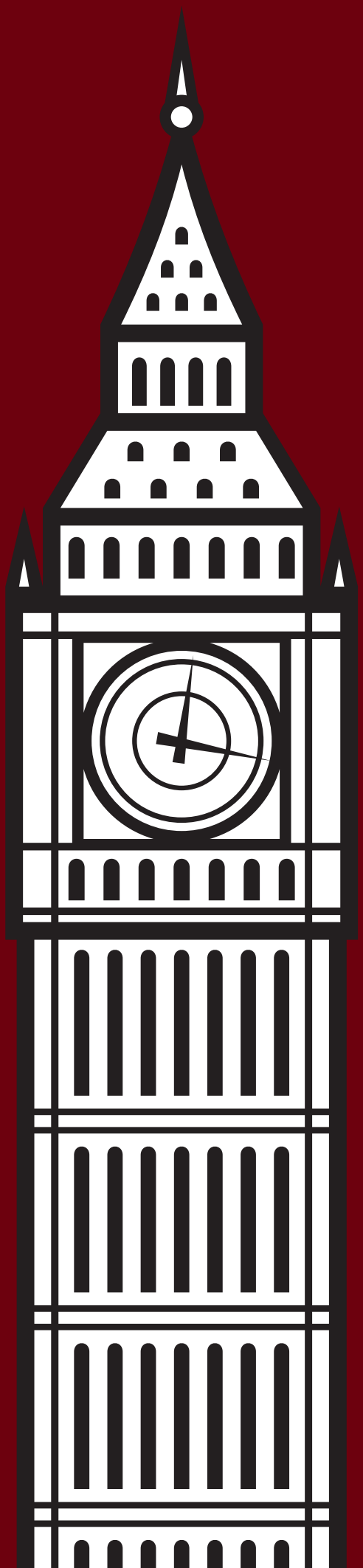


XML LONDON 2013

CONFERENCE PROCEEDINGS

UNIVERSITY COLLEGE LONDON,
LONDON, UNITED KINGDOM

JUNE 15-16, 2013

XML London 2013 – Conference Proceedings
Published by XML London

ISBN 978-0-9926471-0-0

Table of Contents

General Information

Date

Saturday, June 15th, 2013
Sunday, June 16th, 2013

Location

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

Organising Committee

Kate Foster, Socionics Limited
Dr. Stephen Foster, Socionics Limited
Charles Foster, XQJ.net & Socionics Limited

Programme Committee

Abel Braaksma, AbraSoft
Adam Retter, Freelance
Charles Foster (chair), XQJ.net
Dr. Christian Grün, BaseX
Eric van der Vlist, Dyomedea
Jim Fuller, MarkLogic
John Snelson, MarkLogic
Lars Windauer, BetterFORM
Mohamed Zergaoui, Innovimax
Philip Fennell, MarkLogic

Produced By

XML London (http://xmllondon.com)

Sponsors

Gold Sponsor

- OverStory - http://www.overstory.co.uk

Silver Sponsor

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

Bronze Sponsor

- Mercator IT Solutions - http://www.mercatorit.com

Preface

This publication contains the papers presented during the XML London 2013 conference.

This was the first international XML conference held in London for XML Developers – Worldwide, Semantic Web & Linked Data enthusiasts, Managers / Decision Makers and Markup Enthusiasts.

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

The conference took place on the 15th and 16th June 2013 at the Faculty of Engineering Sciences (Roberts Building) which is part of University College London (UCL). The conference dinner and the XML London 2013 DemoJam were held in the Jeremy Bentham Room at UCL, London.

The conference will be held annually using the same format in subsequent years with XML London 2014 taking place in June 2014.

— Charles Foster
Chairman, XML London

Building Rich Web Applications using XForms 2.0

Nick van den Bleeken

Inventive Designers

<nick.van.den.bleeken@inventivegroup.com>

Abstract

XForms is a cross device, host-language independent markup language for declaratively defining a data processing model of XML data and its User Interface. It reduces the amount of markup that has to be written for creating rich web-applications dramatically. There is no need to write any code to keep the UI in sync with the model, this is completely handled by the XForms processor.

XForms 2.0 is the next huge step forward for XForms, making it an easy to use framework for creating powerful web applications.

This paper will highlight the power of these new features, and show how they can be used to create real life web-applications. It will also discuss a possible framework for building custom components, which is currently still missing in XForms.

1. Introduction

Over the last 2 years there is a trend of moving away from browser plug-in frameworks (Adobe Flash, JavaFX, and Microsoft silverlight) in favor of HTML5/Javascript for building rich web-applications. This shift is driven by the recent advances in technology (HTML5 [HTML5], CSS [CSS] and Javascript APIs) and the vibrant browser market on one hand, and the recent security problems in those plug-in frameworks on the other hand.

Javascript is a powerful dynamic language, but a potential maintenance nightmare if one is not extremely diligent. Creating rich web-applications using javascript requires a lot of code. There are a lot of frameworks (like Dojo [DOJO] and jQuery [JQUERY]) that try to minimize the effort of creating user interfaces. Dojo even goes one step further by allowing you to create model-view-controller applications, but you still have to write a lot of javascript to glue everything together.

XForms is a cross device, host-language independent markup language for declaratively defining a data processing model of XML data and its User Interface. It uses a model-view-controller approach. The model consists of one or more XForms models describing the data, constraints and calculations based upon that data, and submissions. The view describes what controls appear in the UI, how they are grouped together, and to what data they are bound.

XForms reduces the amount of markup that has to be written for creating rich web-applications dramatically. There is no need to write any code to keep the UI in sync with the model, this is completely handled by the XForms processor.

XForms 2.0 is the next huge step forward for XForms, making it an easy to use framework for creating powerful web applications. This paper will first discuss the most important improvements in this new version of the specification, followed by an analysis of possible improvements.

2. XForm 2.0

This section will discuss the most important improvements of XForms compared to its previous version. Those improvements make it easier to create powerful web applications that integrate with data available on the web.

2.1. XPath 2.0

XPath 2.0 [XPATH-20] adds a much richer type system, greatly expands the set of functions and adds additional language constructs like 'for' and 'if'. These new language features make it much easier to specify constraints and calculations. At the same time it makes it easier to display the data the way you want in the UI.

doi:10.14337/XMLLondon13.Bleeken01

Example 1. XPath 2.0: Calculate order price
The folowing XPath expression calculates the sum of the multiplication of the price and quantity of each item in the order:

```
sum(
  for $n in order/item
  return $n/price * $n/quantity
)
```

2.2. Attribute Value Templates

Attribute Value Templates [AVT] allow you the use dynamic expressions virtually everywhere in the markup. They are not limited to the XForms elements, but are supported on most host language attributes. Attribute Value Templates enable even more powerful styling of your form based on the data. As an example, a form author can now easily highlight rows in a table based on certain data conditions (overdue, negative values, or complex conditions). In HTML5, this feature enables the form author to declaratively specify when certain css-classes apply to an element.

Example 2. AVT: Higlight overdue jobs

```
<xf:repeat ref="job">
  <tr class="{
    if(current-dateTime() > xs:dateTime(@due))
          then 'over-due' else ''
  }">
  ...
  </tr>
</xf:repeat>
```

2.3. Variables

Variables [VAR] make it possible to break down complex expressions into pieces and make it easier to understand the relationship of those pieces, by using expressive variable names and documenting those individual pieces and their relationships.

Variables also facilitate in de-duplication of XPath expressions in your form. In typical forms the same expression is used multiple times (e.g.: XPath expression that calculates the selected language in a multi-lingual UI).

Example 3. Variables

```
<xf:var name="paging" value="instance('paging')"/>
<xf:group>
  <xf:var name="total"
          value="$paging/@total"/>

  <xf:var name="page-size"
          value="$paging/@page-size"/>

  <xf:var name="page-count"
          value="($total + $page-size - 1)
          idiv $page-size"/>

  <xf:output value="$page-count">
    <xf:label>Number of pages</xf:label>
  </xf:output>
</xf:group>
```

2.4. Custom Functions

Custom functions [CUST_FUNC] like variables allow form authors to simplify expressions and prevent code duplication without using extensions.

Example 4. Custom Functions: Fibonacci

```
<function signature="
  my:fibonacci($n as xs:integer) as xs:integer">
   <var name="sqrt5" value="math:sqrt(5)"
   <result value="(
    math:power(1+$sqrt5, $n) -
    math:power(1-$sqrt5, $n)) div
    (math:power(2, $n) * $sqrt5)" />
</function>
```

2.5. Non-XML data

Because XForms' data model is XML it can consume data from a lot of sources with little effort (SOAP services, XML data bases using XQuery, REST XML services, ...). Starting from XForms 2.0, XForms can natively consume JSON data [JSON]. As more and more services on the web are starting to deliver JSON today this is an important feature. XForms implementations may support other non-XML file formats like CSV, vCard, ...

The form author can use all normal XForms constructs (binds, UI controls, actions,...) independant from the data format of the external source. The XForms processor will build an XPath data model from the recieved data.

2.6. Miscellaneous improvements

Other interesting new features are:

- Model based switching/repeats allows form authors to capture the state of the switch/repeat in the model, which makes it possible to save and restore the actual runtime state of the form.
- The iterate attribute makes it much easier to execute actions for all nodes matching a certain condition.
- Ability to specify the presentation context for the result of a replace-all submission. This feature makes it possible to display the result of a submission in another frame or tab, when you using HTML as your host language.
- MIP functions (e.g.: valid()) which can be used in actions and the user interface. They can for example be used to conditionally show content based on the value of a MIP on a data node.

3. Possible features for a future version of the specification

There are a lot of possible enhancements that could be made to the current specification. Ranging from improving UI events, better expression of conditional default values, structural constraints to better integration with other web platform technologies (Javascript, geo-location, Web Storage, Crypto, ...).

But in my opinion the most important thing that is still missing in the current version of the specification, is a framework for defining custom components that can be re-used in multiple forms/applications. This section will discuss possible requirements for such a framework and a proposal of a possible custom components framework which is heavily based on the framework that is currently implemented by Orbeon Forms [ORBEON].

4. Custom Components

Custom components should be easily re-usable over multiple forms/applications. They should feel and behave the same way as native XForms UI controls. It should also be possible to strongly encapsulate their internal data and logic. The following section will go into more depth about these requirements and how they could be implemented.

4.1. Using Custom Components

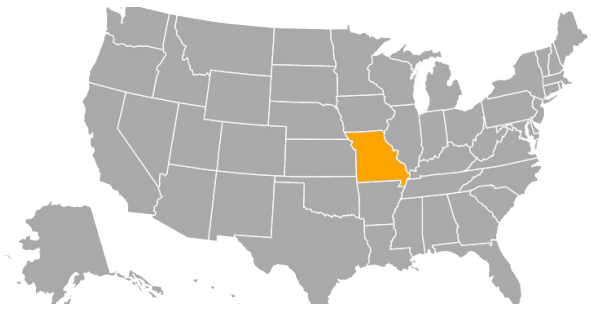
This section will discuss the aspects of a custom component that are relevant to the user of the component and demonstrate how custom controls can be used by the form author in an XForms document.

In general there are two different categories of custom components:

1. Components that are just a different appearance of an existing XForms control. An example of this is a graphical state selection component.

Example 5. Custom appearance: Graphical state selection component

```
<xf:select1 ref="state" appearance="cc:us-states">
  <xf:itemset ref="instance('states')">
    <xf:label ref="@name"/>
    <xf:value ref="@code"/>
  </xf:itemset/>
<xf:select1>
```



2. Advanced and/or complex user interface controls that are not an appearance of an existing XForms User Interface control. An example of such a control is a donut chart:

Example 6. Custom element: Donut chart

```
<cc:chart>
  <cc:slice-serie
    ref="instance('accounts')/account">

    <cc:label value="@label"/>
    <cc:name value="@id"/>
    <cc:value value="@amount"/>
  </cc:slice-serie>
</cc:chart>
```

As shown in the above example, the markup to use custom components is similar to the markup for native XForms controls. To use the first category of controls the form author just has to specify a custom appearance. For the second category new element names should be used, but the same constructs as for native form controls are used (ref-attribute for specifying the repeat sequence, value-attribute for the values, a structure similar to xf:itemset is used for cc:slices).

4.1.1. Container control

Some custom components, such as tabview, act like a container controls (xf:group, xf:switch, xf:repeat). Those controls have typically one or multiple insertion points, to which the custom control host's children are transposed. The transposed children can contain any host language and XForms content, which will be visible from the "outside" (e.g.: IDs are also visible to actions in the host document outside of this custom control).

The following example creates a tab view with two tabs (Host language and xforms markup can be used under the tab elements):

Example 7. Custom Container Control: Tab view

```
<cc:tabview>
  <cc:tab>
    <xf:input ref="foo">...</xf:input>
  </cc:tab>
  <cc:tab>...</cc:tab>
<cc:tabview>
```

4.1.2. Events

XForms uses XML events and XForms actions to execute various tasks during the form execution. The appropriate xforms events are dispatched to the custom control (e.g.: the data node related events like xforms-value-changed and xforms-readonly are sent to the control if the control has a data binding). The form author can attach event listeners to all custom controls just like he does for native XForms controls.

The following example attaches a value change listener to the custom pie chart control:

Example 8. Events: Attach handler to Custom Control

```
<cc:pie-chart ref="account">
  <cc:slices ref="instance('accounts')/account">
    <cc:label value="@label"/>
    <cc:name value="@id"/>
    <cc:value value="@amount"/>
  </cc:slices>

  <xf:action ev:event="xforms-value-changed">
      ...
  </xf:action>
</cc:pie-chart>
```

Custom events, which can be handled by the custom control, can be dispatched to the custom control using the xf:dispatch action.

The following example dispatches an event my-event to the control with id foo-bar when the trigger is activated:

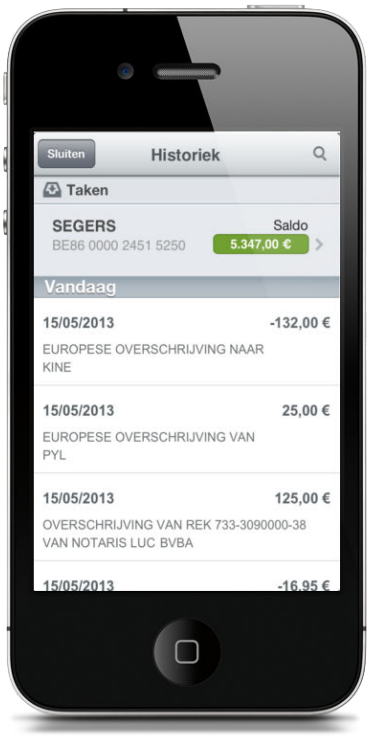
Example 9. Events: Dispatch event to Custom Control

```
<cc:foo-bar id="foo-bar">
...
<xf:trigger>
  <xf:label>Do it</xf:label>

  <xf:dispatch
    ev:event="DOMActivate"
    name="my-event"
    targeted="foo-bar"/>

</xf:trigger>
```

4.1.3. Responsive Design

When targeting a wide variety of devices with different capabilities (screen size/resolution, mouse/touch,…) and usages, it might be desirable to change the appearance of a control depending on the used device and or environment in which it is used. Examples of this are desktop versus mobile, but also landscape versus portrait on a tablet. This is currently not supported but it is something that should be considered for the next version of XForms, and might be related to the custom controls framework.

Example 10. Responsive Design: different appearence depending on orientation



4.2. Creating Custom Components

Implementing a custom component is typically done using a mixture of XForms and host language specific markup. There are a lot of possibilities on how to specify this implementation. A possibility is to extend the work done for XBL 2.0, but because this specification is no longer maintained it is probably better to specify something that is a bit more tailored to the XForms requirements.

A simple custom component that just wraps an input control, has a data binding and supports LHHA (label, hint, help and alert) might look like this:

Example 11. Custom Control: Implementation

```
<xf:component
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:cc="http://www.sample.com/custom-controls"
  id="foo-bar-id"
  element="cc:foo-bar"
  mode="data-binding lhha">
  <xf:template>
    <xf:input ref="xf:binding('foo-bar-id')"/>
  </xf:template>
</xf:component>
```

In the above example the mode attribute on the component element ensures that the custom control will support the data binding attributes (ref, context and bind) and supports the LHHA-elements.

4.3. Data and logic encapsulation

A custom component should be able to have private models to abstract and encapsulate their internal data and logic. This implies that a component can define its own instances, binds and submissions.

4.4. Event encapsulation

Events that are sent from and are targeted to elements inside the component should not be visible to the user of that component. But it should be possible to send events to, and receive events from, the user of the component.

To fulfill these requirements the elements of the custom control will reside in its own 'shadow' tree. But events dispatched to, and listener attached to, the root element of the custom control will be re-targeted to the host element.

4.5. Component lifecycle

Standard XForms initialization events (xforms-model-construct and xforms-model-construct-done and xforms-ready) and destruction events (xforms-model-destruct) will be sent to the models in the custom control when the custom control is created and destroyed respectively. A custom control can be created either when the form is loaded or when a new iteration is added to an xf:repeat. A custom control is destroyed when the XForms Processor is shutdown (e.g.: result of load action or submission with replace all) or if an iteration in an xf:repeat is removed.

The events are sent to the implementation of the custom control and therefore, not traverse any of the host document elements.

4.6. Styling

By default, styling should not cross the boundary between the host document and the component. In other words, the styling rules from the host document should not impact with the styling rules from the component and vice versa. But it should be possible to style parts of the custom control from within the host document that are explicitly specified as being style able by the custom controls' implementation. When using CSS as a styling language it is recommended to use custom pseudo elements, just like defined in Shadow DOM [SHADOW_DOM].

5. Conclusion

The new features in XForms 2.0 like XPath 2.0, Attribute Value Templates and variables make it easier to create web applications with XForms. The support of non-XML data sources ensures that the technology can be used to consume data from a variety of sources. One of the biggest strengths of XForms is its abstraction by declaratively defining its data processing model (dependencies , validations, calculations and data binding). But it is currently missing a standardized way for abstracting re-usable high level components, that can be used to build rich forms/applications. Hopefully such a frame is something that can be added in the next version of XForms.

References

[AVT]	http://www.w3.org/TR/xforms20/#avts
[CSS]	http://www.w3.org/TR/CSS/
[CUST_FUNC]	http://www.w3.org/TR/xforms20/#The_function_Element
[DOJO]	http://dojotoolkit.org/
[DOJO_DECL]	http://dojotoolkit.org/documentation/tutorials/1.8/declarative/
[HTML5]	http://www.w3.org/TR/html51/
[JQUERY]	http://jquery.com/
[JSON]	http://www.json.org/
[ORBEON]	http://www.orbeon.com/
[SHADOW_DOM]	http://www.w3.org/TR/shadow-dom/
[VAR]	http://www.w3.org/TR/xforms20/#structure-var-element
[XFORMS-20]	http://www.w3.org/TR/xforms20/
[XPATH-20]	http://www.w3.org/TR/2012/WD-xforms-xpath-20120807/

When MVC becomes a burden for XForms

Eric van der Vlist

Dyomedea

Abstract

XForms is gaining traction and is being used to develop complex forms, revealing its strengths but also its weaknesses.

One of the latest is not specific to XForms but inherent to the MVC (Model View Controller) architecture which is one of the bases of XForms.

In this talk we see how the MVC architecture dramatically affect the modularity and reusabilty of XForms developments and some of the solutions used to work around this flaw.

1. Practice: a quiz

Let's start with a quiz...

1.1. Basic XForms

1.1.1. Question

Given the following instance:

```
<xf:instance>
  <figures>
    <line>
      <length>
        <value>10</value>
        <unit>in</unit>
      </length>
    </line>
  </figures>
</xf:instance>
</xf:instance>
```

implement a standard XForms 1.1 form displaying the following user interface:

1.1.2. Answer

Model:

```
<xf:model>
  <xf:instance>
    <figures>
      <line>
        <length>
          <value>10</value>
          <unit>in</unit>
        </length>
      </line>
    </figures>
  </xf:instance>
</xf:model>
```

doi:10.14337/XMLLondon13.Vlist01

View:

```
<xf:group ref="line/length">
  <xf:input ref="value">
    <xf:label>Length: </xf:label>
  </xf:input>
  <xf:select1 ref="unit">
    <xf:label></xf:label>
    <xf:item>
      <xf:label>pixels</xf:label>
      <xf:value>px</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>font size</xf:label>
      <xf:value>em</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>font height</xf:label>
      <xf:value>ex</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>inches</xf:label>
      <xf:value>in</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>centimeters</xf:label>
      <xf:value>cm</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>millimeters</xf:label>
      <xf:value>mm</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>points</xf:label>
      <xf:value>pt</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>picas</xf:label>
      <xf:value>pc</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>%</xf:label>
      <xf:value>%</xf:value>
    </xf:item>
  </xf:select1>
</xf:group>
```

1.2. Using instances and actions

1.2.1. Question

Implement the same user interface if the instance uses the CSS2 / SVG 1.1 conventions for sizes:

```
<xf:instance id="main">
  <figures>
    <line length="10in"/>
  </figures>
</xf:instance>
```

1.2.2. Answer

Model:

```
<xf:model>
  <xf:instance id="main">
    <figures>
      <line length="10in"/>
    </figures>
  </xf:instance>
  <xf:instance id="split">
    <line>
      <length>
        <value/>
        <unit/>
      </length>
    </line>
  </xf:instance>
  .../...
</xf:model>
```

View:

```
<xf:group ref="instance('split')/length">
  <xf:input ref="value" id="length-control">
    <xf:label>Length: </xf:label>
  </xf:input>
  <xf:select1 ref="unit" id="unit-control">
    <xf:label/>
    <xf:item>
      <xf:label>pixels</xf:label>
      <xf:value>px</xf:value>
    </xf:item>
    .../...
    <xf:item>
      <xf:label>%</xf:label>
      <xf:value>%</xf:value>
    </xf:item>
  </xf:select1>
</xf:group>
```

Controller:

```
<xf:model>
  .../...
  <xf:action ev:event="xforms-ready">
    <xf:setvalue
            ref="instance('split')/length/value"
            value="translate(
                   instance('main')/line/@length,
                   '%incmptxe',
                    '')" />
    <xf:setvalue ref="instance('split')/length/unit"
              value="translate(
                      instance('main')/line/@length,
                      '0123456789',
                      '')" />
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="length-control">
    <xf:setvalue
      ref="instance('main')/line/@length"
      value="concat(
              instance('split')/length/value,
              instance('split')/length/unit)" />
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="unit-control">
    <xf:setvalue
      ref="instance('main')/line/@length"
      value="concat(
              instance('split')/length/value,
              instance('split')/length/unit)" />
  </xf:action>
</xf:model>
```

1.3. Modularity

1.3.1. Question

Still using XForms 1.1 standard features, extend this user interface to edit the height and width of a rectangle:

```
<xf:instance id="main">
  <figures>
    <rectangle height="10in" width="4em"/>
  </figures>
</xf:instance>
```

Hint: copy/paste is your friend!

1.3.2. Answer

Model:

```
<xf:model>
  <xf:instance id="main">
    <figures>
      <rectangle height="10in" width="4em"/>
    </figures>
  </xf:instance>
  <xf:instance id="height">
    <height>
      <value/>
      <unit/>
    </height>
  </xf:instance>
  .../...
  <xf:instance id="width">
    <width>
      <value/>
      <unit/>
    </width>
  </xf:instance>
  .../...
</xf:model>
```

View:

```
<xf:group ref="instance('height')">
  <xf:input ref="value" id="height-value-control">
    <xf:label>Height: </xf:label>
  </xf:input>
  <xf:select1 ref="unit" id="height-unit-control">
    <xf:label/>
    <xf:item>
      <xf:label>pixels</xf:label>
      <xf:value>px</xf:value>
    </xf:item>
    .../...
  </xf:select1>
</xf:group>
<xh:br/>
<xf:group ref="instance('width')">
  <xf:input ref="value" id="width-value-control">
    <xf:label>Width: </xf:label>
  </xf:input>
  <xf:select1 ref="unit" id="width-unit-control">
    <xf:label/>
    <xf:item>
      <xf:label>pixels</xf:label>
      <xf:value>px</xf:value>
    </xf:item>
    .../...
  </xf:select1>
</xf:group>
```

Controller:

```
<xf:model>
  .../...
  <xf:action ev:event="xforms-ready">
    <xf:setvalue ref="instance('height')/value"
      value="translate(
               instance('main')/rectangle/@height,
               '%incmptxe', '')" />
    <xf:setvalue ref="instance('height')/unit"
      value="translate(
        instance('main')/rectangle/@height,
        '0123456789', '')"/>
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="height-value-control">
    <xf:setvalue
      ref="instance('main')/rectangle/@height"
      value="concat(instance('height')/value,
                    instance('height')/unit)" />
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="height-unit-control">
    <xf:setvalue
      ref="instance('main')/rectangle/@height"
      value="concat(
               instance('height')/value,
               instance('height')/unit)" />
  </xf:action>
  .../...
  <xf:action ev:event="xforms-ready">
    <xf:setvalue ref="instance('width')/value"
      value="translate(
        instance('main')/rectangle/@width,
        '%incmptxe', '')"/>
    <xf:setvalue ref="instance('width')/unit"
      value="translate(
        instance('main')/rectangle/@width,
        '0123456789', '')"/>
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="width-value-control">
    <xf:setvalue
      ref="instance('main')/rectangle/@width"
      value="concat(instance('width')/value,
                    instance('width')/unit)" />
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="width-unit-control">
    <xf:setvalue
      ref="instance('main')/rectangle/@width"
      value="concat(instance('width')/value,
                    instance('width')/unit)"/>
  </xf:action>
</xf:model>
```

1.4. Homework: repeated content

Still using standard XForms features, extend this form to support any number of rectangles in the instance.

Hint: you will not be able to stick to atomic instances for the width and height but act more globally and maintain instances with a set of dimensions which you'll have to keep synchronized with the main instance when rectangles are inserted or deleted.

1.5. What's the problem?

XForms lacks a feature to define and use "components" that would package a group of controls together with their associated model and actions.

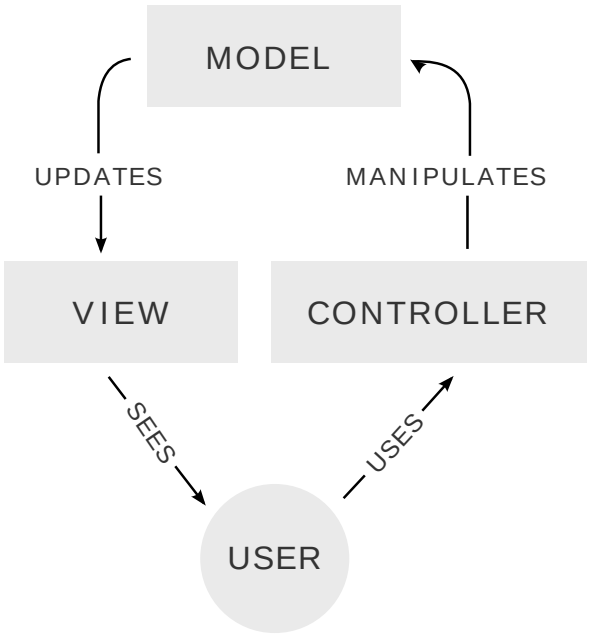
2. Theory: the MVC design pattern

XForms describes itself as a MVC architecture:

> An XForm allows processing of data to occur using three mechanisms:
>
> - a declarative **model** composed of formulae for data calculations and constraints, data type and other property declarations, and data submission parameters
> - a **view** layer composed of intent-based user interface controls
> - an imperative **controller** for orchestrating data manipulations, interactions between the model and view layers, and data submissions.

Micah Dubinko argues that the mapping is more obvious with Model-view-presenter (MVP), a derivative of the MVC software pattern but that's not the point I'd like to make and I'll stick to the MVC terminology where:

- The model is composed of XForms instances and binds
- The view is composed of the XForms controls together with the HTML elements and CSS stylesheets
- The controller is composed of the actions



Mode-view-controler on wikimedia

Orbeon Form Builder/Form Runner go one step forward and add a fourth concern for localization and we get a model/view/localization/controller pattern.

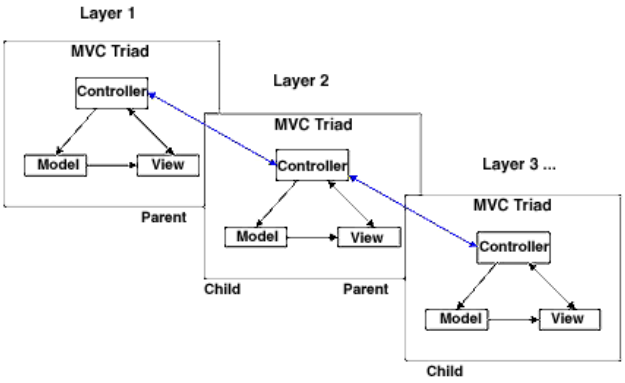
This separation of concerns is great to differentiate different roles and split work between specialists but doesn't play well with modularity and reusability.

I am currently working on a project to develop big and complex forms and this is becoming one of the biggest issues: these forms share a number of common fields and group of fields and, not even speaking of sharing these definitions, this separation of concerns adds a significant burden when copying these definitions from one form to another.

To copy a field from one form to another you need to copy definitions from the model, the view, the localization and the controller and can't just copy a "component".

And of course, there is no easy way to reuse common components instead of copying them.

This kind of issue is common with the MVC design pattern and the Hierarchical model–view–controller (HMVC) has been introduced for this purpose, but how can we use such a pattern with XForms?



Hierarchical model-view-controller in JavaWorld

3. Solutions

A number of solutions are being used to work around this issue with XForms.

3.1. Copy/Paste

This is what we've done for our quiz and we've seen that this is easy -but very verbose and hard to maintain- until we start to deal with repeated content.

I would guess that this is the most common practice when fields (or group of fields) are being reused in XForms though!

3.2. XForms generation or templating

We're XML developers, aren't we? When something is verbose we can use XSLT or any other tool to generate it and XForms is no exception.

XForms can be generated from any kind of model including annotated schemas or other vocabularies such as DDI (we'll be presenting this option at the Balisage International Symposium on Native XML User Interfaces in August.

Projects without any obvious model formats in mind often chose to transform simplified versions of XForms into plain XForms. In that case the approach may tends toward a templating system where placeholders are inserted into XForms documents to be transformed into proper XForms.

We may want for instance to define `<my:dimension/>` placeholders which would look like XForms controls and generate the whole model, view and controller XForms definitions.

The source form would then be something as simple as:

```
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
         xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:my="http://ns.dyomedea.com/my-components/">
  <xh:head>
    <xh:title>Template</xh:title>
    <xf:model>
      <xf:instance id="main">
        <figures>
          <rectangle height="10in" width="4em"/>
        </figures>
      </xf:instance>
    </xf:model>
  </xh:head>
  <xh:body>
    <my:dimension ref="rectangle/@height">
      <xf:label>Height</xf:label>
    </my:dimension>
    <br/>
    <my:dimension ref="rectangle/@width">
      <xf:label>Width</xf:label>
    </my:dimension>
  </xh:body>
</xh:html>
```

A simplistic version of a transformation to process this example is not overly complex. The controls are quite easy to generate from the placeholders:

```
<xsl:template match="my:dimension">
  <xsl:variable name="id"
    select="if (@id) then @id else generate-id()"/>
  <xf:group ref="instance('{$id}-instance')">
    <xf:input ref="value" id="{$id}-value-control">
      <xsl:apply-templates/>
    </xf:input>
    <xf:select1 ref="unit" id="{$id}-unit-control">
      <xf:label/>
      <xf:item>
        <xf:label>pixels</xf:label>
        <xf:value>px</xf:value>
      </xf:item>
      .../...
    </xf:select1>
  </xf:group>
</xsl:template>
```

A model can be appended to the `<xh:head/>` element:

```
<xsl:template match="xh:head">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"
                         mode="#current" />
    <xf:model>
      <xsl:apply-templates select="//my:dimension"
                           mode="model" />
    </xf:model>
  </xsl:copy>
</xsl:template>
```

And the instances and actions can be generated similarly:

```
<xsl:template match="my:dimension" mode="model">
  <xsl:variable name="id"
    select="if (@id) then @id else generate-id()"/>
  <xf:instance id="{$id}-instance">
    <height>
      <value/>
      <unit/>
    </height>
  </xf:instance>
  <xf:action ev:event="xforms-ready">
    <xf:setvalue
      ref="instance('{$id}-instance')/value"
      value="translate(instance('main')/{@ref},
                       '%incmptxe', '')"/>
    <xf:setvalue
      ref="instance('{$id}-instance')/unit"
      value="translate(instance('main')/{@ref},
                       '0123456789', '')"/>
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="{$id}-value-control">
    <xf:setvalue ref="instance('main')/{@ref}"
    value="concat(
             instance('{$id}-instance')/value,
             instance('{$id}-instance')/unit)" />
  </xf:action>
  <xf:action ev:event="xforms-value-changed"
             ev:observer="{$id}-unit-control">
    <xf:setvalue ref="instance('main')/{@ref}"
      value="concat(
        instance('{$id}-instance')/value,
        instance('{$id}-instance')/unit)" />
  </xf:action>
</xsl:template>
```

As always, the devil is in details and this would be far from perfect:

- In actions, references to the main instance do not take into account the context node under which the `<my:dimension/>` placeholder is defined (paths are therefore expected to be relative to the default instance). Mimicking the behavior of an XForms control and its support of the context node would be much more challenging.
- Supporting repetitions would be another challenge.

3.3. Orbeon Forms' XBL implementation

Orbeon's component architecture is inpired by XBL 2.0 which describes itself as:

> XBL (the Xenogamous Binding Language) describes the ability to associate elements in a document with script, event handlers, CSS, and more complex content models, which can be stored in another document. This can be used to re-order and wrap content so that, for instance, simple HTML or XHTML markup can have complex CSS styles applied without requiring that the markup be polluted with multiple semantically neutral div elements.
>
> It can also be used to implement new DOM interfaces, and, in conjunction with other specifications, enables arbitrary tag sets to be implemented as widgets. For example, XBL could be used to implement the form controls in XForms or HTML.
>
> --XBL 2.0

Even if this specification is no longer maintained by the W3C Web Applications Working Group, the concepts described in XBL 2.0 fit very nicely in the context of XForms documents even though the syntax may sometimes look strange, such as when CSS selectors are used where XPath patterns would look more natural in XForms documents.

Note

The syntax of XBL declarations has been changed between Orbeon Forms version 3 and 4. The syntax shown in this paper is the syntax of version 4.

The definition of an XBL component to implement our dimension widget would be composed of three parts: handlers, implementation and template:

```
<xbl:binding id="my-dimension"
             element="my|dimension"
             xxbl:mode="lhha binding value">
  <xbl:handlers>
    .../...
  </xbl:handlers>
  <xbl:implementation>
    .../...
  </xbl:implementation>
  <xbl:template>
    .../...
  </xbl:template>
</xbl:binding>
```

A fourth element could be added to define component specific resources such as CSS stylesheets.

The XForms component's model goes into the implementation:

```
<xbl:implementation>
  <xf:model id="my-dimension-model">
    <xf:instance id="my-dimension-instance">
      <dimension>
        <value/>
        <unit/>
      </dimension>
    </xf:instance>
    .../...
</xbl:implementation>
```

The XForms component's controls are defined into the template:

```
<xbl:template>
  <xf:input ref="value"
            id="my-dimension-value-control"/>
  <xf:select1 ref="unit"
              id="my-dimension-unit-control">
    <xf:label/>
    <xf:item>
      <xf:label>pixels</xf:label>
      <xf:value>px</xf:value>
    </xf:item>
    .../...
  </xf:select1>
</xbl:template>
```

The XForms actions are split between the handlers and the implementation (or the template): handlers are used to define actions triggered by events which are external to the component (such as in our case `xforms-ready`) while traditional XForms actions are used to handle events "internal" to the component such as user actions.
The handlers would thus be:

```
<xbl:handlers>
  <xbl:handler
      event="xforms-enabled xforms-value-changed">

    <xf:setvalue
      ref="instance('my-dimension-instance')/value"
      value="translate(
        xxf:binding('my-dimension'),
        '%incmptxe', '')" />

    <xf:setvalue
      ref="instance('my-dimension-instance')/unit"
      value="translate(
        xxf:binding('my-dimension'),
        '0123456789', '')" />
  </xbl:handler>
</xbl:handlers>
```

And the remaining actions:

```
<xbl:implementation>
  <xf:model id="my-dimension-model">
    .../...
    <xf:setvalue ev:event="xforms-value-changed"
           ev:observer="my-dimension-value-control"
      ref="xxf:binding('my-dimension')"
      value="
        concat(
          instance('my-dimension-instance')/value,
          instance('my-dimension-instance')/unit)" />
    <xf:setvalue ev:event="xforms-value-changed"
                 ev:observer="my-dimension-unit-control"
      ref="xxf:binding('my-dimension')"
      value="
      concat(
        instance('my-dimension-instance')/value,
        instance('my-dimension-instance')/unit)"/>
  </xf:model>
</xbl:implementation>
```

I won't go into the details which are described in Orbeon's XBL - Guide to Using and Writing XBL Components but it is worth noting that there is a strict encapsulation of both the model, the view and the controller of this component that seen from the outside acts as a standard XForms control.

