO usage is on the increase, while on the other hand, CSS is moving to standardise properties for paginated media yet its pagination spec has been forked and Liam Quin, XML Activity Lead at W3C, says "I hope that CSS catches up with XSL-FO over the next two or three years." [28] Recently, the W3C also created a Digital Publishing Interest Group [14] that, starting from a wide-ranging mission as "a forum for experts in the digital publishing ecosystem of electronic journals, magazines, news, or book publishing (authors, creators, publishers, news organizations, booksellers, accessibility and internationalization specialists, etc.)... to align the existing formats and technologies (e.g., for electronic books) with those used by the Open Web Platform", has narrowed its formatting focus to books [23] and is "currently focused on getting more publishing companies to join" [22]. So if you are not a publishing company, you are not looking to represent your data in HTML5, or you need more than CSS or XSL-FO can currently provide [24], then you are out of luck, or out of standards-based solutions, for the immediate future.

1. Comparing XSL-FO and CSS efforts

The XML Print and Page Layout WG [33] shut up shop sometime in 2012. The WG's page was updated in November 2012 to state "This WOrking [sic] Group is no longer active, because of insufficient participation. The specifications are no longer maintained." and the WG's charter expired in January 2013 without any attempt to have it renewed. In October 2013, Liam Quin stated about the WG, "We were down to three people in Working Group teleconference calls, and that was on a good day." [29] The XPPL WG charter [34] stated "To be successful, the XML Layout and Print Working Group is expected to have 6 or more active participants for its duration... to consume two hours per week for each participant, with more time being highly desirable." In contrast, the current draft of the next charter [8] for the CSS WG states "To be successful, the group is expected to have 10 or more active participants for its duration... to consume one work day per week for each participant; two days per week for editors."

For quite some time before it was shuttered, the XPPL WG had no representation from commercial formatter vendors, but nor does the CSS WG even now. However, since CSS has been operating in public for several years, it's not been necessary for the two principal CSS formatter vendors – Prince and Antenna House – be W3C members to follow the work. Even so, Prince has effectively had a proxy in the CSS WG since Håkon Wium Lie of Opera Software is also a director of Prince, and Antenna House has in the past contracted an CSS WG member to develop a CSS module spec in which they were particularly interested. However, from the perspective of the CSS WG, and particularly its chairs, the fact that the main vendors are not members of the WG raises concerns about patent policy and slowness of the communication process [13].

For a long time, the main CSS module for print publications was "Generated Content for Paged Media" (GCPM) [9], which covered running headers and footers, leaders, cross-references, footnotes, and other aspects of styling book-like content. It is largely the work of its editor, Håkon Wium Lie. The CSS WG resolved at a face-to-face meeting in September 2013 to move parts of GCPM to other specs, including parts to a new "Page Floats" spec, and publish the remainder as a new Working Draft. However, before any new drafts were published, Håkon Wium Lie announced in October 2013 that the GCPM and Page Floats specs had moved to WHATWG [32] under the names "Books" and "Figures". In a CSS WG telecon two days later [10], Håkon stated that moving to WHATWG provided "the atmosphere to work more easily" and also that he would "set off, write spec, write tests, and come back in January." Since then, the CSS WG resolved to appoint David Cramer of Hachette Livre as editor of GCPM [11], so there's now potentially two versions of the specs. As of 19 January 2014, "Books" and "Figures" were last updated on 15 January 2014 and 6 January 2014, respectively, while the W3C versions have not been updated since late September 2013.

2. W3C Digital Publishing Interest Group

The DPUBIG has the stated mission "to provide a forum for experts in the digital publishing ecosystem of electronic journals, magazines, news, or book publishing (authors, creators, publishers, news organizations, booksellers, accessibility and internationalization specialists, etc.) for technical discussions, gathering use cases and requirements to align the existing formats and technologies (e.g., for electronic books) with those used by the Open Web Platform.", but as stated in the abstract, they have reduced their current scope to books and are focusing on getting more publishing companies to join the IG, which for most publishers means they need to first join the W3C.

The DPUBIG currently divides its work into eight task forces:

- Latinreq (includes typesetting)

- Page DOM
- Metadata
- Behavioral Adaption
- Annotation
- MathML, STEM
- Security
- Accessibility
- Bridging Offline and Online

The DPUBIG is due to present the first draft of its technical issues to other W3C Groups in January 2014. Since the work is now divided into task forces in a way unanticipated in the charter, the completeness or otherwise of the work will vary across task forces.

Somewhat separately to the work of the IG members, the W3C conducted three Digital Publishing Workshops – February 2013 in New York [15], June 2013 in Tokyo [16], and September 2013 in Paris [27] – that were open to anybody submitting a position paper.

As can be seen from their titles, many of the task forces are specific to electronic books, with really only 'Latinreq' being directly applicable to the page layout aspects of both electronic and dead-tree publishing.

'Latinreq', short for "Requirements for Latin Text Layout and Pagination", "describes requirements for pagination and layout of books in latin languages, based on the tradition of print book design and composition." [23] It is inspired by "Requirements for Japanese Text Layout" [21] produced for Japanese.

The DPUBIG itself is not chartered to produce Recommendations, but, rather, it is to work with other W3C Working Groups "to ensure that the requirements of this particular community are met", so Latinreq is meant mainly as input for the CSS WG. Dave Cramer of Hachette Livre is the Latinreq task force leader, editor of the document, and editor of the CSS GCPM spec, so there should be good cooperation between the DPUBIG and the CSS WG in this matter.

As of 19 January 2014, the Latinreq draft still contains many sections that are just a heading with no other content, but it also has had content added in several sections since the first draft of this paper in early December 2013. The document currently mostly features examples only from narrative texts and includes, for example, a section on optimising the number of pages in a book to be some multiple of eight, sixteen, or thirty-two pages resulting from the binding process.

The current draft also contains several examples of how to use CSS to achieve effects such as running heads. These are due to be removed, but the description of a running head containing the author name, the page number, and an ornament as "quite complex" and needing proprietary extensions illustrates the difference in expectations for CSS pagination compared to XSL-FO pagination.

3. Can you typeset a book with CSS?

That is the title of the talk [2] by Bert Bos, co-inventor of CSS [1], at the 'eBooks & i18n' workshop in Tokyo in June 2013. His assessment is that, no, you can't, or at least not at present. The rest of his presentation covered features that would need to be added to CSS, but provided no timetable for when they would be done. Portions of his concluding summary included:

- It is not possible to make books or e-books with standard CSS
- CSS isn't even up to the level of XSL-FO 1 yet.
- Adding [extensions] to CSS is going to take time

Bert is collecting his own “List of CSS features required for paged media” [3] that has been updated several times in recent months.

Dave Cramer, in his presentation [6] at the 'Publishing and the Open Web Platform' workshop in September 2013 concluded with a list of things that he'd like to see, some of which are already familiar to XSL-FO users:

- Control over pagination and line-breaking
- minimum/optimum/maximum, “break line here!”
- Styling content in margin boxes
- Apply new named pages without page-breaks
- Automation of pagination
- XPath-strength CSS selectors

as well as some that aren't.

4. Can you typeset every book with XSL-FO?

No [24]. Even if the XPPL WG had delivered on all of the XSL-FO 2.0 Requirements [23] it still wouldn't be possible to handle every possible book design.

The problem isn't so much that you can't typeset every book with XSL-FO, it's that you can't use open standards[1] to typeset ever more complex books – or have better ways to typeset books – until such time as CSS catches up and passes what you can currently do with XSL-FO.

As noted above, a running head containing the author name, the page number, and an ornament is seen as “quite complex” for CSS, and Bert Bos's Tokyo presentation discusses how something as simple as an equation in a running header is beyond CSS at present, so Liam Quin's “over the next two or three years” may be politely optimistic.

However, there doesn't seem to be any alternative to having to wait – patiently or impatiently – for CSS to catch up:

[1] Though there's always LaTeX [17].

- The XPPL WG would only be rechartered if there were several dues-paying W3C members willing to provide the majority of the members and task them to do the work and implement the result, but even then it would be difficult to restart the WG in the face of the W3C's emphasis on the Open Web Platform [25] and its running a questionaire on the branding of, not the technology of, CSS [7].
- The Print and Page Layout Community Group [26] at the W3C would be the next-best place to advance XSL-FO, especially since it has permission [30] to host its own version of the XSL-FO spec and ambitions about other things to work on [28], but it has few members and even fewer active contributors, and not even the periodic rumblings about XSL-FO 2.0 on the XSL-List [31] bring new members. Even if the CG did produce a new spec, it would need a functioning WG to take the spec to Recommendation level, which would also require multiple interoperable implementations.

The impatient approach, therefore, would be to engage with the DPUBIG and the CSS WG to help advance their specs and the CSS implementations of their specs. The XSL-FO processor vendors are moving to also support CSS. Some may do it by mapping CSS to XSL-FO 'under the hood' [5], but as the models diverge and CSS incorporates grids, arbitrary shapes, and other features, it will require more than simply mapping CSS to XSL 1.1.

5. Effect on transformations

XSLT started out as the means for transforming arbitrary XML into the XSL Formatting Objects vocabulary. The CSS model, on the other hand, does not re-order content. Even when CSS can do everything you can do with XSL-FO, formatting of arbitrary XML will still require some sort of transformation to re-order, duplicate, or modify content for display.

Bert Bos, in his position paper for the September 2013 workshop [4] notes:

> *CSS does not support document transformations, such as those provided by XSLT. That is to keep CSS easy to understand and use, and to better support WYSIWYG editing of documents. But even so, it could in theory do much more than it does now.*

but he may currently be largely alone in thinking that CSS could do with a transformation capability.

If there was a transformation language for CSS, it would not be based on XPath since the W3C is “moving from XPath to Selectors W3C-wide” [12], yet Jirka Kosek had trouble [12] getting acceptance of a proposal to add a CSS selector for selecting

attributes[2] for use with ITS [20], and no success in proposing that CSS selectors be extended to be able to select all (XDM) node types.

If CSS does get a transformation capability but remains unable to select all node types and unable to use complex predicates or other XPath features, then users of arbitrary XML would have to use a XSLT-CSS hybrid or use a XSLT preprocessor as part of formatting their documents.