Of course, this component can be used as a standard XForms control:

```
<xh:body>
  <my:dimension ref="rectangle/@height">
    <xf:label>Height</xf:label>
  </my:dimension>
  <br/>
  <my:dimension ref="rectangle/@width">
    <xf:label>Width</xf:label>
  </my:dimension>
</xh:body>
```

The complete form with the component definition would be:

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl" type="text/xsl"?>
<?xsltforms-options debug="yes"?>
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml" xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:xxf="http://orbeon.org/oxf/xml/xforms" xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xbl="http://www.w3.org/ns/xbl" xmlns:xxbl="http://orbeon.org/oxf/xml/xbl"
  xmlns:fr="http://orbeon.org/oxf/xml/form-runner" xmlns:my="http://ns.dyomedea.com/my-components/">
  <xh:head>
    <xh:title>Simple XBL Component</xh:title>
    <xbl:xbl script-type="application/xhtml+xml">
      <xbl:binding id="my-dimension" element="my|dimension" xxbl:mode="lhha binding value">
        <xbl:handlers>
          <xbl:handler event="xforms-enabled xforms-value-changed">
            <xf:setvalue ref="instance('my-dimension-instance')/value"
                         value="translate(xxf:binding('my-dimension'), '%incmptxe', '')"/>
            <xf:setvalue ref="instance('my-dimension-instance')/unit"
                         value="translate(xxf:binding('my-dimension'), '0123456789', '')"/>
          </xbl:handler>
        </xbl:handlers>
        <xbl:implementation>
          <xf:model id="my-dimension-model">
            <xf:instance id="my-dimension-instance">
              <dimension>
                <value/>
                <unit/>
              </dimension>
            </xf:instance>
            <xf:setvalue ev:event="xforms-value-changed"
                         ev:observer="my-dimension-value-control"
                         ref="xxf:binding('my-dimension')"
                         value="concat(instance('my-dimension-instance')/value,
                                       instance('my-dimension-instance')/unit)"/>
            <xf:setvalue ev:event="xforms-value-changed"
                         ev:observer="my-dimension-unit-control"
                         ref="xxf:binding('my-dimension')"
                         value="concat(instance('my-dimension-instance')/value,
                                       instance('my-dimension-instance')/unit)"/>
          </xf:model>
        </xbl:implementation>
        <xbl:template>
          <xf:input ref="value" id="my-dimension-value-control"/>
          <xf:select1 ref="unit" id="my-dimension-unit-control">
            <xf:label/>
            <xf:item>
              <xf:label>pixels</xf:label>
              <xf:value>px</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>font size</xf:label>
              <xf:value>em</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>font height</xf:label>
              <xf:value>ex</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>inches</xf:label>
              <xf:value>in</xf:value>
            </xf:item>
```

```
            <xf:item>
              <xf:label>centimeters</xf:label>
              <xf:value>cm</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>millimeters</xf:label>
              <xf:value>mm</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>points</xf:label>
              <xf:value>pt</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>picas</xf:label>
              <xf:value>pc</xf:value>
            </xf:item>
            <xf:item>
              <xf:label>%</xf:label>
              <xf:value>%</xf:value>
            </xf:item>
          </xf:select1>
        </xbl:template>
      </xbl:binding>
    </xbl:xbl>

    <xf:model>
      <xf:instance id="main">
        <figures>
          <rectangle height="10in" width="4em"/>
        </figures>
      </xf:instance>

    </xf:model>
  </xh:head>
  <xh:body>
    <my:dimension ref="rectangle/@height">
      <xf:label>Height</xf:label>
    </my:dimension>
    <br/>
    <my:dimension ref="rectangle/@width">
      <xf:label>Width</xf:label>
    </my:dimension>

    <fr:xforms-inspector/>
  </xh:body>
</xh:html>
```

3.4. Subforms

Subforms are implemented by XSLTForms and betterFORM. They have been considered for inclusion in XForms 2.0 but no consensus have been reached and they won't be included in 2.0.

There are a number of differences between the XSLTForms and Betterform implementations but the principle -and the shortcomings- are the same.

The basic principle behind subforms is to embed (or load) a form within another one. This embedding must be specifically performed using an `<xf:load>` action with a `@show="embed"` attribute. Subforms can also be unloaded.

The fact that subforms are explicitly loaded and unloaded in their "master" form is a key feature for big forms where this mechanism reduces the consumption of resources and leads to important performance improvements.

3.4.1. Subforms, betterFORM flavor

Subforms are described, in the betterFORM documentation, as “a way to avoid redundancies and keep the documents maintainable”:

> As XForms follows a MVC architecture the XForms model is the first logical candidate when decomposing larger forms into smaller pieces. Aggregating more complex forms from little snippets of UI (or snippets of a model) is a limited approach as the interesting parts are located on the bind Elements. This is where controls learn about their constraints, states, calculations and data types. Instead of just glueing pieces of markup together the inclusion of complete models allow the reuse of all the semantics defined within them.
>
> --betterFORM "Modularizing forms"

Joern Turner, founder of Chiba and co-founder of betterFORM, makes it clear that subforms haven't been introduced to implement components, though:

> Sorry i need to get a bit philosophic here but subforms are called subforms as they are *not* components ;) I don't want to dive into academic discussions here but the main difference for us is that from a component you would expect to use it as a part of a form as a means to edit one or probably several values in your form and encapsulate the editing logic inside of it. A subform on the other hand should be designed to be completely standalone. Typically we build our subforms as complete xhtml documents which can be run and tested standalone without being a part of a host document.
>
> --Joern Turner on the betterform-users mailing list

A proper solution for components, based on Web Components) should be implemented in betterFORM 6:

> We have also considered to implement this [XBL] but decided against it due to the acceptance and future of XBL and due to the fact that we found it overly complex and academic. We will come up with our own component model in betterFORM 6 which will orient at more modern approaches (Web Components).
>
> --Joern Turner on the betterform-users mailing list

In the meantime it is still possible to use subforms to design component like features assuming we take into account the following constraints:

- Communications between the master form and the subform are done using either in memory submissions (ContextSubmissionHandler identified by a `model:` pseudo protocol), the `instanceOfModel()` function which gives access to instances from other models or custom events passing context information.
- There is no id collision between the main form and the subforms which are loaded simultaneously.

This second constraint should be released in the future but the current version of the processor doesn't address it. In practice it means that in our sample we cannot load simultaneously an instance of the subform to edit the width and a second instance to edit the height but we can still take a "master/slave approach" where a single instance of the subform will be used to edit the width and the height separately or mimic an "update in place feature" where an instance of the subform will replace the display of the width or height.

A way to implement our example using these principles could be:

- In the master form:
  - Define an instance used as an interface with the subform to carry the value to edit and identify the active subform instance.
  - Include triggers to load and unload subforms.
  - Define actions to load and unload the subforms and maintain the "interface" instance.
  - Control when to display the triggers to avoid that simultaneous loads of the subform.
- In the subforms:
  - Synchronize the local model with the instance used as an interface.
  - Perform all the logic attached to the component.

The master form would then be:

```
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml" xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xf="http://www.w3.org/2002/xforms">
  <xh:head>
    <xh:title>Subforms</xh:title>
    <xf:model id="master">
      <xf:instance id="main">
        <figures>
          <rectangle height="10in" width="4em"/>
        </figures>
      </xf:instance>

      <!-- Instance used as an "interface" with the subform -->
      <xf:instance id="dimension-interface">
        <dimension active=""/>
      </xf:instance>
    </xf:model>

    <!-- Dirty hack to style controls inline -->
    <xh:style type="text/css">

      .xfContainer div {
      display: inline !important;
      }

      .xfContainer span {
      display: inline !important;
      }

    </xh:style>
  </xh:head>
  <xh:body>
    <xf:group ref="rectangle">
      <!-- Height -->
      <xf:group ref="@height">
        <xf:label>Height: </xf:label>
        <!-- This should be displayed when the subform is not editing the height -->
        <xf:group ref=".[instance('dimension-interface')/@active!='height']">
          <xf:output ref="."/>
          <!-- Display the trigger when the subform is not loaded anywhere -->
          <xf:trigger ref=".[instance('dimension-interface')/@active = '']">
            <xf:label>Edit</xf:label>
            <xf:action ev:event="DOMActivate">
              <!-- Set the value of the interface instance -->
              <xf:setvalue ref="instance('dimension-interface')"
                           value="instance('main')/rectangle/@height"/>
              <!-- Remember that we are editing the height -->
              <xf:setvalue ref="instance('dimension-interface')/@active">height</xf:setvalue>
              <!-- Load the subform -->
              <xf:load show="embed" targetid="height" resource="subform-embedded.xhtml"/>
            </xf:action>
          </xf:trigger>
        </xf:group>
        <xh:div id="height"/>
        <!-- This should be displayed only when we're editing the height -->
        <xf:group ref=".[instance('dimension-interface')/@active='height']">
          <xf:trigger>
            <xf:label>Done</xf:label>
            <xf:action ev:event="DOMActivate">
              <!-- Copy the value from the interface instance -->
              <xf:setvalue value="instance('dimension-interface')"
```

```
                        ref="instance('main')/rectangle/@height"/>
            <!-- We're no longer editing any dimension -->
            <xf:setvalue ref="instance('dimension-interface')/@active"/>
            <!-- Unload the subform -->
            <xf:load show="none" targetid="height"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
    </xf:group>
    <br/>
    <!-- Width -->
    <xf:group ref="@width">
      <xf:label>Width: </xf:label>
      <xf:group ref=".[instance('dimension-interface')/@active!='width']">
        <xf:output ref="."/>
        <xf:trigger ref=".[instance('dimension-interface')/@active = '']">
          <xf:label>Edit</xf:label>
          <xf:action ev:event="DOMActivate">
            <xf:setvalue ref="instance('dimension-interface')"
                         value="instance('main')/rectangle/@width"/>
            <xf:setvalue ref="instance('dimension-interface')/@active">width</xf:setvalue>
            <xf:load show="embed" targetid="width" resource="subform-embedded.xhtml"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
      <xh:div id="width"/>
      <xf:group ref=".[instance('dimension-interface')/@active='width']">
        <xf:trigger>
          <xf:label>Done</xf:label>
          <xf:action ev:event="DOMActivate">
            <xf:setvalue value="instance('dimension-interface')"
                         ref="instance('main')/rectangle/@width"/>
            <xf:setvalue ref="instance('dimension-interface')/@active"/>
            <xf:load show="none" targetid="width"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
    </xf:group>
  </xh:body>
</xh:html>
```

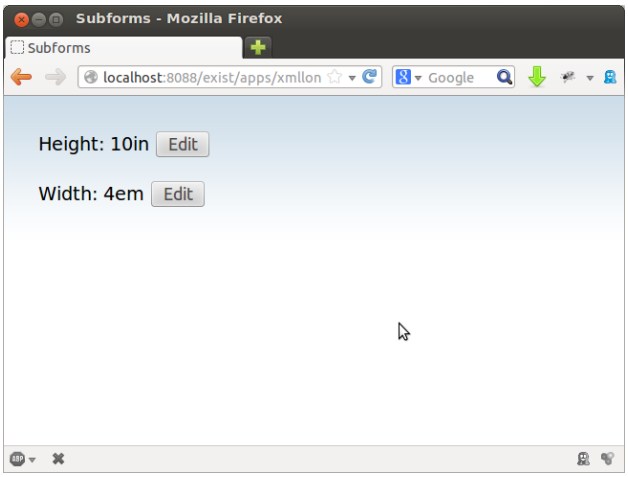
And the subform:

```
<xh:div xmlns:xh="http://www.w3.org/1999/xhtml" xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xf="http://www.w3.org/2002/xforms">
  <xf:model id="dimension-model">
    <xf:instance id="concat">
      <data/>
    </xf:instance>
    <xf:instance id="split">
      <height>
        <value/>
        <unit/>
      </height>
    </xf:instance>
    <!-- Get the value from the "interface" instance and initialize the   -->
    <xf:submission id="get-dimension-value"
                   resource="model:master#instance('dimension-interface')/*"
                   replace="instance" method="get">
      <xf:action ev:event="xforms-submit-done">
        <!--<xf:message level="ephemeral">Subform has updated itself.</xf:message>-->
        <xf:setvalue ref="instance('split')/value"
                     value="translate(instance('concat'), '%incmptxe', '')"/>
        <xf:setvalue ref="instance('split')/unit"
                     value="translate(instance('concat'), '0123456789', '')"/>
      </xf:action>
      <xf:message ev:event="xforms-submit-error" level="ephemeral">
        Error while subform update.
      </xf:message>
    </xf:submission>
    <xf:send ev:event="xforms-ready" submission="get-dimension-value"/>
    <xf:submission id="set-dimension-value" resource="model:master#instance('dimension-interface')/*"
                   replace="none" method="post">
      <xf:action ev:event="xforms-submit-done">
        <!--<xf:message level="ephemeral">Main form has been updated</xf:message>-->
      </xf:action>
      <xf:message ev:event="xforms-submit-error" level="ephemeral">
        Error while main form update.
      </xf:message>
    </xf:submission>
  </xf:model>
  <xf:group ref="instance('split')">
    <xf:input ref="value">
      <xf:action ev:event="xforms-value-changed">
        <xf:setvalue ref="instance('concat')"
                     value="concat(instance('split')/value, instance('split')/unit)"/>
        <xf:send submission="set-dimension-value"/>
      </xf:action>
    </xf:input>
    <xf:select1 ref="unit">
      <xf:action ev:event="xforms-value-changed">
        <xf:setvalue ref="instance('concat')"
                     value="concat(instance('split')/value, instance('split')/unit)"/>
        <xf:send submission="set-dimension-value"/>
      </xf:action>
      <xf:item>
        <xf:label>pixels</xf:label>
        <xf:value>px</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>font size</xf:label>
        <xf:value>em</xf:value>
      </xf:item>
```

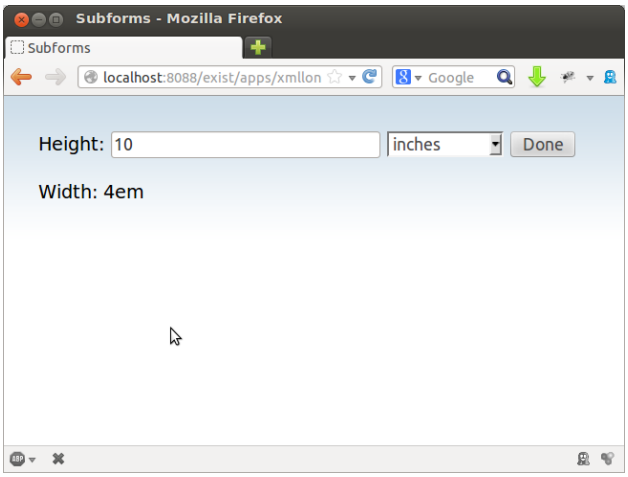
```
      <xf:item>
        <xf:label>font height</xf:label>
        <xf:value>ex</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>inches</xf:label>
        <xf:value>in</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>centimeters</xf:label>
        <xf:value>cm</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>millimeters</xf:label>
        <xf:value>mm</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>points</xf:label>
        <xf:value>pt</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>picas</xf:label>
        <xf:value>pc</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>%</xf:label>
        <xf:value>%</xf:value>
      </xf:item>
    </xf:select1>
  </xf:group>
</xh:div>
```

The code for defining the subform has the same level of complexity than the definition of the XBL in Orbeon Forms but a lot of geeky stuff needs to be added around the invocation of the form which becomes tricky.

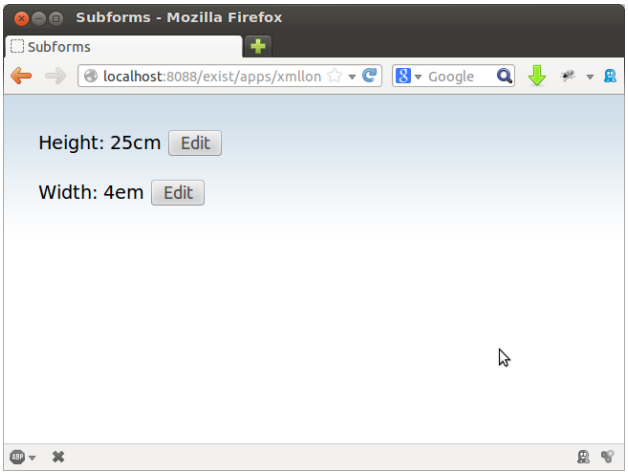
From a user perspective, the page would initially look like:



When a user clicks on one of the "Edit" buttons, the corresponding subform is loaded (note that all the "Edit" buttons have disappeared):



Once the user is done editing the values in this subform, (s)he can click on "Done" to come back to a state where both the height and width are displayed and can be edited:



The presentation can be improved replacing for instance the buttons by trendy icons but we had to bend our requirements to get something that can be implemented with subforms.

Of course here we are misusing subforms to implement components, something which was not a design goal, and it's not surprising that the resulting code is more verbose and that we've had to accept a different user interface. The future component feature announced by Joern Turner should solve these glitches.

3.4.2. Subforms, XSLTForms flavor

The support of subforms in XSLTForms is illustrated by a sample: a writers.xhtml master form embeds a books.xhtml subform.

The main principle behind this subform implementation appears to be the same than for betterFORM but there are some important differences between these two implementations:

- XSLTForms doesn't isolate the models from the master form and its subform and it is possible to access directly to any instance of the master form from the subforms.
- The features to communicate between models implemented by betterFORM are thus not necessary and do not exist in XSLTForms.
- The context node is not isolated and is available directly from the controls in the subform (see the writers/books example for an illustration.
- A specific action (`xf:unload`) is used to unload subforms in XSLTForms while an `xf:load` action with an `@show="none"` attribute is used in betterFORM for the same purpose.

With these differences, the code developed for betterFORM could be adapted to work with XSLTForms as:

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl" type="text/xsl"?>
<?xsltforms-options debug="yes"?>
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
         xmlns:ev="http://www.w3.org/2001/xml-events"
         xmlns:xf="http://www.w3.org/2002/xforms">
  <xh:head>
    <xh:title>Subforms</xh:title>
    <xf:model id="master">
      <xf:instance id="main">
        <figures>
          <rectangle height="10in" width="4em"/>
        </figures>
      </xf:instance>

      <!-- Instance used as an "interface" with the subform -->
      <xf:instance id="dimension-interface">
        <dimension active=""/>
      </xf:instance>
    </xf:model>

    <!-- Dirty hack to style controls inline -->
    <xh:style type="text/css"><![CDATA[

      .xforms-group-content, .xforms-group, span.xforms-control, .xforms-label {
      display:inline;
      }
```

```
    ]]>
  </xh:style>
</xh:head>
<xh:body>
  <xf:group ref="rectangle">
    <!-- Height -->
    <xf:group ref="@height">
      <xf:label>Height: </xf:label>
      <!-- This should be displayed when the subform is not editing the height -->
      <xf:group ref=".[instance('dimension-interface')/@active!='height']">
        <xf:output ref="."/>
        <!-- Display the trigger when the subform is not loaded anywhere -->
        <xf:trigger ref=".[instance('dimension-interface')/@active = '']">
          <xf:label>Edit</xf:label>
          <xf:action ev:event="DOMActivate">
            <!-- Set the value of the interface instance -->
            <xf:setvalue ref="instance('dimension-interface')"
                         value="instance('main')/rectangle/@height"/>
            <!-- Remember that we are editing the height -->
            <xf:setvalue ref="instance('dimension-interface')/@active">height</xf:setvalue>
            <!-- Load the subform -->
            <xf:load show="embed" targetid="height" resource="subform-embedded.xml"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
      <xh:span id="height"/>
      <!-- This should be displayed only when we're editing the height -->
      <xf:group ref=".[instance('dimension-interface')/@active='height']">
        <xf:trigger>
          <xf:label>Done</xf:label>
          <xf:action ev:event="DOMActivate">
            <!-- Copy the value from the interface instance -->
            <xf:setvalue value="instance('dimension-interface')"
                         ref="instance('main')/rectangle/@height"/>
            <!-- We're no longer editing any dimension -->
            <xf:setvalue ref="instance('dimension-interface')/@active"/>
            <!-- Unload the subform -->
            <xf:unload targetid="height"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
    </xf:group>
    <br/>
    <!-- Width -->
    <xf:group ref="@width">
      <xf:label>Width: </xf:label>
      <xf:group ref=".[instance('dimension-interface')/@active!='width']">
        <xf:output ref="."/>
        <xf:trigger ref=".[instance('dimension-interface')/@active = '']">
          <xf:label>Edit</xf:label>
          <xf:action ev:event="DOMActivate">
            <xf:setvalue ref="instance('dimension-interface')"
                         value="instance('main')/rectangle/@width"/>
            <xf:setvalue ref="instance('dimension-interface')/@active">width</xf:setvalue>
            <xf:load show="embed" targetid="width" resource="subform-embedded.xml"/>
          </xf:action>
        </xf:trigger>
      </xf:group>
      <xh:span id="width"/>
      <xf:group ref=".[instance('dimension-interface')/@active='width']">
        <xf:trigger>
```

```
            <xf:label>Done</xf:label>
            <xf:action ev:event="DOMActivate">
              <xf:setvalue value="instance('dimension-interface')"
                           ref="instance('main')/rectangle/@width"/>
              <xf:setvalue ref="instance('dimension-interface')/@active"/>
              <xf:unload targetid="width"/>
            </xf:action>
          </xf:trigger>
        </xf:group>
      </xf:group>
    </xf:group>
  </xh:body>
</xh:html>
```

for the main form and:

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl"
                 type="text/xsl"?>
<?xsltforms-options debug="yes"?>
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
         xmlns:xf="http://www.w3.org/2002/xforms"
         xmlns:ev="http://www.w3.org/2001/xml-events">
  <xh:head>
    <xh:title>A subform</xh:title>
    <xf:model id="subform-model">
      <xf:instance id="split">
        <height>
          <value/>
          <unit/>
        </height>
      </xf:instance>
      <xf:action ev:event="xforms-subform-ready">
        <xf:setvalue ref="instance('split')/value"
          value="translate(
          instance('dimension-interface'),
          '%incmptxe', '')"/>
        <xf:setvalue ref="instance('split')/unit"
          value="translate(
          instance('dimension-interface'),
          '0123456789', '')"/>
      </xf:action>
    </xf:model>
  </xh:head>
  <xh:body>
    <xf:group ref="instance('split')">
      <xf:input ref="value">
        <xf:label/>
        <xf:setvalue ev:event="xforms-value-changed"
          ref="instance('dimension-interface')"
          value="concat(instance('split')/value,
                 instance('split')/unit)"/>
      </xf:input>
      <xf:select1 ref="unit">
        <xf:label/>
        <xf:setvalue ev:event="xforms-value-changed"
          ref="instance('dimension-interface')"
          value="concat(instance('split')/value,
                 instance('split')/unit)"/>
        <xf:item>
          <xf:label>pixels</xf:label>
          <xf:value>px</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>font size</xf:label>
          <xf:value>em</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>font height</xf:label>
          <xf:value>ex</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>inches</xf:label>
          <xf:value>in</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>centimeters</xf:label>
          <xf:value>cm</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>millimeters</xf:label>
          <xf:value>mm</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>points</xf:label>
          <xf:value>pt</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>picas</xf:label>
          <xf:value>pc</xf:value>
        </xf:item>
        <xf:item>
          <xf:label>%</xf:label>
          <xf:value>%</xf:value>
        </xf:item>
      </xf:select1>
    </xf:group>
  </xh:body>
</xh:html>
```

for the subform.

Acknowledging that things could be easier, XSLTForms has introduced a new experimental feature, derived from subforms, to implement simple components:

> I have implemented a new component control in XSLTForms. It is named "`xf:component`" and has two attributes named "`@ref`" and "`@resource`". There are still restrictions within a component: ids cannot be used if the component is to be instantiated more than once. The default instance is local to each instantiated component and the `subform-instance()` function can be used to get the document element of it. From the main form to the component, a binding with a special mip named "changed" is defined. The `subform-context()` allows to reference the node bound to the component control in the main form. The corresponding build has been committed to repositories: http://sourceforge.net/p/xsltforms/code/ci/master/tree/build/
>
> --Alain Couthures on the Xsltforms-support mailing list

With this new experimental feature and another extension (the @changed MIP implemented in XSLTForms), the master form would be:

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl"
                 type="text/xsl"?>
<?xsltforms-options debug="yes"?>
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
       xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xf="http://www.w3.org/2002/xforms">
  <xh:head>
    <xh:title>Subforms</xh:title>
    <xf:model>
      <xf:instance id="main">
        <figures>
          <rectangle height="10in" width="4em"/>
        </figures>
      </xf:instance>

    </xf:model>
  </xh:head>
  <xh:body>
    <xf:group ref="rectangle">
      <!-- Height -->
      <xf:group ref="@height">
        <xf:label>Height: </xf:label>
        <xf:component ref="."
                  resource="component-subform.xml"/>
      </xf:group>
      <br/>
      <!-- Width -->
      <xf:group ref="@width">
        <xf:label>Width: </xf:label>
        <xf:component ref="."
                  resource="component-subform.xml"/>
      </xf:group>
    </xf:group>
  </xh:body>
</xh:html>
```

and the subform (or component):

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl"
                 type="text/xsl"?>
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ev="http://www.w3.org/2001/xml-events">
  <xh:head>
    <xh:title>Size</xh:title>
    <xf:model>
      <xf:instance>
        <size>
          <value xsi:type="xsd:decimal">2</value>
          <unit>cm</unit>
        </size>
      </xf:instance>
      <xf:bind ref="subform-instance()/value"
  changed="translate(subform-context(), '%incmptxe', '')"/>
      <xf:bind ref="subform-instance()/unit"
  changed="translate(subform-context(), '0123456789', '')"/>
    </xf:model>
  </xh:head>
  <xh:body>
    <xf:input ref="subform-instance()/value">
      <xf:label/>
      <xf:setvalue ev:event="xforms-value-changed"
        ref="subform-context()"
        value="concat(subform-instance()/value,
               subform-instance()/unit)"/>
    </xf:input>
    <xf:select1 ref="subform-instance()/unit">
      <xf:label/>
      <xf:item>
        <xf:label>pixels</xf:label>
        <xf:value>px</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>font size</xf:label>
        <xf:value>em</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>font height</xf:label>
        <xf:value>ex</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>inches</xf:label>
        <xf:value>in</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>centimeters</xf:label>
        <xf:value>cm</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>millimeters</xf:label>
        <xf:value>mm</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>points</xf:label>
        <xf:value>pt</xf:value>
      </xf:item>
      <xf:item>
```

```
        <xf:label>picas</xf:label>
        <xf:value>pc</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>%</xf:label>
        <xf:value>%</xf:value>
      </xf:item>
      <xf:setvalue
          ev:event="xforms-value-changed"
          ref="subform-context()"
          value="concat(subform-instance()/value,
                 subform-instance()/unit)"/>
    </xf:select1>
  </xh:body>
</xh:html>
```

The level of complexity of both the definition of the subform component and its invocation are similar to what we've seen with Orbeon's XBL feature. The main difference is the encapsulation (no encapsulation in XSLTForms and a controlled encapsulation in Orbeon Forms which handles the issue of id collisions).

Note that we are escaping the issue caused by id collision because we are accessing the instance from the master form directly from the subform using the `subform-context()` function. This feature allows us to use only one local instance in the subform and we take care of not defining any id for this instance and access it using the `subform-instance()` function. This trick wouldn't work if we needed several instances or if we had to define ids on other elements in the subform.

4. Conclusion

The lack of modularity has been one of the serious weaknesses in the XForms recommendations so far.
A common solution is to generate or "template" XForms but this can be tricky when dealing with "components" used multiple times in a form and especially within `xf:repeat` controls.
Different implementation have come up with different solutions to address this issue (XBL for Orbeon, subforms for betterFORM and XSLTForms).

The main differences between these solutions are:

- The syntax:
  - XBL + XForms for Orbeon Forms
  - XForms with minor extensions for betterFORM and XSLTForms)
- The encapsulation or isolation and features to communicate between the component and other models:
  - complete for betterFORM with extensions to communicate between models
  - either complete or partial for Orbeon Forms with extension to communicate between models
  - no isolation for XSLTForms with extensions to access to the context node and default instance from a component
- The support of id collisions between components and the main form:
  - Id collisions are handled by Orbeon Forms
  - They are forbidden by betterFORM and XSLTForms

The lack of interoperability between these implementations will probably not be addressed by the W3C XForms Working Group and it would be very useful if XForms implementers could work together to define interoperable solutions to define reusable components in XForms.

In this paper, generation (or templating) has been presented as an alternative to XML or subforms but they are by no mean exclusive. In real world projects, hybrid approaches mixing XForms generation (or templating) and components (XBL or subforms) are on the contrary very valuable. They have been demonstrated in a number of talks during the pre-conference day at XML Prague.

These hybrid approaches are easy to implement with common XML toolkits. The generation/templating can be static (using tools such as XProc, Ant or classical make files) or dynamic (using XProc or XPL pipelines or plain XQuery or XSLT) and Orbeon Forms XBL implementation even provides a feature to dynamically invoke a transformation on the content of the bound element).

4.1. Acknowledgments

I would like to thank Erik Bruchez (Orbeon), Joern Turner (betterFORM) and Alain Couthures (XSLTForms) for the time they've spent to answer my questions and review this paper.

XML on the Web: is it still relevant?

O'Neil Delpratt

Saxonica

<oneil@saxonica.com>

Abstract

In this paper we discuss what it means by the term XML on the Web and how this relates to the browser. The success of XSLT in the browser has so far been underwhelming, and we examine the reasons for this and consider whether the situation might change. We describe the capabilities of the first XSLT 2.0 processor designed to run within web browsers, bringing not just the extra capabilities of a new version of XSLT, but also a new way of thinking about how XSLT can be used to create interactive client-side applications. Using this processor we demonstrate as a use-case a technical documentation application, which permits browsing and searching in a intuitive way. We show its internals to illustrate how it works.

1. Introduction

The W3C introduced Extensible Markup Language (XML) as a multi-purpose and platform-neutral text-based format, used for storage, transmission and manipulation of data. Fifteen years later, it has matured and developers and users use it to represent their complex and hierarchically structured data in a variety of technologies. Its usage has reached much further than its creators may have anticipated.

In popular parlance 'XML on the Web' means 'XML in the browser'. There's a great deal of XML on the web, but most of it never reaches a browser: it's converted server-side to HTML using a variety of technologies ranging from XSLT and XQuery to languages such Java, C#, PHP and Perl. But since the early days, XML has been seen as a powerful complement to HTML and as a replacement in the form of XHTML. But why did this not take off and revolutionise the web? And could this yet happen?

XML has been very successful, and it's useful to remind ourselves why:

- XML can handle both data and documents.
- XML is human-readable (which makes it easy to develop applications).
- XML handles Unicode.
- XML was supported by all the major industry players and available on all platforms.
- XML was cheap to implement: lots of free software, fast learning curve.
- There was a rich selection of complementary technologies.
- The standard acquired critical mass very quickly, and once this happens, any technical deficiencies become unimportant.

However, this success has not been gained in the browser. Again, it's a good idea to look at the reasons:

- HTML already established as a defacto standard for web development
- The combination of HTML, CSS, and Javascript was becoming ever more powerful.
- It took a long while before XSLT 1.0 was available on a sufficient range of browsers.
- When XSLT 1.0 did eventually become sufficiently ubiquitous, the web had moved on ("Web 2.0").
- XML rejected the "be liberal in what you accept" culture of the browser.

One could look for more specific technical reasons, but they aren't convincing. Some programmers find the XSLT learning curve a burden, for example, but there are plenty of technologies with an equally daunting learning curve that prove successful, provided developers have the incentive and motivation to put the effort in. Or one could cite the number of people who encounter problems with ill-formed or mis-encoded XML, but that problem is hardly unique to XML. Debugging Javascript in the browser, after all, is hardly child's play.

doi:10.14337/XMLLondon13.Delpratt01

XSLT 1.0 was published in 1999 [1]. The original aim was that it should be possible to use the language to convert XML documents to HTML for rendering on the browser 'client-side'. This aim has largely been achieved. Before the specification was finished Microsoft implemented XSLT 1.0 as an add-on to Internet Explorer (IE) 4, which became an integral part of IE5. (Microsoft made a false start by implementing a draft of the W3C spec that proved significantly different from the final Recommendation, which didn't help.) It then took a long time before XSLT processors with a sufficient level of conformance and performance were available across all common browsers. In the first couple of years the problem was old browsers that didn't have XSLT support; then the problem became new browsers that didn't have XSLT support. In the heady days while Firefox market share was growing exponentially, its XSLT support was very weak. More recently, some mobile browsers have appeared on the scene with similar problems.

By the time XSLT 1.0 conformance across browsers was substiantially achieved (say around 2009), other technologies had changed the goals for browser vendors. The emergence of XSLT 2.0 [2], which made big strides over XSLT 1.0 in terms of developer productivity, never attracted any enthusiasm from the browser vendors - and the browser platforms were sufficiently closed that there appeared to be little scope for third-party implementations.

The "Web 2.0" movement was all about changing the web from supporting read-only documents to supporting interactive applications. The key component was AJAX: the X stood for "XML", but Javascript and XML never worked all that well together. DOM programming is tedious. AJAX suffers from "Impedence mismatch" - it's a bad idea to use programming languages whose type system doesn't match your data.

That led to what we might call AJAJ - Javascript programs processing JSON data. Which is fine if your data fits the JSON model. But not all data does, especially documents. JSON has made enormous headway in making it easier for Javascript programmers to handle structured data, simply because the data doesn't need to be converted from one data model to another. But for many of the things where XML has had most success - for example, authoring scientific papers like this one, or capturing narrative and semi-structured information about people, places, projects, plants, or poisons - JSON is simply a non-starter.

So the alternative is AXAX - instead of replacing XML with JSON, replace Javascript with XSLT or XQuery. The acronym that has caught on is XRX, but AXAX better captures the relationship with its alternatives. The key principle of XRX is to use the XML data model and XML-based processing languages end-to-end, and the key benefit is the same as the "AJAJ" or Javascript-JSON model - the data never needs to be converted from one data model to another. The difference is that this time, we are dealing with a data model that can handle narrative text.

A few years ago it seemed likely that XML would go no further in the browser. The browser vendors had no interest in developing it further, and the browser platform was so tightly closed that it wasn't feasible for a third party to tackle. Plug-ins and applets as extension technologies were largely discredited. But paradoxically, the browser vendors' investment in Javascript provided the platform that could change this. Javascript was never designed as a system programming language, or as a target language for compilers to translate into, but that is what it has become, and it does the job surprisingly well. Above all else, it is astoundingly fast.

Google were one of the first to realise this, and responded by developing Google Web Toolkit (GWT) [3] as a Java-to-Javascript bridge technology. GWT allows web applications to be developed in Java (a language which in many ways is much better suited for the task than Javascript) and then cross-compiled to Javascript for execution on the browser. It provides most of the APIs familiar to Java programmers in other environments, and supplements these with APIs offering access to the more specific services available in the browser world, for example access to the HTML DOM, the Window object, and user interface events.

Because the Saxon XSLT 2.0 processor is written in Java, this gave us the opportunity to create a browser-based XSLT 2.0 processor by cutting down Saxon to its essentials and cross-compiling using GWT.

We realized early on that simply offering XSLT 2.0 was not enough. Sure, there was a core of people using XSLT 1.0 who would benefit from the extra capability and productivity of the 2.0 version of the language. But it was never going to succeed using the old architectural model: generate an HTML page, display it, and walk away, leaving all the interesting interactive parts of the application to be written in Javascript. XRX (or AXAX, if you prefer) requires XML technologies to be used throughout, and that means replacing Javascript not only for content rendition (much of which can be done with CSS anyway), but more importantly for user interaction. And it just so happens that the processing model for handling user interaction is event-based programming, and XSLT is an event-based programming language, so the opportunities are obvious.

In this paper we examine the first implementation of XSLT 2.0 on the browser, Saxon-CE [4]. We show how Saxon-CE can be used as a complement to Javascript, given its advancements in performance and ease of use. We also show that Saxon-CE can be used as a replacement of JavaScript. This we show with an example of a browsing and searching technical documentation.

This is classic XSLT territory, and the requirement is traditionally met by server-side HTML generation, either in advance at publishing time, or on demand through servlets or equivalent server-side processing that invoke XSLT transformations, perhaps with some caching. While this is good enough for many purposes, it falls short of what users had already learned to expect from desktop help systems, most obviously in the absence of a well-integrated search capability. Even this kind of application can benefit from Web 2.0 thinking, and we will show how the user experience can be improved by moving the XSLT processing to the client side and taking advantage of some of the new facilities to handle user interaction.

In our conference paper and talk we will explain the principles outlined above, and then illustrate how these principles have been achieved in practice by reference to a live application: we will demonstrate the application and show its internals to illustrate how it works.

2. XSLT 2.0 on the browser

In this section we begin with some discussion on the usability of Saxon-CE before we give an overview of its internals. Saxon-CE has matured significantly since its first production release (1.0) in June 2012, following on from two earlier public beta releases. The current release (1.1) is dated February 2013, and the main change is that the product is now released under an open source license (Mozilla Public License 2.0).

2.1. Saxon-CE Key Features

Beyond being a conformant and fast implementation of XSLT 2.0, Saxon-CE has a number of features specially designed for the browser, which we now discuss:

1. *Handling JavaScript Events in XSLT*: Saxon-CE is not simply an XSLT 2.0 processor running in the browser, doing the kind of things that an XSLT 1.0 processor did, but with more language features (though that in itself is a great step forward). It also takes XSLT into the world of interactive programming. With Saxon-CE it's not just a question of translating XML into HTML-plus-JavaScript and then doing all the interesting user interaction in the JavaScript; instead, user input and interaction is handled directly within the XSLT code. The XSLT code snippet illustrates the use of event handling:

```
<xsl:template match="p[@class eq 'arrowNone']"
              mode="ixsl:onclick">
  <xsl:if test="$usesclick">
    <xsl:for-each select="$navlist/ul/li">
      <ixsl:set-attribute name="class"
                          select="'closed'"/>
    </xsl:for-each>
  </xsl:if>
</xsl:template>
```

XSLT is ideally suited for handling events. It's a language whose basic approach is to define rules that respond to events by constructing XML or HTML content. It's a natural extension of the language to make template rules respond to input events rather than only to parsing events. The functional and declarative nature of the language makes it ideally suited to this role, eliminating many of the bugs that plague JavaScript development.

2. *Working with JavaScript Functions*: The code snippets below illustrates a JavaScript function, which gets data from an external feed:

```
var getTwitterTimeline = function(userName)
{
  try {
    return makeRequest(timelineUri + userName);
  }
  catch(e) {

    console.log(
      "Error in getTwitterTimeline: " + e );

    return "";
  }
};
```

Here is some XSLT code showing how the JavaScript function can be used; this is a call to the `getTwitterTimeline` function in XSLT 2.0 using Saxon-CE. The XML document returned is then passed as a parameter to the a JavaScript API function `ixsl:parse-xml`:

```
<xsl:variable name="tw-response"
  as="document-node()"
  select="ixsl:parse-xml(
            js:getTwitterTimeline($username)
          )" />
```

3. *Animation*: The extension instruction ixsl:schedule-action may be used to achieve animation. The body of the instruction must be a single call on `<xsl:call-template/>`, which is done asynchronously. If an action is to take place repeatedly, then each action should trigger the next by making another call on `<ixsl:schedule-action />`
4. *Interactive XSLT*: There are a number of Saxon-CE defined functions and instructions which are available. One indispensable useful function is the `ixsl:page()`, which returns the document node of the HTML DOM document. An example of this function's usage is given as follows. Here we retrieve a `div` element with a given predicate and bind it to an XSLT variable:

```
<xsl:variable name="movePiece" as="element(div)"
  select="
    if (exists($piece)) then $piece
    else
      id('board',ixsl:page())/div[$moveFrom]/div
  "/>
```

In the example below, we show how to set the style property using the extension intruction `ixsl:set-attribute` for a current node in the HTML page. Here we are changing the display property to 'none', which hides an element, causing it not to take up any space on the page:

```
<xsl:if test="position() &gt; $row-size">
  <ixsl:set-attribute name="style:display"
                      select="'none'"/>
</xsl:if>
```

In the example below we show how we can get the property of a JavaScript object by using the ixsl:get function:

```
<xsl:variable name="piece" as="element(div)"
   select="ixsl:get(ixsl:window(),
                    'dragController.piece')"/>
```

The full list of the extension functions and extension instructions in Saxon-CE can be found at the following location: *http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extensions* and *http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extension-instructions*

2.2. Saxon-CE Internals

In this section we discuss how we build the client-side XSLT 2.0 processor and how we can invoke it from JavaScript, XML or HTML. The Java code base was inherited from Saxon-HE, the successful XSLT 2.0 processor for Java. The product was produced by cross-compiling the Java into optimized, stand-alone JavaScript files using the GWT 5.2. Although no detailed performance data is available here, all deliver a responsiveness which feels perfectly adequate for production use. The JavaScript runs on all major browsers, as well as on mobile browsers, where JavaScript can run.

The key achievements in the development of Saxon-CE are given below:

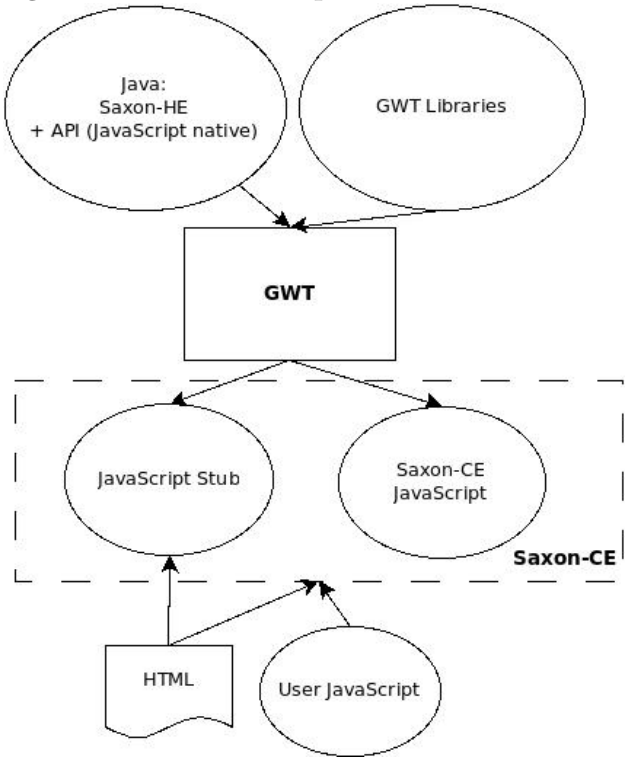
- The size of the Java source was cut down to around 76K lines of Java code. This was mainly achieved by cutting out unwanted functionality such as XQuery, updates, serialization, support for JAXP, support for external object models such as JDOM and DOM4J, Java extension functions, and unnecessary options like the choice between TinyTree and Linked Tree, or the choice (never in practice exercised) of different sorting algorithms. Some internal hanges to the code base were also made to reduce size. Examples include changes to the XPath parser to use a hybrid precedence-parsing approach in place of the pure recursive-descent parser used previously; offloading the data tables used by the `normalize-unicode()` function into an XML data file to be loaded from the

server on the rare occasions that this function is actually used.

- GWT creates a slightly different JavaScript file for each major browser, to accommodate browser variations. Only one of these files is downloaded, which is based on the browser that is in use. The size of the JavaScript file is around 900KB.
- The key benefits of the server-side XSLT 2.0 processor were retained and delivered on the browser. Saxon has a reputation for conformance, usability, and performance, and it was important to retain this, as well as delivering the much-needed functionality offered by the language specification. Creating automated test suites suitable for running in the browser environment was a significant challenge.
- Support of JavaScript events. The handling of JavaScript events changes the scope of Saxon-CE greatly, meaning it can be used for interactive application development. Our first attempts to integrate event handling proved the design of the language extensions was sound, but did not perform adequately, and the event handling is the final product was a complete rewrite. The Events arising from the HTML DOM and the client system, which are understood by GWT, are handled via Saxon-CE. This proxying of event handling in the Java code makes it possible for template rules which have a mode matching the event to overide the default behavour of the browser. Events are only collected at the document node (thus there's only one listener for each type of event). As a result, the events are bubbled up from the event target. This mechanism handles the majority of browser events. There are a few specialist events like `onfocus` and `onblur` which do not operate at the document node, and these events are best handled in JavaScript first. GWT provides relatively poor support for these events because their behaviour is not consistent across different browsers.
- Interoperability with JavaScript. Many simple applications can be developed with no user-written Javascript. Equally, where Javascript skills or components are available, it is possible to make use of them, and when external services are avalable only via Javascript interfaces, Saxon-CE stylesheets can still make use of them.

Figure 1 illustrates the input and output components involved in building the XSLT 2.0 processor, Saxon-CE:

Figure 1. Saxon-CE Development



Static view of the Saxon-CE product and components involved in the build process

As shown in Figure 1 we use GWT to cross-compile the XSLT 2.0 processor. Saxon-HE and GWT libraries are input to this process. In addition, we write the JavaScript API methods in Java using the JavaScript Native Interface (JSNI), which is a GWT feature. This feature proved useful because it provided access to the low-level browser functionality not exposed by the standard GWT APIs. This in effect provides the interface for passing and returning JavaScript objects to and from the XSLT processor.

The output from this process is Saxon-CE, which comprises of the XSLT 2.0 processor and the stub file, both in highly compressed and obfuscated JavaScript. GWT provides separate JavaScript files for each major browser. User JavaScript code can happily run alongside the XSLT processor.

The invoking of Saxon-CE is achieved in several ways. The first method employs a standard `<?xml-stylesheet?>` processing-instruction in the prolog of an XML document. This cannot be used to invoke Saxon-CE directly, because the browser knows nothing of Saxon-CE's existence. Instead, however, it can be used to load an XSLT 1.0 bootstrap stylesheet, which in turn causes Saxon-CE to be loaded. This provides the easiest upgrade from existing XSLT 1.0 applications. The code

snippet below illustrates the bootstrap process of the XSLT 2.0 processor:

```
<?xml-stylesheet type="text/xsl"
                 href="sample.boot.xsl"?>
<dt:data-set xmlns:dt="urn:system.logging.data.xml">
  <dt:rows name="test-data">
  ...
</dt:data-set>
```

The XSLT 1.0 bootstrap stylesheet is given below. It generates an HTML page containing instructions to load Saxon-CE and execute the real XSLT 2.0 stylesheet:

```
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html" indent="no"/>
  <xsl:template match="/">
    <html>
      <head>
        <meta http-equiv="Content-Type"
              content="text/html" />
        <script type="text/javascript"
          language="javascript"
          src="../Saxonce/Saxonce.nocache.js"/>

        <script>
          var onSaxonLoad = function() {
            Saxon.run( {
              source:     location.href,
              logLevel:   "SEVERE",
              stylesheet: "sample.xsl"
            });
          }
        </script>

      </head>
      <!-- these elements are required also -->
    <body><p></p></body>
  </html>
</xsl:template>

</xsl:transform>
```

The second method involves use of the script element in HTML. In fact there are two script elements: one with type="text/javascript" which causes the Saxon-CE engine to be loaded, and the other with type="application/xslt +xml" which loads the stylesheet itself, as shown here:

```
<script type="application/xslt+xml"
        language="xslt2.0" src="books.xsl"
        data-source="books.xml"></script>
```

The third method is to use an API from Javascript. The API is modelled on the XSLTProcessor API provided by the major browsers for XSLT 1.0.

We discussed earlier that the JavaScript API provides an API with a rich set of features for interfacing and invoking the XSLT processor when developing Saxon-CE applications. There are three JavaScript API Sections available: The *Command*, which is designed to be used as a JavaScript literal object and effectively wraps the Saxon-CE API with a set of properties so you can run an XSLT transform on an HTML page in a more declarative way; the *Saxon* object, which is a static object, providing a set of utility functions for working with the XSLT processor, initiating a simple XSLT function, and working with XML resources and configuration; and the *XSLT20Processor*, which is modeled on the JavaScript XSLTProcessor API as implemented by the major browsers. It provides a set of methods used to initiate XSLT transforms on XML or direct XSLT-based HTML updates.

The code snippet below shows a Command API call to run a XSLT transform. Here the stylesheet is declared as `ChessGame.xsl` and the initial template is defined as `main`. We observed the `logLevel` as been set to 'SEVERE'. Saxon-CE provides a debug version which ouputs useful log messages to the JavaScript console, accessable in the browser development tools:

```
var onSaxonLoad = function() {

  proc = Saxon.run( {
    stylesheet:      'ChessGame.xsl',
    initialTemplate: 'main',
    logLevel:        'SEVERE'
  } );

};
```

3. Use Case: Technical documentation application

We now examine a Saxon-CE driven application used for browsing technical documentation in a intuative manner: specifically, it is used for display of the Saxon 9.5 documentation on the Saxonica web site. The application is designed to operate as a desktop application, but on the web.

The documentation for Saxon 9.5 can be found at:

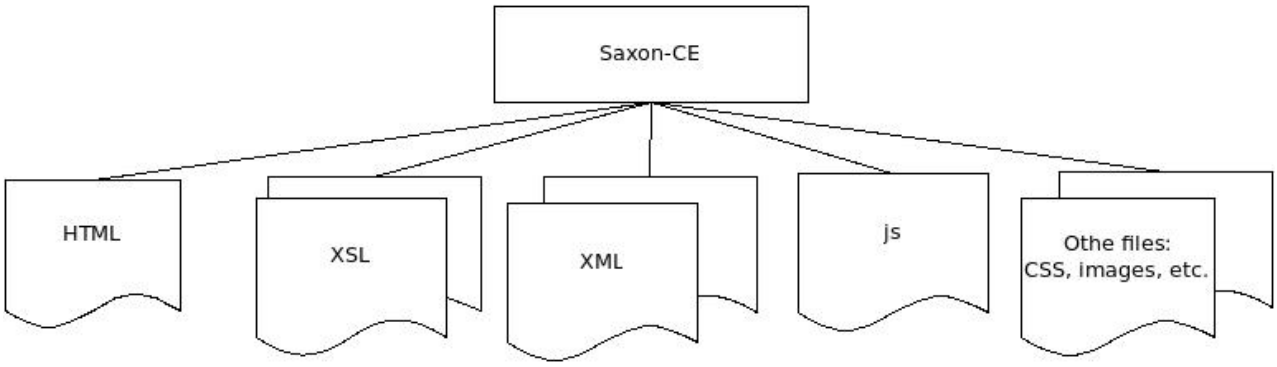
- *http://www.saxonica.com/documentation/index.html*

When you click on this link for the first time, there will be a delay of a few seconds, with a comfort message telling you that Saxon is loading the documentation. This is not strictly accurate; what is actually happening is that Saxon-CE itself is being downloaded from the web site. This only happens once; thereafter it will be picked up from the browser cache. However, it is remarkable how fast this happens even the first time, considering that the browser is downloading the entire Saxon-CE product (900Kb of Javascript source code generated from around 76K lines of Java), compiling this, and then executing it before it can even start compiling and executing the XSLT code.

3.1. Architecture

The architecture of the technical documentation application is shown in Figure 2:

Figure 2. Architecture of Technical Documentation application



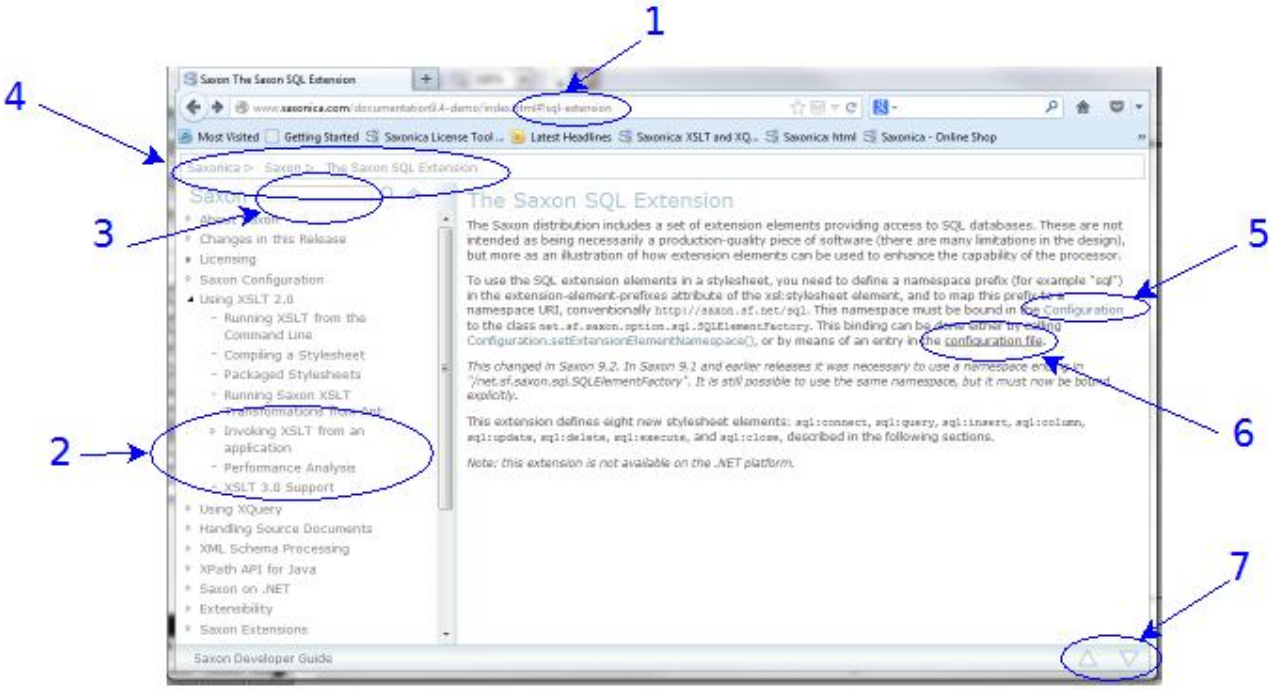
Architectural view of a Saxon-CE application

The application consists of a number of XML documents representing the content data, ten XSLT 2.0 modules, a Javascript file, several other files (CSS file, icon and image files) and a single skeleton HTML webpage; the invariant parts of the display are recorded directly in HTML markup, and the variable parts are marked by empty elements whose content is controlled from the XSLT stylesheets. Development with Saxon-CE often eliminates the need for Javascript, but at the same time it happily can be mixed with calls from XSLT. In this case it was useful to abstract certain JavaScript functions used by the Saxon-CE XSLT transforms.

Key to this application is that the documentation content data are all stored as XML files. Even the linkage of files to the application is achieved by a XML file called catalog.xml: this is a special file used by the XSLT to render the table of contents. The separation of the content data from the user interface means that changes to the design can be done seamlessly without modifiying the content, and vice versa.

The documentation is presented in the form of a single-page web site. The screenshot in Figure 3 shows its appearance.

Figure 3. Technical documentation application in the browser



Screen-shot of the Technical documentation in the browser using Saxon-CE

Note the following features, called out on the diagram. We will discuss below how these are implemented in Saxon-CE.

1. The fragment identifier in the URL
2. Table of contents
3. Search box
4. Breadcrumbs
5. Links to Javadoc definitions
6. Links to other pages in the documentation
7. The up/down buttons

3.2. XML on the Server

This application has no server-side logic; everything on the server is static content.

On the server, the content is held as a set of XML files. Because the content is fairly substantial (2Mb of XML, excluding the Javadoc, which is discussed later), it's not held as a single XML document, but as a set of a 20 or so documents, one per chapter. On initial loading, we load only the first chapter, plus a small catalogue document listing the other chapters; subsequent chapters are fetched on demand, when first referenced, or when the user does a search.

Our first idea was to hold the XML in DocBook form, and use a customization of the DocBook stylesheets to present the content in Saxon-CE. This proved infeasible: the DocBook stylesheets are so large that downloading them and compiling them gave unacceptable performance. In fact, when we looked at the XML vocabulary we were actually using for the documentation, it needed only a tiny subset of what DocBook offered. We thought of defining a DocBook subset, but then we realised that all the elements we were using could be easily represented in HTML5 without any serious tag abuse (the content that appears in highlighted boxes, for example, is tagged as an `<aside>`). So the format we are using for the XML is in fact XHTML 5. This has a couple of immediate benefits: it means we can use the HTML DOM in the browser to hold the information (rather than the XML DOM), and it means that every element in our source content has a default rendition in the browser, which in many cases (with a little help from CSS) is quite adequate for our purposes.

Although XHTML 5 is used for the narrative part of the documentation, more specialized formats are used for the parts that have more structure. In particular, there is an XML document containing a catalog of XPath functions (both standard W3C functions, and Saxon-specific extension functions) which is held in a custom XML vocabulary; and the documentation also includes full Javadoc API specifications for the Saxon code base. This was produced from the Java source code using the standard Javadoc utility along with a custom "doclet" (user hook) causing it to generate XML rather than HTML. The Javadoc in XML format is then rendered by the client-side stylesheets in a similar way to the rest of the documentation, allowing functionality such as searching to be fully integrated. For the .NET API, we wrote our own equivalent to Javadoc to produce class and method specifications in the same XML format.

The fact that XHTML is used as the delivered documentation format does not mean, of course, that the client-side stylesheet has no work to do. This will become clear when we look at the implementation of the various features of the user interaction. A typical XML file fragment is shown below:

```
<article id="changes"
  title="Changes in this Release">
  <h1>Version 9.4 (2011-12-09)</h1>

  <p>Details of changes in Saxon 9.4 are detailed
  on the following pages:</p>

  <nav>
    <ul/>
  </nav>

  <section id="bytecode-94"
           title="Bytecode generation">
    <h1>Bytecode generation</h1>

    <p>Saxon-EE 9.4 selectively compiles
       stylesheets and queries into Java bytecode
       before execution.</p>
...
```

For the most part, the content of the site is authored directly in the form in which it is held on the site, using an XML editor. The work carried out at publishing time consists largely of validation. There are a couple of exceptions to this: the Javadoc content is generated by a tool from the Java source code, and we also generate an HTML copy of the site as a fallback for use from devices that are not Javascript-enabled. There appears to be little call for this, however: the client-side Saxon-CE version of the site appears to give acceptable results to the vast majority of users, over a wide range of devices. Authoring the site in its final delivered format greatly simplifies the process of making quick corrections when errors are found, something we have generally not attempted to do in the past, when republishing the site was a major undertaking.

3.3. The User Interface

In this section we discuss the user interface of the documentation application. The rendition of the webpages is done dynamically, almost entirely in XSLT 2.0. There are a few instances were we rely on helper functions (amounting to about 50 lines) of JavaScript. The XSLT is in 8 modules totalling around 2500 lines of code. The Javascript code is mainly concerned with scrolling a page to a selected position, which in turn is used mainly in support of the search function, discussed in more detail below.

3.3.1. The URI and Fragment Identifier

URIs follow the "hashbang" convention: a page might appear in the browser as:

- *http://www.saxonica.com/documentation/index.html#!configuration*

For some background on the hashbang convention, and an analysis of its benefits and drawbacks, see Jeni Tennison's article at [5]. From our point of view, the main characteristics are:

- Navigation within the site (that is, between pages of the Saxon documentation) doesn't require going back to the server on every click.
- Each sub-page of the site has a distinct URI that can be used externally; for example it can be bookmarked, it can be copied from the browser address bar into an email message, and so on. When a URI containing such a fragment identifier is loaded into the browser address bar, the containing HTML page is loaded, Saxon-CE is activated, and the stylesheet logic then ensures that the requested sub-page is displayed.
- It becomes possible to search within the site, without installing specialized software on the server.

- The hashbang convention is understood by search engines, allowing the content of a sub-page to be indexed and reflected in search results as if it were an ordinary static HTML page.

The XSLT stylesheet supports use of hashbang URIs in two main ways: when a URI is entered in the address bar, the stylesheet navigates to the selected sub-page; and when a sub-page is selected in any other way (for example by following a link or performing a search), the relevant hashbang URI is constructed and displayed in the address bar.

The fragment identifiers used for the Saxon documentation are hierarchic; an example is

- *#!schema-processing/validation-api/schema-jaxp*

The first component is the name of the chapter, and corresponds to the name of one of the XML files on the server, in this case `schema-processing.xml`. The subsequent components are the values of `id` attributes of nested XHTML 5 `<section>` elements within that XML file. Parsing the URI and finding the relevant subsection is therefore a simple task for the stylesheet.

3.3.2. The Table of Contents

The table of contents shown in the left-hand column of the browser screen is constructed automatically, and the currently displayed section is automatically expanded and contracted to show its subsections. Clicking on an entry in the table of contents causes the relevant content to appear in the right-hand section of the displayed page, and also causes the subsections of that section (if any) to appear in the table of contents. Further side-effects are that the URI displayed in the address bar changes, and the list of breadcrumbs is updated.

Some of this logic can be seen in the following template rule:

```
<xsl:template match="*" mode="handle-itemclick">
  <xsl:variable name="ids"
    select="(., ancestor::li)/@id"
    as="xs:string*"/>
  <xsl:variable name="new-hash"
    select="string-join($ids, '/')"/>
  <xsl:variable name="isSpan"
    select="@class eq 'item'"
    as="xs:boolean"/>
  <xsl:for-each select="if ($isSpan) then ..
                        else .">
    <xsl:choose>
      <xsl:when test="@class eq 'open'
                      and not($isSpan)">
        <ixsl:set-attribute name="class"
                            select="'closed'"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="js:disableScroll()"/>
        <xsl:choose>
          <xsl:when test="f:get-hash() eq
                          $new-hash">
            <xsl:variable name="new-class"
              select="f:get-open-class(@class)"/>
            <ixsl:set-attribute name="class"
              select="$new-class"/>
            <xsl:if test="empty(ul)">
              <xsl:call-template
                name="process-hashchange"/>
            </xsl:if>
          </xsl:when>
          <xsl:otherwise>
            <xsl:sequence
              select="f:set-hash($new-hash)"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
```

Most of this code is standard XSLT 2.0. A feature particular to Saxon-CE is the `ixsl:set-attribute` instruction, which modifies the value of an attribute in the HTML DOM. To preserve the functional nature of the XSLT language, this works in the same way as the XQuery Update Facility: changes are written to a pending update list, and updates on this list are applied to the HTML DOM at the end of a transformation phase. Each transformation phase therefore remains, to a degree, side-effect free. Like the `xsl:result-document` instruction, however, `ixsl:set-attribute` delivers no result and is executed only for its external effects; it therefore needs some special attention by the optimizer. In this example, which is not untypical, the instruction is used to change the `class`attribute of an element in the HTML DOM, which has the effect of changing its appearance on the screen.
The code invokes a function `f:set-hash` which looks like this:

```
<xsl:function name="f:set-hash">
  <xsl:param name="hash"/>
  <ixsl:set-property name="location.hash"
                     select="concat('!',$hash)"/>
</xsl:function>
```

This has the side-effect of changing the contents of the `location.hash` property of the browser window, that is, the fragment identifier of the displayed URI. Changing this property also causes the browser to automatically update the browsing history, which means that the back and forward buttons in the browser do the right thing without any special effort by the application.

3.3.3. The Search Box

The search box provides a simple facility to search the entire documentation for keywords. Linguistically it is crude (there is no intelligent stemming or searching for synonyms or related terms), but nevertheless it can be highly effective. Again this is implemented entirely in client-side XSLT.

The initial event handling for a search request is performed by the following XSLT template rules:

```
<xsl:template match="p[@class eq 'search']"
              mode="ixsl:onclick">
  <xsl:if test="$usesclick">
    <xsl:call-template name="run-search"/>
  </xsl:if>
</xsl:template>
<xsl:template match="p[@class eq 'search']"
              mode="ixsl:ontouchend">
  <xsl:call-template name="run-search"/>
</xsl:template>

<xsl:template name="run-search">
  <xsl:variable name="text"
  select="normalize-space(
            ixsl:get($navlist/div/input,
            'value')
          )"/>
  <xsl:if test="string-length($text) gt 0">
    <xsl:for-each
      select="$navlist/../div[@class eq 'found']">
      <ixsl:set-attribute name="style:display"
                          select="'block'"/>
    </xsl:for-each>
    <xsl:result-document href="#findstatus"
                         method="replace-content">
      searching...
    </xsl:result-document>
    <ixsl:schedule-action wait="16">
      <xsl:call-template name="check-text"/>
    </ixsl:schedule-action>
  </xsl:if>
</xsl:template>
```

The existence of two template rules, one responding to an `onclick` event, and one to `ontouchend`, is due to differences between browsers and devices; the Javascript event model, which Saxon-CE inherits, does not always abstract away all the details, and this is becoming particularly true as the variety of mobile devices increases.
The use of `ixsl:schedule-action` here is not so much to force a delay, as to cause the search to proceed asynchronously. This ensures that the browser remains responsive to user input while the search is in progress.The template `check-text`, which is called from this code, performs various actions, one of which is to initiate the actual search. This is done by means of a

recursive template, shown below, which returns a list of paths to locations containing the search term:

```
<xsl:template match="section|article"
              mode="check-text">
  <xsl:param name="search"/>
  <xsl:param name="path" as="xs:string"
             select="''"/>
  <xsl:variable name="newpath"
                select="concat($path, '/', @id)"/>
  <xsl:variable name="text" select="lower-case(
    string-join(*[
      not(local-name() = ('section','article'))
    ],'!'))"/>
  <xsl:sequence
    select="if (contains($text, $search)) then
              substring($newpath,2)
            else ()"/>
  <xsl:apply-templates mode="check-text"
                       select="section|article">
    <xsl:with-param name="search"
                    select="$search"/>
    <xsl:with-param name="path"
                    select="$newpath"/>
  </xsl:apply-templates>
</xsl:template>
```

This list of paths is then used in various ways: the sections containing selected terms are highlighted in the table of contents, and a browsable list of hits is available, allowing the user to scroll through all the hits. Within the page text, search terms are highlighted, and the page scrolls automatically to a position where the hits are visible (this part of the logic is performed with the aid of small Javascript functions).

3.3.4. Breadcrumbs

In a horizontal bar above the table of contents and the current page display, the application displays a list of "breadcrumbs", representing the titles of the chapters/sections in the hierarchy of the current page. (The name derives from the story told by Jerome K. Jerome of how the *Three Men in a Boat* laid a trail of breadcrumbs to avoid getting lost in the Hampton Court maze; the idea is to help the user know how to get back to a known place.)

Maintaining this list is a very simple task for the stylesheet; whenever a new page is displayed, the list can be reconstructed by searching the ancestor sections and displaying their titles. Each entry in the breadcrumb list is a clickable link, which although it is displayed differently from other links, is processed in exactly the same way when a click event occurs.

3.3.5. Javadoc Definitions

As mentioned earlier, the Javadoc content is handled a little differently from the rest of the site.

This section actually accounts for the largest part of the content: some 11Mb, compared with under 2Mb for the narrative text. It is organized on the server as one XML document per Java package; within the package the XML vocabulary reflects the contents of a package in terms of classes, which contains constructors and methods, which in turn contain multiple arguments. The XML vocabulary reflects this logical structure rather than being pre-rendered into HTML. The conversion to HTML is all handled by one of the Saxon-CE stylesheet modules.

Links to Java classes from the narrative part of the documentation are marked up with a special class attribute, for example `<a class="javalink" href="net.sf.saxon.Configuration">Configuration</a>`. A special template rule detects the `onclick` event for such links, and constructs the appropriate hashbang fragment identifier from its knowledge of the content hierarchy; the display of the content then largely uses the same logic as the display of any other page.

3.3.6. Links between Sub-Pages in the Documentation

Within the XML content representing narrative text, links are represented using conventional relative URIs in the form `<a class="bodylink" href="../../extensions11/saxon.message">saxon:message</a>`. This "relative URI" applies, of course, to the hierarchic identifiers used in the hashbang fragment identifier used to identify the subpages within the site, and the click events for these links are therefore handled by the Saxon-CE application.

The Saxon-CE stylesheet contains a built-in link checker. There is a variant of the HTML page used to gain access to the site for use by site administrators; this displays a button which activates a check that all internal links have a defined target. The check runs in about 30 seconds, and displays a list of all dangling references.

3.3.7. The Up/Down buttons

These two buttons allow sequential reading of the narrative text: clicking the down arrow navigates to the next page in sequence, regardless of the hierarchic structure, while the up button navigates to the previous page.

Ignoring complications caused when navigating in the sections of the site that handle functions and Javadoc specifications, the logic for these buttons is:

```
<xsl:template name="navpage">
  <xsl:param name="class" as="xs:string"/>
  <xsl:variable name="ids"
    select="tokenize(f:get-hash(),'/')"/>
  <xsl:variable name="c" as="node()"
    select="f:get-item(
              $ids, f:get-first-item($ids[1]), 1
            )"/>
  <xsl:variable name="new-li"
    select="
      if ($class eq 'arrowUp') then
        ($c/preceding::li[1] union
         $c/parent::ul/parent::li)[last()]
      else ( $c/ul/li union $c/following::li)[1]"/>
  <xsl:variable name="push"
                select="
    string-join(($new-li/ancestor::li union
                 $new-li)/@id,'/')"/>
  <xsl:sequence select="f:set-hash($push)"/>
</xsl:template>
```

Here, the first step is to tokenize the path contained in the fragment identifier of the current URL (variable `$ids`). Then the variable `$c` is computed, as the relevant entry in the table of contents, which is structured as a nested hierarchy of `ul` and `li` elements. The variable `$new-li` is set to the previous or following `li` element in the table of contents, depending on which button was pressed, and `$push` is set to a path containing the identifier of this item concatenated with the identifiers of its ancestors. Finally `f:set-hash()` is called to reset the browser URL to select the subpage with this fragment identifier.

4. Conclusion

In this paper we have shown how Saxon-CE was constructed, with the help of Google's GWT technology, as a cross-browser implementation of XSLT 2.0, performing not just XML-to-HTML rendition, but also supporting the development of richly interactive client-side applications. Looking at the example application shown, there are clear benefits to writing this in XSLT rather than Javascript.

Can this transform the fortunes of XML on the Web? It's hard to say. We are in an industry that is surprisingly influenced by fashion, and that is nowhere more true than among the community of so-called "web developers". The culture of this community is in many ways more akin to the culture of television and film production than the culture of software engineering, and the delivered effect is more important than the technology used to achieve it. The costs related to content creation may in some cases swamp the software development costs, and many of the developers may regard themselves as artists rather than engineers. This is not therefore fertile territory for XML and XSLT with their engineering focus.

Nevertheless, there is a vast amount of XML in existence and more being created all the time, and there are projects putting a great deal of effort into rendering that XML for display in browsers. In many cases, the developers on those projects are already enthusiastic about XSLT (and sometimes very negative about learning to write in Javascript). It is perhaps to this community that we should look for leadership. They won't convice everyone, but a few conspicuous successes will go a long way. And perhaps this will also remind people that there is a vast amount of XML on the web; it's just that most of it never finds its way off the web and into the browser.

5. Acknowledgement

Many thanks to Michael Kay who initiated the Saxon-CE project and reviewed this paper. Also thanks to Philip Fearon for his contribution in the development of the Saxon-CE project, most notably the design of the Javascript event handling and the automated test harness. He also wrote most of the Saxon-CE documentation viewer described in this paper.

References

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

[2] XSL Transformations (XSLT) Version 2.0. W3C Recommendation. 23 January 2007. Michael Kay. W3C.
http://www.w3.org/TR/xslt20

[3] Google Web Toolkit (GWT). Google.
http://code.google.com/webtoolkit/

[4] The Saxon XSLT and XQuery Processor. Michael Kay. Saxonica.
http://www.saxonica.com/

[5] Hash URIs. Jeni Tennison.
http://www.jenitennison.com/blog/node/154

Practice what we Preach

Tomos Hillman

Oxford University Press

<tomos.hillman@oup.com>

Richard Pineger

Tech Doc Direct Limited

<richard.pineger@techdocdirect.com>

Abstract

This case study describes how we use in-house XML skills and the DITA XML semantic architecture to present high-volumes of many-layered technical instructions to the typesetters who capture our XML for publishing.

1. Capturing variety in XML

OUP content types include academic monographs, textbooks, encyclopediae, journals, reported law cases and legislation. They also include bilingual and monolingual dictionaries, but we're not covering those today.

We capture each of these content types to a suite of our own modularised DTD data models. The content architecture used for the DTDs allows for abstraction and re-use of content and processes that use XML for publishing.

Content is captured to XML from manuscripts and (retrospectively) from our back catalogue by typesetters, and these typesetters need instructions.

OUPs advantage in the publishing world is the quality of our academic authors; however, this means that we must accept what is sometimes bespoke content with a disorganised structure with, for instance, special features like custom marginalia and themed boxes.

The original instructions, called Text Capture Instructions (TCIs) were written as a suite of 20-30 Word documents, some specific to products and titles, others specific to core DTD structures that are referenced from the product specific documents. In short, complex, confusing and difficult to maintain and navigate.

2. Challenges caused by the instructions

To reiterate, we had multiple documents with a confusing document hierarchy and precedence of instructions which typesetters needed to relate to a separately maintained hierarchy of DTDs before they could press the first key on a manuscript XML file.

Our maintenance of the instructions was complex with multiple Data Engineers accessing several storage locations in MS SharePoint to retrieve,version, and track the documents. So the web of documents lacked consistency, contained conflicting instructions, and where instructions were replicated in more than one place, caused partial updates to instructions.

It's fair to say that the audience, the typesetters, were confused. The documents didn't always even target them as the audience. Many instructions carried descriptive passages that were only useful for Data Engineers and DTD maintenance, leaving the capturer scratching their head.

Also the language used had started out as passive and uncertain, "sometimes, normally, generally, should, would, might" were used in virtually every sentence. Ironically, this use of language was at least reasonably consistent as generations of Data Engineers fitted with each other's style. But it didn't help the typesetter to know that a structure "should be" captured and sometimes left them thinking, "who by?"

Worse still, for Data Engineers used to precision and certainty, there really was no effective way of automating QA of updates to the instructions.

A team of XML engineers using MS Word? It never made sense... it was time to fix this so we hired an XML technical author... Richard

doi:10.14337/XMLLondon13.Hillman01

3. The DITA XML solution

Technical writing changed in the last 30 years. The web and related technologies like online help means that we no longer want to lock our content into long linear documents as promoted by MS Word or even, dare I say it, DocBook XML. We want the freedom to collaborate and have team members each create small chunks of information like you see in help systems and on websites. Sure, many customers still expect a nice manual with their product or instrument, but we can build those out of chunks. The new kid in town is called "component content management" and when you combine it with "single-sourcing" - the ability to publish to multiple channels, you have solutions that are infinitely more flexible and open to automation. Practically speaking, this means keeping a database of XML topics, mixing and matching them into books, websites, help systems and publishing them through XSL.

Conveniently, and separately, "single-sourcing" is also the track that the publishing industry has been pursuing over the same time frame.

In the technical communications world, one solution came out of IBM 10 or 12 years ago in the form of a set of DTDs called the Darwinian Information Typing Architecture (DITA) [1].

IBM, at one time the largest publisher in the world, had made great progress with the study of what constituted quality technical information, and the capture of semantic information with the content. What they needed was a new scheme for capturing this modular, semantic content, tying it together and publishing to multiple channels. They also needed to do this in an extensible way that meant they could capture new semantic content in a dynamic way, that is, capture the semantic immediately and configure the global teams and publishing engines later. Capture the right content and the rest can follow. An approach that saved them millions - I'm sure Tom can verify the effort required to change a single attribute on a traditional publishing DTD. And that effort obviously translates to cost; and for global organizations, that can represent great deal of cost.

But DITA is more than just a single-sourcing solution, it is a methodology, a paradigm, an ethos, that promotes quality in technical communications. It focusses on the audience and their goals. It promotes minimalism and clear thinking (and therefore clear writing) by removing; the "shoulds", the "woulds", unnecessary tense changes, passive constructions, and superfluous clutter. It structures information around tasks and tells the reader *what to do* instead of building knowledge so that they can *work it out* for themselves. It provides the "minimum effective dose" and no more, which makes it efficient to use although it can take longer to write. As Blaise Pascal explained (translated from the original French), "I'm sorry for the length of this letter, I didn't have time to write a shorter one."

So, hopefully you're now sold on the idea of DITA XML and want to investigate further. Fortunately, you can. IBM open-sourced the whole scheme in about 2001, including DTDs and an Open Toolkit publishing engine, DITA-OT [2].

My project at the OUP can be described simply: I converted their instructions into DITA XML in three passes over 4 months.

- First I rewrote the headings in the Word documents and tagged all the semantic content using MS Word styles
- Next I used Eliot Kimber's excellent word2dita conversion from the DITA4Publishers Open Toolkit plug-in [3]. The word2dita XSL converts MS Office-Open XML (OOXML) into a ditamap and dita topics by referencing an XML mapping file. It creates the semantic nesting that is absent in office documents and can even capture complex constructs such as definition lists and images with captions within figures
- Finally, I worked through in the DITA XML semantic markup and combined and rewrote the 2000+ raw topics into 1300 focussed, definite instructions for typesetters.

Each topic is stored in an XML file with the extension `.dita` and these are tied together into a publication using a ditamap XML file. The topic XML structure is very simple and contains only semantic elements with no layout or style information. The map XML structure is very simple allowing relatively easy manipulation of maps and referenced topics with XSL. Obviously I worked with Tom through the project to automate global changes and incorporate as many data integration features as we could. Tom will tell you about some of those now.

4. Further benefits from XML technologies

Writing and validating XML examples in XML

Since there are so many potential ways in which we can leverage our team's XML expertise with DITA, we leave compiling a complete list (if such a thing is possible!) as an exercise for the reader. Examples range from improvements at quite a low level (e.g. version control on topics through SVN keywords) to sophisticated XML processing (e.g. XSLT transforms or delivery options for global changes).

Instead we will concentrate on one particular issue: example instances of XML.

The particular problem is that an example will normally include contextual mark-up that isn't directly illustrating the main subject. If the data-model or capture rules change, it's difficult or impossible to find every instance of the change in all examples across the entire data set.

A secondary issue was identified in the way in which the code examples are written. Styling the examples by adding `@outputclass` attribute values to every angled bracket, attribute name or value, processing instruction etc. was an obvious time sink.

Our proposed solution was to write the examples in XML, validate them using NVDL [4], then convert them into a DITA shared resource where each example can be referenced and reused dynamically.

4.1. Combining example document types with NVDL

I created a schema for a wrapper XML format to contain multiple examples from the different OUP datamodels.

Creating the schema had some challenges because of OUP's current use of DTDs for our datamodels:

1. OUP DTDs have limited namespace support
2. The editor's implementation of NVDL doesn't support DTDs

These were overcome:

1. By use of NVDL `context` to change `mode` based on element name in the wrapper XML format
2. By converting DTDs to schema.

NVDL allows separate validation at the level of the wrapper format as well as the contained OUP XML examples:

- Validation in the wrapper layer ensures higher level aspects such as the uniqueness of id attributes.
- Validation in the contained layer ensures that each example conforms to the relevant data-model.

Example 1. XML Examples Sample

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-model href="Examples/examples.nvdl" type="application/xml" ❶
3              schematypens="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"?>
4  <x:examples xmlns:x="http://schema.oup.com/dita/examples">
5    <x:example id="OUP_Law_Reports_431_singapore01"> ❷
6      <x:OxDTD><disclaimer> ❸
7    <p>Decision © Singapore Academy of Law under exclusive licence from the Government of Singapore.
8       The Academy reserves all rights in the content, which may not be reproduced without the
9       Academy's written permission.</p>
10 </disclaimer></x:OxDTD>
11   </x:example>
12 </x:examples>
```

❶ Line 2: associates the NVDL schema

❷ Line 5: includes the unique ID which will be used by DITA as a key.

❸ Line 6: the OxDTD element is used by the NVDL engine to choose which validation to apply.

Example 2. NVDL Sample

```
<rules
  xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  xmlns:x="http://schema.oup.com/dita/examples"
  xmlns:sch="http://www.ascc.net/xml/schematron"
  startMode="examples"
  >
  <mode name="examples">
    <namespace ns="http://schema.oup.com/dita/examples">
      <validate schema="examples.rng"> ❶
        <context path="OxDTD"> ❷
          <mode>
            <anyNamespace>
              <validate schema="schema/OxDTD/OxDTD.xsd"/> ❸
              <attach useMode="unwrap"/>
            </anyNamespace>
          </mode>
        </context>
        </validate>
      <validate schema="examples.sch"> ❶
        <context path="OxDTD" useMode="allow"/> ❹
      </validate>
    </namespace>
    <anyNamespace>
      <reject/>
    </anyNamespace>
  </mode>
  <mode name="unwrap">
    <namespace ns="http://schema.oup.com/dita/examples">
      <unwrap/>
    </namespace>
  </mode>
  <mode name="allow">
    <anyNamespace>
      <allow/>
    </anyNamespace>
  </mode>
</rules>
```

❶ Lines 10, 20: The wrapper format is defined as the RelaxNG schema 'examples.rng' and a schematron file 'examples.sch'. The former defines the model, the latter enforces ID uniqueness.

❷ Line 11: defines a change of context when the `OxDTD` element is encountered

❸ Line 14: defines which data-model to use to validate the contained XML

❹ Line 21: excludes the contained XML from the schematron validation

4.2. Code highlighting in DITA XML using XSLT

DITA doesn't contain semantic elements for all the constituent parts of an XML example although it does include phrase `ph` elements and an `outputclass` attribute that transforms to a CSS class in all the HTML-based outputs.

A (remarkably simple) XSL script converts from wrapper format to a dita topic with shared code-highlighted code blocks.

The DITA XML architecture provides a reuse mechanism using a system of content references to other files using keys (stored in the map) and element `id` attributes. To include an example in a topic, the Data Engineer simply adds the `codeblock` element with a `conkeyref` attribute that points via the key to the `codeblock/@id` in the examples topic.

We also added the classes and output classes to a CSS file and used an `@import` CSS statement in the editor CSS to style the examples consistently in the Oxygen [5] author view and HTML output formats.

Figure 1. Generated syntax highlighting in the editor:

Figure: Singapore disclaimer XML

```
<disclaimer>
  <p>Decision © Singapore Academy of Law under exclusive licence from the Government of Singapore.
     The Academy reserves all rights in the content, which may not be reproduced without the
     Academy's written permission.</p>
</disclaimer>
```

Figure 2. Generated syntax highlighting in the output format:

Singapore disclaimer XML

```
<disclaimer>
  <p>Decision © Singapore Academy of Law under exclusive licence from the Government of Singapore.
     The Academy reserves all rights in the content, which may not be reproduced without the
     Academy's written permission.</p>
</disclaimer>
```

Example 3. XSL Listing

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
4                  xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
5                  xmlns:x="http://schema.oup.com/dita/examples"
6                  exclude-result-prefixes="xs xd x"
7                  version="2.0">
8    <xd:doc scope="stylesheet">
9      <xd:desc>
10       <xd:p><xd:b>Created on:</xd:b> Feb 19, 2013</xd:p>
11       <xd:p><xd:b>Author:</xd:b> TFJH</xd:p>
12       <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
13       <xd:p>This stylesheet creates a shared repository of dita code
14             examples from a validated input source.</xd:p>
15     </xd:desc>
16   </xd:doc>
17
18   <xd:doc scope="component">
19     <xd:desc>
20       <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
21       <xd:p>output parameters specify DTD declarations etc</xd:p>
22     </xd:desc>
23   </xd:doc>
24   <xsl:output doctype-public="-//OASIS//DTD DITA Topic//EN"
25               doctype-system="topic.dtd" encoding="UTF-8"/>
26
27   <xd:doc scope="component">
28     <xd:desc>
29       <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
30       <xd:p>The root template creates the DITA superstructure.</xd:p>
31     </xd:desc>
32   </xd:doc>
33   <xsl:template match="/">
34     <topic id="topic_jys_5gr_fj">
35       <title>Common code examples</title>
36       <body>
37         <xsl:apply-templates/>
```

```
38        </body>
39      </topic>
40    </xsl:template>
41
42    <xd:doc scope="component">
43      <xd:desc>
44        <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
45        <xd:p>Each x:example element represents a new codeblock.</xd:p>
46      </xd:desc>
47    </xd:doc>
48    <xsl:template match="x:example">
49      <codeblock id="{@id}">
50       <xsl:apply-templates/>
51      </codeblock>
52    </xsl:template>
53
54    <xd:doc scope="component">
55      <xd:desc>
56        <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
57        <xd:p>Kick off code conversion when encountering a wrapper element.</xd:p>
58      </xd:desc>
59    </xd:doc>
60    <xsl:template match="x:OxDTD">
61      <xsl:apply-templates mode="code"/>
62    </xsl:template>
63
64    <xd:doc scope="component">
65      <xd:desc>
66        <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
67        <xd:p>General handling of elements within code blocks</xd:p>
68      </xd:desc>
69    </xd:doc>
70    <xsl:template match="*" mode="code">
71      <ph outputclass="XmlFurniture">&lt;</ph><ph outputclass="ElementName"><xsl:value-of select="name()
"/></ph>
72      <xsl:apply-templates select="@*" mode="code"/>
73      <ph outputclass="XmlFurniture">&gt;</ph>
74      <xsl:apply-templates select="node() except @*" mode="code"/>
75      <ph outputclass="XmlFurniture">&lt;/</ph><ph outputclass="ElementName"><xsl:value-of select="name(
)"/></ph><ph outputclass="XmlFurniture">&gt;</ph>
76    </xsl:template>
77
78    <xd:doc scope="component">
79      <xd:desc>
80        <xd:p><xd:b>Last Modified:</xd:b> TJFH, 2013-02-19</xd:p>
81        <xd:p>General handling of attributes on elements</xd:p>
82      </xd:desc>
83    </xd:doc>
84    <xsl:template match="@*" mode="code">
85      <xsl:text> </xsl:text><ph outputclass="AttributeName"><xsl:value-of select="name()"/></ph><ph outp
utclass="XmlFurniture">=&quot;</ph><ph outputclass="AttributeValue"><xsl:value-of select="."/></ph><ph
 outputclass="XmlFurniture">&quot;</ph>
86    </xsl:template>
87
88    <xd:doc scope="component">
89      <xd:desc>
90        <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
91        <xd:p>General handling of text nodes in code</xd:p>
92      </xd:desc>
93    </xd:doc>
94    <xsl:template match="text()" mode="code">
95      <xsl:value-of select="."/>
```

```
 96   </xsl:template>
 97
 98   <xd:doc scope="component">
 99     <xd:desc>
100       <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
101       <xd:p>General handling of comments in code</xd:p>
102     </xd:desc>
103   </xd:doc>
104   <xsl:template match="comment()" mode="code">
105     <ph outputclass="Comment">&lt;--<xsl:value-of select="."/>--&gt;</ph>
106   </xsl:template>
107
108   <xd:doc scope="component">
109     <xd:desc>
110       <xd:p><xd:b>Last Modified:</xd:b> TFJH, 2013-02-19</xd:p>
111       <xd:p>General handling of PIs in code</xd:p>
112     </xd:desc>
113   </xd:doc>
114   <xsl:template match="processing-instruction()" mode="code">
115     <ph outputclass="PI">&lt;?<xsl:value-of select="name()"/><xsl:text> </xsl:text><xsl:value-of selec
    t="."/>?&gt;</ph>
116   </xsl:template>
117
118   <xd:doc scope="component">
119     <xd:desc>
120       <xd:p><xd:b>Last Modified:</xd:b> TJFH, 2013-02-19</xd:p>
121       <xd:p>Recursion templates to ensure all elements and attributes match</xd:p>
122     </xd:desc>
123   </xd:doc>
124   <xsl:template match="node()|@*" mode="#default">
125     <xsl:apply-templates select="node()|@*" mode="#current"/>
126   </xsl:template>
127
128 </xsl:stylesheet>
```

4.3. Remaining challenges and developments

DTD/Schema support. The implementation of NVDL [4] in Oxygen [5] lacks support for DTD validation so we used trang to produce schema based versions. Any updates to the DTDs will require a re-transformation to update the schemas.

In the short term, we can mitigate by adding this as an automated part of our release process. In the long term we hope to migrate our datamodels to a compliant schema format.

Example excerpts and highlighting. Each example must be a complete, valid XML instance. This means examples will sometimes contain elements and structures that do not directly relate to the example have to be captured. Consider as an example an IDREF reference to a different part of the document structure!

One potential approach would be to mix elements from the wrapper format among the example data-model. These would be stripped out for validation, but in place to highlight, selectively include or exclude ranges of the example.

XML and Doctype declarations. XML and Doctype declarations would invalidate the data if included verbatim in examples. However it should be straightforward to add options to the wrapper format which can be used to recreate them.

Rule based validation. At the moment OUP use a proprietary format for our rule based validation (XML Probe) [6]. When we move to an NVDL compliant format (viz. Schematron) [7], the system will become far more powerful.

5. Conclusion

One of the major selling points of XML is that it promises reuse not only of data but also of skills. Sometimes it's easy to forget that we can apply these advantages to our own day to day work, such as documentation, as well as the XML which is our product.

We found that any technology that's available for the wider use of XML can introduce opportunities when you decide to use XML with your own processes.

Those principles of reuse are mirrored on a conceptual level with the guidelines introduced by DITA.

We hope that we've shown that there are opportunities to not only produce or process XML, but to use XML ourselves: to practice what we preach.

References

[1] Darwin Information Typing Architecture (DITA)
http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.html

[2] DITA Open Toolkit
http://dita-ot.sourceforge.net/

[3] DITA for publishers
http://dita4publishers.sourceforge.net/

[4] Namespace-based Validation Dispatching Language (NVDL)
http://www.nvdl.org/

[5] Oxygen XML Editor by SyncRO Soft SRL
http://www.oxygenxml.com/

[6] XMLProbe by Griffin Brown Digital Publishing Ltd.
http://www.xmlprobe.com

[7] Schematron
http://www.schematron.com

Optimizing XML for Comparison and Change

Nigel Whitaker

DeltaXML Ltd.

Robin La Fontaine

DeltaXML Ltd.

Abstract

Almost every user of XML will, at some stage, need to do some form of comparison between different versions of their XML data or document. This could be because it is necessary to review the changes that have been made, or to check that the output from one version of some software is the same as the previous version, or to find changes so that they can then be processed in some other way.

Designers of XML formats often do not consider this requirement for comparison when designing an XML format. However, if this is taken into account at the design stage then it can make the usefulness of the XML greater and at the same time reduce the cost of developing software. This paper outlines some of the issues that can usefully be considered by XML designers to make comparison and change easier to handle. The design issues will also be of interest to any user of XML to provide a better understanding of processing change in XML.

1. Introduction

It is possible to achieve better comparison of XML if you have some knowledge of the data or format being processed. At DeltaXML we have produced specific comparison products for formats such as DocBook and DITA and we often help our customers using their own in-house data and document formats. When doing this work we've often said to ourselves “if only they would have done this... it would make their life so much easier”. This paper is a review of some of these issues and recommendations - it could be called a 'wishlist' or 'manifesto' for comparison. Some of this is just 'good advice' that extends beyond comparison per-se. Taking some or all of this advice when designing XML formats will make comparison and other XML processing tools work better out of the box and with less configuration.

We consider comparison to be a problem of identifying similarity, aligning information, and then represent the changes so that they can be processed in various ways. Some data formats have built in mechanisms to describe change, for example the `status` attribute in DITA, similarly `@revisionflag` in DocBook and the `<ins>` and `<del>` elements in HTML. Where these facilities are provided it often makes sense to make a comparison tool that takes two inputs and produce a result which uses the facilities and is also a valid XML result. This goal sometimes conflicts with the design of the schema and some of these issues will be addressed later.

2. Use a DTD or Schema for Whitespace Handling

Consider Figure 1, “Added data” where a new (phone) element has been added to some contact data. The inexperienced user says “I've added an element, it has three children.”. We check if a DTD is used and when not finding one, we reply (perhaps a little pedantically) with: “No you haven't. You've added two nodes, an element and some text (which is a newline and spaces). The element contains seven children - three elements and four text nodes.”.

Figure 1. Added data

```
<contact>
  ...
  <phone type="work">
    <countryCode>44</countryCode>
    <areaCode>020</areaCode>
    <local>7234 5678</local>
  </phone>
</contact>
```

doi:10.14337/XMLLondon13.Whitaker01

Of course, it is possible to make comparison software remove the extra indentation whitespace and provide some configuration options. But this is delegating responsibility to the user, who may not have as good an understanding of the data as the developers.

As a user of a comparison tool - you may see changes that you don't care about and are a nuisance - the four added text nodes in the example above. There are other less obvious implications that we should also mention including performance and alignment.

Most comparison algorithm implementations are similar to those of XSLT transformation, in that you generally need to store the inputs in memory to navigate around them. Storing those extra tree nodes for the unnecessary whitespace affects both performance (it takes time to load and process them), but more importantly the heap space requirements. If you are processing data like that above, its quite possible to halve the heap memory size needed for comparison with proper whitespace handling.

Many comparison algorithms use *Longest Common SubSequence* (or *SubString*) optimization techniques [1]. These work best (give good results and have good performance) when there are reasonably long sequences of similar or identical content. When they are faced with XML data like that above, but where one input has an extra indentation whitespace node for each element and the other file does not (perhaps its a single line of XML without whitespace, which when loaded in an editor makes you go to immediately to the indent button), it is almost a nightmare scenario. The whitespace nodes break-up the sequences of element nodes that can match, furthermore the whitespace nodes are all identical (the same newline and number of spaces giving the same level of indentation) and match each other. This kind of data will often mismatch and is also slow to process.

So what is proper whitespace handling? It is possible to remove the indentation whitespace above by pre-processing the data, or having a post-process which ignores or removes the purely whitespace changes that are identified. But by far the best way of dealing with whitespace is to use a DTD or schema so that the parser can differentiate between element-only and mixed content [4]. When they are used (and associated with the data using a `DOCTYPE` or `@xsi:schemaLocation`), parsers such as Apache Xerces can use the `ignoreableWhitespace` callback method; comparison and other tools then know they can safely ignore that type of content.

3. Using Schemas and DTDs Consistently

We hope the previous section has provided a convincing argument to use a DTD or schema. However, if you go down this route it is worth remembering that using a DTD or schema has a number of implications. In addition to controlling whitespace and use for validity checking they are also used for 'Infoset Augmentation'.

Infoset Augmentation means adding data from DTD or schema to the resulting parsed representation. It is often used to specify values of attributes, for example that a table by default will have a 1 pixel border. It is also, more controversially, used to provide a default namespace to xhtml data. While it is possible in some cases to determine if data was provided by augmentation, we would encourage instead that DTD DocTypes and schema association be used consistently. This will avoid *spurious* attribute change that is often confusing to the user ("I can't see that in either of my inputs") and in the case of xhtml, avoid the issues around an `body` element not matching or aligning with an `xhtml:body` element.

We have recently been working on a comparison tool for a documentation standard. That standard included reference material for each element, in terms of a DTD grammar. It also included, as a download, a large set of modular DTD/entity files and a related set of W3C XML Schema files (.xsd), but nowhere in the standard did it mention how XML instance files were meant to use the DTD or xsd files; no mention of DOCTYPE requirements or use of schemaInstance, or statements about processing expectations. Implementers then chose whether to use a DOCTYPE or not. We are then faced with comparing mixed files and have to deal with the differences due to augmentation between the two inputs. If you provide DTD or xsd files as part of a standard or format definition and they are used for augmentation, then a little guidance to implementers on how you expect them to be used would sometimes be appreciated!

4. Use Appropriate Data Types

There's another benefit to using a DTD or schema, particularly for data-orientated XML. A DTD provides a limited form of data typing; attributes can be declared of type `ID` which constrains uniqueness, and whitespace normalization for `CDATA` attributes is different to that for other types of attribute. A richer form of data-typing is provided by W3C and RelaxNG schema languages. These use the XML Schema datatypes [2] to describe the types of elements and attributes. This information can then be used for normalization purposes or post-processing to remove unexpected change. For example, you may consider these timestamped elements to be equivalent: `<element time='2013-03-14T14:35:00Z'>` and `<element time="Thu 14 Mar 2013 14:35:00 GMT">`.

When using floating point (or 'double') numbers most developers programming in Java or C# are warned about doing direct comparison operations and told to use epsilon or 'units of least precision' (ULPs). Similar consideration should be given to XML data values, whether serialized as characters in an XML file or when loaded into an in-memory tree.

It's often simpler to think of all of the attribute and other datatypes as strings particularly when developing both reader and writer software. However, proper definition of the datatypes has benefits when for comparison and also for other more general forms of XML processing such as XSLT 2.0 transformation.

5. Consider using xml:space

We have discussed how an XML parser's whitespace handling is affected by the use of a DTD or schema. The `xml:space` attribute also has an effect on whitespace handling, but not within the parser. Instead it is used to control how downstream applications handle whitespace. Think of `xml:space="preserve"` as a *hands off my whitespace* instruction!

It is often needed because many text processing applications of XML will normalize text prior to presentation, collapsing sequences of spaces to a single space, normalizing between line breaks and spaces and fitting or *breaking* lines to fit the page or display width. There is an expectation that comparison tools do likewise, for example, not reporting where two spaces in one document correspond to three spaces in another. Therefore our default processing of textual data involves a whitespace normalization process. It is good practice to then provide `xml:space` support to users who wish to avoid this normalization.

The examples given in the XML spec for `xml:space` are of the `poem` and `pre` element. Many XML specifications then allow use of `xml:space` on various elements and this gives the user the opportunity to turn-off the default behaviour described above. We would suggest that grammar designers *sprinkle* the `xml:space` attribute around their text or mixed content whenever possible independently of whether they fix or default its behaviour on elements such as `pre`. This allows the user to refine the space handling on a per instance basis.

6. Consider Using xml:lang

Users of comparison tools like to see changes at fine granularity, they generally don't want to see which text nodes in an XDM tree have changed, but rather which 'words' have changed. In order to do this text is usually *segmented* or broken into smaller chunks for alignment and comparison. This process is harder than it initially appears. The concept of a word varies significantly in different languages and character systems. In Latin/European alphabets words are often delimited by spaces or punctuation whereas in eastern scripts 'words' are often represented by a single glyph or Unicode character.

Software is available to do this kind of text segmentation (the Java `BreakIterator` classes, or the widely used ICU [3] library which supports better internationalization). However in order to do its job properly it needs to be told which language is being used. This is where `xml:lang` is very useful. It is a small part of the XML specification [4], but often bypassed. Please consider adding it to your DTD or schema even if you don't envisage having multiple languages in a single XML file. If you add `xml:lang` to the root element, then it is possible for software to provide a default value, even if the user does not. This could perhaps be based on some user preferences, or computer locale settings.

Using `xml:lang` has benefits beyond comparison. Consider the case of single, double or triple clicking a range of text in an editor. The behaviour of these actions is also something that is typically language dependant and could utilise this attribute. Another useful general benefit, discovered when preparing this paper, is that `xml:lang` can be used to control spell checking dictionaries.

7. Data Ordering Issues

Some forms of data have an associated order and others do not. We will use a contact record as an example, as shown in Figure 2, "Mixed ordered and orderless contact data". In this example the order of the `addressLine` elements is significant for postal delivery. However users may not care about which order the various `phone` elements appear.

Figure 2. Mixed ordered and orderless contact data

```
<contact>
  <name>John Smith</name>
  <addressLine>25 Green Lane</addressline>
  <addressLine>Bloomsbury</addressline>
  <addressLine>London</addressline>
  <addressLine>UK</addressline>
  <postcode>W1 2AA</postcode>
  <phone type="office">+44 20 1234 5678</phone>
  <phone type="fax">+44 20 1234 5680</phone>
  <phone type="mobile">+44 7123 123456</phone>
 </contact>
</contact
```

Our comparison algorithms, and we suspect most others, are suited to either ordered/list based processing, perhaps optimizing a metric such as edit distance, or are suited to set-based or 'orderless' processing perhaps using hashing techniques and Map based data structures. Mixing these algorithms together so as to process the example above increases complexity enormously. Instead we prefer to treat all of the children of an element as either ordered or orderless and use the algorithms separately.

So, for the above data, rather than having a DTD like this :

```
<!ELEMENT contact
  (name, addressLine*, postcode, phone*)>
```

we prefer to use something like this:

```
<!ELEMENT contact (name, address, phone*)>
<!ELEMENT address (addressLine*, postcode)>
```

or perhaps this:

```
<!ELEMENT contact
  (name, addressLine*, postcode, phoneNumbers)>
<!ELEMENT phoneNumbers (phone*)>
```

Introducing these grouping elements to the contact data makes it possible to subdivide the ordered and orderless content. It also allows us to attach attributes to them to control how the comparison is then performed at a given level of the tree. These attributes are easy to add using XSLT if there is an XPath that corresponds to an orderless container. It is also possible to specify, and to some extent document, the orderless nature of the element in the DTD, for example:

```
<!ATTLIST phoneNumbers deltaxml:ordered
  (true|false) "false")>
```

8. Ids and Uniqueness in XML

The XML specification allows attributes to be declared of type `ID` or `IDREF`. One use-case of this facility is for cross-referencing within a document. For example a user inserts a table into a document and gives it an id, for example `<table xml:id="forecast">`, and when talking about the table would later write: `... In <xref linkend="forecast"/> we describe the ....` The assumption being that the processing system would put replace the `xref` with an appropriate reference.

The XML specification requires that such ids are unique and this is supported by the editing and processing applications. For comparison and change control we would like to recommend another property that such ids should have, is that of persistence. The cross reference is in some respect a cross-tree pointer in XML. That's fine, but it can also change and when it does we are faced with the problem of working out if the user has changed what is being pointed to, or if the thing being pointed to has changed. Perhaps it is the same thing, but it has also changed slightly? Working out what's happened gets very difficult in these cases. We would recommend that if you write software to read and write such content then as well as considering id uniqueness please also consider persistence through read/write cycles.

We've seen some examples of content which is converted from a word-processor format into XML markup where sections, tables etc are automatically numbered with a scheme such as "sect1_1, sect1_2 .. ". From a comparison perspective this is almost next to being useless, the same information is usually extractable using xsl:number or count(preceding-sibling::*) type operations. When the user creates a cross-reference please don't change the user-provided text. Adding artificial ids is usually a hindrence for comparison, particularly when a new chapter or section is interspersed into the content and causes subsequent elements to be renumbered differently. Finally we would suggest that schema designers do not make id attributes REQUIRED so that application developers and users do not have to come up with, probably artificial, numbering schemes.

The example we have shown above is a good use-case for XML ids. We don't recall many others. There is a danger of id/xml:id mis-use. Consider the case of a data file with National Insurance (NI) numbers. Don't be tempted to declare these of type ID/xs:ID because NI numbers are considered unique. Sooner or later you may need to add vehicle registration numbers of some other similar unique identifier and then face the problem that ID/xs:ID provides a single global namespace. Consider using schematron or perhaps XSD 1.1 assertions to specify uniqueness, they tend to be more flexible and also extensible for future needs.

9. Grammar Design Issues

When generating a comparison output, making use of the facilities provided by the data or document format being processed to describe the changes is useful as it allows users to make use of existing publication and other software available for that format. If that software validates its input then the comparison result should ideally also be valid. However there are some design issues that make this quite hard. We will illustrate this type of issue using DITA.

The DITA task specialization has a DTD model essentially similar to:

```
<!ELEMENT taskbody
  ((steps | steps-unordered), result)>
```

Using the status attributes we normally say whether an element has been added or deleted and this can then be styled appropriately with red or green colouring or strike-through. When one element is changed to another we can normally represent this as one being deleted and another being added. However the design of the DITA grammar precludes this when steps is changed to steps-unordered or vice-versa.

We can see why the designers thought that there should only be a single sequence of steps in a task. But from our perspective its difficult to represent this type of change in standard DITA. We currently provide a parameter in our DITA products which provides control over how the output is represented using either of these elements and include a comment which describes the details of the change.

From a comparison perspective grammars which allow *singleton choice* are harder to deal with, adding a repetition qualifier such as * or + makes representing change easier.

10. Formatting Element Patterns

The process of marking up mixed content with *formatting* information (by wrapping text in element such as: b, i, span or emphasis) is something which users (document editors or authors) typically do not see or appreciate in XML terms. Explaining that they have deleted some text and replaced it with an element containing some text rarely enlightens the situation. They are generally interested in the words that have changed and as a secondary concern what formatting may have changed. However, they do like to see the added or deleted text represented as it was formatted in the two input files.

To meet this requirement we tend to *flatten* formatting information. The result is a paragraph or other container of mixed content where all of the words are siblings at the same level of the XML tree. When compared in this re-organized form changes to the text are easier to align and identify. There are a number of flattening techniques that can be used, including using start and end markers, so that for example
`some <b>bold</b> text` becomes
`some <b-start/>bold<b-end/> text` or by moving formatting information to some out-of-band location.

When trying to generate a result we need to reconstruct as much of the original flattened hierarchies around the added and deleted content and in some cases those hierarchies can be in conflict.

We have found two properties of grammars that are very useful when reconstructing a good representation of the input hierarchies in the comparison result which we informally call removability and nestability:

Removability
: Can the wrapping element be removed still leaving a valid result? This is true for example, if the elements in the content model of a span or other formatting element, are the same as the content

model at the point where the span is used in the containing element.

Nestability
: Can the formatting element contain an immediate child of the same type? This is usually true when recursive reuse is used in a content model, for example allowing a span element directly containing another span element.

These properties are often true for established document formats such as DocBook, DITA and (x)html, however for simpler document formats they are not always true and cause difficulty for our algorithms which try to reconstruct the formatting hierarchies. As well as for comparison, we suspect these would also be good properties if the document format were to be supported by a WYSIWIG or authoring editor.

11. Processing Instructions (PIs) and Comments

Comments and Processing Instructions (PIs) are often used by developers as a simple or ad-hoc extensibility mechanism: “The grammar doesn't allow me to put this information here.... I know, I'll write it in a PI and change the reading code to understand those PIs.”

However, it is difficult to describe changes to them because they lack structure. PIs are often used to represent changes using accept/reject or review systems used in editors, and thus are a possible way of describing comparison results. However, when used in this way its almost impossible for accept/reject PI mechanisms to describe changes to PIs. Similarly CSS styling using attributes cannot be used for highlighting comment change.

We would therefore caution their use as an ad-hoc extensibility mechanism. It may work in limited scenarios with writer/reader agreement, but a general purpose comparison has a hard time describing the changes without resorting to similar ad-hoc mechanisms to do so.

Bibliography

[1] Binary codes capable of correcting deletions, insertions, and reversals. Vladimir I. Levenshtein. Soviet Physics Doklady, 1966.

[2] XML Schema Part 2: Datatypes Second Edition. W3C Recommendation. 28 October 2004. http://www.w3.org/TR/xmlschema-2/

[3] International Components for Unicode http://sites.google.com/site/icusite/

[4] Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. 26 November 2008. http://www.w3.org/TR/REC-xml/

What you need to know about the Maths Stack

MathML, MathJax, HTML5, and EPUB 3

Ms. Autumn Cuellar

Design Science

Mr. Paul Topping

Design Science

Abstract

MathML is a well-known and widely-used standard for encoding mathematics within XML workflows, but what you may not know is that MathML is not just a standard that affects your internal workflow, used only for storage and converted to images when you need to present your content to your audience. MathML is a key part of the digital publishing revolution towards enriched content. Its recent inclusion into the HTML5 and EPUB 3 standards is helping to bring to fruition the promise of interactive content for maths-based industries around the world.

All of the major browser vendors have pledged support for HTML5, and many EPUB readers are built on browser engines, thus full browser and e-reader support of MathML is expected in the future. Gecko-based browsers including Firefox already do include native MathML support. WebKit, the engine behind Safari and Chrome, also has some MathML support, though only Safari has included this feature in releases. Until universal support for MathML is available, MathJax fills in the gaps. MathJax is an open-source Javascript library for displaying MathML in all modern browsers and has been included with success in EPUB readers, such as Readium.

MathJax helps both content creators and content consumers realize the full advantages of MathML. One challenge that producers of digital content are faced with is the range of devices, screen sizes, and screen resolutions that must be supported. Unlike with bitmap images, MathJax easily adjusts equations to match the surrounding text without losing any quality. MathJax also leaves the MathML content available for re-use. This is particularly important to audiences who'd like to play with the equations in your content by loading (such as through copy and paste) the equations into Microsoft Excel or other calculation or modeling software, most of which accept MathML as an input format. Another feature of MathML that MathJax accommodates is its accessibility to audiences with vision and learning disabilities. MathML has been embraced by these communities and most accessible technology software includes support of MathML by reading the equation aloud, highlighting parts of the expression, and allowing control of the navigation of the equation. Finally, by leaving the MathML in the content, the equations remain searchable, allowing your content to be found.

The key elements to moving your maths forward into this bright new interactive publication era are a base doctype that allows MathML, tools to create MathML, XSL, and MathJax. Many standard doctypes already include MathML or are on their way to adding MathML support. OASIS, for example, keeps a DocBook MathML Document Type, and the next version of DITA (1.3) will also include MathML. All other doctypes, since XML is the eXtensible Markup Language, can be extended to include MathML. Though MathML can be manually written, it's a verbose language that was never intended to be written by humans. To create MathML, you will need a conversion tool to convert existing maths content to MathML and/or a MathML editor, such as MathFlow. To publish the XML to its digital output format, HTML5 or EPUB 3, a wide range of publishing tools are available on the market, most of which use XSL in some part of the process. The XSL or publishing tool can usually be easily modified such that the output includes the MathJax libraries. Lastly, you will need the MathJax libraries.

We are only just now starting to get a taste of the exciting future of digital content thanks to the HTML5 and EPUB 3 standards, and MathJax is filling the need of publishers with maths in their content until the browser and e-reader technology catches up to the standards. The benefits of MathML can only be fully realized when MathML is supported not just as an internal storage format but also as a delivery format, and with the Maths Stack, consisting of MathML, MathJax, HTML5, and EPUB 3, the maths in your digital publications are interoperable, accessible, and searchable.

doi:10.14337/XMLLondon13.Cuellar01

1. Introduction

The digital publishing era promises to revolutionize how we consume content. Plain text is now being enhanced not only with images but also with, for example, video, music, and interactive applications. At the heart of this revolution in publishing is XML. Recent developments in XML-based standards, such as HTML5 and EPUB 3, are facilitating the trend towards enhanced content.

The great news for those of us with any interest in scientific, technical, engineering, or mathematical (STEM) content is that mathematics, often an afterthought in web technologies, are a key part of the recent advancements toward enriched content. In this paper we'll discuss the latest developments in the XML standards for mathematics, web pages, and e-books; how MathML brings to fruition the promise of enhanced content for maths-based industries; the current status of support for these standards; and, finally, what you need to do to prepare your content for the bright new future of digital publishing.

2. Recent Standards Developments

2.1. MathML

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 some felt needed to be addressed. However, the challenges for representing maths on the web seemed too mountainous to be tackled in an update to HTML, so the MathML working group was formed to address the problems surrounding maths in XML/HTML.

The MathML 1 spec was finalized in 1998. Five years later, in 2003, MathML 2 was unveiled to correct flaws 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 now served by taking line breaks to the next level: for long equations that may extend past the width of pages or columns, content providers can now specify both a maximum width (so 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, such as stacked 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.

2.2. HTML5

While MathML has been developed and revised, it has established itself as a standard for encoding mathematics. It has been incorporated into a number of other standards from biological modeling languages (CellML and SBML) to medical journal archives (NLM). Tool support for MathML has advanced, with MathML import and/or export available from a variety of software applications ranging from word processors to computer algebra systems. Most importantly, from the perspective of the HTML Working Group, MathML rendering engines have arrived on the scene to provide expert layout of complex equations.

Mathematics did not make the cut for inclusion into HTML3 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. [Fernandes, 2012]

2.3. EPUB 3

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 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.

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.

3. How MathML Fits in with the HTML/EPUB Promise of Enriched Content

Maths content has long been included in e-books and web pages as images, plain text (often referred to as ASCII Math), or, for the enthusiasts, as TeX/LaTeX. However, by delivering the mathematics in your content as MathML, you can enable your audience new levels of interaction with the information presented. MathML is integral to HTML5's and EPUB 3's promises of enhanced publications. MathML offers the following advantages in usability and presentation over alternative formats.

3.1. Searchability

Annually, EMC, a maker of storage and big data systems, publishes a study called "Digital Universe" measuring the amount of digital data available. By the end of 2012, EMC measured 2.5 zettabytes of data available worldwide. The study showed the amount of data available worldwide had doubled in just two years. The same study predicts that by 2020, the total data count will exceed 40 zettabytes. "To put it in perspective, 40 zettabytes is 40 trillion gigabytes -- estimated to be 57 times the amount of all the grains of sand on all the beaches on earth." [Mearian, 2012]

With so much data available at our fingertips, search is becoming ever more important. An ambition in the field of mathematics is how best to search mathematical notation. Maths search could have several applications including aiding research in scientific communities, helping identify common patterns between remote disciplines, and serving as an educational tool. [Miner, 2004] Most of the studies that have had success in this area, such as MathDex, EgoMath, and MathWebWorld, have used alorithms relying on MathML. [Misutka, 2008]

3.2. Localization

While maths is a universal language and semantically one expression might be the same as the next, the notation used to express a given equation might vary significantly from region to region. For instance, notation for expressing long division differs distinctly depending on where you are in the world. With XML stylesheets, one can target the display of mathematical content to the region where it's being observed.

3.3. Flexibility of Display

Since MathML is a part of the document and is rendered at load time, environmental settings can be factored into the display of the maths. Therefore, if a visitor or reader has his browser or e-reader font size set to a larger font size than the default setting, the equations will also appear at the larger base font size. The audience will also see high quality equations no matter the resolution or pixel density of the device loading the web page or e-book. This is a special shortcoming of using images for maths display, as images intended for lower pixel density monitors display poorly on high pixel density monitors and vice versa.

MathML, like XML, gives you power of presentation, allowing content providers to apply a certain style to equations along with the rest of the document without having to go through and change each equation individually. For example, equations in tables can be displayed at a smaller font size than the rest of the document or maths in change tracking systems can be highlighted with a foreground (or background) color to indicate changes to an expression.

3.4. Accessibility

Content providers may be required by government statute to supply accessible materials to readers with vision and learning disabilities, however, even if government regulations don't apply, there is no reason to limit the available audience. The accessibility community favors MathML because MathML's encoding of a mathematical expression is precise enough that it may be used to translate the equation to Braille or speech text and can also be used to navigate a complex equation so that parts of the equation can be repeated in a semantically relevant way. Most standards intended to support the accessibility community, including DAISY for Digital Talking Books and NIMAS for instructional materials, require maths to be encoded in MathML.

3.5. Interoperability

Perhaps the most exciting feature of MathML is that it is receiving growing support in a wide variety of applications: computer algebra systems, graphing applications, calculators, modeling software, assessment creation systems, educational whiteboards, etc. Even today one can copy and paste (or import and export) MathML between applications to make better use of data. It isn't difficult to envision the future of MathML: Imagine a doctor is reading a journal article describing mathematical functions behind chemical pathways in the body. To help him visualize how changes in the level of sodium affect the pathways, he sends the equations in the journal article to a modeling system. Within seconds he is able to start adjustments until a life-saving solution is found. Alternatively, the journal article may include a simple modeling system that allows the reader to adjust the parameters of the mathematical functions in the article for hands-on reinforcement of the concepts being described. The potential for interoperability is only limited by the imagination.

4. Status of MathML Support in HTML5 and EPUB 3

The vendors of the major browsers have been vocal in their support of HTML5. In fact, all of the browser vendors have representatives on the HTML Working Group. Support for MathML is already available in the Gecko rendering engine, which is used by Firefox. WebKit, the engine behind Safari, can render a subset of MathML. Opera currently uses CSS to provide support for MathML. As part of HTML5, browser support for MathML will no doubt only improve in the coming years.

Since EPUB 3 is based on HTML5, and since many e-readers are built on browser technology, some e-readers have inherited MathML support. Apple's iBooks e-reader is a primary example. Based on WebKit, iBooks has limited support of MathML. With the enthusiasm that's been shown for EPUB 3, the backing of all of its major features, including MathML, is expected in future versions of all e-readers.

However, knowing that support is coming is of little use to those who want to start taking advantage of all of the benefits of MathML today. With the gaps in browser and e-reader display of MathML today, this is an obstacle. No one wants to produce content so that the intended audience has to jump through hoops to view it. Without a doubt, every website visitor has been on the wrong end of content delivery requirements: a website might display a message stating that you do not have the required version of Flash or that the website is optimized for a certain version of browser, which is (of course) not the browser you are using. You are left to contemplate whether the content you have come to consume is worth the effort of updating software or switching browsers, and sometimes it's not.

Luckily, a Javascript project called MathJax has been developed to eliminate the gaps in browser and e-reader support of MathML.

4.1. MathJax

MathJax is an open-source Javascript library for rendering MathML in all modern browsers. It is developed by a consortium consisting of Design Science, American Mathematical Society, and the Society for Industrial and Applied Mathematics. Internet Explorer and Google Chrome, two browsers that have no native support of MathML, using MathJax, can now display HTML5 pages with MathML in them without the use of a client-installed plug-in. It also adds more complete support of the MathML specification than Firefox, Safari, and Opera have alone.

Because MathJax improves browser display of MathML, all e-readers built on browser technology can improve their EPUB 3 support by implementing the MathJax Javascript library. Calibre is a popular e-reader application that has already included MathJax to display e-books with mathematical content.

5. Enriching Content with MathML

Your excitement to get started with your maths stack is palpable. Without delving into too much technical detail, the four areas requiring consideration when setting up your maths stack are your doctype, MathML editors, the conversion process, and MathJax.

Assuming you are working in a source format other than HTML5 or EPUB 3, first you'll want to make sure your XML doctype includes MathML. Some standard doctypes, such as Docbook and NLM, do already include MathML. If your doctype does not already, you'll need to modify your schema to include the MathML schema.

Once your schema has been set up to include MathML, the next step is to start creating MathML. While MathML is human-readable, it's a verbose language that doesn't lend itself to manual creation. Quite possibly your XML editor of choice already includes a MathML editor or has a plug-in available for download. If not, Design Science offers a suite of MathML editors for integration with your workflow. Most MathML editors have a WYSIWYG interface so that you never have to see the MathML.

After creating your content, you might run it through a conversion process, usually involving XSLT, to produce HTML5 and EPUB 3 output. The conversion process needs to be modified to include the calls to MathJax in the header. Also, the MathML will need to be passed through the process untouched. By default, XSLT processors will remove the MathML tags and only pass the CDATA through; thus, you will need to add a few lines to your XSLT library to make sure the MathML tags make it to the output.

Finally, you may need to set up MathJax on your server. The MathJax Consortium does host a version of MathJax to which you can link your pages, but if you have heavy traffic, like to have control of the libraries used on your site, or want to customize MathJax at all, you will need to download and install MathJax.

6. Conclusion

Recent developments in digital publishing standards are changing the way we present and consume information. Mathematics are a key part of this revolution with MathML's inclusion in HTML5 and EPUB 3. MathML brings to fruition the promise of enriched content for STEM industries by providing the following advantages: MathML is searchable and accessible, it allows easy localization of content, it is flexible in its display across multiple devices and in its support of stylesheets, and finally MathML's use as an exchange format provides endless possibilities in the way mathematical content in web and e-book publications can be enhanced for your audience.

MathML is already supported by a number of browsers and e-readers. To fill the gaps, an open-source Javascript library by the name of MathJax provides MathML rendering for all modern browsers. MathJax is also being used to successfully add proper display of MathML in e-readers.

The maths stack consisting of MathML, MathJax, HTML5 and EPUB 3 can be implemented by modifying your XML schema to include MathML, adding a MathML editor to your XML editor, making sure your conversion pipeline passes the MathML through untouched, and setting up MathJax on your server.

Bibliography

[Mearian, 2012] Lucas Mearian. 11 December 2012. *By 2020, there will be 5,200 GB of data for every person on Earth*. Computerworld Inc.

[Miner, 2004] Robert Miner. 8 June 2004. *Enhancing the Searching of Mathematics*. Design Science.

[Misutka, 2008] J. Mišutka. Copyright © 2008. *Indexing Mathematical Content Using Full Text Search Engine*. 240-244. *WDS '08 Proceedings of Contributed Papers* . Part I.

[Fernandes, 2012] Rossi Fernandes. 15 March 2012. *What HTML5 means to you*. Tech2.com India.

Small Data in the large with Oppidum

Stéphane Sire

Oppidoc

<s.sire@oppidoc.fr>

Christine Vanoirbeek

EPFL

<christine.vanoirbeek@epfl.ch>

Abstract

The paper addresses the topic of frameworks intended to speed up the development of web applications using the XML stack (XQuery, XSLT and native XML databases). These frameworks must offer the ability to produce exploitable XML content by web users - without technical skills – and must be simple enough to lower the barrier entry cost for developers. This is particularly true for a low-budget class of applications that we call Small Data applications. This article presents Oppidum, a lightweight open source framework to build web applications relying on a RESTful approach, sustained by intuitive authoring facilities to populate an XML database. This is illustrated with a simple application created for editing this article on the web.

Keywords: XML, web development, framework, XQuery, XSLT, RESTful approach, Oppidum

1. Introduction

There are still millions of individually operated web applications that contain only a few "megabytes" of data. These are not Big Data applications although their addition still constitutes a remarkable part of the web. As a matter of fact, it currently exists an amazing number of web applications that run small to medium size corporate web sites or aim at integrating the work of associations in various domains.

This category of applications is often based on a PHP/MySQL stack to deal with factual data and requires aside use of an office suite (Word, Excel, etc.) to deal with document-oriented information. Functionalities range from publishing news, blogs, agenda, and catalogue of members to provision of a set of services such as registration, on-line shopping or more specific processes.

We call this class of applications Small Data application. Characteristics of those applications are the following ones : mostly publication oriented, asymmetrical (few contributors and few to many readers), evolutive (frequent updates) and extensible (through modules, e.g. image galleries, shopping cart, registration, e-book generation, etc.).

Because semi-structured data models procure the possibility to encompass both data and document-oriented representation of information [1], there are many reasons why such applications would benefit from an XML technology stack. For instance, to build custom schema aware search engines for better information retrieval, for single-source and cross-media publishing, or most importantly for data portability.

There is always the alternative to build Small Data applications on a hosting platform with an embedded site authoring tool (e.g. Weebly or Google site). However it does not easily support extensibility and data reuse.

For other Small Data applications, teams of developers are using classical (aka non-XML) long standing technologies (aka relational databases). This is most probably because there are now dozens of popular server-side frameworks to lower the barrier entry cost for developers, when not totally commodifying development to customizing a set of configuration files in the so-called Content Management Systems (CMS).

We believe there are two reasons preventing the adoption of XML technologies for Small Data applications. The first one is the lack of adequate browser's based-editing facilities. Most of the CMS use rich-text-editors and HTML forms. Thus they miss the capability to really structure the input space. The second one is the complexity of building XML applications : teams developing Small Data applications work under small budget constraints (usually a few thousands Euro/ Dollar per project) and they can't afford a long learning curve and/or development cycle.

doi:10.14337/XMLLondon13.Sire01

To some point XForms could have solved the first issue. However it has been designed for form-based data and is less powerful for handling document-oriented semi-structured data (see for instance these developer's question on Stack Overflow [3] [4]).

The blossoming of many XML development environments based on XQuery (BaseX, eXist-DB, MarkLogic, Sausalito) could solve the second issue. However each one comes with its own extensions and conventions to build applications, often still at very low abstraction level compared to over MVC frameworks available for other platforms (e.g. Ruby on Rails).

Despite these obstacles we have started to develop Small Data applications with XML during the last three years. We have solved the first issue by developing a Javascript library for XML authoring in the browser called AXEL [13]. It uses the concept of template to allow developers to create customized editing user interfaces guaranteeing the validity of data [6]. We resolve the second issue by providing a lightweight framework called Oppidum that is described in this paper.

Oppidum is an open source XML-oriented framework written in XQuery / XSLT. It is designed to create custom Content Management Solutions (CMS) involving lightweight XML authoring chains. It is currently available as a Github repository to be deployed inside an eXist-DB host environment.

The paper is organized as follows. The first section describes the architecture of Oppidum and its two-steps execution model. The second section presents the way it manages some common web application design patterns. The third section presents some example applications done with it. Finally the fourth section discusses some design choices of Oppidum compared with over XML technologies.

2. Oppidum architecture

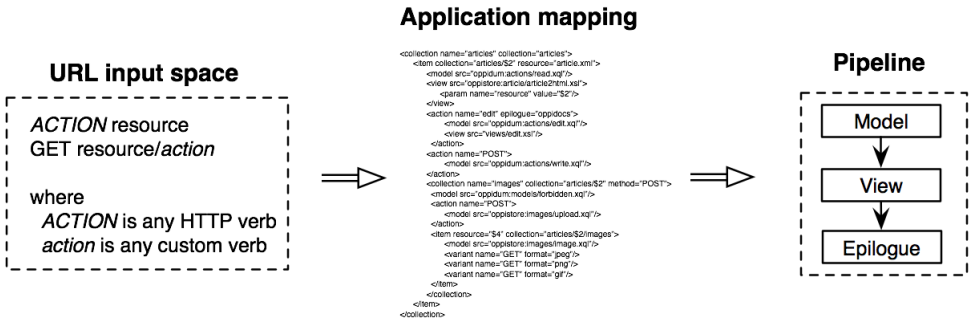
2.1. Application model

Oppidum provides developers with a simple application model that allows them to rapidly and efficiently deploy a RESTful application; it relies on a few fundamental principles:

- the application must be entirely defined in terms of *actions* on *resources*;
- each action is defined declaratively in a mapping file as a sequential pipeline composed of three steps :
  - the first step, called the *model*, is always an XQuery script;
  - the second step, called the *view*, is always an XSLT transformation;
  - the third step, called the *epilogue*, is always an XQuery script that must be called `epilogue.xql`;
- a pipeline may have only the first step, the first and second steps, the first and third steps, or the three steps.

It is always possible to extend a pipeline by invoking one or more XSLT transformations from XQuery in the first and the third steps.

Figure 1. The application mapping defines a pipeline for each resource / action

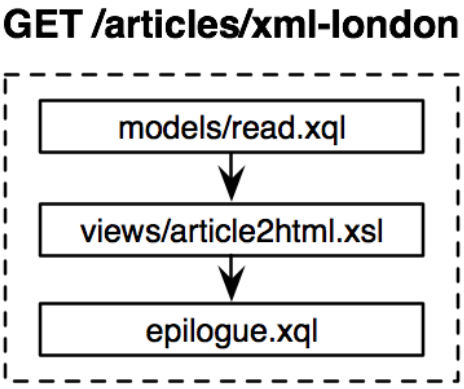


The application mapping describes the mapping of the URL space with the pipeline definitions.

An action is either an HTTP verb (e.g. GET, POST, PUT, DELETE) or a custom name. In the first case the HTTP request corresponding to the action is the request using the same verb. In the second case the corresponding HTTP request is any request with a request URI path ending by the action name.

In application mapping terms the `GET articles/xml-london` HTTP request is matched as calling the `GET` action on the xml-london resource. The Figure 2, "Example of a pipeline to view this article using Oppidum" shows an example of a pipeline that implements the rendering of the resource as an HTML page : the 1st step calls a `models/read.xql` script that returns the raw XML data of the article. It could also perform side effects such as checking user's access right or updating access logs. The 2nd step calls an XSLT transformation `views/article2html.xsl` that generates an HTML representation. Finally the 3rd step is an `epilogue.xql` script. It inserts the HTML representation of the article into an application page template with extra decorations such as a navigation menu and/or some buttons to edit or publish the article.

Figure 2. Example of a pipeline to view this article using Oppidum

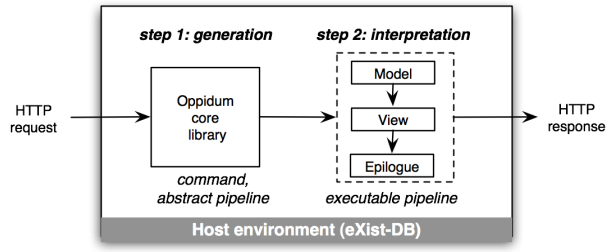


The previous example is equivalent to a RESTful operation to get a representation for a resource [2]. It is also possible to consider extended RESTful operations by considering the resource as a controller and to define custom verbs. In that case the verb must be appended to the end of the resource-as-a-controller URL. For instance, the `GET articles/xml-london/publish` request could be matched as calling a custom publish action onto the xml-london resource. Currently custom actions cannot be targeted at a specific HTTP verb, this is a limitation. As a consequence it is up to the developer to enforce specific semantics for different HTTP verbs if required.

2.2. Execution model

Oppidum execution model is a two steps process. The first step takes the client's HTTP request and the application mapping as inputs. It analyses the request against the mapping and generates a pipeline to execute in the host environment. The second step executes the pipeline and returns its output into the HTTP response.

Figure 3. The execution model



In the eXist-DB host environment, the first step is invoked in a `controller.xql` script file. That scripts calls an Oppidum `gen:process` method that returns an XML structure specifying a pipeline to be executed by the URLRewriter servlet filter of eXist-DB. The second step is entirely left to that filter, that executes the pipeline.
The `gen:process` method executes a sequential algorithm with three steps : the first step produces an XML

structure that we call the command, the second step transforms the command into an internal abstract pipeline definition, finally the third step generates the executable pipeline for the host environment.
The command is persisted into the current request as an attribute. Thus it is available for introspection to the XQuery and XSLT scripts composing the pipeline. This is useful to write more generic scripts that can access data copied into the command from the target resource or the target action mapping entry.

2.3. Conventions

Oppidum uses a few conventions, although it is very often possible to bypass them writing more code.

As per eXist-DB there is only one way to invoke Oppidum: it is to create a `controller.xql` file at the application root. The script will be invoked by the eXist servlet with the HTTP request. It must call the `gen:process` method with the mapping and a few environment variables as parameters. Similarly the epilogue must be called `epilogue.xql` and placed at the application root. The same controller.xql file can be copy / pasted from a project to another.

All the code in production should be stored in the database inside a `/db/www/:app` collection (where :app is the application name). Configuration files (error messages, mapping resource, skin resource, etc.) must be placed inside the `/db/www/:app/config` collection. It is recommended to store all application's user generated content inside a `/db/sites/:app` collection (and sub-collections). In our current practice those conventions ease up multiple applications hosting within the same database.

The library enforces conventions onto the URL mappings mostly to offer debug services:

- adding `.xml` to a URL executes only the 1st step of the pipeline and returns its result;
- adding `.raw` to a URL executes only the 2nd step of the pipeline and returns its result;
- adding `.debug` (or a `debug=true` request parameter) prevents the pipeline execution, instead it returns a dump of the command and of the generated pipelines.

Consequently, developers should not use the epilogue to apply state changes or side effects to the database.

3. Oppidum design patterns

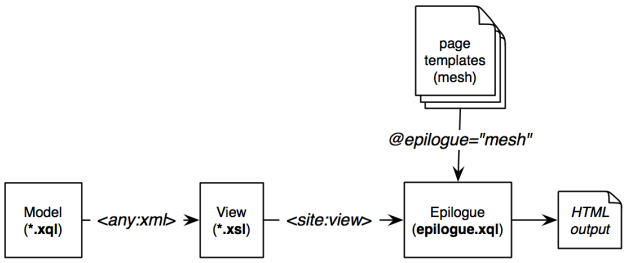
Oppidum architecture and execution model support common web application design patterns. In some cases we have extended the mapping language and/or the Oppidum API to support more of them.

3.1. Template system

The pipeline generator generates the epilogue step of the pipeline if and only if the target mapping entry has an epilogue attribute. The `epilogue.xql` script can interpret the value of this attribute as the name of a page template file defining common page elements such as an application header, footer and navigation menu. We call this template a mesh.

This is summarized on Figure 4, "Conventional pipeline for using the template system". The template system also relies on a pipeline where the view step must output an XML document with a `site:view` root element containing children in the `site` namespace defined for that purpose.

Figure 4. Conventional pipeline for using the template system



A typical `epilogue.xql` script applies a typeswitch transformation to the mesh [14]. The transformation copies every XHTML element. When it finds an element in the `site` namespace, called an extension point, it replaces it with the content of the children of the `<site:view>` input stream that has the same local name. If not available, it calls an XQuery function from the epilogue file named after the element and replaces the element with the function's result. So if an element from the mesh is `<site:menu>`, and there is no `<site:menu>` element in the input stream, it will be replaced with the result of the `site:menu` function call.

The typeswitch function at the heart of the template system can be copied / pasted between applications. It relies on a variable part for the extension points that is customized for each application.

3.2. Skinning applications

The template system can be used to decouple the selection of the CSS and JS files to be included in a page from the direct rendering of that page.

A typical `epilogue.xql` script defines a `site:skin` function to be called in place of a `<site:skin>` element from the head section of a mesh file. That function dynamically inserts the links to the CSS and JS files. Oppidum provides a `skin` module for managing the association between string tokens and sets of CSS and JS files, called profiles.

The module supports several profile levels:

- profiles associated with a given mesh name;
- profiles associated with keywords generated and passed to the epilogue script as a `skin` attribute of the `<site:view>` root element of the pipeline input stream;
- a catch-all `*` profile applying to every page rendered through the epilogue together with some exceptions declared using predicates based on a micro-language.

The keyword profiles are useful for a fine-grain control over the selection of CSS and JS files to be included by generating the appropriate keywords from the XQuery model scripts or XSLT views. The catch-all profile is useful to insert a favicon or web analytics tracker code.
The profile definitions are stored in a `skin.xml` resource in a conventional location inside the database.

3.3. Error management

Another common pattern is to signal errors from the scripts composing the rendering pipeline, and to display these errors to the user. Oppidum API provides a function to signal an error and another function to render an error message in the `epilogue.xql` script.
A typical `epilogue.xql` script defines a `site:error` function to be called in place of a `<site:error>` element placed anywhere inside a mesh file. That function calls Oppidum error rendering function. The skinning mechanism is also aware of the error API since it allows to define a specific skin to render for the displaying and disposal of the error messages.

The error management code is robust to page redirections, so that if a pipeline execution ends by a redirection, the error message is stored in a session parameter to be available to the rendering of the redirected target page.

There is an identical mechanism to display messages to the users. The messages are stored in an `errors.xml` (resp. `messages.xml`) resource in a conventional location inside the database for internationalization purposes.

3.4. Data mapping

It is common to generate pages from content stored in the database. Thus it is very frequent to write code that locates that content in terms of a collection and a resource and that simply returns the whole resource content or that extracts some parts of it. In most cases, the collection and resource paths are inferred from segments of the HTTP request path.

For instance, in a blog, the `posts/123` entry could be resolved as a resource `123.xml` stored in the `posts` collection.

The mapping file allows to associate a reference collection and a reference resource with each URL. They are accessible from the model script with the `oppidum:path-to-ref-col` and `oppidum:path-to-ref` methods that return respectively the path to the reference collection and the path to the reference resource. This mechanism fosters the writing of generic model scripts that adapt to different data mapping.

In addition, the reference collection and resource can be declared using variables that will be replaced with specific segments of the HTTP request path. For instance, if a reference resource is declared as `resource=$3`, the `$3` variable will be replaced by the third segment of the request path.

3.5. Form-based access control

It is frequent to restrict access to some actions to specific users or groups of users. Thus, a common pattern is to check users' rights before doing any further action and ask for user identification. Using the Oppidum constrained pipeline that would mean to always invoke the same kind of code in the model part of a pipeline.

Oppidum alleviates this constraint with some extensions to the application mapping syntax that allows to declare access control rules. When rules have been defined, they are checked directly by the `gen:process` function, before generating the pipeline. An alternative pipeline is generated in case of a refusal. It redirects clients to a login page with an explanation.

The access to each resource or action can be restricted individually by defining a list of role definitions. The roles are defined relatively to the native database user definitions and resource permissions : the `u:`name role restricts access to the user named name; the `g:`name role restricts access to users belonging to the group named name; finally the `owner` role restricts access to the owner of the reference resource declared in the data mapping.

3.6. Development life cycle

It is common to have different runtime environments when developing, testing or deploying an application (e.g. dev, test and prod environments). For instance the application may be developed with eXist-DB in standalone mode, while the final application may be deployed under Tomcat. This may impose a few constraints on some portions of the code. For that purpose the application mapping defines a `mode` attribute that remains accessible with the Oppidum API. It can be used to adapt functionalities to the environment. For instance it is possible to apply different strategies for serving static resources while in dev or in prod, or to enable different debug functionalities.
It is also recommended to share the application code with other developers using a code repository system such as Git. For that purpose we have found useful to develop applications directly from the file system. This way it is possible to commit code to the repository at any time and to move all of the application code to the database only when in prod. Oppidum supports the turn over with an installation module. It provides a declarative language for the application components and their localization into the database. This module supports the creation of an installer screen to automate installation. The installer may also be used to copy some configuration files and/or some initial data or some test sets to the database while in dev or in test modes.

Since the release of eXist-DB 2.0, that comes with a complete IDE for in-browser development, the life-cycle based on source code written in the file system may no longer be the most efficient way to work. Thus we are considering to update the installation module so that it is possible to work directly from the database and to package the library and the applications as XAR archives. However, at the moment, we are still unsure how this would integrate seamlessly with code versioning systems such as Git.

4. Example applications

4.1. Illustrative example

This article has been edited with an application written with Oppidum. Its mapping and its mesh are shown in the code extracts below. The article editor itself is an XTiger XML document template [12] as explained in [6] [13].

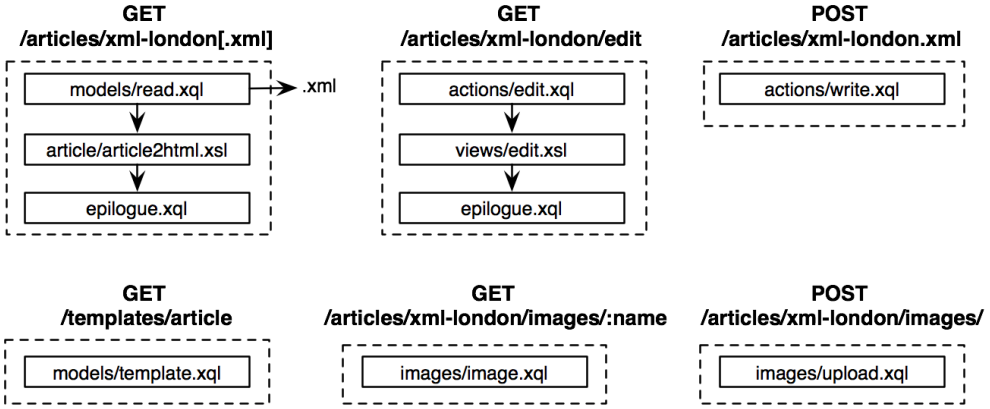
The mapping defines the following URLs:

- `GET /articles/xml-london` returns the article as HTML for screen display
- `GET /articles/xml-london/edit` returns an HTML page containing an editor based on the AXEL library, that page loads two additional resources :
  - `GET /templates/article` returns the XTiger XML template
  - `GET /articles/xml-london.xml` returns the article as raw XML
- `POST /articles/xml-london.xml` saves the article back to the database
- some other URLs manages image upload and retrieval :
  - `POST /articles/xml-london/images/` saves a new image to the database
  - `GET /articles/xml-london/images/:name` returns the image stored in the resource name

The raw XML version of the article shares the same pipeline as the HTML version of the article. This exploits the .xml suffix convention that short-circuits any pipeline to return the output of the first step.

Four of the application pipelines are implemented as a single step pipeline executing an XQuery script. This is because either they directly returns a resource from the database (this is the case for the XTiger XML template or to serve an image previously stored into the database), or because they are called from an XHR request implementing a custom Ajax protocol where the expected result is coded as raw XML. For instance, the protocol to save the article only requires the POST request to return an HTTP success status (201) and to set a location header. It then redirects the window to the location header address which is the representation of the article as HTML. These Ajax protocols depend on the AXEL-FORMS javascript library that we are using to generate the editor [11], which is out of the scope of this article.

Figure 5. Pipelines for the application used to edit this article



The code extract below shows most of the application mapping defining the previous pipelines.

Example 1. Mapping extract for the application used to edit this article