Bibliography

[1] Bert Bos. http://www.w3.org/People/#bbos

[2] Bos, Bert, Can you typeset a book with CSS?. http://www.w3.org/Talks/2013/0604-CSS-Tokyo/

[3] Bos, Bert, List of CSS features required for paged media. http://www.w3.org/Style/2013/paged-media-tasks

[4] The limits of "single-source publishing" with XML and CSS. http://www.w3.org/2012/12/global-publisher/statements-of-interest/18-bert-bos-0020.html

[5] Re: [xsl] xsl 2.0?. http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201311/msg00010.html

[6] The Exotic World of Trade Publishing Part One: Culture and Workflow. http://www.w3.org/2012/12/global-publisher/slides/Day2/P1-w3c-paris-hachette.pdf, slide 48

[7] CSS Branding. https://www.w3.org/2002/09/wbs/1/cssbranding/

[8] Cascading Style Sheets (CSS) Working Group Charter. http://www.w3.org/Style/2013/css-charter

[9] CSS Generated Content for Paged Media Module. http://www.w3.org/TR/css3-gcpm/

[10] Minutes Telecon 2013-10-16. http://lists.w3.org/Archives/Public/www-style/2013Oct/0460.html

[11] Minutes TPAC F2F 2013-11-10 Sun I: Agenda, GCPM, Canvas and Video and CSS Image, and Device Pixel Ratio. http://lists.w3.org/Archives/Public/www-style/2013Nov/0349.html

[12] Minutes TPAC F2F 2013-11-11 Mon I: display: none on Fragmentainers, Selecting Attributes. http://lists.w3.org/Archives/Public/www-style/2013Nov/0358.html

[2] Some of the support may have been to "take the few small steps necessary to kill most remaining uses of XPath." [12]

[13] Minutes Paris F2F 2013-09-12 III: GCPM. http://lists.w3.org/Archives/Public/www-style/2013Sep/0876.html

[14] Digital Publishing Interest Group. http://www.w3.org/dpub/IG/wiki/Main_Page

[15] eBooks: Great Expectations for Web Standards. http://www.w3.org/2012/08/electronic-books/

[16] eBooks & i18n: Richer Internationalization for eBooks. https://www.w3.org/2013/06/ebooks/

[17] Re: [xsl] xsl 2.0?. http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201311/msg00038.html

[18] Example 5 - Copyfitting by adjusting 'font-size'. http://www.w3.org/community/ppl/wiki/FOPRunXSLTExt#Example_5_-_Copyfitting_by_adjusting_.27font-size.27

[19] Bals, Klaas, Extensible Stylesheet Language (XSL) Requirements Version 2.0. http://www.w3.org/TR/xslfo20-req/

[20] Internationalization Tag Set (ITS) Version 2.0. http://www.w3.org/TR/its20/

[21] Requirements for Japanese Text Layout. http://www.w3.org/TR/jlreq/

[22] Private email from W3C staff contact to author and group-digipub-chairs@w3.org, 13 November 2013

[23] Requirements for Latin Text Layout and Pagination. http://w3c.github.io/dpub-pagination/

[24] Re: [xml-dev] Re: XML As Fall Guy. http://markmail.org/message/ytp7z2dge6jhz6n7

[25] Open Web Platform. http://www.w3.org/wiki/Open_Web_Platform

[26] Print and Page Layout Community Group. http://www.w3.org/community/ppl/

[27] Publishing and the Open Web Platform. http://www.w3.org/2012/12/global-publisher/

[28] Re: [xsl] xsl 2.0?. http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201311/msg00014.html

[29] Re: [xsl] xsl 2.0? http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201310/msg00152.html

[30] Re: Prerequisites for modifying XSL 2.0 spec or producing API. http://lists.w3.org/Archives/Public/public-ppl/2013Feb/0046.html

[31] xsl 2.0?. http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201310/msg00150.html

[32] CSS Books & CSS Figures. http://blog.whatwg.org/css-books-css-figures

[33] XML Print and Page Layout Working Group. http://www.w3.org/XML/XPPL/

[34] XML Print and Page Layout Working Group Charter. http://www.w3.org/XML/2010/10/xslfo-charter

ProXist - XProc Processes in eXist

Ari Nordström
<ari.nordstrom@gmail.com>

Abstract

ProX is an abstraction layer around XProc pipelines, an XML-based blueprint that lists any processes built around XProc pipelines in a system, including the pipelines themselves, any input they might accept (XSLT, schemas, etc), as well as any configuration options the pipelines accept when run. When narrowed down to an instance, ProX describes a specific process with a specific pipeline and a specific configuration. For example, a generic print publishing process might allow choosing between several related pipelines that can use different XSLTs configured with different input parameters and other options, which need to be narrowed down to an instance with all of the choices made.

The ProX instance XML can then be used to generate a script that runs the selected process, configuring it and the resources used with any runtime values.

ProXist is a ProX implementation for eXist, run using a wrapper XQuery and accompanying pipelines. The wrapper preprocesses its input and presents the ProX blueprint to a user in an XForm, allowing the user to make choices and narrow the ProX blueprint down to an instance that, when saved, is used to generate an XQuery that runs the child pipeline represented by the selected process and configuration.

1. Intro

1.1. What Is ProX?

ProX is an attempt to define an XML-based abstraction layer around XML processing with XProc. While XProc is XML, running XProc pipelines using an actual engine involves a lot more, usually batch or shell scripts that configure the engine and whatever inputs and options, etc, that the pipeline defines, which is something of a pain. Offering the resulting configuration options to an end user in a GUI is difficult at best and a nightmare for any conscientious developer.

Enter ProX. an XML-based abstraction layer that lists all those configuration options, putting the XProc in a context. The XML is made available to a user of a document management system so she can select and configure pipelines and whatever options they have, as defined by the ProX XML, and save the configured

process as a ProX instance that is used to generate a shell script with the configurations and any runtime values included. This script then runs the configured pipeline, greatly simplifying handling the process.

ProX is probably best regarded as a *blueprint* that lists *all* available processes in the system, their associated pipelines, the command lines that configure these pipelines, including any available input files used by the pipelines and the parameters used to configure the inputs. It's a description of what is possible and the choices that need to be made before there can be a specific pipeline to run.

Let's say that one process is about delivering documents to an end user and another about reviewing said documentation in-house. Both of these would result in both print or online publishing (and more) but using slightly different options, for change bars, comments, etc, and while these are run with XProc pipelines, the processes have to be configured first. It's a top-down logic where selecting a process limits the available pipelines to those listed inside that process, choosing a pipeline limits the available command lines to those defined for that pipeline, and so on, like this:

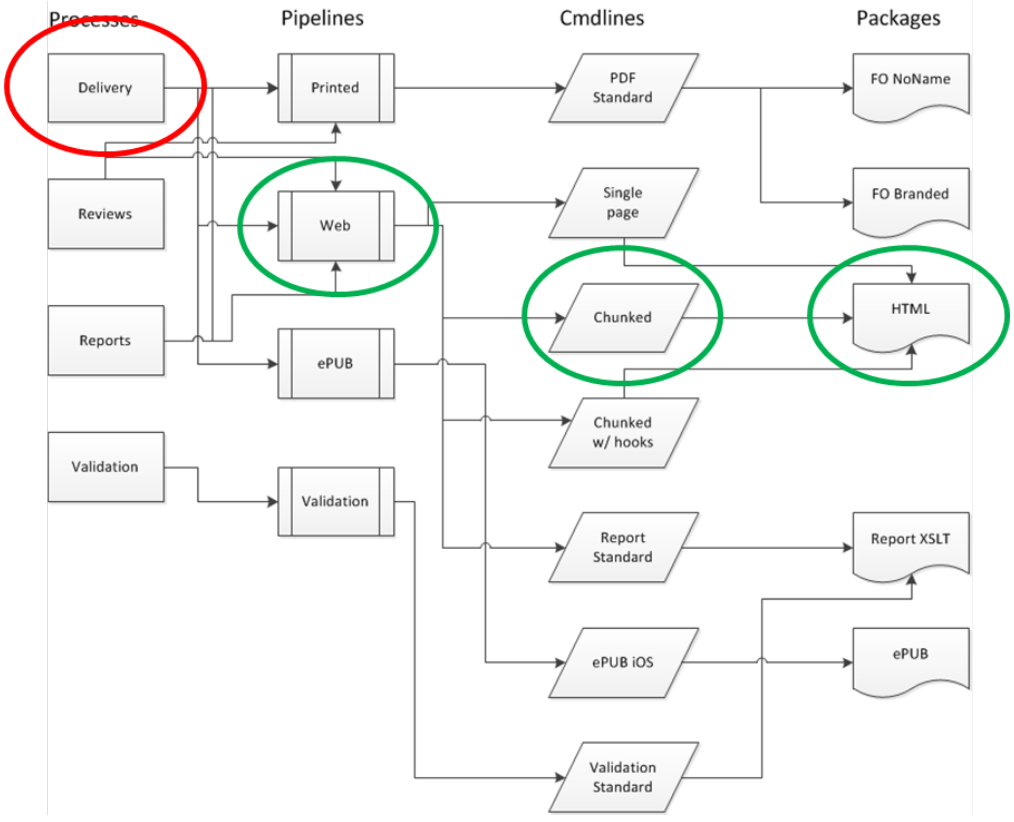


Figure 1. ProX Logic

The above image attempts to explain the available selections and what they might result in. In this case, a web-based delivery process and pipeline, and its command

line options, is selected. All of the above is defined in an XML blueprint that adheres to the ProX Relax NG schema. The structure looks like this:

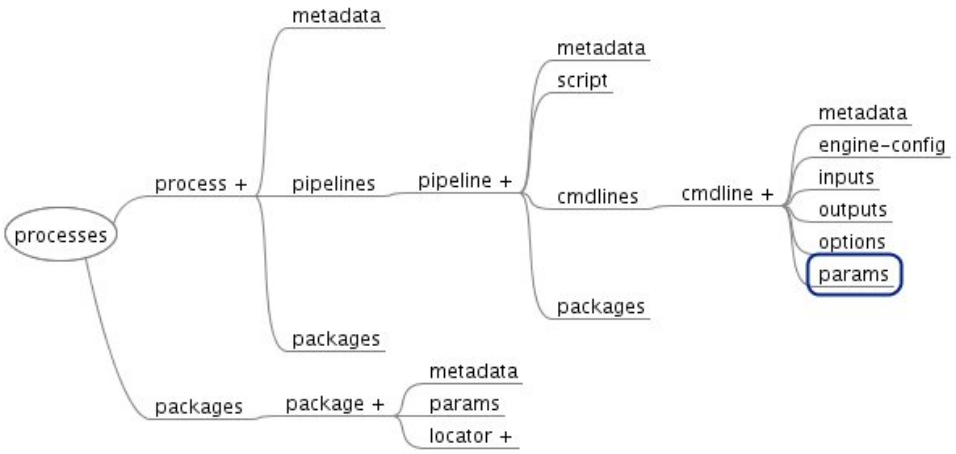


Figure 2. ProX Structure

The ProX blueprint lists one or more processes, each of which includes one or more pipelines. Each pipeline will also list command line options with any and all files used by the pipelines, from XSLT to schemas to images to everything else. As many such files are actually groups of related files (an XSLT stylesheet is very frequently a package comprising several modules), the ProX schema allows listing each and every module, in context, clearly indicating where it belongs.

The command line groups include configuration options (such as XSLT parameters or XProc options, engine configuration, and basically anything else that an XProc engine might expose to the command line).

The ProX blueprint also lists any runtime values required by a pipeline input or output port, etc. For example, the delivery process for print and web publishing requires an input XML file or files (if modular), usually a named output, and so on.

Here's a ProX instance. Note that it needs to be processed, replacing URNs with their corresponding URLs, before it can be used to generate a script. It is a complete example of a specific ProX process, however.

```
<processes>
  <process id="id-pdf-process">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-">
      <title id="title-2013-4-9-16-53-8-39562387-">Print Publishing</title>
      <description id="description-2013-4-9-16-53-8-39562387-">
        <p id="p-2013-4-9-16-53-8-39562387-">Print publishing for COSML documents</p>
      </description>
    </metadata>
    <pipelines id="pipelines-2013-4-9-16-53-8-39562387-">

      <!-- PDF Pipeline -->
      <pipeline id="id-pipeline-pdf-1">
        <metadata id="metadata-2013-4-9-16-53-8-39562387-1">
```

```
  <title id="title-2013-4-9-16-53-8-39562387-1">Publish PDF</title>
  <description id="description-2013-4-9-16-53-8-39562387-1">
    <p id="p-2013-4-9-16-53-8-39562387-1">Normalizes, validates and converts a COSML
      document to PDF</p>
  </description>
</metadata>
<script xmlns:xlink="http://www.w3.org/1999/xlink" type="pkg"
  id="script-2013-4-9-16-53-8-39562387-"
  xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-xproc-pdf"
  xlink:title=" XProc Pipeline for Normalize, Validate and PDF
               Normalizes, validates and publishes in PDF a COSML document "/>
<cmdlines id="cmdlines-2013-4-9-16-53-8-39562387-">

  <!-- COSML Internal XSL -->
  <cmdline id="id-cmdline-cos-internal-pdf">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-2">
      <title id="title-2013-4-9-16-53-8-39562387-2">COS Internal Template</title>
      <description id="description-2013-4-9-16-53-8-39562387-2">
       <p id="p-2013-4-9-16-53-8-39562387-2">Configures the pipeline for the "COS Internal"
          template</p>
      </description>
    </metadata>
    <engine-config>
      <config xmlns:xlink="http://www.w3.org/1999/xlink" type="pkg"
        xlink:href="#id-conf-calabash"/>
    </engine-config>
    <inputs id="inputs-2013-4-9-16-53-8-39562387-">
      <input choice="no" id="input-2013-4-9-16-53-8-39562387-">
        <port id="port-2013-4-9-16-53-8-39562387-">document</port>
        <value xmlns:xlink="http://www.w3.org/1999/xlink" type="external"
          input-type="doc-root" xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-"
          mimetype="application/xml">DOCUMENT-PLACEHOLDER</value>
      </input>
      <input choice="no" id="input-2013-4-9-16-53-8-39562387-1">
        <port id="port-2013-4-9-16-53-8-39562387-1">stylesheet</port>
        <value xmlns:xlink="http://www.w3.org/1999/xlink" type="pkg"
          xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-xslfo-cosml"
          xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-1"
          xlink:title=" XSL-FO Package for COSML PDF Converts COSML
                        documents to XSL-FO format for COS PDF layout "/>
        <params id="params-2013-4-9-16-53-8-39562387-">
          <!-- Index generation -->

          <!-- XEP Extensions -->
          <param choice="no" id="param-2013-4-9-16-53-8-39562387-1">
            <port id="port-2013-4-9-16-53-8-39562387-3">xslt-params</port>
            <name id="name-2013-4-9-16-53-8-39562387-1">xep.extensions</name>
            <value xmlns:xlink="http://www.w3.org/1999/xlink" type="string"
              xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-3">0</value>
          </param>
          <!-- XSL-FO Bookmark Generation -->

          <!-- TOC Generation -->

          <!-- TOC Depth -->
          <param choice="yes" ctype="list1" id="param-2013-4-9-16-53-8-2385485-2"
            group="value-2013-7-10-16-53-8-764625737-3">
            <port id="port-2013-7-10-16-34-8-9283444-4">xslt-params</port>
```

```
                <name id="name-2013-7-10-16-50-3-1946564-2">toc.depth</name>
                <value xmlns:xlink="http://www.w3.org/1999/xlink" type="string"
                  xlink:type="simple" id="value-2013-7-10-16-53-8-764625737-4">2</value>
                <value id="value-13-07-10-12345-1" type="string">1</value>
                <value id="value-13-07-10-12345-2" type="string">3</value>
              </param>
            </params>
          </input>
          <input choice="no" id="input-2013-4-9-16-53-8-39562387-2">
            <port id="port-2013-4-9-16-53-8-39562387-5">stylesheet-norm</port>
            <value xmlns:xlink="http://www.w3.org/1999/xlink" type="pkg"
              xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-normalize"
              xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-5"
              xlink:title=" Normalize XSLT Stylesheet for applics filtering
                                and module normalization for COSML documents "
            />
          </input>
        </inputs>
        <options id="options-2013-4-9-16-53-8-39562387-">
          <option choice="no" id="option-2013-4-9-16-53-8-39562387-">
            <name id="name-2013-4-9-16-53-8-39562387-3">pdf</name>
            <value xmlns:xlink="http://www.w3.org/1999/xlink" type="external"
              output-type="primary" mimetype="application/pdf" xlink:type="simple"
              id="value-2013-4-9-16-53-8-39562387-6">PDF-PLACEHOLDER.pdf</value>
          </option>
        </options>
      </cmdline>

      <!-- COSML Formal XSL -->
    </cmdlines>
  </pipeline>
</pipelines>

<!-- Packages for Print -->
<packages xml:base="file:///e:/SGML/DTD/Cassis/Process-XML/"
  id="packages-2013-4-9-16-53-8-39562387-">

  <!-- XProc Normalize, Validate, XSLFO Pipeline Package -->
  <package id="id-xproc-pdf">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-4">
      <title id="title-2013-4-9-16-53-8-39562387-4">XProc Pipeline for Normalize, Validate and
        PDF</title>
      <description id="description-2013-4-9-16-53-8-39562387-4">
       <p id="p-2013-4-9-16-53-8-39562387-4">Normalizes, validates and publishes in PDF a COSML
          document</p>
      </description>
    </metadata>
    <!-- publish-cosml-pdf.xpl -->
    <locator xmlns:xlink="http://www.w3.org/1999/xlink" type="main"
      xlink:href="urn:x-cassis:r1:cos:00002715:sv-SE:0.1"
      id="locator-2013-4-10-10-32-24-12830403-"/>
  </package>


  <!-- COSML Internal XSL-FO Package -->
  <package id="id-xslfo-cosml">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-5">
      <title id="title-2013-4-9-16-53-8-39562387-5">XSL-FO Package for COSML PDF</title>
```

```
  <description id="description-2013-4-9-16-53-8-39562387-5">
   <p id="p-2013-4-9-16-53-8-39562387-5">Converts COSML documents to XSL-FO format for COS
      PDF layout</p>
  </description>
</metadata>

<!-- Stylesheet parameters -->
<params id="params-2013-4-9-16-53-8-39562387-1">
  <!-- Index generation -->
  <param id="param-2013-4-9-16-53-8-39562387-3">
    <port id="port-2013-4-9-16-53-8-39562387-9">xslt-params</port>
    <name id="name-2013-4-9-16-53-8-39562387-5">generate.index</name>
    <value type="string" id="value-2013-4-9-16-53-8-39562387-11">0</value>
  </param>
  <!-- XEP Extensions -->
  <param id="param-2013-4-9-16-53-8-39562387-4">
    <port id="port-2013-4-9-16-53-8-39562387-10">xslt-params</port>
    <name id="name-2013-4-9-16-53-8-39562387-6">xep.extensions</name>
    <value type="string" id="value-2013-4-9-16-53-8-39562387-12">0</value>
  </param>
  <!-- XSL-FO Bookmark Generation -->
  <param id="param-2013-4-9-16-53-8-39562387-5">
    <port id="port-2013-4-9-16-53-8-39562387-11">xslt-params</port>
    <name id="name-2013-4-9-16-53-8-39562387-7">xslfo.bookmarks</name>
    <value type="string" id="value-2013-4-9-16-53-8-39562387-13">1</value>
  </param>
</params>

<!-- XSLT -->
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000232:sv-SE:0.6" xlink:title="COS Internal XSLT"
  type="main" id="locator-2013-4-9-16-53-8-39562387-1"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000074:sv-SE:0.11"
  id="locator-2013-4-9-16-53-8-39562387-2"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000059:sv-SE:0.2"
  id="locator-2013-4-9-16-53-8-39562387-3"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000070:sv-SE:0.15"
  id="locator-2013-4-9-16-53-8-39562387-4" xlink:title="Layout"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000876:sv-SE:0.2"
  id="locator-2013-4-9-16-53-8-39562387-5" xlink:title="bookmarks.xsl"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000075:sv-SE:0.17"
  id="locator-2013-4-9-16-53-8-39562387-6"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000072:sv-SE:0.10"
  id="locator-2013-4-9-16-53-8-39562387-7" xlink:title="meta-data.xsl"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000078:sv-SE:0.9"
  id="locator-2013-4-9-16-53-8-39562387-8" xlink:title="TOC"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000062:sv-SE:0.9"
  id="locator-2013-4-9-16-53-8-39562387-9"/>
<locator xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="urn:x-cassis:r1:cos:00000233:sv-SE:0.8"
```

```
        id="locator-2013-4-9-16-53-8-39562387-10"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000061:sv-SE:0.29"
        id="locator-2013-4-9-16-53-8-39562387-11"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000065:sv-SE:0.6"
        id="locator-2013-4-9-16-53-8-39562387-12"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000071:sv-SE:0.6"
        id="locator-2013-4-9-16-53-8-39562387-13"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000077:sv-SE:0.6"
        id="locator-2013-4-9-16-53-8-39562387-14"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000079:sv-SE:0.7"
        id="locator-2013-4-9-16-53-8-39562387-15"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000060:sv-SE:0.7"
        id="locator-2013-4-9-16-53-8-39562387-16"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000064:sv-SE:0.8"
        id="locator-2013-4-9-16-53-8-39562387-17"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000066:sv-SE:0.2"
        id="locator-2013-4-9-16-53-8-39562387-18"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000069:sv-SE:0.3"
        id="locator-2013-4-9-16-53-8-39562387-19"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000063:sv-SE:0.3"
        id="locator-2013-4-9-16-53-8-39562387-20"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000785:sv-SE:0.6"
        id="locator-2013-4-9-16-53-8-39562387-21"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000076:sv-SE:0.10" type="aux" xlink:title="Strings"
        id="locator-2013-4-9-16-53-8-39562387-22"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000230:sv-SE:0.1" type="aux"
        id="locator-2013-4-9-16-53-8-39562387-23"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000426:sv-SE:0.1" type="aux"
        id="locator-2013-4-9-16-53-8-39562387-24" xlink:title="tux.jpg"/>
    </package>
  </packages>
</process>
<packages id="packages-2013-4-9-16-53-8-39562387-1">

  <!-- XSLT for Normalizing COSML -->
  <package id="id-normalize" type="xslt">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-9">
      <title id="title-2013-4-9-16-53-8-39562387-9">Normalize XSLT</title>
      <description id="description-2013-4-9-16-53-8-39562387-9">
        <p id="p-2013-4-9-16-53-8-39562387-9">Stylesheet for applics filtering and module
          normalization for COSML documents</p>
      </description>
    </metadata>
    <!-- No parameters required. -->
```

```
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00000073:sv-SE:0.4"
        id="locator-2013-4-9-16-53-8-39562387-26" type="main" xlink:title="Normalize XSLT"/>
    </package>

    <!-- Calabash Engine Configuration File -->
    <package id="id-conf-calabash">
      <metadata id="metadata-2013-5-2-21-40-30-37001288-">
        <title id="title-2013-5-2-21-40-30-37001288-">Calabash Configuration</title>
        <description id="description-2013-5-2-21-40-30-37001288-">
          <p id="p-2013-5-2-21-40-30-37001288-">Configures Calabash</p>
        </description>
      </metadata>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
       xlink:href="urn:x-cassis:r1:cos:00002745:sv-SE:0.1" type="main" id="id-loc-calabash-config"
      />
    </package>



    <!-- Wrapper ProX Resources -->
    <package id="id-wrapper-resources">
      <metadata id="metadata-2013-5-2-21-40-30-37001288-1">
        <title id="title-2013-5-2-21-40-30-37001288-1">Wrapper Pipeline Processing</title>
        <description id="description-2013-5-2-21-40-30-37001288-1">
          <p id="p-2013-5-2-21-40-30-37001288-1">These files are used for running the wrapper
            pipeline.</p>
        </description>
      </metadata>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00002735:sv-SE:0.1" id="id-wrapper-xpl"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00002732:sv-SE:0.1" id="id-prox-fix"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00002733:sv-SE:0.1" id="id-urn2url"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00002731:sv-SE:0.1" id="id-prox2bat"/>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:href="urn:x-cassis:r1:cos:00002734:sv-SE:0.1" id="id-prox2shell-config"/>
    </package>

    <!-- XForms -->
    <package id="id-xform">
      <metadata>
        <title>ProX XForms Package</title>
        <description>
          <p>XForms for selecting and configuring a process, based on a ProX blueprint.</p>
        </description>
      </metadata>
      <locator xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href="urn:prox:xform:0.1"
        type="main" id="id-loc-xform"/>
    </package>
  </packages>
</processes>
```

A few things of note, above:

- Elements with `@type="pkg"` pinpoint their resources using a fragment identifier link to a `package` element, elsewhere in the file; the package then references the actual files, using `locator` elements. The `script` element near the top, for example, points out the XProc pipeline *package*, and the `value[@type="pkg"]` elements in `input` elements identify XSLT packages. The actual XProc script is referenced from inside that package.