```
<collection name="articles" collection="articles"
            epilogue="standard">
  <item collection="articles/$2"
        resource="article.xml"
        supported="edit" method="POST"
        template="templates/article"
        epilogue="oppidocs">
    <access>
      <rule action="edit POST" role="g:authors"
            message="author"/>
    </access>
    <model src="oppidum:actions/read.xql"/>
    <view src="article/article2html.xsl">
      <param name="resource" value="$2"/>
    </view>
    <action name="edit" epilogue="oppidocs">
      <model src="actions/edit.xql"/>
      <view src="views/edit.xsl">
        <param name="skin"
               value="article axel-1.3-with-photo"/>
      </view>
    </action>
    <action name="POST">
      <model src="actions/write.xql"/>
    </action>
    <collection name="images"
                collection="articles/$2"
                method="POST">
      <model src="models/forbidden.xql"/>
      <action name="POST">
        <model src="images/upload.xql">
        </model>
      </action>
      <item resource="$4"
            collection="articles/$2/images">
        <model src="images/image.xql"/>
      </item>
    </collection>
  </item>
</collection>
```

Without entering into too much details of the mapping language, the target resources are defined either by a `collection` element if they are supposed to contain an indefinite number of resources, or by an `item` element if they are supposed to contain a finite number of resources. Actions are defined by an `action` element. The hierarchical structure of the mapping file follows the hierarchical structure of the URL input space : the name attribute matches the corresponding HTTP request path segment in the hierarchy and anonymous item elements (ie. without a name attribute) match any segment string at the corresponding level.

The use of anonymous `item` elements to define resources allows to create generic mappings that work with collections of resources such as collection of articles or collection of images inside articles. As such the xml-london resource illustrating this article is mapped with the anonymous `item` element on the second line.

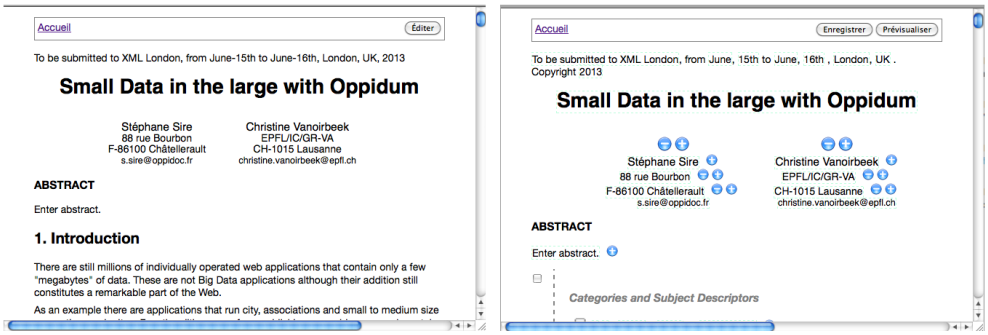
Some notations are supported to inject segment strings from the request path into the mapping using positional $ variables. For instance resolving $2 against the `/articles/xml-london` URL returns the xml-london string.

The mapping language also makes use of annotations to support some of the design patterns or specific features:

- the `epilogue` attribute selects a mesh to render the page in the epilogue as explained in the template system design pattern;
- the `access` element supports the access control design pattern;
- the `collection` and `resource` attributes support the data mapping design pattern;
- the `template` attribute indicates the URL of the XTiger XML template for editing a resource;
- the `param` element implements mapping level parameters transmitted to the model or view scripts.

The application screen design is quite simple as it displays a menu bar at the top with either an Edit button when viewing the article, as shown on Figure 6, “Screen shots of the article editing application with the shared menu bar”, or a Save and a Preview buttons when editing it. Both screen are generated with the mesh shown below.

Figure 6. Screen shots of the article editing application with the shared menu bar



Example 2. mesh to display the article or to edit the article

```
<html xmlns:site="http://oppidoc.com/oppidum/site"
      xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <site:skin/>
    </head>
    <body>
        <div id="menu">
            <site:commands/>
        </div>
        <div id="article">
            <site:content/>
        </div>
    </body>
</html>
```

The mesh defines three extension points in the `site` namespace. The `<site:skin>` extension point calls a `site:skin` XQuery function as explained in the skinning design pattern. The `<site:commands>` extension point calls a `site:commands` XQuery function that generates some buttons to edit (when not editing), or to save (when editing) the article. Finally the `<site:content>` extension point is a place-holder for the homonym element's content to be pulled from the input stream and that contains either the article when viewing or an HTML fragment that defines an editor using AXEL-FORMS when editing.

4.2. Other applications

During the last three years, several Small Data applications have been developed using Oppidum. We mention a few of them, emphasizing the features they present accordingly to our definition of Small Data applications: mostly publication oriented, asymmetrical, evolutive and extensible.

The first one is a publication-oriented application for the editing and publication of a 4 pages newsletter provided by Platinn (an organization that offers coaching support to Swiss enterprises). The newsletter data model has been derived from a set of legacy newsletters with two goals : first the ability to define an XSLT transformation that generates a CSS/XHTML representation suitable for conversion to PDF ready to be sent to the print shop, second the ability to export the newsletter to a content management system. The application design is quite close to the illustrative example above, some additional resources have been defined to manage authors profiles and a log mechanism to track and display the editing history, mainly to prevent concurrent editing of the same newsletter. Until now 10 newsletters have been written by a redaction team of up to 5 redactors, with more than 30000 printed copies.

The second one is an application for maintaining and publishing a multilingual (french and english) database of startup companies members of the science park (societes.parc-scienfique.ch) at EPFL (École Polytechnique Fédérale de Lausanne). A contact person and the staff of the science park can edit a public company profile with several modules. It illustrates the “asymmetrical” and “evolutive” characteristics since the goal of the application is to encourage editors to frequently and accurately update their presentation. For that purpose we are using statistics about the company profiles to generate some recommendations to improve their presentation, and to advertise most recently updated companies. There are currently about two hundreds companies using this system. The semi-structured document-oriented approach has been validated when we have been asked to open a web service to feed an internal advertising TV screen network in the different buildings of the park with some company profile extracts to avoid multiple inputs.

The third and fourth applications are web sites of associations. One is a craftsmen's association of Belgium called Union des Artisans du Patrimoine de Belgique (uniondesartisansdupatrimoine.be), and the other one is the Alliance association (alliance-tt.ch) that provides assistance in building partnerships between industry and academic research in Switzerland. Both are centred on traditional content management oriented features to maintain a set of pages and to publish articles about events and/or members that could have been done with non-XML frameworks. However we have been able to extend them with custom modules such as a moderated classified ads services reserved for members in the case of the craftsmen's association, and an event registration module for the Alliance association. This last module has benefited from XML technologies in that it has been turned into an editorial chain with the ability to edit custom registration forms for each event, and to generate different types of documents such as a list of badges for printing and participants' list to distribute. Moreover these extensions, thanks to the application model enforced by Oppidum, are clearly separated from the other parts of their original application. Thus they can be ported to other projects by grouping and copying the pipeline files and by cut-and-paste of the corresponding application mapping parts.

5. Discussion

We do a quick comparison between Oppidum and Orbeon Forms, RESTXQ, Servlex and XProc.

The Orbeon Forms page flow controller dispatches incoming user requests to individual pages built out of models and views, following the model / view / controller (MVC) architecture [5]. This is very close to the Oppidum architecture presented in Section 2, "Oppidum architecture" however there are a few differences. Orbeon page flow controller usually integrates page flows definitions stored within the folders that make up the application code. With Oppidum the application mapping, which is equivalent to the page flows, is a monolithic resource, although we have been experimenting with modularization techniques for importing definitions not described in this article. One of the reasons is that we see the application mapping as a first order object, and hence as a document which can be stored in the database as the other user-generated content. It could ultimately be dynamically generated and or edited by end-users.

The syntax is also quite different : the page flows determines the correspondence between URLs and their implementation with the implicit directory structure of the source code and with regular expressions for less implicit associations; in Oppidum this is the implicit structure of the application mapping tree. The principal reason is that the application mapping in Oppidum aims at decoupling the RESTful hierarchy from the code hierarchy which is difficult to achieve with Orbeon Forms. Another side reason is that this allows to define some kind of cascading rules to inherit the reference collection and resource which have no equivalent with Orbeon Forms.

Like RESTXQ [10] Oppidum proposes a complete RESTful mapping of an application. However it diverges in the granularity of this mapping. While RESTful XQuery proposes a very elegant syntax to bind individual server-side XQuery functions with RESTful web services, Oppidum granularity is set at a coarser pipeline grain level. As a consequence Oppidum mapping is also decoupled from its implementation and is maintained in a single document which can be further processed using XML technologies as explained above. On the opposite RESTXQ mappings are defined as XQuery 3.0 annotations intertwined with function definitions only available to the developers of the application.

Both solutions provide different means to select the target mapping entry, and to parse and to communicate parameters from the request to the code generating the response. They also use syntactic artefacts to select different targets in the mapping based on different request's properties (URL path segments, HTTP verbs, HTTP headers, etc.). In this regards Oppidum is much more limited than RESTXQ since it only discriminates the targets from the path segments and the HTTP verbs. We could envision to extend Oppidum mapping language with further conditional expressions, however it seems a better solution could be to mix both approaches : in a first step, Oppidum mapping could be used to do a coarse grain target selection, then in a second step, RESTXQ could be used inside the model part of each pipeline to select between a set of functions to call.

We can see this need emerging in Oppidum as for limiting the number of XQuery files we found useful under some circumstances to group together related functionalities inside a single XQuery script shared as a model step between several pipelines. For instance, it is tempting to group reading and writing code of a given type of resource, together with some satellite actions, to create a consistent code unit (e.g. use a single script to create, update and read a participant's registration in an application). As a consequence the file starts by checking more properties of the request to select which function to call, which seem to be one of the reason that led Adam Retter to propose RESTXQ in replacement of the verbish circuitry inside former eXist-DB `controller.xql` files [9]. To our current perception, parts of these efforts tend to address the same kind of issues as object relational mapping in non-XML frameworks, without yet a clear solution for XML applications.

To some extent Oppidum shares goals similar to the Servlex EXPath Webapp framework [7]. Like it it defines how to write web applications on server-side, using XML technologies (currently XSLT and XQuery) and it defines their execution context, as well as some functions they can use [8]. However Oppidum is more restrictive as it imposes a RESTful application model and some constraints on the generated pipeline composition, thus it looks like a restriction of this framework. It will be interesting, in the future, to check if applications written with Oppidum could be automatically converted and/or packaged as EXPath Webapp applications and what would be the benefits. In particular we see a strong interest in accessing the host environment functionalities which are database dependent today (like accessing to the request or response objects) and that could be abstracted into common APIs with the EXPath framework.

Finally Oppidum does not currently makes use of XProc although it's execution model is based on simple pipelines. The first reason is historical since we have started to work with Orbeon Forms XML pipeline language (XPL) before Oppidum. The lack of design patterns and good practices led us to overcomplexify simple developments, and thus led us to invent the Oppidum self-limiting three steps pipeline model in reaction. But now with the return on experience, we are more confident in opening up Oppidum to support direct inclusion of XProc definitions within our application model. A quick solution could be to support pipeline files as models or views (as it is the case in Orbeon Forms) or eventually as epilogue. For that purpose, it would be feasible to rewrite the Oppidum pipeline generator to directly generate pipelines written in XProc instead of the URLRewrite filter pipeline format currently imposed by eXist-DB.

6. Conclusion

The XML Stack clearly proposes a valuable technological option to deal with data manipulated by Small Data applications. It offers the great potential to adopt a uniform representation of information that bridges two often separated paradigms: document and database systems. Adopting so-called semi-structured data models allows capturing in a homogeneous way the structure of information, at a very fine granularity level, if needed. It significantly enhances reuse of content in different purposes either for developing new services or delivering information through many channels.

The resulting benefits are particularly important even for low-budget class of applications. For end-users it avoids spending time in potentially manipulating the same pieces of information through different user interfaces or copying/pasting information, for instance, from a document to a web form. For developers existing semi-structured data may be reused to cost effectively add a wide range of new functionalities to a web application as, for instance, the generation of badges and list of participants who registered to an event. Finally, it is obvious, the XML format is especially well adapted to generate publishable documents or to deliver content on cross-media platforms.

Two main challenges need to be taken up to promote the adoption of the XML Stack in Small Data applications: the capability offered to end-users to easily provide valid content on the web and the provision to developers with a framework that can be rapidly mastered. The paper presented Oppidum, a lightweight framework to build such applications. It integrates the use of AXEL, a template-driven editing library that guarantees the validity of data provided by end-users.

The past and ongoing developments made us confident that Oppidum may evolve, while keeping it simple, to address another concern of Small Data applications: the cooperation aspects. Relying on a declarative approach, we believe that, with proper data modelling, it would be easy to develop simple workflows to support cooperative processes among stakeholders of a web application.

In this perspective, our further research work is targeting two main inter-connected issues:

- the first one is addressing the modeling of cooperative/communication processes to be supported in order to make efficient the production, the sharing and the exploitation of XML information by a community of users. It covers, amongst other preoccupations, the following issues: awareness, notification and role controls;
- the second one is addressing the collaborative editing of content itself. For the time being, it does not exist too many XML web-based collaborative authoring environments. We are interested in investigating about models and specification of an appropriate language that help to sustain the collaborative production task as well as associated actions.

Bibliography

[1] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset and Pierre Senellart: Web Data Management. Cambridge University Press 2011.

[2] Subbu Allamaraju: RESTful Web Services Cookbook. O'Reilly Yahoo! Press.

[3] Anonymous (user1887755): Web-based structured document authoring solution (Stack Overflow). http://stackoverflow.com/questions/13777777/web-based-structured-document-authoring-solution

[4] Anonymous (user1887755): Is Drupal suitable as a CMS for complex structured content? (Stack Overflow). http://stackoverflow.com/questions/14214439/is-drupal-suitable-as-a-cms-for-complex-structured-content

[5] Erik Bruchez: Orbeon Developer and Administrator Guide : Page Flow Controller. http://wiki.orbeon.com/forms/doc/developer-guide/page-flow-controller

[6] Francesc Campoy-Flores and Vincent Quint and Irène Vatton: Templates, Microformats and Structured Editing. Proceedings of the 2006 ACM Symposium on Document Engineering, DocEng 2006. doi:10.1145/1166160.1166211.

[7] Florent Georges: Servlex. https://github.com/fgeorges/servlex

[8] Florent Georges: Web Application EXPath Candidate Module 9 March 2013 (in progress). http://expath.org/spec/webapp/editor

[9] Adam Retter: RESTful XQuery Standardised XQuery 3.0 Annotations for REST. XML Prague 2012. http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[10] Adam Retter: RESTXQ 1.0: RESTful Annotations for XQuery 3.0. http://exquery.github.com/exquery/exquery-restxq-specification/restxq-1.0-specification.html

[11] Stéphane Sire: AXEL-FORMS Web Site. http://ssire.github.com/axel-forms/

[12] Stéphane Sire: XTiger XML Language Specification. http://ssire.github.com/xtiger-xml-spec/

[13] Stéphane Sire and Christine Vanoirbeek and Vincent Quint and Cécile Roisin: Authoring XML all the Time, Everywhere and by Everyone. Proceedings of XML Prague 2010. http://archive.xmlprague.cz/2010/files/XMLPrague_2010_Proceedings.pdf

[14] Priscilla Walmsley: XQuery, Search Across a Variety of XML Data. O'Reilly

Extremes of XML

Philip Fennell
<philip.fennell@gmail.com>

1. Introduction

The Extensible Markup Language (XML) is a meta language, it is used to describe other languages and as such it has been phenomenally successful, it has been drawn into just about every information domain imaginable, not to mention some you really can't. Although it might be unfair to term the extremes as lurching from the sublime to the ridiculous it is, however, fair to say that some applications of XML are incredibly useful whilst others are insanely convoluted and frustrating to work with.

XML, originally envisaged as a lingua franca for the Web, was born of the Standard Generalized Markup Language (SGML) but with rather neater, tighter rules to make it more concise. Some of the earliest applications of XML hinted at the diversity of uses to which it would be put. However, of more recent times the XML community has been reflecting upon the very nature of XML, its processing rules and serialisation, and has been looking for ways to simplify it in order that it might find a more amicable position along side, for example, the JavaScript Object Notation (JSON).

As hinted at above, XML has been applied, sometimes ill advisedly, throughout the IT industry and beyond. Over the last 15 years, looking at the applications of XML as a whole, they can be broadly boiled-down to the following categories:

- Describing
- Modeling
- Processing
- Publishing
- Interacting
- Presenting

At the heart of what we do with XML is describe things, essentially it is used for marking-up information to be published unless it is metadata about those published entities or how the marked-up information is to be modelled, processed,interacted with or presented.

When looking at the extremes of XML, what I find fascinating is not how much or how fast but the breadth of applications to which XML has been applied, and this is what one could call the 'XML Envelope'.

Whilst periods of introspection are often fruitful, we should also, on occassion, turn our attention outwards towards the edges of this envelope and see where XML has pushed itself into some new or unusual applications.

2. Describing

Describing things is not an edge-case for XML, it is at the centre because it is at the heart of what we do with XML. By description we actually mean information about things, metadata which, we capture as annotations within or in reference to these structures. We'll push outwards towards the edges of our envelope and work around the categories.

3. Modeling

We use XML to help us capture how we model a specific information domain using XML Schema (XSD) and Relax NG, however, at the very edge of current data modeling interest lies the Semantic Web where the boundary between XML and the Semantic Web has been and still is somewhat blurred.

doi:10.14337/XMLLondon13.Fennell01

XML and the Semantic Web have something of a mixed and, it should be said, confused relationship. XML has become, for many applications, a singular standard for information representation and interchange whilst the Semantic Web has managed to adopt at least four, and rising, representations of the Resource Description Format (RDF) and for many people the existence of one of them, RDF/XML, has posed a problem. A very real problem because it was seen as an information markup language rather that an XML representation of a graph. There is a subtle but important difference.

Whilst it can still be advocated that there's is nothing wrong with using XML technologies, like XSLT for transform RDF/XML, the querying of it with non-graph based languages, like XQuery, is analogous to querying XML with text based techniques like Regular Expressions. Whilst you can do it, in their respective cases you fail to see the underlying structures to the information without a lot of additional work.

When we stand at the edge, the interface between the Document and Semantic Web, we see that XML's usage within Semantic Web technologies is somewhat on the slide. Compared to Notation 3 (N3), N-Triples, Turtle and the upcoming JSON-LD. RDF/XML is increasingly becoming a legacy format and with RDF 1.1 it will no longer be the 'required' interchange format for RDF implementations.

So, leaving aside questions of graph serialisation, XML content does still intersect with the Semantic Web because much of the information it marks-up has, or is, metadata that could, and should, be modelled as RDF Linked Data. The question is how do the two sides meet and to what extent should they overlap. By way of a related example, mappings between Relational Data and RDF already exist in the form of the W3C's 'RDB to RDF Mapping Language' (R2RML) and the 'Direct Mapping of Relational Data to RDF'. These two concepts provide a means to create RDF 'views' of relational data that could be queried with the SPARQL query language. This is a solution for structured data but what of semi-structured data?

Schema Lifting and Lowering are terms coined by the Semantic Web community to describe the process of mapping XML Schemas to RDF Ontologies and back again. The idea being that you can enhance the basic semantics of XML fragments by converting them to RDF. Buried deep within the myriad of W3C Web Services recommendations lies the 'Semantic Web Annotations for WSDL and XML Schema' (SAWSDL) which uses foreign attributes to embed references between schema types and ontology classes and properties.

```
<xs:complexType xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:sawsdl="http://www.w3.org/ns/sawsdl" name="entryType"
                sawsdl:modelReference="http://bblfish.net/work/atom-owl/2006-06-06/#Entry">
  <xs:annotation>
    <xs:documentation> The Atom entry construct... </xs:documentation>
  </xs:annotation>
  ...
</xs:complexType>
```

To bring SAWSDL into line with the next part of this we'll translate the model references into proper schema appinfo annotations.

```
<xs:complexType xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sawsdl="http://www.w3.org/ns/sawsdl" name="entryType">
  <xs:annotation>
    <xs:appinfo>
      <sawsdl:modelReference
        href="http://bblfish.net/work/atom-owl/2006-06-06/#Entry"/>
    </xs:appinfo>
    <xs:documentation> The Atom entry construct... </xs:documentation>
  </xs:annotation>
  ...
</xs:complexType>
```

The next question is, how do you utilise these annotations to build a mapping from a document tree to a graph? An XSLT transform is a good option but the variability in schema construction has always made this task more complicated than we would care for. Alternatively, Type Introspection in XQuery, as described by Mary Holstege in her paper 2012 Balisage paper of the same title, is a useful tool to aid this process. The ability to access schema annotations, through an XQuery API, whilst processing an instance of that schema provides a quick and easy root for interpreting the schema to ontology references and thus extracting an RDF graph derived from information within an XML document that has not been otherwise explicitly marked-up as such in the first place.

```
<!-- OWL Class definition of an Atom Entry. -->
<owl:Class xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:rdfschema="http://www.w3.org/2000/01/rdf-schema#"
           rdf:about="http://bblfish.net/work/atom-owl/2006-06-06/#Entry">
  <rdfs:label xml:lang="en">Entry Class</rdfschema:label>
  <rdfs:comment xml:lang="en">see 4.1.2 of the rfc 4287 spec</rdfs:comment>
  ...
</owl:Class>

<!-- Source Atom XML Fragment. -->
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Atom-Powered Robots Run Amok</title>
  ...
</entry>

# Triple from the shadow graph (Turtle).
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix awol: <http://bblfish.net/work/atom-owl/2006-06-06/#> .

[ a awol:Entry ] .
```

This technique may have uses beyond the domain of Web Services as it has the potential to describe mappings between semi-structured XML data and RDF ontologies that enable hybrid Content / Graph stores to index content for both tree and graph based querying from the same source XML. To be able to almost seamlessly mix XQuery and SPARQL querying over the same content would be a powerful toolset in deed. Initial experiments have gone some way to proving that this technique works by generating an RDF dataset that can be loaded into a graph store and queried separately, but to fully realise this concept would require an indexer to be able to build the 'shadow' graph index via the these rules rather than create an entire graph and thus duplicate much, if not all of the original information.

In a world that's throwing itself at the notion of schemaless databases it seems to fly-in-the-face of the this current vogue to suggest using schemas and ontologies to define these mappings up-front. But, where better should we do such a thing than in declarative languages that are independent of implementation.

4. Processing

There are many processes that can been seen as a series of transformations of static information from one form or structure to another but the same can also be said for application state. As surely as a document's structure is a snapshot of its current revision so is an application's data structures a snapshot of its current state. XML has often been used to describe an application's initial configuration or to maintain a set of preferences but what about using it in the actual manipulation of that application's state. Crossing the line from passive state representation to active state manipulation.

The XML Pipeline Language (XProc) succeeds as a mechanism for orchestrating sequences of XML processing steps but what is potentially a more interesting and useful aspect of XProc is its ability to create a Domain-specific Language (DSL) for a particular set of steps. You can create, what could be call 'naked' pipelines that use the underlying XProc grammar or, alternatively, you can abstract that away behind typed step descriptions.

The new state processing grammar that you create becomes your DSL for this particular task and which is hosted within the XProc framework. A practical example of this being the ability to manipulate the configuration of an application; the consumption and transformation of application state and the execution of update instructions to the application.

```
<admin:get-configuration/>

<admin:forest-create forest-name="forest1"
    host-id="hp6910-772" data-directory=""/>

<admin:database-create ml-database-name="data"
    security-db="Security" schema-db="Schemas"/>

<admin:save-configuration-without-restart/>
```

Application configuration can be viewed as a pipeline of instructions that involve a source input as the current state, a set of actions that may be ordered due to dependencies, and an instruction to commit the changes. Such a set of steps, that make a specific configuration change, can be regarded as compound action and can themselves be composed into large sets of changes.

```
<admin:get-configuration/>

<mla:create-database database="data"
    schema-db="Schemas" security-db="Security">
  <p:input port="forests">
    <p:inline>
      <mla:forests>
        <mla:forest name="forest1" host="hp6910-772"
                    data-dir=""/>
      </mla:forests>
    </p:inline>
  </p:input>
</mla:create-database>
```

The composability of XProc processing steps is an incredibly powerful tool that, allied with the ability to define your own steps and abstractions, has applications outside of its originally intended domains.

5. Publishing

For the most part, publishing appears to be a reasonably straightforward process of marking-up textual content with semantically meaningful elements that tell us more about the author's intent than just the plain words, it gives structure. There are extensive grammars for this process, DocBook and XHTML being the more notable but markup is only the first part.

The separation of concerns tells us to keep the markup of content free from presentation and layout. Whilst keeping them separate is the goal, at some point they meet in order to present a finished result.

Taking the publishing of information in an intriguing direction is the Document Description Format (DDF), a framework that looks at documents as Functions over a dataset and as such presents new opportunities for the merging of data and documents. As we attempt to tackle the problems of handling vast amounts of data and getting it to the point of consumption in a manner tailored to the recipient, DDF is, quite possibly unique, as a publishing format in that it can consume many and varied sources of information, transform them to a common structure and then apply presentation constraints that can build flexible page layouts that adapt to variability in the source data.
DDF defines three layers:

- Data Binding
- Common Structure
- Presentation

```
<doc>
  <data>
    Source data for the document.
  </data>
  <struct>
    Structure and Transforms.
  </struct>
  <pres>
    Presentation and Transforms.
  </pres>
</doc>
```

Through these layers the separation of concerns in aggregation, structuring and presenting information are maintained, within a portable document structure.

DDF's main design decision was to 'consider the document as a function of some variable application data'. When the document is evaluated, data (may be from a number of sources), is transformed into a common structure from which a view can be generated for presentation. Embedding XSLT syntax and semantics into the document provides the mechanism for transforming within and between the layers.

Partial evaluation is also possible where processing results in new transformations to be applied in subsequent document workflow steps.

Along with the idea of 'document as a function', another aspect of DDF that is worth noting is the presentation system that extends the notions of layout beyond both the conventional flow and and copy-hole concepts. Although supporting flows, as can be found in HTML viewers, the introduction of constraints between components that ensure their spatial relationship is maintained within a variable data environment.

6. Interaction

An area where XML applications have found mixed success is in user interaction. XForms is our most notable success which, despite having a long history of partial implementations, which have come and gone over the years, XForms has managed to retain and has even grown enthusiasm for it with the more recent rise of Native XML databases.

That said, there's another area of interaction, that goes beyond conventional form filling, and that involves animation. The Synchronised Multimedia Integration Language (SMIL) has a whole subset of its extensive specification set aside for animation and it is a specification that is shared by Scalable Vector Graphics (SVG).

Whilst there is a lot of effort that goes into building scripting libraries for every form of user interaction and animation these are mostly reinventions of the same basic principles of user interface or temporal event driven alterations to the properties of some object in the page. SMIL, like XForms has been there, in the background, for many years and SMIL 1.0 is the oldest of the applications of XML, becoming a recommendation in June 1998 (four months after XML 1.0 itself).

SMIL Animation has, historically, been tied to its host document types: SMIL, HTML+TIME and SVG but with the advent of SMIL Timesheets that dependecmcy was broken, enabling the principles of event sequencing and orchestration being applied, out-of-line. Normally we might consider animation techniques being applied to presentation, timing of slide decks, linking steps in a configuration wizard or just simple page display effects. But, if you can orchestrate the display processes of information in a web page, why not orchestrate the aggregation and transformation of information in a workflow?

A value of XProc has been to lift the orchestration of XML transformations out of programming language dependency and into the domain of declarative programming. However, any additional sequencing of pipeline executions is still controlled outside of the pipeline. XProc step execution can be seen as inherently event driven due to the fact that steps start when an input is received at a source input port and end when a result appears on a result output port.

The sequencing is implementation dependant therefore allowing the possibility of asynchronous step processing. Now, imagine an XProc Pipeline that aggregates information from one or more sources. Through the use of SMIL Timesheets it is quite conceivable that you can orchestrate the execution of steps in a pipeline to explicitly execute steps concurrently, like requesting information from multiple web services:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
    xmlns:terms="http://example.org/service/terms/"
    name="term-aggregation"
    version="1.0">

  <p:pipeinfo>
    <smil:timesheet
        xmlns:smil="http://www.w3.org/ns/SMIL30">
      <smil:par>
        <smil:item select="#service1"
         begin="term-aggregation.begin" dur="30s"/>
        <smil:item select="#service2"
         begin="term-aggregation.begin" dur="30s"/>
        <smil:item select="#service3"
         begin="term-aggregation.begin" dur="30s"/>
      </smil:par>
    </smil:timesheet>
  </p:pipeinfo>

  <terms:get xml:id="service1" name="OpenCalais"
      href="http://opencalais.com/"/>
  <p:sink/>
  <terms:get xml:id="service2" name="MetaCarta"
      href="http://www.metacarta.com/"/>
  <p:sink/>
  <terms:get xml:id="service3" name="Yahoo"
      href="http://search.yahooapis.com/"/>
  <p:sink/>

  <p:wrap-sequence xml:id="aggregate-terms"
      name="aggregate"
      wrapper="terms:group">
    <p:input port="source">
      <p:pipe step="OpenCalais" port="result"/>
      <p:pipe step="MetaCarta" port="result"/>
      <p:pipe step="Yahoo" port="result"/>
    </p:input>
  </p:wrap-sequence>
</p:pipeline>
```

The above example illustrates how three calls to separate web services, that would conventionally run sequentially, could orchestrated to run explicitly in parallel. Each step is defined to begin when the pipeline starts and to have a maximum buration (timeout) of 30 seconds.

This is another example where bringing together seemingly unrelated applications of XML can extend their respective usefulness and push the boundaries of what can be achieved with XML.

7. Presentation

The final step from information to presentation is the conversion of text, values and graphics into a visual representation, a rasterization of shapes for display or printed media.

When it comes to vector graphics on the web, SVG is on the ascendant again which is primarily due to increased browser support. As a vector format, SVG has had the lion's share of people's attention whilst X3D, the XML representation of the Virtual Reality Modeling Language (VRML) has never really taken the web by storm. Alongside these public facing vector graphics markup grammars there are 'behind the scenes' formats like Collada which is squarely aimed at 3D digital assets interchange and archiving. But, in all these cases the objects or scenes they represent end-up being rendered into raster image formats for presentation on some form of device or media.

Experts at Pixar, the Computer Generated Imagery (CGI) film studio that brought us Toy Story and the like developed a 3D image rendering architecture called Reyes. The principle behind the Reyes rendering architecture is to take complex 3D graphics primitives and break them down into smaller, simpler components that can be more easily processed for visibility, depth, colour and texture. Reyes defines a processing pipeline that applies a series of transforms to the source primitives until they are finally at the point where sampling can take place to generate the resultant pixels.

But what has this to do with XML? Another boundary, at the edge of the XML envelope is that between text and binary data formats. XSLT is designed for generating text-based representations of information. When XSLT 2.0 came along it introduced the concept of sequences. A sequence can be made of nodes or atomic values, including integers. A rasterized image is just one long sequence of integer values with some header info to tell you, amongst other things, the format of a pixel's colour values. To push the boundaries of the XML envelope still further, using XSLT 2.0 and SVG, it is possible to develope a constrained implementation of the Reyes rendering pipeline and a TIFF encoder that proves the point that XSLT is not just limited to text and markup output but can deliver, in effect, binary data too.

The image you see here was created from a simple SVG graphic where all the rectangles are sub-divided down into pixel sized micro-polygons upon which depth sorting, transparency, colour and final sampling calculations are applied to generate a sequence of pixel value integers before finally encoded according to the TIFF format and then serialised as a Base64 encoded file, to make it possible to output the resultant image data. The icing on the cake would be to create a new serializer for the XSLT processor so that a binary file could be generated directly as the output.

This example is somewhat extreme as it is highly questionable as to whether XSLT is the right technology for rendering simple images let alone complex photo realistic ones. Certainly it is not fast and it is not very efficient either but it illustrates that the boundaries are not hard and fast. In fact the main components of the pipeline processing were relatively straight forward as XSLT provides the tools to transform and XML the means to represent the graphic primitive information.

8. Overlapping Edges

In the XML Envelope there is no clear dividing line between the categories:

1. All the categories require description.
2. Modeling relies upon processing to enable mappings between data models.
3. Information and application state can be transformed through processing.
4. Publishing utilises processing at every point of its life-cycle.
5. Interaction can be applied to the orchestration processing.
6. Processes of presentation can be enhanced by mechanisms of interaction.

9. Conclusion

This walk around the edges of this so called XML Envelope has illustrated, I believe, that whilst so much has been accomplished over the last 15 years and from which we have very robust tools, techniques and design patterns that we can apply, I'm convinced that XML does not really have any hard and fast limits to its application and whilst we may dwell upon issues of serialisation and simplification the essence of what has been created is right and good and that there are many more applications to which XML and its attendant technologies can be put than we might have originally imagined...

The National Archives Digital Records Infrastructure Catalogue: First Steps to Creating a Semantic Digital Archive

Rob Walpole

Devexe Limited / The National Archives

<rob.walpole@devexe.co.uk>

Abstract

This paper describes the context and rationale for developing a catalogue based on Semantic Web technologies for The National Archives of the United Kingdom as part of the Digital Records Infrastructure project, currently in progress at Kew in London. It describes the original problem that had to be resolved and provides an overview of the design process and the decisions made. It will go on to summarise some of the key implementation steps and the lessons learned from this process. Finally, it will look at some of the possible future uses of the catalogue and the opportunities presented by the use of Semantic Web technologies within archives generally.

1. Background

1.1. The National Archives

The National Archives (TNA) are the official archives of the UK Government. TNA holds over 11 million historical government and public records [1] in the form of documents, files and images covering a thousand years of history. The vast majority of the documents currently held are on paper. However, as the digital revolution continues, this will soon be overtaken by a tsunami of digital files and documents for which a new and permanent home must be found that will allow controlled access for future generations.

These documents can take many forms including standard office documents, emails, images, videos and sometimes unusual items such as virtual reality models. Digital preservation brings a myriad of challenges including issues such as format recognition, software preservation and compatibility, degradation of digital media and more. Some of these issues were clearly demonstrated by the problems encountered by the BBC Domesday Project [2].

1.2. The Digital Records Infrastructure

TNA have been at the forefront of meeting this digital preservation challenge and have made great strides in finding solutions to many of the issues along with colleagues from other national archives, libraries and academia. In 2006, they deployed the Digital Repository System (DRS) which provided terabyte scale storage. Unfortunately DRS can no longer meet the vastly increased volumes of information produced by the Big Data era or the "keep everything" philosophy that cheap storage allows.

Now a new and far more extensible archive system, the Digital Records Infrastructure (DRI), is being built on the foundations of DRS to provide a quantum leap in archive capacity. This new system will allow long term controlled storage of a huge variety of documents and media. Digitised Home Guard records from the Second World War were used for the proof of concept and many more record collections, such as the Leveson Enquiry and 2012 Olympic Games (LOCOG), are now awaiting accession into the new system. At its core DRI provides its massive storage using a robot tape library. Although tapes provide highly resilient storage if treated and monitored carefully, they are not suited to frequent access. Therefore, the archive is designed to be a "dark archive". In other words, it is powered down until an access request is received.

doi:10.14337/XMLLondon13.Walpole01

Although there will be frequent demands for access to the data in the archive, many of these requests can be met by substitutes from a disk cache. For example, scanned documents can be substituted with a lower quality JPEG file from disk, instead of the original JPEG 2000 held on tape. Whenever data is retrieved it will be cached on disk for the next time so that frequently requested items are always promptly available.

1.3. The DRI Catalogue

The DRI Catalogue is perhaps best described as an inventory of the items held within the repository. It is distinct from the TNA Catalogue. The latter is a comprehensive accessioning, editorial management and public access system spanning both paper and digital documents.

As the tape archive cannot easily be searched, it is vital that rich metadata is readily available to tell archivists and other users what data is being held. Some of this metadata comes from the records' providers themselves, usually a government department. Some is generated as part of the archiving process while some is obtained by inspecting or transcribing the documents. With each collection of data stored, a comprehensive metadata document is built up. A copy of this metadata is placed in the archive and another copy is sent to Discovery [3], TNA's public access search portal, provided the record is open to the public.

Controlled release of material from the archive is of paramount importance. Although the majority of data in the archive is open to the public, some is not. This may be for reasons of national security, commercial interest or simply because it would breach someone's privacy. For example, a service and medical record is held for each member of the Home Guard. Service records are opened to the public when the soldier in question is known to be deceased. Medical records on the other hand are only released some time later, usually not until the record itself is 100 years old. Because some members of the Home Guard were very young when they served, it is possible they would still be alive today.

This crucial need to provide fine-grained control over record closure lies at the heart of the DRI Catalogue's requirements and has provided some of the key challenges during implementation, which will be discussed in more detail further on.

2. Requirements

Archiving records is only of value if those records can be accessed and viewed by researchers and members of the public. TNA's search portal for the archives is Discovery which holds over 20 million descriptions of records. Once a user has located a record of interest from searching Discovery they can request to see the original item. In cases where the record has been digitised a built-in image viewer allows the record to be viewed on-line. Until recently, the majority of records were on paper and painstaking work by the cataloguing team provided this metadata which was stored in the original electronic catalogue system (PROCAT) which has now been replaced by Discovery. In future the majority of records will come via DRI. DRI has two fundamental responsibilities with regard to Discovery which we can classify as closure and transfer which are explained in more detail below.

2.1. Closure

Until recently, most public records were closed for 30 years, however the government is now progressively reducing the closure period to 20 years. Some records, particularly those containing personal data, are closed for longer periods - up to 100 years. However the justifications for closing records for longer periods are scrutinised by a panel of academics and other experts [4].

Closure of records has two possible forms: the record itself can be closed but the description may be open or, alternatively, the record and the description may both be closed. In either situation it is very important that DRI does not release any closed records to the public.

Closure can apply at various levels. In one case a document may be open whereas in another only the metadata could be open. In some cases, even the metadata could be closed or possibly a whole collection of data, depending on the content and the reasons for closure.

2.2. Transfer

DRI transfers information on all of the records it holds to Discovery for public access. In the case of a closed record what the public sees depends on whether just the record, or the record and the description are closed. If the record is closed but there is a public description this will be shown, albeit with no possibility to see the actual record. In the case of a closed record and description they will be able to see that there is a record but not what the record is about. In other words, whilst providing as much public access to the records as possible, closed information must be filtered from the public view at all times.

2.3. Initial Approach

In order to establish the most effective way of dealing with the closure problem, three different approaches were prototyped simultaneously. These approaches were based on three fundamentally different models of the catalogue data. These models can be categorised as the *relational*, *hierarchical* and *graph* approach.

- *Relational* – this approach was to retain the existing relational database management system for storing catalogue data but to rewrite the SQL queries used to establish record closure status. On the face of it this would seem to be the most obvious and expedient solution.
- *Graph* – the second approach was to re-structure the catalogue as a graph using RDF and then query it using SPARQL. This was the least well understood approach of the three but its increasingly common usage in large scale applications suggested it was a viable solution.
- *Hierarchical* - the third approach was to use XML to describe the DRI catalogue, store the catalogue data in a native XML database and query the data using XQuery. The nature of the catalogue is strongly hierarchical and so this seemed a good solution.

Before any of these approaches could be tested extra data needed to be added the existing DRI catalogue. It was essential that items in the catalogue knew their ancestry, or at least the item that represented their parent. To achieve this a simple Scala program was written which connected to the database, built up a map of catalogue entries and their descendants and then added a parent reference column to the main catalogue table by looking up the entry from the map.

2.3.1. Results

Relational

Rewriting the SQL used to extract catalogue data led to a dramatic improvement in query response times. Whereas the original query took hours to complete, the new query using the extra parent column information completed in minutes. Optimising this query may well have further reduced the query response times.

Graph

Considerable effort had to be made to create an environment where the graph approach could be tested. These steps will briefly be described here and covered in more detail later on.

1. Create a mapping from the relational database column entries to triples using D2RQ [5]
2. Export the data from the relevant relational tables into triples using D2RQ.
3. Load these new triples into a triple-store (Jena TDB [6]) which could be accessed via a SPARQL endpoint (Jena Fuseki[7]).
4. Write SPARQL queries using SPARQL 1.1 property paths [8] to establish closure status.

Once all of this was done the results of this experiment were stunning. It was possible to establish the closure status of any catalogue item based on its ancestors and descendants in seconds or split-seconds. The performance far outstripped that of the relational database queries. It was also possible to write queries that showed the ancestors and descendants of any item and verify beyond doubt that the results were correct.

Hierarchical

Testing of the hierarchical approach was abandoned; the reasons for abandoning this approach were threefold:

1. It was felt that the graph approach offered a good solution to closure problem.
2. The graph tests had led to a better understanding of this approach and, with this understanding, a number of new and interesting possibilities had arisen in terms of what could be done with the catalogue. It was felt that the hierarchical approach did not offer these same possibilities.
3. Finally, and sadly, project deadlines and cost overheads meant that, although it would have been interesting to complete the hierarchical test, the fact that a good solution had been found obliged the project to move on.

2.3.2. Conclusion

The issue of closure had meant that it was necessary for the DRI project team to fundamentally question the nature of the DRI catalogue. Which of the available models best represented the catalogue? While relational tables may be very good at representing tabular data such as you find in a financial institution they were found to be less suited to describing the complex relationships within the catalogue.

Because TNA accessions data from the full range of government activities it is difficult to predict what structure this data will have and what information will need to go in the catalogue. The hierarchical model offers a good solution for documents and publications but falls down when attempting to describe the poly-hierarchical structures that you find in archives. For example a scanned document may contain data about many people. How do you nest this information in a hierarchy without repeating yourself many times over?

Fundamentally the archive holds information about people, their relationships and activities over the course of time. These things are complex and varied – they are after all the nature of the world around us. The conclusion of the experiment was that the graph approach not only solved the immediate problem of closure but also most closely modelled our complex world and would in the future provide a powerful tool for discovering information held within the archive.

3. Design

3.1. Technology

The catalogue trials had provided valuable experience in a number of technologies. This included tools for working with RDF such as D2RQ and Apache Jena plus experience of new standards-based formats and languages such as Turtle [9] and SPARQL. An important factor in the technology choices made was the preference for using open source and open standards specified by the UK Government in the Government Services Design Manual:

"..it remains the policy of the government that, where there is no significant overall cost difference between open and non-open source products that fulfil minimum and essential capabilities, open source will be selected on the basis of its inherent flexibility." [10]

And also for using open standards as stipulated by criteria 16:

"Use open standards and common Government platforms (e.g. Identity Assurance) where available" [11]

All of these technologies met this criteria as being either open source (D2RQ, Jena) or open standards (Turtle, SPARQL).

Furthermore the trials had given developers a head start with these particular tools and there was felt there was no particular benefit to be gained by switching to an alternative solution at this stage. Having said that, the use of open standards means that, should the existing open source technology cease to meet TNA's requirements, the overhead in moving to a new tool-set should be kept to a minimum.

Another significant reason for choosing the Apache Jena framework was the excellent Java API provided. DRI is predominantly a Java based system. Java was chosen originally because the underlying technology of DRI (Tessella's Safety Deposit Box – SDB [12]) was written in Java and therefore Java was the natural choice for extending SDB functionality. The DRI development team naturally had strong Java skills and Jena's API provided a straightforward way for developers familiar with Java to start working with RDF.

3.2. The Catalogue Services

The DRI catalogue is required to provide a number of key services:

Accessioning

Accessioning Firstly, it must accept new entries in the catalogue when data is initially accessioned into DRI. For each item accessioned it must provide a unique identifier which is persisted with the entry. In fact the identifiers generated must be globally unique identifiers [13].

Currently the catalogue recognises a number of item types. These are:

- *Collection* – this describes a large related group of documents, for example the Durham Home Guard records are represented by a single collection.
- *Batch* – this represents a batch of records on disk. This is how records are initially received by TNA and usually represents a substantial volume but it may or may not be a whole collection.
- *Deliverable Unit* – this represents a single item of information. It is a term coined from the paper archive world and represents something that can be handed to someone else. This may be a box or records, a folder or a single document. Similar criteria are used to for digital records.

- *Manifestation* – there are different types of manifestation. For example images have preservation and presentation manifestations. Preservation manifestations of these would be the highest quality images while presentation ones are a lower quality for display purposes.
- *File* – these are the actual digital files held within the archive.

Each of these types comes with a set of properties which must be retained in the catalogue, including things like TNA Catalogue references and closure information.

Closure Updates

The catalogue must provide the functionality to alter the closure status of a record (subject to review by an human operator).

Export

The catalogue must provide the functionality for exporting records. This is normally done in order to transfer the record to Discovery. The export process itself involves numerous steps in a controlled work-flow. The catalogue's responsibility is to allow a review of items for export and to maintain a record of the status of the export work-flow.

Lists

The catalogue must also provide the ability to persist lists of records. These are generic lists of records which may be used for a variety of purposes. Currently they are used to group records due for opening and export but there are likely to be other uses in the future.

From these requirements it can be gathered that the catalogue must provide read, write, update and delete access to the triple-store. The Apache Jena framework provides a straightforward approach to these requirements.

- *Reading* data can be done using the SPARQL Query Language [14].
- *Writing* data can be done by creating and persisting new triples
- *Updating* and *deleting* can be done using the SPARQL 1.1 Update Language [15].

There are a number of different ways of performing these tasks including:

1. Using the Jena command line utilities
2. Using the Jena API to work directly with the triple-store.
3. Using a SPARQL server such as Fuseki to perform these tasks

The SPARQL server provides an attractive solution which allows queries to be executed quickly and conveniently over HTTP as well as allowing new sets of triples to be posted to the triple-store. The Jena Fuseki SPARQL server also includes a built in version of the TDB triple-store. As TDB can only be accessed safely from within one Java Virtual Machine [16] it makes sense to use this built-in version with the SPARQL server approach. This server has a number of endpoints built in including a SPARQL query endpoint which can be accessed via a web browser. This not only provides a useful tool for developers but could in the future be exposed to staff within TNA, or even to the general public, who could then query the data for themselves given a little SPARQL knowledge.

Jena Fuseki with embedded TDB was chosen as the solution.

One hurdle to acceptance of this semantic technology stack within TNA however was the need to develop skills around semantic search and in particular in terms of learning RDF syntax and the SPARQL query language. One solution to this problem is the Linked Data API [17]. This API offers a way for SPARQL queries to be pre-configured and then accessed via RESTful URLs. For example you can configure a SPARQL query that locates a catalogue entry's descendants and then access this via a pre-set URL structure e.g.

```
http://{server-name}/{catalogue-identitifer}/descendant
```

Elda [18] is an open source implementation of the API written in Java. The SPARQL queries within Elda are configured using the Turtle format so in this case the configuration for this specific URL would look something like this:

```
spec:catalogueItemDescendant a apivc:ListEndpoint
 ; apivc:uriTemplate "/catalogue/{uuid}/descendant"
 ; apivc:variable [apivc:name "uuid";
                   apivc:type xsd:string]
 ; apivc:selector [
     apivc:where """
       ?catalogueItem dcterms:identifier ?uuid .
         {
           ?item dri:parent+ ?catalogueItem .
         }
     """
   ];
 .
```

Once this endpoint is registered (also within the configuration file) any requests that match this URI template will execute a SPARQL SELECT statement returning any matching catalogue items represented by the variable `?item`. The string value of the UUID passed in via the URI is allocated to the `?uuid` variable when the statement is executed.

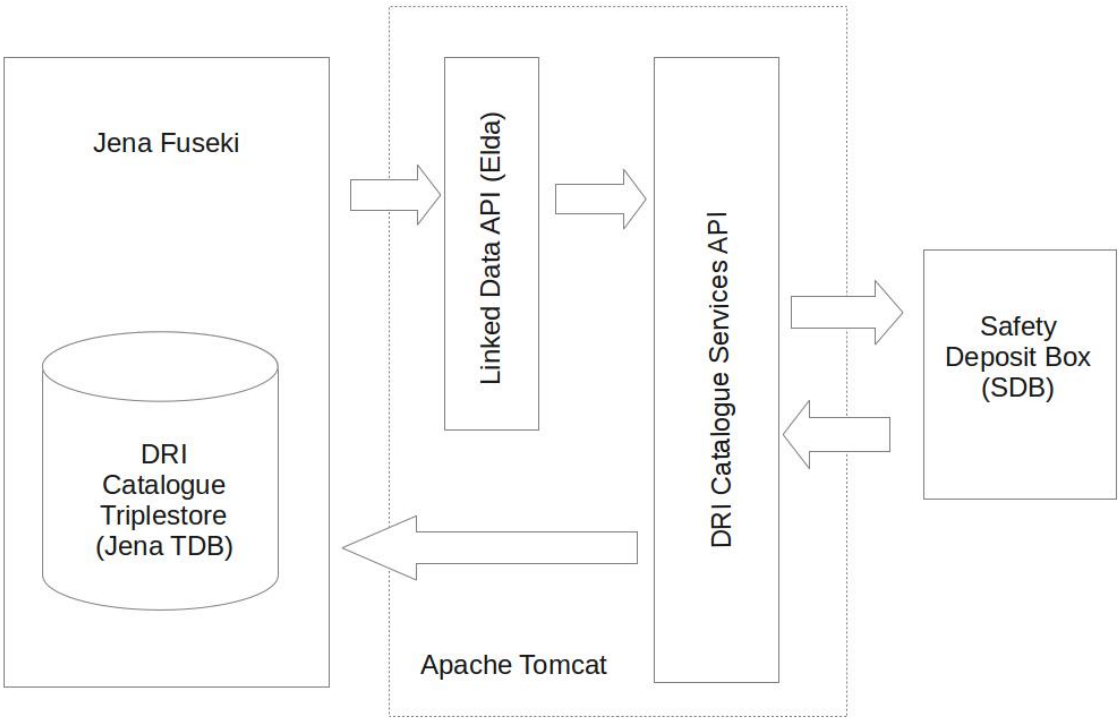
Within this statement you will notice the dri:parent+ property. The + signifies a SPARQL 1.1 property path, in other words it will find the parent and the parent's parent and so on until there are no more matches. The dri prefix indicates that this parent property is part of the DRI vocabulary which is discussed in more detail later.

Elda was chosen as the read API in front of Jena Fuseki. This meant that all reads would go via Elda and all writes, updates and deletes would go directly to Fuseki.

One final problem remained with regards to the acceptance of this solution within TNA. It was perceived that there was some risk involved in using such a (relatively) new technology stack and therefore the impact on other systems, in particular SDB, had to be kept to a minimum. To solve this problem it was decided to develop a simple bespoke Catalogue Service API between SDB and the DRI Catalogue. Having SDB talk to this API meant that if the new DRI Catalogue failed to deliver the expected results for some reason then it could be swapped for another solution with only the Catalogue Service API needing to be re-engineered.

Both Elda and the Catalogue Service API would be run within an Apache Tomcat web container in common with other components of DRI and SDB. Jena Fuseki however would need to be run as a standalone application as there is currently no mechanism for deploying Fuseki as a web archive within Tomcat, although it is a feature that has been discussed in depth [19].

The final design of the catalogue system is shown below.



3.3. DRI Vocabulary

During the initial testing of a graph based catalogue it was quickly discovered that a vocabulary for catalogue terms would need to be developed in order to be able to describe catalogue specific entities and relationships. This vocabulary would need to be established, as far as possible, in advance of the initial import of catalogue data. Although it would be possible to reclassify terms once the data was converted into RDF, it is more expedient to resolve these terms up front and, as far as possible, avoid renaming later.

In keeping with the W3C's guidelines laid out in the Cookbook for Open Government Linked Data[20] existing vocabularies are re-used as much as possible. Extensive use is therefore made of OWL [21], Dublin Core [22], RDF Schema [23] and XML Schema [24] however quite a few terms are very specific to the Catalogue. Although catalogue items have "parents" suggesting use of the FOAF [25] vocabulary it was decided that catalogue items are emphatically not people and the rules around people's parents (one mother and one father) do not apply in this case. Use of the FOAF vocabulary could therefore cause confusion at a later date. A DRI parent term was therefore created.

The full vocabulary is described in Appendix A, *The DRI Vocabulary*.

3.4. Implementation

The first stage of the implementation was to extract the existing catalogue data from the RDBMS where it was held. D2RQ was to be used for this as had been done for the initial trial. The difference now was that we had established a vocabulary the terms to be used in the new catalogue. With this in place it was possible to map the columns in the database to the terms that would be used in the RDF. This was done using the D2RQ mapping file, a sample of which is shown below.

```
# Table TNADRI.COLLECTION
map:collection a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "http://nationalarchives.gov.uk/dri/catalogue/collection/@@TNADRI.COLLECTION.UUID@@";
  d2rq:class dri:Collection;
  d2rq:classDefinitionLabel "TNADRI.COLLECTION";
  .
```

In this example a row from the TNADRI.COLLECTION table is mapped to an instance of the dri:Collection class and assigned a URI based on the UUID column of the table. In means that we end up with a resource described by the following RDF triple (in Turtle format).

```
<http://nationalarchives.gov.uk/dri/catalogue/collection/example1234>
  rdf:type dri:Collection .
```

In other words a resource of type collection.

Using this mapping technique the contents of the RDBMS catalogue were dumped into text files in the Turtle format, which totalled approximate 1Gb of data. Approximately 8 million triples.

The second stage was to load these triples into TDB using the *tdbloader* command line tool which is part of the Apache Jena framework. However this raw data was still not in the desired format for use within the new catalogue. For starters the closure information was not linked to the resources it referred to. Closure information is comprised of five elements:

- *Closure period* – how long the record must remain closed
- *Description status* – whether the description is open or closed
- *Document status* – whether the document is open or closed
- *Review Date* - when the closure is due for review or when the record was opened
- *Closure type* – gives more information about the type of closure

However closure cannot be said to be a resource in its own right as it only exists in the context of a catalogue item. RDF has a concept for this type of information which seemed highly appropriate, the *blank node* or *bNode*. Creating these blank nodes would require some transformation of the data however. While there is no equivalent of XML the transformation language XSLT for RDF the SPARQL language itself allows a transformation through use of CONSTRUCT queries. In this case new triples can be created based on existing triples.

By loading the data into TDB and then using the SPARQL query endpoint in Fuseki to run construct queries, it was possible to generate new triples in the desired format that could be downloaded in Turtle format from Fuseki and then reloaded into a new instance of TDB. The following CONSTRUCT query shows how the new triples could be created. In this case by combining file and closure information into a new set of triples relating to a single resource with the closure information held in a blank node signified by the square brackets in the construct query:

```
PREFIX closure:  <http://nationalarchives.gov.uk/dri/catalogue/closure#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX dri: <http://nationalarchives.gov.uk/terms/dri#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT
{
  ?file rdf:type dri:File ;
      rdfs:label ?name ;
      dri:directory ?directory ;
        dcterms:identifier ?identifier ;
      dri:closure [
      rdf:type dri:Closure ;
      dri:closurePeriod ?closureCode ;
      dri:descriptionStatus ?descriptionStatus ;
          dri:documentStatus ?documentStatus ;
      dri:reviewDate ?openingDate ;
          dri:closureType ?newClosureType
    ] ;
  .
}
WHERE
{
  ?file rdf:type dri:File ;
    rdfs:label ?name ;
    dri:directory ?directory ;
      dcterms:identifier ?identifier .
  ?closure rdfs:label ?identifier ;
    dcterms:creator ?creator ;
      dri:closureCode ?closureCode ;
      dri:closureType ?closureType ;
    dri:descriptionStatus ?descriptionStatus ;
      dri:documentStatus ?documentStatus ;
      dri:openingDate ?openingDate ;
    BIND(IF(?closureType = 1, closure:A, closure:U)
         AS ?newClosureType)
      .
}
```

With the data cleaned, refactored and accessible via Fuseki, development of the Catalogue Service API could begin.

3.5. Catalogue Service API

The Catalogue Service API is a RESTful Jersey JAX-RS [26] application that reads and writes data to the TDB triple-store. As per the design, reading data is done via Elda and writing data is done via Fuseki. The actual API itself is extremely simple and returns data in an XML or JSON format. For example, in the case of creating a new catalogue entry it simply takes in the TNA catalogue reference as a request parameter and generates an entry in the DRI catalogue of the appropriate type (this depends on the URI called) and, if successful, returns the unique identifier in a snippet of XML as follows:-

```
<result
  xmlns="http://nationalarchives.gov.uk/dri/catalogue">
  <uuid>e9d8f987-5d49-40f2-869b-a2172e3d362c</uuid>
</result>
```

In the process it generates a new unique ID, writes out the triple to file using the Jena API to then it to Fuseki over HTTP and, if all goes well, returns the UUID in the response as shown above.

To create the new triples it first creates an ontology model using the Jena API which is populated with the required classes from our vocabulary:

```
protected Model createModel() {

  OntModel model =
    ModelFactory.createOntologyModel(
      OntModelSpec.RDFS_MEM);

  collectionClass = model.createClass(
                    DRI_TERMS_URI + "Collection");

  batchClass = model.createClass(
                DRI_TERMS_URI + "Batch");

  deliverableUnitClass = model.createClass(
        DRI_TERMS_URI + "DeliverableUnit");

  preservationManifestationClass =
    model.createClass(
      DRI_TERMS_URI + "PreservationManifestation");

  exportClass = model.createClass(
    DRI_TERMS_URI + "Export");

  recordListClass = model.createClass(
    DRI_TERMS_URI + "RecordList");

  model.setNsPrefix("dri", DRI_TERMS_URI);
  model.setNsPrefix("dcterms", DCTerms.getURI());

  return model;
}
```

It then creates the necessary resources and literals which are added to the model.

```
private String addCollection(Model model,
                             String collectionRef,
                             String label) {
  UUID uuid = UUID.randomUUID();

  Resource collection = model.createResource(
                COLLECTION_URI + uuid.toString());
  collection.addLiteral(RDFS.label, collectionRef);
  collection.addProperty(RDF.type,collectionClass);
  collection.addLiteral(DCTerms.created,
          new XSDDateTime(Calendar.getInstance()));
  collection.addLiteral(DCTerms.identifier,
                        uuid.toString());
  collection.addLiteral(DCTerms.description,label);
  return uuid.toString();
}
```

The model is then written to a file in Turtle format which is posted to Fuseki via HTTP.

In the case of a SPARQL update it writes out a SPARQL file and posts this to Fuseki.

```
public ResultWrapper createRecordListAddItemFile(
  String recordListUuid,
  String itemUuid) {

  Model model = createModel();

  Resource recordList = model.createResource(
    RECORD_LIST_URI + recordListUuid);

  Literal itemUuidLiteral =
    model.createTypedLiteral(itemUuid);

  QuerySolutionMap parameters =
    new QuerySolutionMap();

  parameters.add("recordList", recordList);
  parameters.add("itemUuid", itemUuidLiteral);

  ParameterizedSparqlString paramString =
    new ParameterizedSparqlString(
      getQueryProlog() +
      getRecordListItemAddString(), parameters);

  UpdateRequest update = paramString.asUpdate();

  File queryFile = getFileHandler().writeUpdateFile(
    update, "insert" + "_" +
    getDtf().print(new DateTime()) + ".rq");

  ResultWrapper rw =
    new ResultWrapper(queryFile, null);

  return rw;
}
```

In the above example the getQueryProlog() and getRecordListItemAddString() methods generated the necessary text for the SPARQL update as follows:

```
protected String getQueryProlog() {
  String prologString =
    "PREFIX dcterms: <"+DCTerms.getURI() + "> \n" +
    "PREFIX dri: <" + DRI_TERMS_URI + "> \n" +
    "PREFIX rdf: <" + RDF.getURI() + "> \n" +
    "PREFIX rdfs: <" + RDFS.getURI() + "> \n" +
    "PREFIX owl: <" + OWL.getURI() + "> \n" +
    "PREFIX xsd: <" + XSD.getURI() + "> \n";
  return prologString;
}

private String getRecordListItemAddString() {
  StringBuilder deleteBuilder =new StringBuilder();
  deleteBuilder.append(
    "INSERT { ?recordList dri:recordListMember ?item . }");
  deleteBuilder.append(
    "WHERE { ?item dcterms:identifier ?itemUuid . }");
  return deleteBuilder.toString();
}
```

In the case of reading the data it accesses Elda, requesting XML format (for which a schema has been developed) and unmarshalls this into JAXB objects from which it extracts the required information. The results are then marshalled into the much simpler XML format described above.

3.6. Insights, Issues and Limitations

3.6.1. Elda

Whilst Elda and the Linked Data API provide enormous benefits for users in terms of simplifying access to triple-stores it has provided some challenges to the developers wanting to implement SPARQL queries and make use of the XML result format.

Elda extension

Early on in the development process it came to light that Elda had a significant limitation in the type of SPARQL queries it could run. Although Elda provides read access to the underlying triple-store it was found to be impossible to create new triples through the use of CONSTRUCT queries. There was an early requirement to know whether a record was open, closed or partially closed. This information is not held within the knowledge-base but has to be generated as the result of a query. Although you can embed SPARQL SELECT queries within Elda there was no working mechanism for a CONSTRUCT query. As Elda is open source and written in Java, it was feasible for TNA to add this functionality as an extension, which was subsequently done.

As the project has continued however, we have found this functionality to be of limited benefit. Although there is a temptation to use CONSTRUCT queries for all kinds of things, quite often there is a more efficient way to achieve the desired result, for example by configuring a viewer within the API to control the properties returned.

Furthermore it was found that complex construct queries that relied on a number of SELECT sub-queries became difficult to debug as there was limited visibility of what was happening within the query. This led to a rethink whereby more complex queries were built up gradually using the Jena API calling a combination of simpler sub-queries from Elda. This enabled us to embed logging within the Java which could show the results of sub-queries and this also gave us a structure for providing much better error handling within the application.

Whilst the Elda construct extension is still in use, it is likely to be gradually phased out in the future.

Elda caching

Elda is very efficient at caching query results. Whilst this reduces query times for frequently called queries it can cause some confusion and frustration when the data in the underlying triple-store is being frequently updated. The cache can however be cleared by calling...

`http://{server-name}/catalogue/control/clear-cache`

...and so the catalogue API calls this before any query which is likely to be affected by updates.

Elda XML

An early design decision was to use the XML results from Elda and then unmarshall the results as JAXB objects with the Catalogue Service from where the required values could be extracted and returned. This meant creating our own schema for the XML as there is no publicly available one and, in any case, the XML returned is dependent on the underlying vocabularies being used. Because we had created our own vocabulary we had no choice but to create our own schema.

An important point to note with Elda is that in the case of item endpoints (i.e. where only one result will ever be returned) a *primaryTopic* element is used to contain the result. In the case of a list endpoint, which can return zero to many results, an *items* element is returned containing one *item* element for each result. Understanding this enabled us to write generic services for reading and writing these two types of result.

Elda Turtle

It is proposed that in future the DRI Catalogue Service will make use of Turtle format results from Elda instead of XML. Turtle response times seem to be considerably faster than the equivalent XML (presumably because this is closer to the native format of the data) and Jena provides a well developed API for reading RDF, meaning that Plain Old Java Objects could be used within the Catalogue Service rather than specifically JAXB objects.

3.6.2. TDB

One point that was established early on in the design process was that TDB could only be safely accessed from one Java Virtual Machine. We had therefore chosen to use Fuseki. This presented us with a problem however when it came to performing backups. We wanted the backups to be controlled by a CRON job run on the server itself. How could the server safely carry out a backup if TDB was being controlled by Fueski?