  This has the advantage of grouping any related resources so that the whole group may be referenced by the ProX process: Note, for example, the long list of `locators` referencing the XSLT stylesheet modules for the XSL-FO package. The locators include URN-based links to *specific versions* of the stylesheet modules but the package that groups them is used to ensure that these versions work together.
- Elements with `@type="external"`, on the other hand, identify runtime values. For example, there is an `input/value[@type="external"]` for the input XML and an `option/value[@type="external"]` for a named output for this particular process.
- Packages are grouped inside specific processes and immediately below the root `processes`. This is a convention indicating that in the former case, the package(s) may only be used by that process while the latter allows them to be used everywhere.

ProX is modular, so a process may reuse packages, command lines, etc, from other processes, and the whole thing can, of course, be edited using the same XML tools as the XML processed by the system that uses ProX.

The idea is to convert the blueprint to a GUI from which the user can make selections and then save the result as a ProX instance. Here's a simple XForm that narrows the available choices:

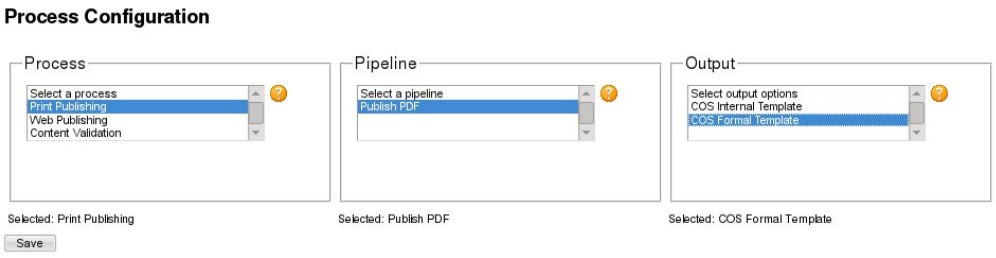


Figure 3. ProX XForm

The above selects first an appropriate process, then any pipelines made available for the process, and finally any "output options" (basically command line configurations) for the selected pipeline.

The “output options” here group XSLT parameters for the pipeline[1], presented to the user like this:

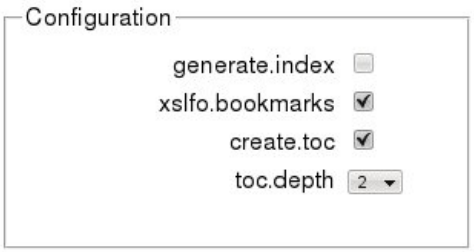


Figure 4. ProX Output Options

Here, the XForm exposes[2] parameters for an XSL-FO stylesheet.

```
<params id="params-2013-4-9-16-53-8-39562387-">

  <!-- XEP Extensions -->
  <param
    choice="no"
    id="param-2013-4-9-16-53-8-39562387-1">
    <port id="port-2013-4-9-16-53-8-39562387-3"
      >xslt-params</port>
    <name id="name-2013-4-9-16-53-8-39562387-1"
      >xep.extensions</name>
    <value
      xmlns:xlink="http://www.w3.org/1999/xlink"
      type="string" xlink:type="simple"
      id="value-2013-4-9-16-53-8-39562387-3">0</value>
  </param>

  <!-- TOC Generation -->
  <param choice="yes" ctype="list1"
    id="param-2013-4-9-16-53-8-2385485-2"
    group="value-2013-7-10-16-53-8-764625737-3">
    <port id="port-2013-7-10-16-34-8-9283444-4"
      >xslt-params</port>
    <name id="name-2013-7-10-16-50-3-1946564-2">toc.depth</name>
    <value xmlns:xlink="http://www.w3.org/1999/xlink"
      type="string" xlink:type="simple"
```

[1]While a pipeline can use more than one set of XSLT stylesheets, these are hidden in this abstraction. The user is only aware of the different output configurations, including both XSLT and whatever parameters they use.

[2]Only the parameters that the author of the ProX blueprint wants to make available are in fact made available. These might vary, depending on the user's permissions or something else. Several different ProX blueprints may be used by a single system.

```
        id="value-2013-7-10-16-53-8-764625737-4">2</value>
      <value id="value-13-07-10-12345-1" type="string">1</value>
      <value id="value-13-07-10-12345-2" type="string">3</value>
    </param>
  </params>
```

Some notes:

- `@ctype` indicates the type of parameter, used by the XForm that displays the parameters to the user.
- `@choice` indicates if the parameter is configurable.
- `@group` is an `IDREF` to a related parameter and indicates a dependency to that parameter. For example, a parameter may be used to set the table of contents depth, but it is useless if another parameter has turned off the TOC generation. The first parameter needs to include a `group` `IDREF` to the second so only relevant options are made available when configuring a ProX blueprint.

With every choice made, the user can save the XForm and produce a ProX instance. The instance is a single process, with every choice made, and is then converted to a shell script (or, in the case of this paper, an XQuery) that runs the selected process and pipeline.

The above very briefly describes a ProX demo implementation, first shown at Balisage 2013 (see [2]). The demo runs a wrapper XProc pipeline that preprocesses the ProX blueprint, makes it available via an Apache server and *XSLT Forms*, allowing configuration in an XForm, postprocesses the saved ProX instance, and eventually generates a shell script that runs the selected child process. This results in a PDF or an XHTML file, depending on the choices made.

While fun, the demo is crude and full of bizarre limitations, including some imposed by XProc (for example, the XProc spec does not specify any way to wait for user input, causing problems when opening an XForm) and others by the fact that it's really just a demo on an Apache server, with the XML, XSLT and XProc in a `htdocs` subfolder and the demo and the XSLT Forms implementation in a subfolder to that. Better would be to use something more mature, with built-in XML handling, XSLT, XProc, etc... something like eXist.

2. ProXist

ProXist, then, is a ProX implementation for eXist. At the time of this writing, it is a work in progress, with some limitations but also some promise.

In the Balisage demo, the ProX wrapper is simply a pipeline that a) allows the user configure a ProX blueprint using an XForm, narrowing it down to a ProX instance, b) generates a shell script for the child pipeline with runtime values inserted, and c) runs the child pipeline using the generated shell script. This, I figured, would be straight-forward to translate to eXist, generating an XQuery instead of a shell

script, but until recently, the Calabash extension module would only allow two inputs to the `xmlcalabash:process` function[3], namely an output URI and the pipeline URI.

Luckily, Jim Fuller who wrote the extension, graciously offered to help, adding options and bindings to the module, and I am now more or less back on track.

2.1. ProX Wrapper Process

A ProX wrapper process basically involves the following:

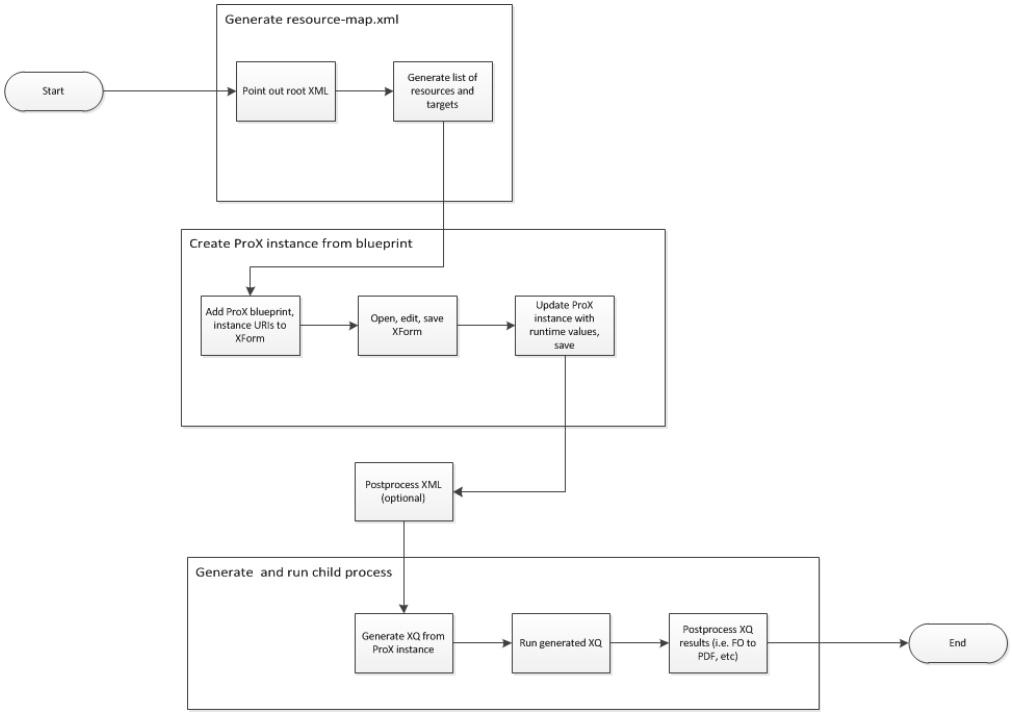


Figure 5. ProXis Wrapper process

Here's what happens:

1. First, we point out an input XML file and parse it for any linked resources, XML and otherwise. The linked XML also have to be parsed for more XML and other resources, until there's no more XML (or other resources) to be parsed.
2. The linked resource URIs are listed in an XML file, `resource-map.xml`, that also lists other resources (XProc scripts, XSLT, etc) needed by ProX, including the ProX blueprint to be used and and any runtime or "target" values that might be needed as specified by the ProX blueprint (for example, the output PDF file

[3]This had me thinking about ways to avoid writing inputs and options to my pipelines. This is not uncomplicated and involves an XQuery-based wrapper that inserts any required inputs, options, etc, directly into the child pipelines and then runs them without any external options.

needs to be named if the process needs to produce a PDF). The resource map is used as a lookup table for the ProX processes.

> **Note**
> `resource-map.xml` is generated using an XSLT stylesheet applied on the input XML, the ProX blueprint and a resource map template. The latter is a ProX resource list generated from the ProX blueprint's package lists.

3. The ProX XForm is updated with a URL to the ProX blueprint, used as input data, and a URL that names the ProX instance that will result. The XForm is then opened and the wrapper process now pauses to wait for the user's saved selections.
4. The user makes choices in the XForm (selects a process, pipeline, and output options according to the blueprint), and thereby defines a ProX instance.
5. The wrapper pipeline resumes operations when the ProX instance is saved; the pipeline waits for a change to the ProX instance URI.
6. The next step(s) may postprocess both the ProX instance (with any missing runtime values) and the input XML files (with, for example, temporary URLs to linked resources).

> **Note**
> In the Balisage demo, the resources used by both the input XML and ProX (XML, XSLT, XProc, etc) were all linked to using URNs rather than URLs, as evident in the above examples. The URNs were replaced with the URLs listed in `resource-map.xml` in a postprocessing XSLT step.

7. With the postprocessing done, the ProX instance (input values, options, etc) is used as an input to an XSLT that generates an XQuery.
8. The XQuery is saved and its permissions and ownership are changed to allow it to be run.
9. The wrapper runs the child pipeline using the generated XQuery.
10. The results from the child pipeline are postprocessed. For example, as noted in Section 3, the PDF generation step takes place after the wrapper finishes.

2.2. The ProX Blueprint

The ProX blueprint that resulted in the example instance, above, looks like this in a shortened form:

```
<?xml-model href="http://localhost:8080/exist/rest/db/work/system/prox/relaxng/processes.rnc" ▶
type="application/relax-ng-compact-syntax"?>
<processes
```

```
xmlns:xlink="http://www.w3.org/1999/xlink"
id="processes-2013-4-9-16-53-8-39562387-">

<!-- Print Publishing Process -->
<process id="id-pdf-process">
  <metadata id="metadata-2013-4-9-16-53-8-39562387-">
    <title id="title-2013-4-9-16-53-8-39562387-">Print Publishing</title>
    <description id="description-2013-4-9-16-53-8-39562387-">
      <p id="p-2013-4-9-16-53-8-39562387-">Print publishing for COSML documents</p>
    </description>
  </metadata>
  <pipelines id="pipelines-2013-4-9-16-53-8-39562387-">

    <!-- PDF Pipeline -->
    <pipeline id="id-pipeline-pdf-1">
      <metadata id="metadata-2013-4-9-16-53-8-39562387-1">
        <title id="title-2013-4-9-16-53-8-39562387-1">Publish PDF</title>
        <description id="description-2013-4-9-16-53-8-39562387-1">
          <p id="p-2013-4-9-16-53-8-39562387-1">Normalizes, validates
            and converts a COSML document to PDF</p>
        </description>
      </metadata>
      <script type="pkg" id="script-2013-4-9-16-53-8-39562387-"
        xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-xproc-pdf"
        xlink:title=" XProc Pipeline for Normalize, Validate and PDF
        Normalizes, validates and publishes in PDF a COSML document "/>
      <cmdlines id="cmdlines-2013-4-9-16-53-8-39562387-">

        <!-- COSML Internal XSL -->
        <cmdline id="id-cmdline-cos-internal-pdf">
          <metadata id="metadata-2013-4-9-16-53-8-39562387-2">
            <title id="title-2013-4-9-16-53-8-39562387-2">COS Internal
              Template</title>
            <description id="description-2013-4-9-16-53-8-39562387-2">
              <p id="p-2013-4-9-16-53-8-39562387-2">Configures the pipeline for
                the "COS Internal" template</p>
            </description>
          </metadata>
          <engine-config>
            <config type="pkg" xlink:href="#id-conf-calabash"/>
          </engine-config>
          <inputs id="inputs-2013-4-9-16-53-8-39562387-">
            <input choice="no" id="input-2013-4-9-16-53-8-39562387-">
              <port id="port-2013-4-9-16-53-8-39562387-">document</port>
              <value type="external" input-type="doc-root" xlink:type="simple"
                id="value-2013-4-9-16-53-8-39562387-" mimetype="application/xml"
                >DOCUMENT-PLACEHOLDER</value>
            </input>
            <input choice="no" id="input-2013-4-9-16-53-8-39562387-1">
              <port id="port-2013-4-9-16-53-8-39562387-1">stylesheet</port>
              <value type="pkg"
                xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-xslfo-cosml"
                xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-1"
                xlink:title=" XSL-FO Package for COSML PDF Converts COSML documents
                to XSL-FO format for COS PDF layout "/>
              <params id="params-2013-4-9-16-53-8-39562387-">
                <!-- Index generation -->
                <param choice="yes" ctype="boolean"
```

```
        id="param-2013-4-9-16-53-8-39562387-">
        <port id="port-2013-4-9-16-53-8-39562387-2"
          >xslt-params</port>
        <name id="name-2013-4-9-16-53-8-39562387-"
          >generate.index</name>
        <value type="string" xlink:type="simple"
          id="value-2013-4-9-16-53-8-39562387-2">false</value>
      </param>
      <!-- XEP Extensions -->
      <param choice="no" id="param-2013-4-9-16-53-8-39562387-1">
        <port id="port-2013-4-9-16-53-8-39562387-3"
          >xslt-params</port>
        <name id="name-2013-4-9-16-53-8-39562387-1"
          >xep.extensions</name>
        <value type="string" xlink:type="simple"
          id="value-2013-4-9-16-53-8-39562387-3">0</value>
      </param>
      <!-- XSL-FO Bookmark Generation -->
      <param choice="yes" ctype="boolean"
        id="param-2013-4-9-16-53-8-39562387-2">
        <port id="port-2013-4-9-16-53-8-39562387-4"
          >xslt-params</port>
        <name id="name-2013-4-9-16-53-8-39562387-2"
          >xslfo.bookmarks</name>
        <value type="string" xlink:type="simple"
          id="value-2013-4-9-16-53-8-39562387-4">true</value>
      </param>
      <!-- TOC Generation -->
      <param choice="yes" ctype="boolean"
        id="param-2013-4-9-16-53-8-39514778-2">
        <port id="port-2013-4-9-16-53-8-9653444-4"
          >xslt-params</port>
        <name id="name-2013-4-9-16-53-8-1928364-2">create.toc</name>
        <value type="string" xlink:type="simple"
          id="value-2013-7-10-16-53-8-764625737-3">true</value>
      </param>
      <!-- TOC Depth -->
      <param choice="yes" ctype="list1"
        id="param-2013-4-9-16-53-8-2385485-2"
        group="value-2013-7-10-16-53-8-764625737-3">
        <port id="port-2013-7-10-16-34-8-9283444-4"
          >xslt-params</port>
        <name id="name-2013-7-10-16-50-3-1946564-2">toc.depth</name>
        <value type="string" xlink:type="simple"
          id="value-2013-7-10-16-53-8-764625737-4">2</value>
        <value id="value-13-07-10-12345-1" type="string">1</value>
        <value id="value-13-07-10-12345-2" type="string">3</value>
      </param>
    </params>
  </input>
  <input choice="no" id="input-2013-4-9-16-53-8-39562387-2">
    <port id="port-2013-4-9-16-53-8-39562387-5">stylesheet-norm</port>
    <value type="pkg"
      xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.1#id-normalize"
      xlink:type="simple" id="value-2013-4-9-16-53-8-39562387-5"
      xlink:title=" Normalize XSLT Stylesheet for applics filtering
      and module normalization for COSML documents "
    />
```

```
          </input>
        </inputs>
        <options id="options-2013-4-9-16-53-8-39562387-">
          <option choice="no" id="option-2013-4-9-16-53-8-39562387-">
            <name id="name-2013-4-9-16-53-8-39562387-3">pdf</name>
            <value type="external" output-type="primary"
              mimetype="application/pdf" xlink:type="simple"
              id="value-2013-4-9-16-53-8-39562387-6"
              >PDF-PLACEHOLDER.pdf</value>
          </option>
        </options>
      </cmdline>

      <!-- COSML Formal XSL -->
      <cmdline id="id-cmdline-cos-formal-pdf">
        ...
      </cmdline>
    </cmdlines>
  </pipeline>
</pipelines>

<!-- Packages for Print -->
<packages xml:base="file:///e:/SGML/DTD/Cassis/Process-XML/"
  id="packages-2013-4-9-16-53-8-39562387-">

  <!-- XProc Normalize, Validate, XSLFO Pipeline Package -->
  <package id="id-xproc-pdf">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-4">
      <title id="title-2013-4-9-16-53-8-39562387-4">XProc Pipeline for Normalize,
        Validate and PDF</title>
      <description id="description-2013-4-9-16-53-8-39562387-4">
        <p id="p-2013-4-9-16-53-8-39562387-4">Normalizes, validates and publishes in
          PDF a COSML document</p>
      </description>
    </metadata>
    <!-- publish-cosml-pdf.xpl -->
    <locator type="main" xlink:href="urn:x-cassis:r1:cos:00002715:sv-SE:0.1"
      id="locator-2013-4-10-10-32-24-12830403-"/>
  </package>


  <!-- COSML Internal XSL-FO Package -->
  <package id="id-xslfo-cosml">
    <metadata id="metadata-2013-4-9-16-53-8-39562387-5">
      <title id="title-2013-4-9-16-53-8-39562387-5">XSL-FO Package for COSML
        PDF</title>
      <description id="description-2013-4-9-16-53-8-39562387-5">
        <p id="p-2013-4-9-16-53-8-39562387-5">Converts COSML documents to XSL-FO
          format for COS PDF layout</p>
      </description>
    </metadata>

    <!-- Stylesheet parameters -->
    <params id="params-2013-4-9-16-53-8-39562387-1">
      <!-- Index generation -->
      <param id="param-2013-4-9-16-53-8-39562387-3">
        <port id="port-2013-4-9-16-53-8-39562387-9">xslt-params</port>
        <name id="name-2013-4-9-16-53-8-39562387-5">generate.index</name>
```

```
          <value type="string" id="value-2013-4-9-16-53-8-39562387-11">0</value>
        </param>
        <!-- XEP Extensions -->
        <param id="param-2013-4-9-16-53-8-39562387-4">
          <port id="port-2013-4-9-16-53-8-39562387-10">xslt-params</port>
          <name id="name-2013-4-9-16-53-8-39562387-6">xep.extensions</name>
          <value type="string" id="value-2013-4-9-16-53-8-39562387-12">0</value>
        </param>
        <!-- XSL-FO Bookmark Generation -->
        <param id="param-2013-4-9-16-53-8-39562387-5">
          <port id="port-2013-4-9-16-53-8-39562387-11">xslt-params</port>
          <name id="name-2013-4-9-16-53-8-39562387-7">xslfo.bookmarks</name>
          <value type="string" id="value-2013-4-9-16-53-8-39562387-13">1</value>
        </param>
      </params>

      <!-- XSLT -->
      <locator xlink:href="urn:x-cassis:r1:cos:00000232:sv-SE:0.6"
        xlink:title="COS Internal XSLT" type="main"
        id="locator-2013-4-9-16-53-8-39562387-1"/>
      ...
    </package>
  </packages>
</process>

<!-- Wep PublishingProcess -->
<process id="id-web-process">
  <metadata id="metadata-2013-4-9-16-53-8-39562387-6">
    ...
  </metadata>
  <pipelines id="pipelines-2013-4-9-16-53-8-39562387-1">

    <!-- Pipeline for HTML -->
    <pipeline id="id-pipeline-web-1">
      <metadata id="metadata-2013-4-9-16-53-8-39562387-7">
        ...
      </metadata>
      <script id="script-2013-4-9-16-53-8-39562387-1"
       xlink:href="urn:x-cassis:r1:cos:00002712:sv-SE:0.6#package-2013-5-19-11-12-49-71312191-1"
        xlink:title="XProc COSML2XHTMLNormalises, validates and converts COSML to XHTML."
        type="pkg"/>
      <cmdlines id="cmdlines-2013-4-9-16-53-8-39562387-1">

        <!-- Single-file HTML Config -->
        <cmdline id="id-cmdline-single-file-HTML-1">
          ...
        </cmdline>
      </cmdlines>
    </pipeline>
  </pipelines>

  <!-- Web Publishing Packages -->
  <packages>

    <!-- XProc for COSML to XHTML -->
    <package id="package-2013-5-19-11-12-49-71312191-1">
      ...
    </package>
```

```
    <!-- XSLT for COSML to XHTML -->
    <package id="package-2013-5-19-11-12-49-71312191-">
      ...
    </package>
  </packages>
</process>

<!-- Content Validation Process -->
<process id="process-2013-5-19-11-12-49-71312191-">
  <metadata>
    ...
  </metadata>

  <!-- Content Validation Pipelines -->
  <pipelines>

    <!-- Xref Check Pipeline -->
    <pipeline id="pipeline-2013-5-19-11-12-49-71312191-">
      ...
    </pipeline>
  </pipelines>
  <packages>

    <!-- XProc for Xref Check -->
    <package id="package-2013-5-19-11-12-49-71312191-3">
      ...
    </package>

    <!-- XSLT for Xref Check -->
    <package id="package-2013-5-19-11-12-49-71312191-2">
      ...
    </package>
  </packages>
</process>

<packages id="packages-2013-4-9-16-53-8-39562387-1">

  <!-- XSLT for Normalizing COSML -->
  <package id="id-normalize" type="xslt">
    ...
  </package>

  <!-- Calabash Engine Configuration File -->
  <package id="id-conf-calabash">
    ...
  </package>



  <!-- Wrapper ProX Resources -->
  <package id="id-wrapper-resources">
    ...
  </package>

  <!-- XForms -->
  <package id="id-xform">
    ...
```

```
    </packages>
  </processes>
```

As explained above, the above lists a number of `processes`, each of which contains one or more `pipelines` that in turn include one or more command line (`cmdline`) option lists defining the required (and allowed) bindings for Calabash. Note that the pipeline identifies the XProc scripts using fragment IDs to `packages` that group the `locators` to the actual files.