The answer was provided by the Fuseki management console. A little documented but extremely useful feature of Fuseki which meant that the backup could be controlled safely via the management console by calling the relevant URL from the server shell.

```
wget -O - --post-data='cmd=backup&dataset=/
catalogue' http://{server-address}/mgt
```

3.6.3. Xturtle

Working with a semantic web technology stack means that you frequently have to manually edit RDF files in Turtle format. For example the mapping files for D2RQ and configuration files for Elda are written in Turtle. Without a syntax highlighter, errors in these files can be difficult to spot, especially as they get larger.

Xturtle [27], which comes as a plug-in for Eclipse, provided us with a very useful tool for editing these files. Especially as the majority of our developers were already using Eclipse as a Java IDE.

3.6.4. Scardf

Scala is being used increasingly within the DRI project. The reasons for this are its scalability, compatibility with Java (Scala programs compile to JVM byte-code) and concise syntax which results in less lines of code, better readability and fewer opportunities for errors.

For this reason, *scardf* [28] is being considered for future development of the Catalogue Service API within DRI. The *ScardfOnJena* API appears to offer a slot in replacement for the Java API currently being used. If the project were being started over again now this may well have been the preferred language for the Catalogue Service API, rather than Java.

3.6.5. Scale and performance

So far the DRI Catalogue has only been used for the modest amount of data currently held within DRI (the Durham Home Guard Records). Whilst no performance issues have been encountered so far it is likely that this will become more of a concern as data accumulates. Actual volumes are very difficult to predict and the nature of an archive is that it will always grow and never shrink. For this reason extensibility has been a primary concern in all aspects of DRI development and the catalogue is no different. Whilst we cannot predict future performance we are encouraged by a number of factors:

Firstly the graph solution was chosen because of its superior performance.

Secondly we are confident that the existing catalogue architecture can be scaled horizontally. This has already been done with other components. Additional servers could be added, running further instances of the triple-store, possibly containing completely separate graphs. Indeed there is a certain logic to one graph per catalogue collection. With a SPARQL endpoint configured on each collection it would be possible to have a "catalogue of catalogues" which would provide us with a pan-archive search capability.

Finally, if the existing open source framework fails to meet expectation the standards-based approach means that we can move to an alternative framework. For example, TNA has previously worked with Ontotext in the development of the UK Government Web Archive [29] which uses an OWLIM [30] triple-store containing billions of triples.

We are confident that whatever issues arise with scale in the future, we are in a strong position to address them.

4. The Future

So far what has been achieved with the DRI Catalogue is a fundamental remodelling of the catalogue using Semantic Web technologies and a successful implementation of a solution to the problem of closure. With these new technologies in place further enhancements of the catalogue can be considered which were not previously possible.

4.1. Named Entity Recognition

Using a semantic approach enables an Open World Assumption. This means that it is "implicitly assumed that a knowledge base may always be incomplete" [31]. It is always possible to add more information as we learn it. As we extract more information from the content going into the archive we can add it to our knowledge-base and this information can be transferred to Discovery where it can be viewed and searched.

What could this really mean for people viewing the record on the Web?

As a human being with a reasonable level of education we can look at the following entry and make a reasonable assumption that it refers to a soldier receiving an award for conduct during the Second World War.

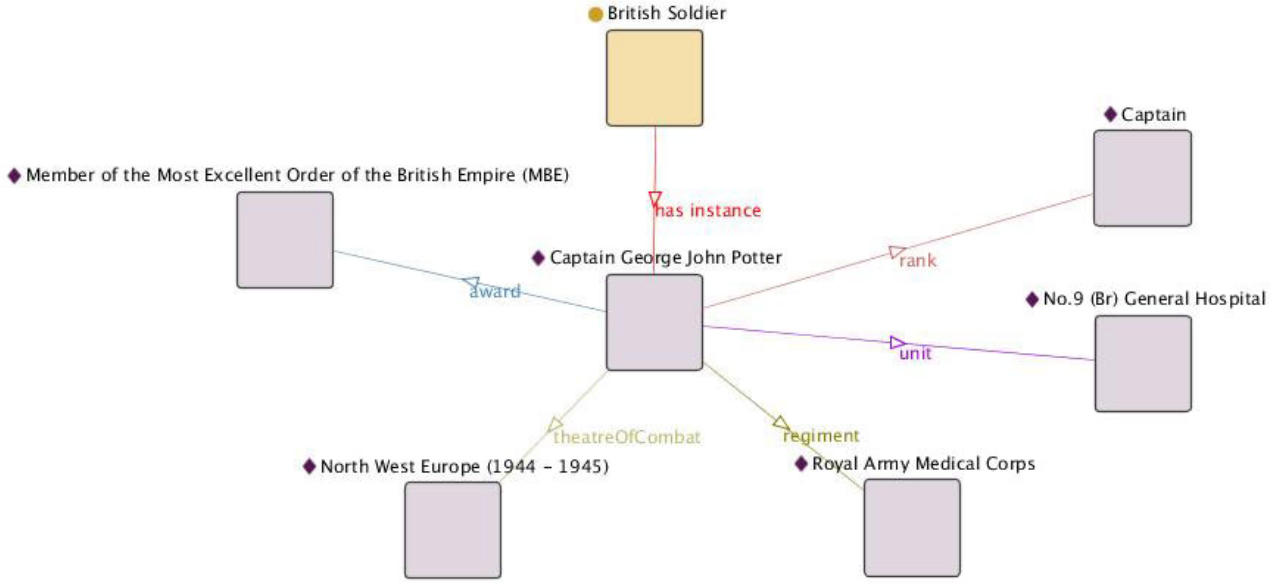
Reference: WO 373/83/774

Description:

Name	Potter, George John
Rank:	Captain
Service No:	171371
Regiment:	Royal Army Medical Corps
Theatre of Combat or Operation:	North West Europe 1944-45
Award:	Member of the Most Excellent Order of the British Empire
Date of announcement in London Gazette:	29 March 1945

It is easy to forget that the machine has no idea about these concepts. As far as the computer is concerned this is just a collection of string values. If we want the computer to help us search for information we need to tell it about the concepts that we already understand. We need to tell it that George John Potter is a Captain which is a kind of officer, which is a kind of soldier, which is a kind of person who serves in the army. We can tell the computer that he served in a "regiment" which is part of an "army" and the regiment is called the "Royal Army Medical Corps". We can also tell it that we know this to be a regiment within the "British Army". If we identify the Royal Army Medical Corps as a single entity we can then say that other people also served in the Royal Army Medical Corps and point at the same resource. With the addition of these simple pieces of information suddenly the computer can help us enormously in our search. The "Royal Army Medical Corps" is no longer just a piece of text but a concept that the computer can relate to other entities it also knows about. It can now tell us whatever it knows about the Royal Army Medical Corps. For example who else served in the Royal Army Medical Corps? What theatres of combat or operations did it take part in and when?

The follow is a machine readable representation of Captain George John Potter's record previously discussed. The nodes in the graph represent either resources or literal values and the lines connecting the nodes are the properties.



4.2. Ontology-driven NLP

Let's say that we build up a dictionary (or ontology) of these terms. As the documents are loaded into the archive they could be scanned for these terms using a Natural Language Processing tool such as GATE [32].

Let's take a real example from Discovery. The following is the text of the recommendation for award for the aforementioned George John Potter:

"*From 1 Aug 44 to 20 Oct 44 Bayeux*, *Rouen* and *Antwerp*. During the period from 1 Aug to date this *officer* has carried the principal strain of establishing and re-establishing the *hospital* in three situations. His unrelenting energy, skill and patience have been the mainstay of the unit. His work as a *quartermaster* is the most outstanding I have met in my service. (*A.R.ORAM*) *Colonel* Comdg. *No.9 (Br) General Hospital*"

I have underlined the terms that could reasonably be understood by the computer with the help of a dictionary. We have place names, a date range, a keyword "hospital" a rank (Colonel) the name of a unit (No.9 British General Hospital) and the name of a person (A.R.Oram). The computer could reasonably be expected to recognise A.R.Oram as he is himself the subject of a record (he also received an award). Although the computer would know him as Algar Roy Oram, it would be computationally simple to add another name value with this common abbreviation. Likewise "Br" could be recognised as an abbreviation for British.

As a result of this process the computer could perform *reification*. In other words it could make statements that may or may not be true but which are plausible conclusions. For example, it could say "Colonel Algar Roy Oram served with Captain George John Potter" or "Colonel Algar Roy Oram was based in Antwerp". This is inferred knowledge and may not be factual but it could be marked as a theory in the knowledge-base, until such a time as it can be shown to be (or not to be) a fact.

4.3. Semantic Search

Keyword searches tend to deliver a large volume of irrelevant results because it is not usually possible to communicate the desired context to the search engine mechanism. Semantic searching on the other hand allows for selection of ontological terms when searching. For example if you search for "George John Potter" in Discovery you get 361 results whereas in fact, there is only one result that actually refers to a person with this name. Imagine if you were able to say that you were looking for a match on person's name. To paraphrase Bill Gates, *context is king* when doing Semantic Search. This technique is known as *query string extension*.

Semantic search offers other possibilities as well. If it knows you are searching for a person named George John Potter it would be possible for a semantic knowledge base to look for terms that were closely associated to matches. For example, George Potter's award was given by a Colonel A.R. Oram. This information could be held within the graph of knowledge and therefore a semantic search could bring back results linking to Colonel Algar Roy Oram as well. Through painstaking research it is possible to discover this information now and this reveals that Algar Oram's records describe in great detail the difficult and dangerous situation that he shared with George Potter in Antwerp during the Second World War. In this way we can discover new information about George Potter's experiences during the war that are not directly expressed but strongly implied. A semantic search could've provided this information in seconds. This technique is referred to as *cross-referencing*.

Because the knowledge base is a graph, it is possible to go further into searches than traditional keyword searching. For example in the case of George Potter you could continue the search using the link to Algar Oram. In this case you would find that he too was awarded a medal and that this medal was awarded by a very senior officer in the Royal Army Medical Corps who had himself been decorated many times. Some of the medals this individual received were for extraordinary acts of bravery that saved many lives and the letters of recommendation make gripping reading. This may not be what you were looking for originally but it provides interesting context that you would not otherwise have found. This is known as *exploratory search*.

Another possibility would be to allow machine reasoning within the catalogue. This would enable rules to be applied. For example if the machine knew that George Potter “served” with something called “No.9 (British) General Hospital” in the “theatre of combat” called “North West Europe (1944-1945)” it would be possible to reason that certain other people “served with” George Potter. This is new knowledge that is not currently available in the catalogue. It is an example of the using the graph and the ontology to do *reasoning*.

4.4. Linked Data (for the things the archive doesn't have)

Whilst TNA is a huge national source of information, there is lots of pieces of official information that are simply not there, but kept elsewhere. For example birth, marriage and death certificates from 1837 onwards are held by the General Register Office or at local register offices - prior to 1837 they are kept in parish registers; military service records for soldiers and officers after 1920 are maintained by the Ministry of Defence; the London Gazette [33] is the official UK Government Newspaper and has it's own knowledge-base of historical articles. When it comes to places, an organisation such as the Ordnance Survey would be a excellent source of information. As long as an archive is kept in a silo, its world view will remain incomplete.

Linked Data [34], the brainchild of World Wide Web inventor Tim Berners-Lee, provides a way of un-siloing data and it is based on the standardised technologies of the Semantic Web. By providing its information as Linked Data, i.e. in RDF-based machine readable formats, other organisations or individuals can connect the dots between items of data. In the case of George Potter we know from his record that he set-up field hospitals in Bayeux, Rouen and Antwerp. Whilst the archive may not be an authority on these places, someone could make a connection between a record held in the archive and a resource that gives much more information about these places. For example, DBPedia [35], which contains key information taken from Wikipedia provides a great deal of useful and increasingly reliable data. This could work both ways so not only could external users make these connection but the archive itself it could pull in information from other places that provide Linked Data. For example at TNA, UK location information could be pulled in from the Ordnance Survey, allowing relevant maps and other information to be shown within Discovery. At the same time the archive could contribute knowledge back to DBPedia, adding new entries and improving the quality of existing entries on which it is an authority.

4.5. Crowd-sourced linking

TNA has a vast amount of information and it is unrealistic to think that a machine is going to be able to read everything and make all the right connections. It is going to make mistakes and it is going to miss things. This is particularly the case with digitised documents, i.e. scanned paper documents. Even with modern OCR technology it is not possible to accurately read all these items. Providing a way for researchers, archivists and the general public to make connections would add enormous value to the knowledge base. Allowing users to add tags to a document is a common way to crowd-source information but the terms tend to be very specific and individualistic. Imagine if it was possible to select dictionary terms from a semantic knowledge-base to apply to these items, many new and previous overlooked connections could be made.

I am grateful to a host of people whose knowledge and expertise have made this work possible. In particular to Bob DuCharme for his excellent book on SPARQL (now worn out), Dr Harald Sack at the Hasso Plattner Institute for his thorough course on Semantic Web Technologies, the guys at Epimorphics who have given us invaluable support and of course, above all, the folks at The National Archives whose foward thinking has enabled this project, in particular David Thomas, Tim Gollins, Diana Newton, Peter Malewski and Adam Retter.

Bibliography

[1] The National Archives. *Our new catalogue: the Discovery service.* *http://www.nationalarchives.gov.uk/about/new-catalogue.htm*

[2] BBC. *Domesday Reloaded* *http://www.bbc.co.uk/history/domesday/story*. Copyright © 2013 BBC.

[3] The National Archives. *Discovery* *http://discovery.nationalarchives.gov.uk*.

[4] The National Archives. *Freedom of Information Act 2000* *http://www.legislation.gov.uk/ukpga/2000/36/contents*.

[5] Free University of Berlin. *D2RQ* *http://d2rq.org/*.

[6] Apache Software Foundation. *Apache Jena TDB* *http://jena.apache.org/documentation/tdb/*. Copyright © 2011-2013 Apache Software Foundation.

[7] Apache Software Foundation. *Apache Jena Fuseki* *http://jena.apache.org/documentation/serving_data/*. Copyright © 2011-2013 Apache Software Foundation.

[8] W3C. *SPARQL 1.1 Property Paths* *http://www.w3.org/TR/sparql11-property-paths/*. Copyright © 2010 W3C.

[9] W3C. *Terse RDF Triple Language* *http://www.w3.org/TR/turtle/*. Copyright © 2013 W3C.

[10] GOV.UK. *Choosing technology* *https://www.gov.uk/service-manual/making-software/choosing-technology.html#level-playing-field*.

[11] GOV.UK. *Digital by Default Service Standard* *https://www.gov.uk/service-manual/digital-by-default*.

[12] Tessella. *Safety Deposit Box* *http://www.digital-preservation.com/solution/safety-deposit-box/*. Copyright © 2013 Tessella.

[13] Oracle. *UUID* *http://docs.oracle.com/javase/6/docs/api/java/util/UUID.html*. Copyright © 1993, 2011 Oracle and/or its affiliates.

[14] W3C. *SPARQL 1.1 Query Language* *http://www.w3.org/TR/sparql11-query/*. Copyright © 2013 W3C.

[15] W3C. *SPARQL 1.1 Update* *http://www.w3.org/TR/sparql11-update/*. Copyright © 2013 W3C.

[16] The Apache Software Foundation. *TDB Transactions.* *http://jena.apache.org/documentation/tdb/tdb_transactions.html*. Copyright © 2011-2013 The Apache Software Foundation.

[17] *Linked Data API* *http://code.google.com/p/linked-data-api/*.

[18] Epimorphics. *Elda Linked Data API Implementation* *http://code.google.com/p/elda/*.

[19] Apache Software Foundation. *Deliver Fuseki as a WAR file* *https://issues.apache.org/jira/browse/JENA-201*.

[20] W3C. *Linked Data Cookbook* http://www.w3.org/2011/gld/wiki/Linked_Data_Cookbook#Step_3_Re-use_Vocabularies_Whenever_Possible.

[21] W3C. *OWL Web Ontology Language Overview* http://www.w3.org/TR/owl-features/. Copyright © 2004 W3C.

[22] Dublin Core Metadata Initiative. *DCMI Metadata Terms* http://dublincore.org/documents/dcmi-terms/. Copyright © 1995 - 2013 DCMI.

[23] W3C. *RDF Vocabulary Description Language 1.0: RDF Schema* http://www.w3.org/TR/rdf-schema/. Copyright © 2004 W3C.

[24] W3C. *XML Schema* http://www.w3.org/XML/Schema. Copyright © 2000 - 2007 W3C.

[25] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.98* http://xmlns.com/foaf/spec/. Copyright © 2000 - 2010 Dan Brickley and Libby Miller.

[26] java.net. *Jersey JAX-RS (JSR 311) Reference Implementation* https://jersey.java.net/. Copyright © 2013.

[27] Agile Knowledge Engineering and Semantic Web (AKSW). *Xturtle: an eclipse / Xtext2 based editor for RDF/ Turtle files* http://aksw.org/Projects/Xturtle.html.

[28] *scardf Scala RDF API* http://code.google.com/p/scardf/.

[29] The National Archives. *UK Government Web Archive* http://www.nationalarchives.gov.uk/webarchive/.

[30] Ontotext. *Ontotext OWLIM* http://www.ontotext.com/owlim. Copyright © 2000-2013 Ontotext.

[31] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. Copyright © 2010 Taylor & Francis Group, LLC. >978-1-4200-9050-5. Chapman & Hall/CRC. *Foundations of Semantic Web Technologies*.

[32] The University of Sheffield. *GATE General Architecture for Text Engineering* http://gate.ac.uk/. Copyright © 1995-2011 The University of Sheffield.

[33] The London Gazette. *The London Gazette* http://www.london-gazette.co.uk/.

[34] *Linked Data - Connect Distributed Data across the Web* http://linkeddata.org/.

[35] DBPedia. *DBPedia* http://dbpedia.org/.

A. The DRI Vocabulary

This vocabulary has been laid out using the Turtle W3C format.

```
<http://nationalarchives.gov.uk/terms/dri>
      rdf:type owl:Ontology ;
      owl:imports <http://purl.org/dc/elements/1.1/> .

:Batch
      rdf:type rdfs:Class ;
      rdfs:label "Batch"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:Closure
      rdf:type rdfs:Class ;
      rdfs:label "Closure"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:ClosureType
      rdf:type rdfs:Class ;
      rdfs:label "Closure type"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:Collection
      rdf:type rdfs:Class ;
      rdfs:label "Collection"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:DeliverableUnit
      rdf:type rdfs:Class ;
      rdfs:label "Deliverable unit"^^xsd:string ;
      rdfs:subClassOf :Item .

:Directory
      rdf:type rdfs:Class ;
      rdfs:label "Directory"^^xsd:string ;
      rdfs:subClassOf :Resource .

:Export
      rdf:type rdfs:Class ;
      rdfs:label "Export"^^xsd:string ;
      rdfs:subClassOf rdfs:Container .

:ExportStatus
      rdf:type rdfs:Class ;
      rdfs:label "Export status"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:File
      rdf:type rdfs:Class ;
      rdfs:label "File"^^xsd:string ;
      rdfs:subClassOf :Resource .

:Item
      rdf:type rdfs:Class ;
      rdfs:label "Item"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:Manifestation
      rdf:type rdfs:Class ;
```

```
      rdfs:label "Manifestation"^^xsd:string ;
      rdfs:subClassOf :Item .

:PresentationManifestation
      rdf:type rdfs:Class ;
      rdfs:label "Presentation manifestation"^^xsd:string ;
      rdfs:subClassOf :Manifestation .

:PreservationManifestation
      rdf:type rdfs:Class ;
      rdfs:label "Preservation manifestation"^^xsd:string ;
      rdfs:subClassOf :Manifestation .

:RecordList
      rdf:type rdfs:Class ;
      rdfs:label "Record list"^^xsd:string ;
      rdfs:subClassOf rdfs:Container .

:Resource
      rdf:type rdfs:Class ;
      rdfs:label "Resource"^^xsd:string ;
      rdfs:subClassOf rdfs:Class .

:batch
      rdf:type rdf:Property ;
      rdfs:domain :Item ;
      rdfs:label "batch"^^xsd:string ;
      rdfs:range :Batch .

:closure
      rdf:type rdf:Property ;
      rdfs:domain :DeliverableUnit ;
      rdfs:label "closure"^^xsd:string ;
      rdfs:range :Closure .

:closurePeriod
      rdf:type rdf:Property ;
      rdfs:domain :Closure ;
      rdfs:label "closure period"^^xsd:string ;
      rdfs:range xsd:decimal .

:closureType
      rdf:type rdf:Property ;
      rdfs:domain :Closure ;
      rdfs:label "closure type"^^xsd:string ;
      rdfs:range :ClosureType .

:collection
      rdf:type rdf:Property ;
      rdfs:domain :Item ;
      rdfs:label "collection"^^xsd:string ;
      rdfs:range :Collection .

:completedDate
      rdf:type rdf:Property ;
      rdfs:label "completed date"^^xsd:string ;
      rdfs:range xsd:date ;
      rdfs:subPropertyOf dcterms:date .

:descriptionStatus
      rdf:type rdf:Property ;
      rdfs:domain :Closure ;
```

```
      rdfs:label "description status"^^xsd:string ;
      rdfs:range xsd:decimal .

:directory
      rdf:type rdf:Property ;
      rdfs:domain :Resource ;
      rdfs:label "directory"^^xsd:string ;
      rdfs:range xsd:string .

:documentStatus
      rdf:type rdf:Property ;
      rdfs:label "document status"^^xsd:string ;

:exportMember
      rdf:type rdf:Property ;
      rdfs:label "export member"^^xsd:string ;
      rdfs:subPropertyOf rdfs:member .

:exportStatus
      rdf:type rdf:Property ;
      rdfs:label "export status"^^xsd:string ;
      rdfs:range :ExportStatus .

:file
      rdf:type rdf:Property ;
      rdfs:domain :Manifestation ;
      rdfs:label "file"^^xsd:string ;
      rdfs:range :File .

:parent
      rdf:type rdf:Property ;
      rdfs:domain :Item ;
      rdfs:label "parent"^^xsd:string ;
      rdfs:range :Item .

:recordListMember
      rdf:type rdf:Property ;
      rdfs:label "record list member"^^xsd:string ;
      rdfs:subPropertyOf rdfs:member .

:reviewDate
      rdf:type rdf:Property ;
      rdfs:domain :Closure ;
      rdfs:label "review date"^^xsd:string ;
      rdfs:range xsd:dateTime ;
      rdfs:subPropertyOf dcterms:date .

:username
      rdf:type rdf:Property ;
      rdfs:label "username"^^xsd:string ;
      rdfs:range xsd:string
```

From trees to graphs: creating Linked Data from XML

Catherine Dolbear

Oxford University Press

<cathy.dolbear@oup.com>

Shaun McDonald

Oxford University Press

<shaun.mcdonald@oup.com>

Abstract

This paper describes the use case at Oxford University Press of migrating XML content to Linked Data, the business drivers we have identified so far and some of the issues that have arisen around modelling RDF from an XML base. We also discuss the advantages and limitations of schema.org markup relative to our much richer XML metadata, and describe our experimental system architecture combining stores for both XML documents and RDF triples.

1. Introduction: What We Need

Oxford University Press publishes a wide range of academic content in its online products, such as Oxford Scholarship Online, Oxford Art Online, the Dictionary of National Biography, Oxford Medicine Online, Oxford Law Online, Oxford Scholarly Editions Online and Oxford Islamic Studies Online to name just a few. The content includes journals, chaptered books, reference content, legal case studies and original literary texts (ranging from Shakespeare to the Bible and translations of the Qur'an) across many different disciplines.

Since our products are largely subscription based, there are varying levels of freely discoverable content on our product sites, so our "Discoverability Programme" has been working to release metadata and content to a wide array of users and to search engines. A major way of doing this is the Oxford Index site, a publicly available website described as our "Discoverability Gateway". Each document (book chapter, journal article, original text or entry in a reference work) in each of our online products is given its own "index card" page on the Oxford Index. We describe the Oxford Index to new users as a "digital card catalogue", to use a library metaphor. This is stored "under the hood" as an XML file of all the document's metadata. We have already found that the exposure of titles, authors, abstracts or short snippets of text from the online document to the "open" web has increased the volume of our content that's indexed by Google and others.

doi:10.14337/XMLLondon13.Dolbear01

As well as Search Engine Optimisation (SEO), another major business driver is the need to improve user journeys within and between our various online products. For example, we want to be able to suggest to a user looking at a reference entry on medical ethics from the Oxford Textbook of Medicine that they might also be interested in a recent journal article on the same subject; or to suggest to someone reading one of John Donne's poems on Oxford Scholarly Editions Online that they might be interested in a monograph chapter also about John Donne. While some of these links may be generated dynamically (automatically generated recommendations of the sort: "people who read what you're reading also looked at this other page") or driven by an underlying search query based on metadata elements, other links can be more definitively stated (such as citations, relationships between people, authorship etc.) and may have already been manually edited. We call this latter category "static" links and have been storing these links as independent XML documents in our metadata database which we call the "metadata hub". We have also grouped index cards about the same topic or person together into what we call an "Overview", providing another way of navigating around a topic. Index cards are linked to their overviews using the "hasPrimaryTopic" link. As the number of links have increased, it has become more apparent that a graph model would be better suited to storing the links, and so our thoughts have turned to RDF.

2. Metadata Hub and OxMetaML: Where We're At

Developing a single XML schema for all this metadata was no easy task, because the entire body of content as data originates from many different sources, including Mark Logic databases, FileMaker databases, SQL Server stores and even spreadsheets, using a number of different DTDs or none at all, and each having been developed to its particular product lifecycle by its particular product team.

We analysed each data set identifying gaps and gathering product metadata where available. Initial findings across larger sets helped identify a core set of bibliographic fields to serve as a requisite benchmark. Further findings uncovered where data was unavailable, lacked standardization or sufficient workflows to facilitate an additional "metadata" delivery stream. Change requests were filtered into two categories: those with little to no adverse impact to the product's business, and those requiring more significant changes. All non-impactful changes were implemented – adding identifiers, streamlining repositories, and authoring scripts to automate upstream extraction. The analysis results were updated, the core set of fields were extended, and a draft schema model was created.

At the time, we had planned to implement the existing Dublin Core model. There were a few factors serving as major constraints. Oxford University Press uses third party developers to build and maintain its product websites. There are advantages to such a business model, however, we realized we would no longer have complete control over data changes. Further, cost efficiencies gained by code reuse when new sites are launched are only realized if changes to existing data models are kept to a minimum. Finally, when new models are introduced, in such conditions, they tend to conflate with constructs solely intended to support site functionality. Thus, while RDF/XML was the intent, business needs for site development required a model much more XML intensive. This coupled with the inherent challenge of filtering variant data from myriad product silos into one, meant that our current model uses an internal abstraction layer comprised of OxMetaML - an XML schema suite of Dublin Core nested within proprietary constructs, nested within simple RDF constructs, stored in a file store, the Metadata Hub.

The Metadata Hub (Hub) is our XML file store loaded via a Pre-Ingestion Layer (PIL), where a collection of automated, proprietary pipelines transform the various source data into OxMetaML metadata records. One of the advantages of OxMetaML is that it establishes a single vocabulary for all links that had previously existed in disparate silos. Each link is defined as an XML-serialized triple with a named predicate. Because many of the links are embedded in the record from which they link, each link is encoded as an RDF resource with no specific identifier. That is, as a blank node, which makes further linking or querying difficult from an RDF perspective. It is, however, efficient as XML, and for post XML processing.

One of the post processing use cases calls for dynamically creating static links. To do this, we index the data with a full text search server, based on a Java search engine library extended with XML, among other things. It is a powerful search engine, allowing us to pinpoint potential relationships between the subject of a query (a single piece of content) and the entire corpus. It is also normally quite fast. There are trade offs, however, when dealing with XML.

The search engine uses an inverted, terms based index, where each term points to a list of documents that contain it. For text documents, even for HTML where markup is less strict, it is potent. The repetitive nature of bibliographic metadata in highly structured XML, however, inherently increases the time it takes to complete searches, especially when documents number in the millions. An author name like Shakespeare could appear in hundreds of thousands of documents. Therefore, great care must go into the search engine server's configuration.

One aspect controlled by configuration is segmenting, where the index is segmented into distinct disk locations for greater efficiency. Segment sizes are controllable via the configuration file. From a file system standpoint, this is highly efficient. Basically, writing a 5 Gig file (or a collection of files amounting to it) to a single disk space makes quick access nearly impossible. With highly structured, repetitve XML, it becomes problematic as it essentially requires multi-value fields.

"Multi-value field" is the term used for XML elements that repeat per document. For instance, and not insignificantly, keywords or index terms, which are very relevant to a document's relational position within a corpus, would be captured as multi-value fields. If the use case requires that these fields be stored as well, the search will only return the last value indexed for each one. In order to retrieve all relevant values from a document, all values for a multi-value field must be concatenated into a single, preferably separate, field, and the engine must know to return that field. Further, consideration must be given to which fields can be queried, and which returned between the multi-value field and its corresponding concatenated field.

The increase in query time is negligible by comparison to configuration and load, which itself may contribute to increased query time. Neither the schema or configuration file, the heart of the index, can be modified on the fly. There's no tweaking the index as each configuration change requires reindexing all the documents. In addition, while it is possible to use near realtime search (NRT), it can only be accomplished via what are known as soft commits, where index changes (updates) are only made visible to search, as opposed to hard commits that ensure the changes have been saved properly. Obviously, NRT is accomplished via a combination of soft and hard commits. However, if your use case requires that each query be followed up by document retrieval in a matter of seconds, you are left with the choice of either forgoing NRT, or performing some impressive, architectural alchemy.

3. An RDF Graph Model: Where We're Going

Although our current metadata is stored in the XML filestore serialised as RDF/XML, we still have some way to go to represent the knowledge that it contains as an RDF graph. This has come about for two reasons. Firstly, because of the very different mindsets required for XML, thinking in terms of documents and elements, trees and sequential order; versus RDF where our building blocks are vertices and arcs. Where documents are important, XML is a far better choice; where relationships between things are important, RDF has the advantage. One common tip is "XML for content, RDF for metadata" and this is where we are now heading. Since our metadata still includes abstract texts, which can run to several paragraphs, we do not expect to completely ditch the XML for RDF triples, but as more of the concepts encoded in the XML metadata, such as authors, become linked in their own right, and more importantly, are given their own identifiers, potentially through initiatives such as the author identification scheme ORCID , we expect to migrate more information to the graph model.

The second reason we are still at the development stage for our RDF model is because we really have two different types of metadata, and are working towards demarcating them more clearly. The first type is that better known in the publishing world - bibliographical information about the author, title, ISBN of the book or journal to which the document belongs and other such related metadata. In this case, since our primary output for the bibliographical information is in HTML form on the Oxford Index website, we have chosen to retain the data as XML, which is easily parsed into HTML, and only store the static links between documents, such as "references" links, as RDF.

The second type of metadata is information concerning what the document is about, often called contextual or semantic metadata. For example, we currently tag each document with the academic subject to which it belongs, and some documents, the Oxford Index Overviews, may also be tagged with information about whether the Overview is about a Person, Event or other type of entity. There is significant scope for expansion here, to provide more detail about the content of the document. Semantic publishing workflows usually do this by linking documents to an ontology, stored as RDF or OWL-DL, which contains background knowledge about the domain. For example, in the domain of American Revolutionary history, we could store the facts: John Adams and Samuel Adams were

cousins, contemporaries of George Washington. John Adams succeeded George Washington as President, and was the father of President John Quincy Adams. Some of this information is available as Linked Data in DBpedia, and we could augment it with additional information as we expand our ontology. Therefore, if we identify documents as being about John Adams, George Washington and John Quincy Adams, say, based on named entity recognition techniques we can then use the ontology links to generate links between the documents directly. Additionally, the ontology provides information about the link type "father of", "successor of" etc. that can help the user know how the two documents are related to each other, thus further improving the user journey.

We are storing the document-to-document links that are considered part of the bibliographic metadata as RDF triples. However, we also need to record information about the links themselves, for example, whether they have been approved by a curator, the accuracy of the link if it has been automatically generated, the date of creation etc. There are several ways of storing a fourth piece of information in RDF. The recommended option for Linked Data is to use named graphs, or "quads". This assigns a name (URI) to each separate graph (or group of triples), and hence allows additional information to be attached to that graph URI. Although this is supported in SPARQL, the RDF query language, unless we were to put each triple in a separate graph, this approach would not fulfil our needs, since we need to assign descriptive information to each triple, not just each group of triples.

The alternative option is to assign a URI to each triple and treat it as a resource in itself, so that further statements can be made about the triple. This is known as reification. For example the triple about Barack Obama:

```
<http://metadata.oup.com/10.1093/oi/authority.20110803100243337>
  <http://metadata.oup.com/has_occupation>
    "President of the United States"
```

could be reified to:

```
<http://metadata.oup.com/Statements/12345>
  rdf:subject
    <http://metadata.oup.com/10.1093/oi/authority.20110803100243337>.

<http://metadata.oup.com/Statements/12345>
  rdf:predicate
    <http://metadata.oup.com/has_occupation>.

<http://metadata.oup.com/Statements/12345>
  rdf:object
    "President of the United States".
```

Then additional triples can be added about the Statement:

```
<http://metadata.oup.com/Statements/12345>
  oup:is_valid_from
    "20 January 2009".
```

This approach is usually not recommended for linked data because it increases the number of triples, and requires a large number of query joins in order to return the RDF statements' metadata. However, because we are not directly publishing our document links, but simply storing them for insertion into XML which is later transformed into HTML, reification is still an option for us.

We have adopted a modified form of this reification model, whereby each type of link is considered an RDFS class, and we create instances of that class which are objects of the oup:has_link predicate, and subjects of the oup:has_target predicate:

```
<http://metadata.oup.com/10.1093/oi/authority.20110803100243337>
  oup:has_link
    <http://metadata.oup.com/links/12345>.

<http://metadata.oup.com/links/12345>
  rdf:type
    <http://metadata.oup.com/isPrimaryTopicOfLink>.

<http://metadata.oup.com/links/12345>
  oup:has_target
    <http://metadata.oup.com/10.1093/acref/9780195167795.013.0922>.

<http://metadata.oup.com/links/12345>
  oup:matchStatus
    "approved".
```

Much has been said about the drawbacks of the RDF/XML syntax for serialising RDF. Unlike XML, RDF is fundamentally defined as a data model, not a syntax. Although there are best practices for using RDF/XML [Dodds2012], in our opinion, it is better to use one of the alternative serialisations of RDF, such as Turtle [W3C2013] , to encode triples, and steer clear of RDF/XML completely. Not only is RDF/XML verbose, and permissive of syntactic variation to describe the same knowledge, it actually makes it harder on XML experts to focus on modelling issues. The questions that need to be answered in the design of an RDF Schema are what concepts are we trying to encode, and what types of links should we have between them? Issues of element order, and whether to use rdf:about or rdf:Description are orthogonal to knowledge modelling and don't need to be addressed if the Turtle syntax is used.

Initially, we are encoding the links between our XML documents as triples, using a number of common vocabularies such as Friend of a Friend/ and continuing to use the Dublin Core terms we already had in the XML. We have also explored the use of bibo and PRISM bibliographic vocabularies. We are testing whether to store inverse triples explicitly, or to rely on inference or SPARQL CONSTRUCT queries to generate the inverse at the output, as we deliver data to our front end websites. Storage of links in both "directions" for example, statements that a chapter "is part of" a book, and the book "has part" that chapter is obviously more costly in terms of storage, but improves delivery performance. Since we have to recombine the triples back into the XML document for presentation on the front end, generating the inverse triple using an inference engine is unlikely to be the most efficient method to acquire the information about the links, and requires the use of OWL, rather than the less complex RDFS.

4. Semantic Publishing Experiences: Where Others Are

A number of solutions to the kind of problems we're facing have been discussed in the XML and RDF communities: the various configurations of system architecture "beast" combining an RDF triple store with an XML database were discussed at XML Prague 2013 [Greer2013]. Greer's "Consumer" model pulls in the RDF triples via an XQuery application to combine them with XML from a MarkLogic database, and another option he mentions is to tag a node set with a URI identifier, which corresponds to an RDF resource in a triple store. This then explicitly links the XML documents to the RDF triples, and is something we are implementing through the use of DOIs (Digital Object Identifiers) used both to identify the XML document and, expressed as URIs, to identify the subject and object of a triple expressing a relationship between two XML documents.

The BBC have combined the use of XML databases and RDF triple stores for their dynamic semantic publishing for BBC Sport, the 2012 Olympics and now BBC News sites [Rayfield2012]. Their journalists tag assets (such as video clips) according to an ontology, and this metadata is captured in the OWLIM triple store, then combined with external sports statistics and content assets encoded in XML stored in a MarkLogic database. Content can be aggregated using a combination of SPARQL for domain querying and XQuery for asset selection. The content transaction manager combines SPARQL and XQuery so that queries for sports statistics related to a particular concept, like "The league table for the English Premiership" can be carried out across the triple store and MarkLogic database. Another example of combining XML with RDF is at Nature Publishing Group [Hammond2013], where they use XMP [Adobe], a vocabulary using a subset of RDF that can be embedded in XML documents, to bridge between their XML data store and their triple store. While their XML content is distributed across the organisation in a number of different silos, the triple store enables them to query the integrated picture of all their data.

5. RDFa and schema.org

While semantic markup does not increase search rankings directly, it has been shown to improve click through rates significantly, as search results are more eye-catching and it's clearer to the user that the retrieved document is a relevant answer to their query. For example, when Best Buy added RDFa to their product pages, traffic to their site increased by 30%, and Yahoo! has reported a 15% increase in click through rate for enriched links. We are still evaluating a number of options for embedding structured metadata in our discoverability pages. Although RDFa allows for richer descriptions, and can provide our full metadata "under the hood", the advantage of schema.org markup is that it is fully supported by the major search engines. As an example, we can add some simple markup to the Overview on Barack Obama on the Oxford Index like so:

```
<div vocab="http://schema.org/" typeof="Person"
     about="http://oxfordindex.oup.com/view/10.1093/oi/authority.20110803100243337">
  <span property="name">Barack Obama</span> <p/>
  <span property="jobTitle">American Democratic statesman</span> <p/>
  born <span property="birthDate">4 August 1961</span> <p/>
</div>
```

This does however require a mapping of our XML elements such as "occupation" to schema.org vocabulary terms like "jobTitle", which can introduce semantic mismatch. (Is "American Democratic statesman" really a job title?). Other schema.org CreativeWork schemas, such as Book and Article may map more closely on to our XML, but overall, the drawback of schema.org is that only very simple markup can be used, so it does not provide a full alternative to an API on our metadata or full linked data publication.

6. Conclusion

We are still in the early days of our journey from XML to Linked Data, and a number of issues remain to be resolved. Firstly, we need to re-assess our business case to identify the most effective output. Secondly, we need to identify what proportion of our information should be stored as triples, versus as XML: our strategy to date is to migrate slowly, as more resources are assigned a URI and more links created between those URIs, we can store the new triples in the triple store. It is also a modelling issue of how much of the data is persistent - the more data changes, the better it is to leave it as well-indexed XML that can easily be searched, such that new links can be dynamically created, rather than stored statically as RDF. Thirdly, we need to decide whether to publish the triples as RDFa embedded markup, or go the whole hog and publish Linked Data, though again these may be two stages on a longer journey. And finally, we have yet to prove that a combination of an XML store for documents and triple store for links is really the best architectural solution for our needs.

Bibliography

[Rayfield2012] Jem Rayfield. Sport Refresh: Dynamic Semantic Publishing. 17 April 2012. http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html

[Greer2013] Charles Greer. XML and RDF Architectural Beastiary. Proceedings of the XML Prague 2013 Conference. February 2013, 189-205. http://archive.xmlprague.cz/2013/files/xmlprague-2013-proceedings.pdf

[Hammond2013] Tony Hammond. Techniques used in RDF Data Publishing at Nature Publishing Group. March 2013. http://www.slideshare.net/tonyh/semweb-meetupmarch2013

[Adobe] Adobe. Extensible Metadata Platform. http://www.adobe.com/products/xmp/

[W3C2013] World Wide Web Consortium. Turtle Terse RDF Triple Language W3C Candidate Recommendation. 19 February 2013. http://www.w3.org/TR/turtle/

[Dodds2012] Leigh Dodds. Principled Use of RDF/XML. 12 June 2012. http://blog.ldodds.com/2012/06/12/principled-use-of-rdfxml/

xproc.xq - Architecture of an XProc processor

James Fuller

MarkLogic

Abstract

XProc is a markup language that describes processing pipelines which are composed of discrete steps that apply operations on sets of XML documents. This paper details out the architecture, model and process flow of xproc.xq, an XProc processor, implemented using XQuery 3.0.