2.3. Resource Map

The resource map XML is just a long list of mapped resources, XML, ProX and otherwise, used as a lookup table when running the processes defined in the ProX blueprint. It is generated using an XSLT stylesheet immediately after selecting the input XML and lists the selected input, any target output(s), and all resources used by ProX and its allowed child processes[4].

Here's an example resource map in a somewhat shortened form:

```
<resource-map>

  <!-- Source Modules Listed here -->
  <docs>

    <doc id="">

      <!-- Root document from Process Manager confguration -->
      <!-- ProX instance needs this value -->
      <!-- //*/@type=''external' and //*/@input-type='doc-root' -->
      <root>
        <resource>
          <urn>urn:testroot</urn>
          <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/test-root.xml</url>
          <type>doc-root</type>
          <prox-id>value-2013-4-9-16-53-8-39562387-</prox-id>
          <prox-id>id-html-docroot</prox-id>
        </resource>
      </root>

      <!-- All modules linked from root or its descendants -->
      <!-- XML, images, etc -->
      <modules>
        <resource>
          <urn>urn:image1</urn>
          <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/image3.jpg</url>
          <type>jpg</type>
        </resource>
        <resource>
          <urn>urn:inset1</urn>
          <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/inset1.xml</url>
          <type>xml</type>
```

[4]Many of the resources are fixed, determined statically, as they have been uploaded and defined long before a specific process is run.

```
      </resource>
      <resource>
        <urn>urn:inset2</urn>
        <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/inset2.xml</url>
        <type>xml</type>
      </resource>
      <resource>
        <urn>urn:inset3</urn>
        <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/inset3.xml</url>
        <type>xml</type>
      </resource>
      <resource>
        <urn>urn:inset4</urn>
        <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/inset4.xml</url>
        <type>xml</type>
      </resource>
      <resource>
        <urn>urn:block-inset1</urn>
        <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/block-inset1.xml</url>
        <type>xml</type>
      </resource>
    </modules>
  </doc>
</docs>


<!-- Runtime targets -->
<!-- Should use xmldb:exist for most -->
<!-- Do they need http or webdav variants? -->
<targets>
  <resource>
    <urn>URN-FOR-OUTPUT</urn>
    <url>xmldb:exist:///db/work/docs/test/my-pdf-internal-file.pdf</url>
    <type>primary</type>
    <prox-id>value-2013-4-9-16-53-8-39562387-6</prox-id>
  </resource>
  <resource>
    <urn>URN2-FOR-OUTPUT</urn>
    <url>xmldb:exist:///db/work/docs/test/my-pdf-formal-file.pdf</url>
    <type>primary</type>
    <prox-id>value-2013-4-9-16-53-8-39562387-10</prox-id>
  </resource>
  <resource>
    <urn>URN-FOR-XREF-XHTML-LOG</urn>
    <url>xmldb:exist:///db/work/docs/test/my-xref-check.htm</url>
    <type>primary</type>
    <prox-id>id-value-xref-htm</prox-id>
  </resource>
  <resource>
    <urn>URN-FOR-FILES-LIST-XML</urn>
    <url>xmldb:exist:///db/work/docs/test/files.xml</url>
    <type>fixed</type>
    <prox-id>files</prox-id>
  </resource>
  <resource>
    <urn>URN-FOR-HTM-OUT</urn>
    <url>xmldb:exist:///db/work/docs/test/my-xhtml-out.htm</url>
    <type>primary</type>
```

```
    <prox-id>id-htm-out</prox-id>
  </resource>
  <resource>
    <urn>URN-FOR-NORMALIZED-HTML</urn>
    <url>xmldb:exist///db/work/docs/test/normalized-for-debug.xml</url>
    <type>secondary</type>
    <prox-id>id-normalized-html</prox-id>
  </resource>
</targets>

<!-- ProX blueprint and saved instance(s) -->
<prox>
  <!-- Blueprint used to get instance is here -->
  <blueprints>
    <resource id="id-prox-blueprint">
      <urn>URN-OF-PROX-BLUEPRINT</urn>
      <url>http://localhost:8080/exist/rest/db/work/system/prox/xml/prox-blueprint.xml</url>
      <!--<url>file://prox-blueprint.xml</url>-->
      <type/>
      <prox-id/>
    </resource>
  </blueprints>

  <!-- Saved ProX instances -->
  <!-- Input to wrapper pipeline -->
  <instances>
    <resource id="id-prox-saved-instance">
      <urn>URN-OF-SAVED-PROX-INSTANCE</urn>
      <url>xmldb:exist:///db/work/docs/test/prox-instance.xml</url>
      <!-- Insert file URI here for local testing -->
      <type/>
      <prox-id/>
    </resource>
  </instances>
</prox>

<!-- Resources used by ProX Processes -->
<prox-resources>

  <!-- PDF Publishing XProc -->
  <package>
    <name>XProc Pipeline for Normalize, Validate and PDF</name>
    <resources>
      <resource>
        <urn>urn:x-cassis:r1:cos:00002715:sv-SE:0.1</urn>
        <url>xmldb:exist:///db/work/system/cosml/xproc/publish-cosml-pdf.xpl</url>
        <prox-id>locator-2013-4-10-10-32-24-12830403-</prox-id>
      </resource>
    </resources>
  </package>

  <!-- PDF Publishing XSL-FO, Internal -->
  <package>
    <name>XSL-FO Package for COSML PDF</name>
    <resources>
      <resource>
        <urn>urn:x-cassis:r1:cos:00000232:sv-SE:0.6</urn>
        <url>http://localhost:8080/exist/rest/db/work/system/cosml/fo/cos-fo-internal.xsl</url>
```

```
        <prox-id>locator-2013-4-9-16-53-8-39562387-1</prox-id>
      </resource>
      <resource>
        <urn>urn:x-cassis:r1:cos:00000074:sv-SE:0.11</urn>
        <url>http://localhost:8080/exist/rest/db/work/system/cosml/fo/param.xsl</url>
        <prox-id>locator-2013-4-9-16-53-8-39562387-2</prox-id>
      </resource>
      ...

    </resources>
  </package>

  <!-- XHTML Publishing XProc -->
  <package>
    <name>XProc COSML2XHTML</name>
    ...
  </package>

  <!-- XHTML Publishing XSLT -->
  <package>
    <name>COSML XHTML XSLT</name>
    ...
  </package>

  <!-- Xref Check XProc -->
  <package>
    <name>XProc Xref Check</name>
    ...
  </package>

  <!-- Xref Check XSLT -->
  <package>
    <name>XSLT Xref Check</name>
    ...
  </package>

  <!-- Standard Normalize XSLT for Publishing -->
  <package>
    <name>Normalize XSLT</name>
    ...
  </package>

  <!-- Calabash Engine Configuration -->
  <package>
    <name>Calabash Configuration</name>
    ...
  </package>
</prox-resources>


<!-- Wrapper stuff -->
<wrapper-pipeline>

  <!-- Wrapper Pipeline Resources -->
  <package>
    <name>Wrapper Pipeline Processing</name>
    <resources>
      <!-- Wrapper Pipeline -->
```

```
        <resource>
          <urn>urn:x-cassis:r1:cos:00002735:sv-SE:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/work/system/prox/prox-wrapper.xpl</url>
          <prox-id>id-wrapper-xpl</prox-id>
        </resource>
        <!-- ProX Instance Update -->
        <resource>
          <urn>urn:x-cassis:r1:cos:00002732:sv-SE:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/work/system/prox/xslt/prox-fix.xsl</url>
          <prox-id>id-prox-fix</prox-id>
        </resource>
        <!-- URN2URL for XML Input -->
        <resource>
          <urn>urn:x-cassis:r1:cos:00002733:sv-SE:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/work/system/prox/xslt/urn2url.xsl</url>
          <prox-id>id-urn2url</prox-id>
        </resource>
        <!-- ProX Instance Conversion to Shell Script -->
        <resource>
          <urn>urn:x-cassis:r1:cos:00002731:sv-SE:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/work/system/prox/xslt/prox2shell.xsl</url>
          <prox-id>id-prox2shell</prox-id>
        </resource>
        <!-- ProX Instance Conversion to XQ -->
        <resource>
          <urn>urn:x-cassis:r1:cos:00012731:sv-SE:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/work/system/prox/xslt/prox2xq.xsl</url>
          <prox-id>id-prox2xq</prox-id>
        </resource>
        <!-- Engine parameters required by ProX to Shell Script conversion -->
        <resource>
          <urn>urn:x-cassis:r1:cos:00002734:sv-SE:0.1</urn>
         <url>http://localhost:8080/exist/rest/db/work/system/prox/xml/prox2shell-config.xml</url>
          <prox-id>id-prox2shell-config</prox-id>
        </resource>
      </resources>
    </package>

    <!-- XForm for ProX Process Configuration -->
    <package>
      <name>ProX XForm</name>
      <resources>
        <!-- XForm for proX Blueprint Handling -->
        <resource>
          <urn>urn:prox:xform:0.1</urn>
          <url>http://localhost:8080/exist/rest/db/apps/prox-xform.xml</url>
          <prox-id>id-loc-xform</prox-id>
        </resource>
      </resources>
    </package>
  </wrapper-pipeline>
</resource-map>
```

Some notes:

- The input XML document, defined in `doc/root/resource`, includes several `prox-id` elements:

```
<resource>
  <urn>urn:testroot</urn>
  <url>http://localhost:8080/exist/rest/db/work/docs/pdftest/test-root.xml</▶
url>
  <type>doc-root</type>
  <prox-id>value-2013-4-9-16-53-8-39562387-</prox-id>
  <prox-id>id-html-docroot</prox-id>
</resource>
```

Each `prox-id` identifies the input binding for a child ProX process, that is, an input binding in an XProc pipeline defined in the ProX blueprint, such as the following (note the matching `@id` in the `value` element:

```
<input
  choice="no"
  id="input-2013-4-9-16-53-8-39562387-">
  <port
    id="port-2013-4-9-16-53-8-39562387-">document</port>
  <value
    type="external"
    input-type="doc-root"
    xlink:type="simple"
    id="value-2013-4-9-16-53-8-39562387-"
    mimetype="application/xml">DOCUMENT-PLACEHOLDER</value>
</input>
```

- The `targets` structure lists named output targets for the child process[5]. The target names are generated by an XSLT stylesheet, based on the input XML name and how they are defined in the ProX blueprint. The primary output retains the file-name but with a new file suffix based on the output's MIME type, also defined in the blueprint. Here's an example of such a binding in the blueprint:

```
<option
  choice="no"
  id="option-2013-4-9-16-53-8-39562387-">
  <name
    id="name-2013-4-9-16-53-8-39562387-3">pdf</name>
  <value
    type="external"
    output-type="primary"
    mimetype="application/pdf"
    xlink:type="simple"
    id="value-2013-4-9-16-53-8-39562387-6">PDF-PLACEHOLDER.pdf</value>
</option>
```

[5]Note that the wrapper, for example, might require additional runtime values, for example, the ProX instance.

This `option` might result in the following target:

```
<resource>
  <urn>URN-FOR-OUTPUT</urn>
  <url>xmldb:exist:///db/work/docs/test/test-root.pdf</url>
  <type>primary</type>
  <prox-id>value-2013-4-9-16-53-8-39562387-6</prox-id>
</resource>
```

- The ProX definitions are listed in `prox`. Currently, it lists one blueprint, used as the data for the XForm, and one instance, the XML that results when saving the XForm.
- The resources used by ProX processes are listed in `prox-resources`. These include any XSLT, FO, XProc, etc.
- The ProX wrapper resources are listed in `wrapper-pipeline`. These include the wrapper XProc, the XForm, etc.

But most importantly: currently, any URI used by a pipeline to save something in eXist *must* be given as an `xmldb:exist` URI, due to the unfortunate Calabash URI handling problem, described above.

2.4. Running ProX in eXist

My first XQuery wrapper for running ProX in eXist was just something that ran the wrapper XProc:

```
query version "3.0";
let $result := xmlcalabash:process("xmldb:exist:///db/work/docs/xproc/▶
WRAP-2.xpl",
("-imap=http://localhost:8080/exist/rest/db/work/system/common/xml/▶
resource-map.xml",
"-oresult=-"),
("normalized=xmldb:exist:///db/work/docs/test/test.xml"))
return
   $result
```

In other words, simply something that invoked the wrapper XProc, using the resource map as its only input. When toying with eXist's web development capabilities in its eXide editor, however, I realised that there is no need to create one, unified wrapper. We still need to do the following, but we don't need to do them as a single process:

1. Select an input XML file.
2. Generate a resource map lookup table for ProX, based on the input and a "resource map template" that lists the resources that aren't determined at runtime.

3. Start the XForm used to configure ProX with (presenting the configuration options available in the ProX blueprint) and save a configured instance that describes a single ProX child process.
4. Run the configured child ProX process and handle results.

Better to present each of these on an HTML page in eXist, like this:

1. Select an input file. This involves an XForm and a simple XQuery for listing files in the DB and filtering out those that aren't XML or have the wrong root element for further processing. With apologies for the crudeness of my test XForm:

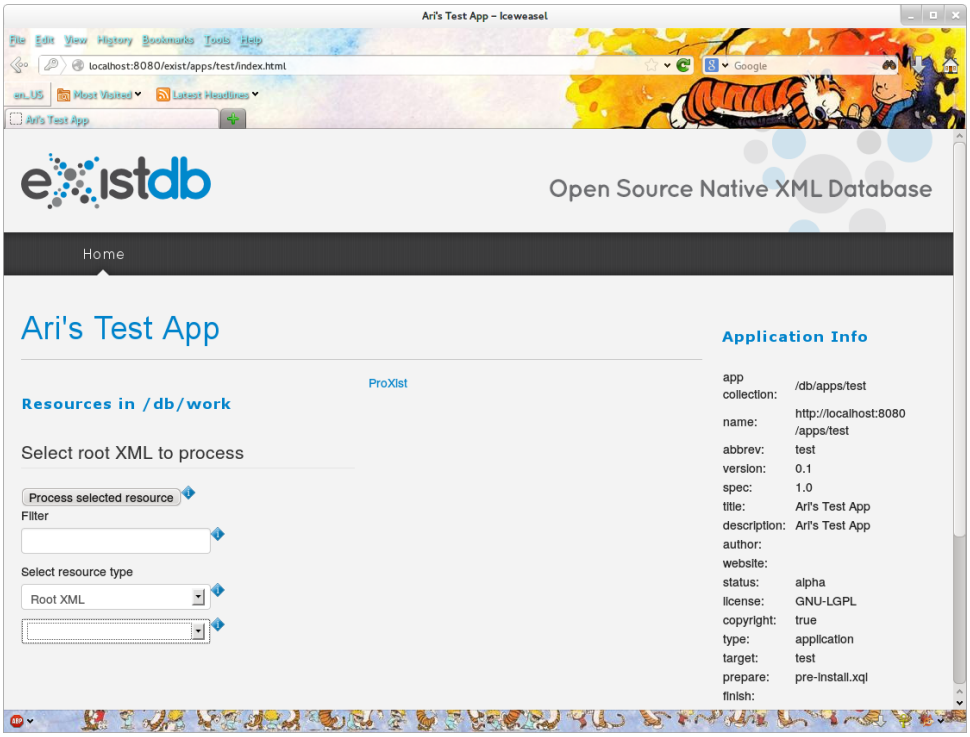


Figure 6. Select Input XML

The XForm fetches its data using something like this:

```
let $xml := collection(concat('/db/work/','docs'))

return <data>

  {
  for $doc in ($xml)
  order by base-uri($doc)

  return

  if (contains(base-uri($doc),'.xml') and local-name($doc/*)='cos')
    then <item>
```

```
      <string>{tokenize(base-uri($doc),'/')[last()]} - root XML, ▶
({base-uri($doc)})</string>
      <value>{base-uri($doc)}</value>
      <type>root</type>
    </item>
    else if (contains(base-uri($doc),'.xml') and local-name($doc/*)!='cos')
      then <item>
      <string>{tokenize(base-uri($doc),'/')[last()]} ▶
({tokenize(base-uri($doc),'\.')[last()]} XML module, {base-uri($doc)})</▶
string>
      <value>{base-uri($doc)}</value>
      <type>xmlmodules</type>
    </item>
    else (<item>
      <string>{tokenize(base-uri($doc),'/')[last()]} ▶
({tokenize(base-uri($doc),'\.')[last()]} module, {base-uri($doc)})</string>
      <value>{base-uri($doc)}</value>
      <type>other</type>
    </item>)
  }
</data>
```

2. When hitting Process selected resource, an XQuery runs the XSLT to generate `resource-map.xml` and save it where it can be found by the next step (configuring the XForm). Here's a test:

```
let $files := request:get-data()

for $file in tokenize($files//value,' ')

  (: We only allow root XML as input :)
  let $input := if ((contains($file,'.xml') and local-name(doc($file)/▶
*)='cos'))
    then $file
    else ""

  let $filename := tokenize($file,'/')[last()]

  let $parameters := <parameters><param name="root-xml" value="{$file}"/></▶
parameters>

  let $result := if ($input !='') then
      (transform:transform(doc($file),'http:localhost:8080/exist/rest/db/▶
system/cosml/xslt/doc-resources.xsl', $parameters))
      else ""

  return if ($result!='')
```

```
      then xmldb:store("xmldb:exist:///db/work/docs/▶
test",$resource-map,$result) (:$result:)
      else ""
```

3. An additional XQuery then runs the ProX XForm, using the generated resource map as input.

   Note
   If the first two steps are skipped, the last available `resource-map.xml` is used instead or, if there is none available, the user warned and the process interrupted.

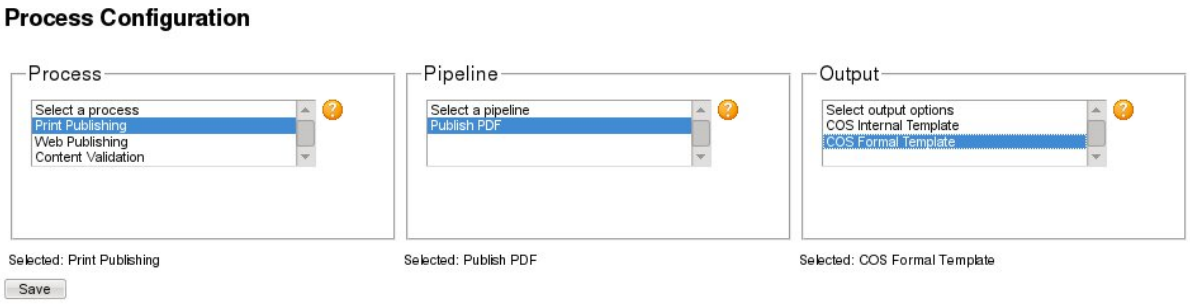


Figure 7. The ProXXForm

4. The Save saves the configured ProX instance and runs the ProX wrapper XProc.

2.5. The Wrapper Pipeline

The Balisage version of ProX ran with a wrapper XProc script that configured and ran the XForm, updated the ProX instance that resulted with runtime values, post-processed the input XML files, converted the ProX instance to a shell script to run the child process with, and ran that shell script. Published output resulted.

The ProXist version also uses a wrapper XProc, but leaves the preprocessing before and including the XForm to XQueries and an eXist web app, described above. Also, as the wrapper XProc steps are run exclusively in eXist rather than on a file system, some of them are XQueries[6] invoked form the XProc.

The following is the wrapper XProc as it appears at the time of this writing. It works but still invokes the XForm by running a new profile of the browser rather than in an XQuery as described in Section 2.4. This is both cumbersome and unnecessary, but the XQuery was not finished in time for the XML Prague paper deadline.

[6] A prime example is the snippet required to change the ownership and permissions of the XQuery that is generated from the ProX instance.

Also, as the current Calabash seems to have problems with URI handling, the XProc wrapper cannot currently produce a PDF; this also needs to be handled by an XQuery[7], relying on eXist's FO processor integration.

```
<p:declare-step
  xmlns:c="http://www.w3.org/ns/xproc-step"
  xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:cx="http://xmlcalabash.com/ns/extensions"
  xmlns:xf="http://www.w3.org/2002/xforms"
  name="main"
  version="1.0">


  <!-- Wrapper XProc for ProX
         Requires resource map file as an input. -->

  <!-- Inputs -->

  <!-- Resource map document -->
  <!-- Contains all URN/URL for XSLT, XPL, XML modules, targets, etc -->
  <p:input port="map" sequence="true"/>

  <!-- Global XSLT params -->
  <p:input port="xsltparams" kind="parameter"/>

  <!-- Outputs -->
  <p:output port="result" sequence="true">
    <p:pipe port="result" step="med"/>
  </p:output>

  <!-- Extension steps -->
  <p:import href="http://xmlcalabash.com/extension/steps/library-1.0.xpl"/>

  <!-- ProX Blueprint URL -->
  <p:variable
    name="prox-blueprint"
    select="//prox/blueprints/resource[@id='id-prox-blueprint']/url/normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- ProX XForm Target Instance URL (webdav) -->
  <p:variable
    name="target-prox-instance"
    select="//prox/instances/resource[@id='id-prox-xform-target-instance']/url/▶
normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- ProX XForm Target Instance URL (xmldb) -->
  <p:variable
    name="xmldb-target-prox-instance"
    select="//prox/instances/resource[@id='id-prox-xform-xmldb-instance']/url/▶
```

[7]Either in the XQuery wrapper or by changing the PDF publishing pipeline to only produce FO and then handling that FO in the generated XQuery instead. The latter option seems to be the more likely one, as I write this.

```
normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- ProX Saved Instance URL (rest) -->
  <p:variable
    name="saved-prox-instance"
    select="//prox/instances/resource[@id='id-prox-saved-instance']/url/normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- XForm URL -->
  <p:variable
    name="xform-url"
    select="//wrapper-pipeline//resource[prox-id='id-loc-xform']/url/normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- prox2shell config URL -->
  <p:variable
    name="prox2shell-config"
   select="//wrapper-pipeline//resource[prox-id='id-prox2shell-config']/url/normalize-space(text())">
    <p:pipe port="map" step="main"/>
  </p:variable>

  <!-- Temp URL -->
  <p:variable name="tmp-url" select="'xmldb:exist:///db/work/docs/test/'">
    <!-- substring-before(base-uri(/*),tokenize(base-uri(.),'/')[last()]) -->
    <!--<p:pipe port="map" step="main"/>-->
    <!-- Should use base URI of a target output (ensures writable collection) -->
  </p:variable>

  <!-- OS ('osx', 'win', 'linux', 'exist' allowed) -->
  <p:variable name="os" select="'exist'"/>



  <!-- Open ProX Blueprint in Browser -->
  <!-- Opens with an XForms profile in order
         to start a separate browser instance -->
  <p:choose name="browse">
    <!-- Linux -->
    <p:when test="$os='linux'">
      <p:exec command="/usr/bin/iceweasel">
        <p:input port="source">
          <p:empty/>
        </p:input>
        <p:with-option
          name="args"
          select="concat('-P &#34;XForms&#34; -no-remote ',$xform-url)"/>
      </p:exec>
      <p:sink/>
    </p:when>

    <!-- eXist -->
    <p:when test="$os='exist'">
      <p:exec command="/usr/bin/iceweasel">
        <p:input port="source">
```

```
          <p:empty/>
        </p:input>
        <!-- Add variable ref to the following? -->
        <p:with-option
          name="args"
          select="concat('-P &#34;xforms&#34; -no-remote ','http://localhost:8080/exist/rest/db/▶
apps/form.xq?form=prox-xform.xml')"/>
      </p:exec>
      <cx:wait-for-update pause-after="3">
        <!-- Needs to monitor webdav URI of ProX instance, changed by XForm -->
        <p:with-option name="href" select="$target-prox-instance"/>
      </cx:wait-for-update>
      <p:sink/>
    </p:when>
  </p:choose>

  <!-- Insert runtime values to ProX instance -->
  <p:xslt name="prox-urn2url" cx:depends-on="browse">
    <!-- Input source is ProX instance saved by XForm -->
    <p:input
      port="source"
      select="doc(//prox/instances/resource[@id='id-prox-saved-instance']/url/▶
normalize-space(text()))">
      <p:pipe port="map" step="main"/>
    </p:input>
    <p:input
      port="stylesheet"
      select="doc(//wrapper-pipeline/package/resources/resource[prox-id='id-prox-fix']/url/▶
normalize-space(text()))">
      <p:pipe port="map" step="main"/>
    </p:input>
    <p:with-param name="map-url" select="base-uri()">
      <p:pipe port="map" step="main"/>
    </p:with-param>
  </p:xslt>

  <p:identity name="id">
    <p:input port="source"/>
  </p:identity>

  <!-- Store ProX instance with URLs -->
  <p:store name="save-prox" cx:depends-on="id">
    <p:with-option
      name="href"
      select="'xmldb:exist:///db/work/docs/test/tmp-prox-instance.xml'"/>
  </p:store>

  <!-- Convert instance to XQ -->
  <p:xslt name="xsltbat" cx:depends-on="id">
    <p:input port="source">
      <p:pipe port="result" step="id"/>
    </p:input>
    <p:input
      port="stylesheet"
      select="doc(//wrapper-pipeline/package/resources/resource[prox-id='id-prox2xq']/url/▶
normalize-space(text()))">
      <p:pipe port="map" step="main"/>
    </p:input>
```

```
    <p:with-param name="map-url" select="base-uri()">
      <p:pipe port="map" step="main"/>
    </p:with-param>
  </p:xslt>

  <p:store
    name="save-xq"
    cx:depends-on="xsltbat"
    media-type="text/plain"
    method="text">
    <p:with-option
      name="href"
      select="'xmldb:exist:///db/work/docs/test/out.xq'"/>
  </p:store>

  <p:xquery name="xq">
    <p:input port="source">
      <p:pipe port="result" step="xsltbat"/>
    </p:input>
    <p:input port="query">
      <!-- Change permissions, group and owner -->
      <p:data
        href="http://localhost:8080/exist/rest/db/work/docs/xq/chown-test.xq"
        content-type="text/plain"/>
    </p:input>
  </p:xquery>

  <p:xquery name="run-xq">
    <p:input port="query">
      <!-- Run generated XQuery -->
      <p:data
        href="http://localhost:8080/exist/rest/db/work/docs/test/out.xq"
        content-type="text/plain"/>
    </p:input>
  </p:xquery>

  <p:sink/>

  <!-- Return Results -->
  <p:identity name="med">
    <p:input port="source">
      <p:inline>
        <p>Success!</p>
      </p:inline>
    </p:input>
  </p:identity>
</p:declare-step>
```

Some notes:

- While the above example includes OS-dependent steps, they are no longer needed. Also, the XForm does not actually need to be opened in a separate browser instance, as there is a `cx:wait-for-update` that pauses the pipeline while monitoring a change to the ProX instance URL.

- The wrapper concludes with two XQueries. The first invokes a helper XQuery[8] that changes the file ownership and permissions of the XQuery that was generated from the ProX instance by a previous step, and the second runs that generated XQuery.

  Note
  While this arrangement works, there may be advantages to placing these two steps in the wrapper XQuery, running them *after* the wrapper XProc has finished. Most importantly, it is far easier to handle the child process results there rather than in an XProc that for now is something of a foreign entity inside eXist.

2.6. Processing before Running the Child Process

Some processing to both the input XML and the ProX components is required before the XQuery for the child process can be generated and run:

- Before running the XForm, it needs a URI to the ProX blueprint, to be used as input, and a temporary ProX instance URI for, to be used as output.
- The ProX instance saved by the XForm needs to be updated with runtime values for the input XML and any outputs before it can be used as the input for generating the child process XQuery. The runtime values are fetched from the resource map XML.
- The input XML may need to be processed, for example, replacing URLs with URNs in links.

  Note
  All these steps require an input resource map XML file, to be used as a lookup table. See Section 2.3.

2.7. Generating an XQuery to Run a Child Process

An XQuery for running a child pipeline looks something like this (the example is a test for a child process producing PDF output; this particular test currently only produces the FO, not the converted PDF) :

```
xquery version "3.0";
let $result := xmlcalabash:process("xmldb:exist:///db/work/docs/xproc/▶
publish-cosml-pdf-TEST.xpl",
("-istylesheet=http://localhost:8080/exist/rest/db/work/system/cosml/fo/▶
```

[8] Including this directly in the wrapper XProc does not work, likely because of the permissions the XProc runs with.

```
cos-fo-internal.xsl",
"-istylesheet-norm=http://localhost:8080/exist/rest/db/work/system/cosml/xslt/▶
normalize-2.xsl",
"-idocument=http://localhost:8080/exist/rest/db/work/docs/pdftest/▶
test-root.xml"),
("normalized=xmldb:exist:///db/work/docs/test/test.xml",
"pdf=xmldb:exist:///db/work/docs/test/out.pdf"))
return
   $result
```

The above child process requires a root XML input file, two XSLT stylesheets (one for normalising the XML into a single file, the other for converting the normalised XML to FO), an option for saving the normalised XML for debugging and finally a named PDF output filename. Given a ProX instance as input, the above is easily produced with an XSLT stylesheet [9].

The generated XQuery is saved to a temporary collection. Its permissions are then changed using a `p:xquery` step referencing a stored XQuery. This allows the generated XQuery to be run by the next XProc step, another `p:xquery` step.

3. Limitations, Hacks and Additions

Some limitations and some solutions:

- The demo wrapper used a `p:exec` to start a browser and open the XForm URL in the operating system's command line. This did pause the wrapper pipeline *until the browser was closed*, allowing the user to make choices and save the XForm, IF the browser was not already open in which case the wrapper wouldn't understand that it was supposed to pause.

  Norm Walsh was present at the demo and wrote an extension to Calabash, `cx:wait-for-update`, that pauses a pipeline until a URI changes, *before the day was over*. This very neatly solves the pause for user input problem as the wrapper pipeline now only has to monitor the ProX instance URI for changes.
- The XML Calabash eXist extension module now works with the currently latest XML Calabash (1.0.16-94 as of this writing) and accepts XProc options, inputs and other bindings as defined by Calabash, with some quirks and limitations[10].
- At the time of this writing, a Calabash URI handling bug is imposing some limitations to my child pipeline processes. The `p:xsl-formatter` step, for example, appears to require that `file:` is used in `@href`, which, for now, means that the resulting PDF cannot be saved in eXist from a pipeline; currently, the Calabash

[9] It would probably be just as easy to write the wrapper ProX process in XQuery, limiting the use of XProc to the child processes. The solution presented here is the continuation of the author's earlier work, indicating his preferences rather than an objectively preferred way.

[10] For example, input bindings need to be sequences, and only one output port is allowed.

module and eXist require `xmldb:exist:`. An "outside" XQuery for the FO to PDF conversion, and a subsequent save, is required.

- This, on the other hand, causes a problem because that outside step needs to identify what it receives from the previous step. This, at the time of this writing, I cannot yet do.

> **Note**
> The demo ProX implementation left all of the child processing to the child pipeline, which is far easier because the wrapper does not need to know what it processes, it just needs to pass on the user's choices and the runtime values to the child. This, of course, is what I would prefer to happen in eXist as well.

4. What's Next?

While ProXist does work at the time of this writing, there are fixes and improvements to be made, both in time for the conference and later:

- The wrapper XQuery that initiates and runs the ProX wrapper process needs to be updated to handle selecting the input XML, running the XSLT that generates the resource map XML, updating, opening and saving the XForm, and only then running the wrapper XProc pipeline. See Section 2.4.
- ProXist needs to be packaged properly. eXist includes a terrific web application development and packaging kit, which vastly simplifies both packaging it and writing the wrapper XQuery (as well as other resources) itself.
- The XProc pipelines, ProX and child processes alike, currently include very little in the way of error handling and logging, which is something that needs to be remedied
- The ProX Relax NG schema is somewhat inconsistent in its use of repeating and common semantics, and there are problems with how bindings are declared. For example, currently identifying target (runtime) bindings is cumbersome because of how their IDs are defined; two different pipelines cannot share the same basic output binding because the output values use `ID` attributes to identify them.

5. Some Final Words

My deepest and most heartfelt gratitude must go to to Jim Fuller who took pity to my code-impaired self and added the missing options and Calabash bindings to his eXist Calabash module. He also discovered bugs in Calabash's handling of URIs in the in- and output bindings, and *fixed them, too*.

My thanks also to Norm Walsh, who listened to my Balisage talk about ProX and wrote the `cx:wait-for-update` extension step for Calabash before I was done. Not sure if it means I was boring or interesting and afraid to ask.

And thanks also to my boss who allowed me to open-source all of ProX.

Finally, both ProXist and Prox the Balisage edition are available on Github (see [5]). While I'm aware of the fact that what's there now is a chaotic mess, I hope to bring some order to it all in time for the conference. Perhaps even document it.

Bibliography

[1] Using XML to Implement XML, Ari Nordström
http://www.balisage.net/Proceedings/vol8/html/Nordstrom01/BalisageVol8-Nordstrom01.html

[2] ProX: XML for interfacing with XML for processing XML (and an XForm to go with it)
http://www.balisage.net/Proceedings/vol11/html/Nordstrom02/BalisageVol11-Nordstrom02.html

[3] XProc: An XML Pipeline Language, Recommendation
http://www.w3.org/TR/xproc/

[4] XML Calabash http://xmlcalabash.com/

[5] ProX on Github https://github.com/sgmlguru

Jiří Kosek (ed.)

XML Prague 2014
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 XEP.

1st edition

Prague 2014

ISBN 978-80-260-5712-3