Keywords: XProc, XQuery 3.0

1. Introduction

This article provides an in-depth overview of the primary architectural components of xproc.xq, an XProc [1] processor which has been built using XQuery 3.0, on top of the MarkLogic database [2]. Where there is time I highlight some of the more novel aspects of the system and provide background on key design decisions.

The goals of developing xproc.xq (as with most of my open source work) are of an entirely selfish nature;

- Testbed XProc implementation for experimentation
- Learn about XQuery and functional programming development 'in the large'
- Observe performance characteristics of XProc within database context

2. XProc Background

The ubiquity of XML creates the need for programmers to be able to implement complex, scalable and extensible processing work flows which work on sets of XML documents using the broad and deep stack of XML technologies available today. XProc [1], the XML Pipeline language defined by the W3C, attempts to provide developers with a tool that helps create complex document work flows using a declarative description in the form of pipelines.

A pipeline is a well worn abstraction in computing, yet loosely defined. For our purposes, we define pipelines as a declarative model that prescribes a set of operations which are to be applied to a set of XML documents. Each operation has a consistent data interface allowing for the flow of XML documents generating data for the next operation to consume.

Our first example shows how XProc describes a pipeline using XML markup.

Example 1. XProc simple example

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            version="1.0"
            mylimit="10">
  <p:identity>
    <p:input port="source" select="//p"/>
  </p:identity>
  <p:count limit="$mylimit"/>
</p:pipeline>
```

An XProc processor consumes the pipeline and applies processing to XML document(s) supplied as input. The example pipeline copies document(s) using the `p:identity` step then counts how many document(s) there are with the `p:count` step.

Having a declarative definition of a work flow, separates the *how to process* from the *what to process*, leaving the XProc processor free to handle the '*how*' as implementation details and pipeline authors to describe the '*what*' to process with a catalog of operations. For those who have never experienced XProc I will pass over over many of the finer details of this example for now, but I revisit in the XProc refresher section.

Unix pipelines are the oft quoted analogy when explaining XProc. Unix pipelines work by allowing each individual shell command to consume and emit lines of text data. Such a consistent data interface is similarly found in XProc's ability to generate and consume XML, though its important to note that in Unix pipes, shell commands work on each line of text versus an entire document.

Much of the utility of Unix pipes comes from the fact that there are a lot of useful shell commands, correspondingly XProc comes with a large set of built in steps as well as the facility to create your own steps. We could extend the analogy by observing that shell commands share a consistent selection language in the form of regular expressions where XProc leverages XPath.

doi:10.14337/XMLLondon13.Fuller01

If you work with XML today, its likely that you've encountered or already built your own ad-hoc pipelines using your favourite XML processor. XProc simply formalises what could be viewed as the natural evolution from Unix pipeline 'line by line' text processing to a richer, more complex work flow style of document processing. This frees up the other XML technologies to focus on what they are good at and for XProc to act as the 'main control loop', orchestrating work flow processes at a higher level of abstraction.

2.1. Goals

The XProc specification lists out twenty+ goals, embodying the guiding principles for development of the language.

I've taken the liberty to summarise into the following list;

- The language must be expressed as declarative XML and be rich enough to address practical interoperability concerns but concise
- The language must allow the inputs, outputs, and other parameters of a components to be specified with information passed between steps using XML
- The language must define the basic minimal set of mandatory input processing options and associated error reporting options required to achieve interoperability.
- Given a set of components and a set of documents, the language must allow the order of processing to be specified.
- Agnostic in terms of parallel, serial or streaming processing
- The model should be extensible enough so that applications can define new processes and make them a component in a pipeline.
- The model could allow iteration and conditional processing which also allow selection of different components as a function of run-time evaluation.

2.2. History

Pipelines in computing are an old concept, and in work flow processing similarly ancient for markup languages.

As far back as 2004, a W3C Note set out requirements for an XML processing model: "XML Processing Model Requirements,"[1] W3C Working Group Note 05 April 2004.

The following year, in 2005, another W3C member submission was proposed: "XML Pipeline Language (XPL) Version 1.0" (draft), submitted by Orbeon, Inc., on 11 March and published on 11 April.[2]

It was identified as a goal to promote an interoperable and standard approach to the processing of XML documents and the working group started meetings late 2005.

A set of use cases were developed and published in 2006[3] and work on the spec itself proceeded.

Several interim draft candidates were developed, with the WG editor, Norman Walsh and member Vojtech Toman developing in parallel reference implementations.

As is typical with any specification process, it took XProc much longer to achieve W3C Recommendation status, ratified in May 2010 [1].

2.3. Brief refresher

An XProc pipeline document has a document root element of p:pipeline or p:declare-step which contains one or several steps. Steps are either implicitly or explicitly connected with documents flowing between them. Each step type determines the kind of processing it does on documents.

The example shown in the Introduction section, illustrated how XProc describes a pipeline.

```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            version="1.0"
            mylimit="10">
  <p:identity>
    <p:input port="source" select="//p"/>
  </p:identity>
  <p:count limit="$mylimit"/>
</p:pipeline>
```

At first glance, you maybe able to deduce that this is a pipeline that has two components ('steps' is the XProc term); an identity step which copies an XML document and a count step that counts the number of documents being passed to it from the `p:identity` step, but with the twist that it `p:count` stops counting once it hits a certain limit.

[1] http://www.w3.org/TR/2004/NOTE-proc-model-req-20040405/
[2] XML Pipeline Language (XPL) Version 1.0 (Draft) W3C Member Submission 11 April 2005 - http://www.w3.org/Submission/xpl/
[3] XProc Use Cases - http://www.w3.org/TR/xproc-requirements/

What is not entirely clear with this example is;

> What defines the starting set of XML document(s) ?
>
> How do documents *know* how to flow from step to step ? What connects them ?
>
> What does it *mean* that `p:identity` child `p:input` has a select attribute xpath value of '`//p`' ?
>
> How does the `p:pipeline` attribute `@mylimit` define a reusable value that `p:count` uses within its own limit attribute ?

We can rewrite this pipeline to be more explicit, which answers some of our questions.

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

  <p:input port="source" primary="true"
           sequence="true"/>
  <p:output port="result" primary="true"
           sequence="true">

    <p:pipe step="mycount" port="result"/>
  </p:output>
  <p:option name="mylimit" select="10"/>
  <p:identity name="myidentity">
    <p:input port="source" select="//p">
      <p:pipe step="mypipeline" port="source"/>
    </p:input>
  </p:identity>
  <p:count name="mycount">
    <p:input port="source">
      <p:pipe step="myidentity" port="result"/>
    </p:input>
    <p:with-option name="limit" select="$mylimit">
      <p:empty/>
    </p:with-option>
  </p:count>
</p:declare-step>
```

Documents come in from the '*outside* world' via the primary `p:input` child defined under `p:declare-step`. The `p:pipeline` was a kind of shorthand alias for the more general `p:declare-step` step which incidently is also used to define new steps.

The `$mylimit` value is an XProc option on the `p:declare-step`, the `p:count` has a defined option, whose value is set using `p:with-options`.

None of this explains the select attribute xpath expression on `p:identity`'s `p:input` element. Its purpose is to instruct the processor to filter incoming documents on the input source port with an xpath expression, which is very useful to select what you want to work on. Each matched `<p/>` element is turned into a document and will be copied to the `p:identity` output result port. XProc relies heavily on XPath as its built in selection mechanism and we will see it pops up everywhere.

With the magic of default readable ports, we can now understand how each step's `p:input` uses a `p:pipe` element to define where documents flow from, even if we don't explicitly instruct them in our pipeline. The `p:pipe` uses a step's name attribute and port to unambiguously identify the binding and provides a rigid 'flow' path at runtime, through which documents flow.

In summary, this pipeline takes a set of documents (from primary `p:input`) and sends the input into `p:identity` (as defined with its `p:input`). The `p:identity` step copies the `p` elements from them and passes them (as a sequence of documents) to the `p:count` step. The `p:count` step counts the number of items in the sequence up to a maximum of 10, outputting the count result to console (which is the job for primary `p:output` result port).

I've chosen this example to illustrate that XProc has some disconcerting 'warts', eg. the `<p:empty/>` within the `p:with-option` is anachronistic at best. You should now have a sense of Process's defaulting 'story'. Our description of a pipeline needs a sufficient level of detail for an XProc processor to be able to puzzle out *how* to process the documents. XProc provides implicit defaults with its syntax to help make an already verbose language, well less verbose. There is work needed at the W3C XML Processing WG level to help make even more syntatical changes to reduce verbosity.

Note

For those who are curious, I would highly recommend installing what I consider the reference XProc implementation, Norm Walsh's XML Calabash[1] and run against the example

```
calabash -isource=test.xml -isource=test2.xml test.xpl
```

This would yield the following result, outputted to console.

```
<c:result
  xmlns:c="http://www.w3.org/ns/xproc-step">
  2
</c:result>
```

Where the value of the `c:result` element reflects depending on how many `<p/>` elements were passed from the `p:identity` step, up to a limit of 10.

2.3.1. Process Model

In classic work flow systems it is common to implement methods like Finite State Machine (FSM) [3] which embed state information within each document.

In such systems, changes to state are the events which control processing and the apparent 'flow' of documents. Document flow based on state transitions implies that there are no fixed paths between the processing 'steps' enabling at runtime highly dynamic and complex work flows. In practice, event driven pipelines are also difficult to implement and diagnosing issues with performance or optimisations not very straightforward.

A saying attributed to several programming luminaries declares that 'state' is the enemy of dynamic computation and I think this applies to XProc. A more amenable approach was needed to take advantage of functional programming principles associated with its declarative format.

In XProc, inputs flow into a pipeline from one step to the next with results emitted at the end. The order of the steps is constrained by the input/output connections between them rather then state contained within the documents themselves. Document flow is a consequence of the implicit and explicit binding of input source ports to output result ports.

This allows implementations to be able to statically analyze how to execute steps, be it in sequential or parallel fashion or to take advantage of some characteristic of the environment. With a minor loss of freedom in specifying highly complex work flows (though its still possible with XProc) we gain a lot of freedom in terms of process execution.

XProc also not have too bother maintaining the state of each document which itself can be complicated and costly in terms of performance. By eschewing with document state altogether, XProc avoids the issues associated with recovering state when it gets lost or corrupted.

Using a 'stateless' process flow model means that XProc itself minimises constraining parallel or streaming processing scenarios. Streaming is a particularly difficult scenario to enable, for example, if there is significant reordering between the input document trees and output document trees, its been observed [4] that one should avoid even attempting to stream, conversely even mature XML processor like Saxon [5] has only partial streaming in place for its XML Schema validator, xpath processor and XQuery/XSLT processor [6].

Lastly, be aware that there are plenty of pipelines one could author that can cause side effects which invalidate streaming or parallel processing. This caveat is more to do with the technologies underlying any specific step processing (`p:eval`, `p:xslt`, etc) versus XProc.

2.3.2. Steps

Steps can have options and parameters, some steps are called 'compound' steps and embody multiple nested pipelines.
New steps can be defined, using `p:declare-step`, that are used in exactly the same manner as the built-in steps.
Note that whenever you have create a pipeline, that pipeline itself can also be reused as a step in other pipelines.
Custom steps can be bundled up into library (`p:library`) and reused in other pipelines by importing using a `p:import` element.

[1] http://xmlcalabash.com/

Compound and Multi container Steps

A compound step contains a subpipeline. Multi-container steps contain two or more subpipeline(s).

`<p:declare-step>`
: Declares an XProc pipeline. This step can define a new reusable XProc step for use by other pipelines. When used within another pipeline it acts as if it was an atomic step, regardless if it contains more subpipelines.

`<p:pipeline>`
: Is an alias of `p:declare-step` with default implied inputs and outputs, itself reusable as a step in other pipelines. When invoked within another pipeline it acts as if it was an atomic step, regardless if it contains more subpipelines.

`<p:choose>`
: Multi container step which selects one of a number of alternative pipelines based on test criteria

`<p:for-each>`
: Iterates over a sequence of documents with a specific subpipeline

`<p:group>`
: Groups a sequence of steps together as a subpipeline

`<p:try>`
: Multi container step that provides a try subpipeline which if fails is caught with an error exception handling subpipeline

`<p:viewport>`
: Iterates a pipeline over inner selections of each document in a set of documents.

Atomic Steps

These steps are the basic building blocks of XProc pipelines with each carrying out a single XML operation. Atomic steps fully encapsulate the processing they apply. Most atomic steps accept input and emit output. All atomic steps will never themselves contain subpipeline(s).

Required steps: These steps are provided by a conformant XProc processor.

`<p:add-attribute>`
: Add a single attribute to a set of matching elements.

`<p:add-xml-base>`
: Explicitly add or correct `XML:base` attributes on elements.

`<p:compare>`
: Compare two documents for equivalence.

`<p:count>`
: Count the number of documents in source input sequence.

`<p:delete>`
: Delete items matched by pattern from the source input.

`<p:directory-list>`
: Enumerate a directory's listing into result output.

`<p:error>`
: Generate an error that throws at runtime.

`<p:escape-markup>`
: Escape XML markup from source input.

`<p:filter>`
: Filter documents with dynamically created select expressions

`<p:http-request>`
: Interact with resources identified by Internationalized Resource Identifiers (IRIs) over HTTP.

`<p:identity>`
: Make an exact copy of an input source to the result output.

`<p:insert>`
: Insert an XML selection into the source input.

`<p:label-elements>`
: Create a label (ex. `@XML:id`) for each matched element, and store the value of the label within an attribute.

`<p:load>`
: Load an XML resource from an IRI providing it as result output.

`<p:make-absolute-uris>`
: Make the value of an element or attribute in the source input an absolute IRI value in the result output.

`<p:namespace-rename>`
: Rename the namespace declarations.

`<p:pack>`
: Merge two document sequences.

`<p:parameters>`
: Make available a set of parameters as a `c:param-set` XML document in the result output.

`<p:rename>`
: Rename elements, attributes, or processing instruction.

`<p:replace>`
: Replace matching elements.

`<p:set-attributes>`
: Set attributes on matching elements.

`<p:sink>`
: Accept source input and generate no result output.

`<p:split-sequence>`
: Divide a single sequence into two.

`<p:store>`
: Store a serialized version of its source input to a URI.

`<p:string-replace>`
: Perform string replacement on the source input.

`<p:unescape-markup>`
: Unescape the source input.

`<p:unwrap>`
: Replace matched elements with their children.

`<p:wrap>`
: Wrap matching nodes in the source document with a new parent element.

`<p:wrap-sequence>`
: Produce a new sequence of documents.

`<p:xinclude>`
: Apply XInclude processing to the input source.

`<p:xslt>`
: XSLT evaluation on style sheet input source.

Optional Steps: These steps are optionally provided by an XProc processor.

`<p:exec>`
: Apply an external command to the input source.

`<p:hash>`
: Generate a cryptographic hash (message digest, digital fingerprint) and inserts in document.

`<p:uuid>`
: Generate a Universally Unique Identifier (UUID).

`<p:validate-with-relax-ng>`
: Validate the input XML with RelaxNG schema.

`<p:validate-with-schematron>`
: Validate the input XML with Schematron schema.

`<p:validate-with-xml-schema>`
: Validate the input XML with XML schema.

`<p:www-form-urldecode>`
: Decode the x-www-form-urlencoded string into a set of XProc parameters.

`<p:www-form-urlencode>`
: Encode a set of XProc parameter values as an x-www-form-urlencoded string.

`<p:xquery>`
: XQuery evaluation on xquery input source.

`<p:xsl-formatter>`
: Render an XSL version 1.1 document (as in XSL-FO).

Additional steps: The XML Processing WG from time to time publishes W3C notes on additional steps that exist in the XProc step namespace.

`<p:template>`

`<p:in-scope-names>`

A developer may also define their own steps (using `p:declare-step`) which when combined with `p:library` provides a powerful reuse componitisation.

Community defined extensions

There are also many extensions being defined by the community which are being defined which some XProc processors may support.

2.3.3. Known Implementations

The following is transcribed from a list of XProc processors being tested at tests.xproc.org[1]:

Calabash
: Norman Walsh is building an open-source implementation in Java. Calabash is built on top of the Saxon APIs and uses XPath 2.0 as its expression language.

Calumet
: EMC's Java-based XProc processor. The processor features an extensible architecture and is easy to embed in other Java applications. Free for developer use.

QuiXProc Open
: Innovimax's GPL, Java implementation based on XML Calabash adding Streaming and Parallel Processing. The is also a Commercial product at http://quixproc.com.

Tubular
: Tubular is a Java implementation based on immutable objects, in order to facilitate the addition of parallelism support, thus reducing the need for locking mechanisms.

XProcerity
: XProcerity is a Java implementation focused primarily on high performance in multi-threaded environments, such as high-traffic enterprise web applications.

xprocxq
: An earlier implementation of Jim Fuller's **xprocxq** is an experimental bootstrap implementation of W3C XProc Draft Specification, written in XQuery, for the eXist XML Database [7].

[1] XProc Known Implementations - http://xproc.org/implementations/

2.3.4. XProc vnext

The following is a sampling of non trivial deployments of XProc in use today, Anecdotal evidence seems to point towards the fact that for the right scenario, XProc can be quite a power tool.

http://mesonet.info/ - real-time citizen world wide weather station

http://code.google.com/p/daisy-pipeline/wiki/XProcOverview- The DAISY Pipeline is an open-source, cross-platform framework for document-related pipelined transformation. It supports the migration of digital content to various formats efficiently and economically, facilitating both production and distribution of DAISY Digital Talking Books

http://balisage.net/Proceedings/vol8/html/Williams01/BalisageVol8-Williams01.html - validating RESTful services

https://github.com/gimsieke/epubcheck-xproc - epub checker implemented using XProc

Unfortunately, there is just as strong feedback that indicates that there are many 'rough edges' in XProc v1.0 which is a barrier to wider adoption in both the XML and broader development communities.

- too verbose or the need to be overly explicit
- some constructs unwieldy (parameters)
- deficient (only use string values in options and variables)
- hard to work with other non-XML data
- we require mechanism for sanctioning step definitions without full blown W3C specification

Adoption, while slow, has had a steady uptake over the ensuing two and half years since becoming a W3C Recommendation. The W3C XML Processing WG is now preparing for work on version 2.0 of the specification, by first creating a draft set of requirements[1] with a short set of goals that attempt to address deficiencies;

1. Improving ease of use (syntactic improvements)
2. Improving ease of use (ex. increasing the scope for working with non XML content)
3. Addressing known shortcomings in the language
4. Improve relationship with streaming and parallel processing

The requirements document also traces how well XProc v1.0 satisfied previous use case requirements. This 'score card' helps focus working on scenarios that were not at all or partially addressed.

The next version of XProc is very much a case of 'fix what is broken' and be judicious with adding only the bare minimum required.

The following list is a sample of concrete actions being considered;

- **Fix Parameters** - Change parameters to be more like options which imply adopting the XSLT 3.0 extensions to the data model and functions and operators to support maps
- **Non XML document processing** - Provide native processing of non XML processing with a constrained scope (possibly using a resource manager)
- **Drop XPath 1.0 support** - Remove any *must* type requirements for supporting XPath 1.0.
- **Allow options and variable to contain arbitrary fragments** - Relax the constraint that variables and options can ony be defined as a string or `xs:untypedAtomic`.
- **Fix "non-step wrapper"**- Remove the concept of 'Non-step wrappers' by making `p:when`/`p:otherwise` in `p:choose` and `p:group`/`p:catch` in `p:try` compound steps and get rid of the notion "non-step wrapper".
- **Syntax changes** - For example, allow Attribute Value Template (AVT).

This is an ongoing discussion, so please feel free to offer suggestions and join the debate (Details at XML Processing WG home page[2])

3. The xproc.xq project

The xproc.xq project is an open source project hosted at github [8] and offers use under the liberal Apache v2.0 license[3]. xproc.xq is an implementation of an XProc processor using XQuery v3.0 with vendor specific plugins. Currently, only support for MarkLogic [2] exists but plans are in place to support other XQuery processors that support XQuery 3.0 (eXist, Saxon, XQilla).

In late 2008, the author created a prototype XProc processor using XSLT v2.0 [9] under eXist XML Database [7]. This proof of concept led to the start of development of **xprocxq**, with an initial implementation created as an extension to the eXist XML database server.

The goals were set out as follows;

- creation of an XProc processor with XQuery
- avoid using XSLT, mainly as the author had engaged in development of pipeline like processing in XSLT and wanted to avoid reliance upon it
- use eXist support for first class functions to underpin execution control

[1] XProc vnext language requirements - http://www.w3.org/XML/XProc/docs/langreq-v2.html
[2] XML Processing Working Group - http://www.w3.org/XML/Processing/
[3] *Apache v2.0 License - http://www.apache.org/licenses/LICENSE-2.0.html*

- acknowledge limitations of XQuery by implementing a usable subset of XProc (at the time both XQuery 3.0 and XProc v1.0 were draft standards)
- leverage performance of an XProc processor embedded in a database context

eXist [7] has excellent extensibility characteristics making it is easy to develop extensions, but it became quite a challenge to get a usable subset of XProc conformance. Most of the issues were related to XQuery v1.0 suitability as a language for implementing such a processor versus anything particularly difficult with eXist itself.

XQuery v1.0 lack of first class function support meant a heavy reliance on eXist specific functions, more troubling was that many fundamentals of xprocxq process flow were controlled by util:eval() (eXist extension for dynamically executing constructed XQuery). Additionally, due to a reluctance to employ XSLT there were several issues in the implementation of XProc steps, which turns out to be a mistake as XProc makes heavy use of XSLT style match [9] expressions.

Tip

Incidently, it is now possible to use XQuery 3.0 to fully emulate XSLT matching expressions as exemplified by John Snelson's transform.xq [10]

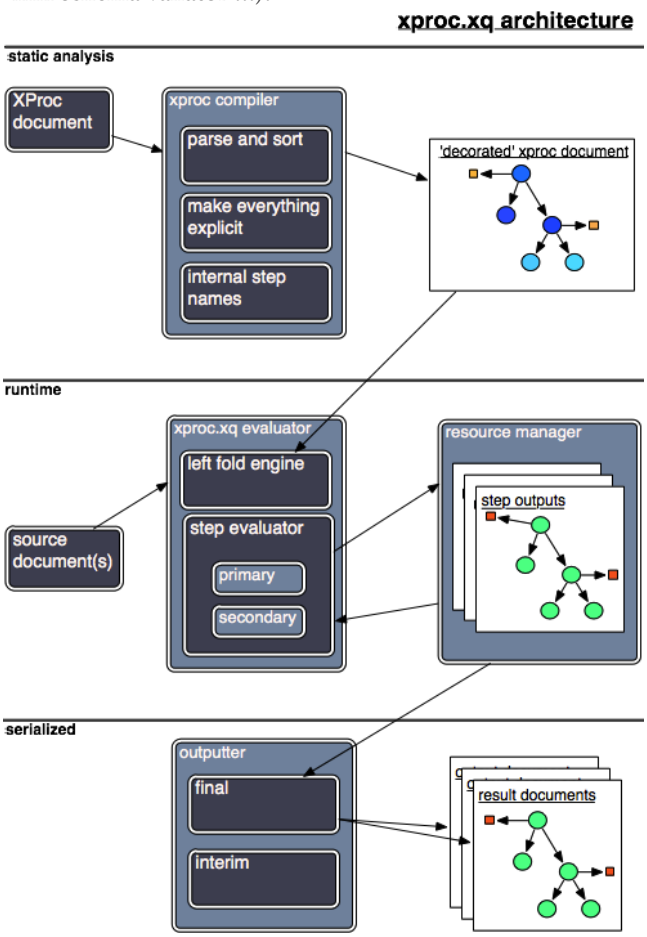
In 2010, when XProc became a recommendation, plans changed to completely refactor **xprocxq** to work with Saxon [5], which started to support many of the emerging ideas in XQuery 3.0. This had a positive effect on the internal processing and with robust first class function support was able to remove most evals but still a heavy reliance on vendor specific extensions throughout the codebase.

The final iteration of the XProc processor, known as xproc.xq, started in 2012 and is built on top of MarkLogic [2] XQuery v3.0 support. This refactor focused on isolating as much of the codebase as possible into pure XQuery v3.0 and push any vendor specific code into a pluggable module. This enables future versions of xproc.xq to support other XQuery vendors.

4. xproc.xq architecture

4.1. Design

The diagram provides a 'fly over' view of xproc.xq application architecture, in the context of its process flow. Using XQuery allows us to focus on the application architecture aspects, which is why there are no lower level components on display (xml parser, xpath engine, xml schema valiator ...).



The XProc processor advances through three stages when processing an XML pipeline.

- **static analysis**- consume and parse the XProc pipeline, generating a runnable representation of said pipeline
- **dynamic evaluation**- engine that dynamically evaluates the runnable pipeline representation
- **serialization**- output interim results for further processing or final results

Before we precede a word of caution. I am using nomenclature which is more appropriately applied to development within compiled languages. Building an XProc processor using a dynamic, interpreted language like XQuery often brings the terminology in use into question, mainly because engineering tradeoffs are being considered which in compiled languages would feel like 'cutting corners'. This comes with the territory of pushing XQuery beyond its intended limits and with that said we now drill down into more detail of each phase.

4.2. Static Analysis Phase

The static analysis phase consumes the original pipeline, parsing it and generates a highly decorated version of this pipeline.

This decorated pipeline version can be considered the internal model used throughout all subsequent processing and provides a 'single point of truth' in terms of lookup, namespace declarations, step bindings, variable & option definitions. The static phaze also takes care of adding a unique internal default name to each step, as well as reordering steps according to the flow as defined by connections (bindings) between steps.

When a pipeline is created, the syntax represents an authors intent but itself does not contain enough information to be 'runnable' by an XProc processor. As we've seen in previous sections, XProc has a suite of implied processing behaviors which need to be 'teased' out, we also need to take care of other concerns, like ensuring the order of steps follow how their implicit and explicit bindings have been created. To illustrate what I mean, nothing stops a developer writing an *obfuscated* pipeline where the order of step processing is unclear.

The following is an example (taken directly from the W3C XProc test suite[1]) which illustrates the kind of problems that the static analysis phase needs to account for.

Example 2. unordered XProc Example

```
<p:declare-step version='1.0' name="main">
  <p:input port="source"/>
  <p:output port="result">
    <p:pipe step="i1" port="result"/>
  </p:output>

  <p:identity name="i1">
    <p:input port="source">
      <p:pipe step="i3" port="result"/>
    </p:input>
  </p:identity>

  <p:identity>
    <p:input port="source">
      <p:pipe step="main" port="source"/>
    </p:input>
  </p:identity>

  <p:identity name="i3"/>

</p:declare-step>
```

In this example, the `p:identity` step named '`i1`' input port means its not the first step in the XProc work flow. Its actually the second step defined which takes in the pipeline's input that is the first to be processed. And perhaps what will surprise most readers is that the first step is actually the last to process, based on its step binding with `p:identity` '`i3`'. Deriving the final process order is a function of making all step binding relationships explicit.

The next example shows a pipeline where the author took care to layout the steps in their true process order.

Example 3. ordered XProc Example

```
<p:declare-step version='1.0' name="main">
  <p:input port="source"/>
  <p:output port="result">
    <p:pipe step="i1" port="result"/>
  </p:output>

  <p:identity>
    <p:input port="source">
      <p:pipe step="main" port="source"/>
    </p:input>
  </p:identity>

  <p:identity name="i3"/>

  <p:identity name="i1">
    <p:input port="source">
      <p:pipe step="i3" port="result"/>
    </p:input>
  </p:identity>

</p:declare-step>
```

[1] XProc Test Suite - http://tests.xproc.org/

Its a feature of XProc that the author does not have to explicitly control process order by ordering XML elements. Being able to insert a step or make changes to step bindings without having to go trace through an entire pipeline checking ordering makes life easier for developers.

Side Effects in pipelines

Connections between steps define order in XProc but this is not a promise or guarantee of the actual process order. For pipelines that rely upon side effects, unexpected results may occur (non-deterministic processing, like downloading a file from the internet or saving a file to the file system). The XProc specification discusses this in the XProc specification[1]

Now lets take a look at the abstract syntax tree that gets generated by the static analysis phase, using a variation of our original xproc example.

Example 4. XProc example

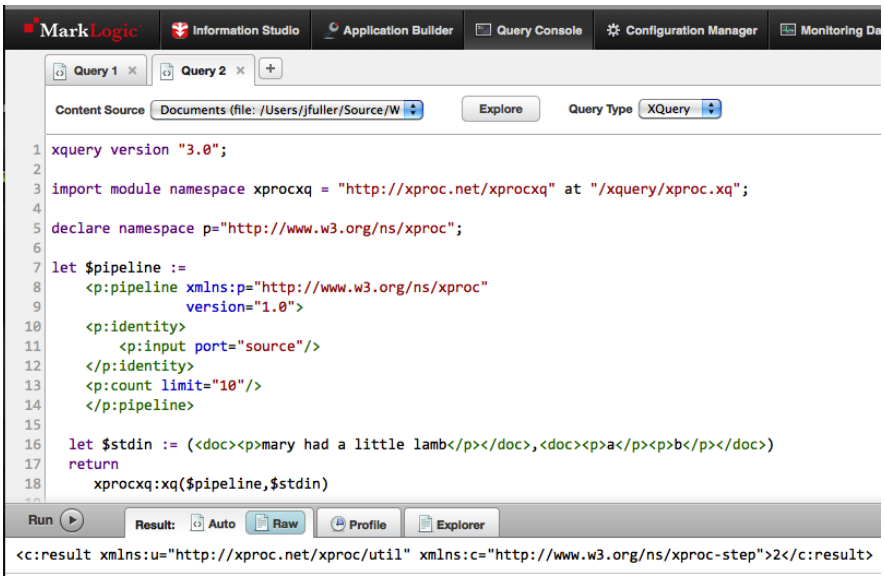
```
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            version="1.0">
  <p:identity>
    <p:input port="source"/>
  </p:identity>
  <p:count limit="10"/>
</p:pipeline>
```

If you have not previously installed xproc.xq to run on your MarkLogic instance, now is the time to review xproc.xq installation to install and deploy xproc.xq.
The default method of running xproc.xq is to import the xprocxq XQuery library module and invoke with the simple `xprocxq:xq()` entry point, as shown in the following code listing.

Example 5. xproc.xq entry point

```
xquery version "3.0";

import module namespace xprocxq =
  "http://xproc.net/xprocxq" at "/xquery/xproc.xq";

declare namespace p="http://www.w3.org/ns/xproc";

let $pipeline :=
  <p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
              version="1.0">
    <p:identity/>
    <p:count limit="10"/>
  </p:pipeline>

let $stdin := (
  <doc>
    <p>mary had a little lamb</p>
  </doc>,
  <doc>
    <p>a</p>
    <p>b</p>
  </doc>
)
return
  xprocxq:xq($pipeline,$stdin)
```

The simplest way to run this is to cut and paste into MarkLogic query console (`http://localhost:8000/qconsole`) and choose the content source that xproc.xq has been set up with.



[1] XProc specification H. Sequential Steps, parallelism, and side-effects - http://www.w3.org/TR/xproc/#parallelism

Running this will return a c:result element containing the count of unique documents

Example 6. result of XProc processing

```
<c:result xmlns:c="http://www.w3.org/ns/xproc-step">
  2
</c:result>
```

But what we really want is to be able to analyze the decorated tree version that is generated 'under the covers' during the static analysis phase. This can be achieved using an overloaded version of the `xprocxq:xq()` function, demonstrated below;

Example 7. getting debug output from xproc.xq

```
xquery version "3.0";

import module namespace xprocxq =
  "http://xproc.net/xprocxq" at "/xquery/xproc.xq";

declare namespace p="http://www.w3.org/ns/xproc";

let $pipeline :=
<p:pipeline xmlns:p="http://www.w3.org/ns/xproc"
            version="1.0">
  <p:identity>
    <p:input port="source"/>
  </p:identity>
  <p:count limit="10"/>
</p:pipeline>

let $stdin    := (<doc>
                    <p>mary had a little lamb</p>
                  </doc>,
                  <doc>
                    <p>a</p>
                    <p>b</p>
                  </doc>
                 )
let $bindings := ()
let $options  := ()
let $outputs  := ()
let $dflag    := 1
let $tflag    := 0
return
  xprocxq:xq($pipeline, $stdin, $bindings,
             $options, $outputs, $dflag, $tflag)
```

The parameters the function accepts are defined as;

- `$pipeline` - the XProc pipeline being processed
- `$stdin` - A sequence of XML Document(s) to be placed on the primary input source port of the pipeline
- `$bindings` - n/a
- `$options` - Sequence containing options that will override pipeline option values
- `$outputs` - n/a
- `$dflag` - When set to 1 outputs decorated tree representation of XProc pipeline and ll input/output port values
- `$tflag` - When set to 1 outputs timing information

By supplying a value of `1` for the `$dflag`, we are instructing xproc.xq to emit the decorated pipeline tree. We also get returned all values of any input or output port which for the time being we will ignore.
Cut and paste the above into query console (`http://localhost:8000/qconsole`) and run.

Example 8. debug output

```
<xproc:debug episode="11600574566574829649" xmlns:xproc="http://xproc.net/xproc">
  <xproc:pipeline>
    <p:declare-step version="1.0" mylimit="10" xproc:type="comp-step"
                    xproc:default-name="!1" xmlns:p="http://www.w3.org/ns/xproc">
      <ext:pre xproc:default-name="!1.0" xproc:step="true"
               xproc:func="ext:pre#4" xmlns:ext="http://xproc.net/xproc/ext">
        <p:input port="source" select="/" xproc:type="comp" primary="true">
          <p:pipe port="result" xproc:type="comp" step="!1" xproc:step-name="!1"/>
        </p:input>
        <p:output xproc:type="comp" port="result" primary="true" select="/"/>
      </ext:pre>
      <p:identity xproc:step="true" xproc:type="std-step"
                  xproc:func="std:identity#4" xproc:default-name="!1.1">
        <p:input port="source" select="//p" xproc:type="comp" primary="true">
          <p:pipe port="result" xproc:type="comp" step="!1.0" xproc:step-name="!1.0"/>
        </p:input>
        <p:output xproc:type="comp" port="result" sequence="true" primary="true" select="/"/>
      </p:identity>
      <p:count limit="10" xproc:step="true" xproc:type="std-step"
               xproc:func="std:count#4" xproc:default-name="!1.2">
        <p:input port="source" select="/" xproc:type="comp" primary="true">
          <p:pipe port="result" xproc:type="comp" step="!1.1" xproc:step-name="!1.1"/>
        </p:input>
        <p:output xproc:type="comp" port="result" primary="true" select="/"/>
        <p:with-option xproc:type="comp" name="limit" select="10"/>
      </p:count>
      <ext:post xproc:step="true" xproc:func="ext:post#4"
                xproc:default-name="!1!" xmlns:ext="http://xproc.net/xproc/ext">
        <p:input port="source" primary="true" select="/" xproc:type="comp">
          <p:pipe port="result" xproc:type="comp" step="!1.2" xproc:step-name="!1.2"/>
        </p:input>
        <p:output primary="true" port="result" xproc:type="comp" select="/"/>
      </ext:post>
    </p:declare-step>
  </xproc:pipeline>
  <xproc:outputs>
    ..... snipped for brevity .....
  </xproc:outputs>
</xproc:debug>
```

We are specifically interested in the `xproc:pipeline` element which contains the abstract syntax tree (decorated pipeline).
The code highlighted in bold shows how each step now has a unique internal `@xproc:default-name` attribute. The naming convention for these default names is outlined within the XProc specification [1] itself and provides a unique id to each step element. Nesting as well as a way to determine its nesting level, which is needed to deal with nested subpipelines. The input and output port `p:pipe` elements now point to their sources using these default-names.
The following table outlines how `@xproc:default-name` are used;

Table 1.

Level #	Level #.#
!1 p:declare-step	
	!1.0 ext:pre step - internal step that is responsible for bringing input into the pipeline via `p:pipe` on `p:input` reference to !!
	!1.1 `p:identity` step - standard atomic step that takes its input from the result of !1.0
	!1.2 `p:count` step - standard atomic step that takes its input from the result port of !1.1
	!1! ext:post step - internal step that takes its input from the result port of !1.2 and responsible for placing the result to the outputter.

The `ext:pre` and `ext:post` steps are not extension steps to be specified by pipeline authors but added during static analysis to facilitate piping source and result ports to the parent of the pipeline. All these steps do is copy their inputs to their outputs (similar to `p:identity` in that respect) to facilitate bringing in data from outside the pipeline itself.

For our single branch pipeline example, this means that the !1 `p:declare-step p:input` is passing the standard input (set during `xprocxq:xq()` invoke) to `ext:pre p:input`. In the same manner, the !1! `ext:post p:output` is being piped to !1 `p:declare-step p:output` ending upon being emitted as the return result from the `xprocxq:xq()` function invoke.

These extension steps are also responsible for making sure XML documents flow correctly between steps and subpipelines.

Moving on from extension steps, lets now remark upon the `xproc:type` attribute which exist on every element. This attribute identifies an element's component type, within the XProc vocabulary; explicitly marking out elements makes dynamic evaluation simpler. The following list outlines all the available types;

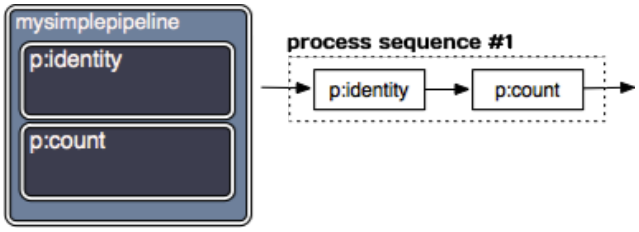
- `comp-step`: indicates a compound step
- `*-step`: indicates a (standard | optional | extension) atomic step
- `comp`: indicates an ancillary component (all other elements that are not steps themselves e.g.. `p:input`, `p:output`, `p:pipe`, `p:option`, etc...)

Finally, the `@xproc:step` and `@xproc:func` attributes are used to map the step element with the internal function that does the processing for the step, which we explain in more detail in the next section.

4.3. Dynamic Evaluation Phase

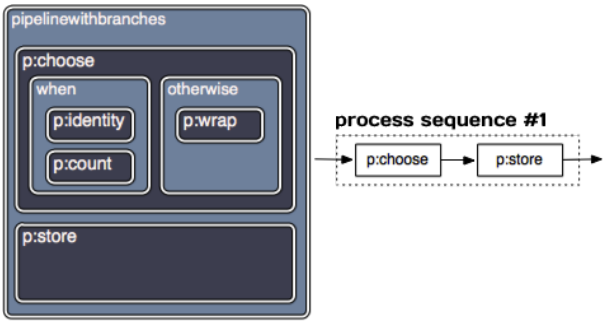
Once a decorated pipeline has been built, its the job of the evaluation stage to execute the pipeline and manage the inputs and outputs sloughing off each step of the process.

From the view of the evaluator, every type of pipeline is constructed to be an ordered sequence of steps. The diagram shows this for our example pipeline.

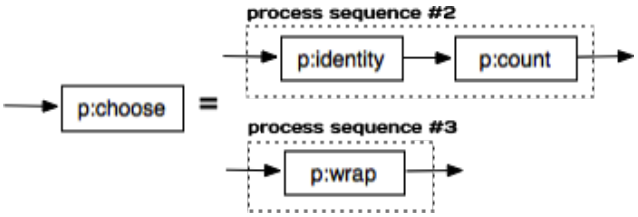


But what about pipelines that contain compound or multi container steps? One of the strengths of XProc is its ability to construct complex multi branching work flows, but how can we model these branching work flows in a form thats easy for the evaluator to process?

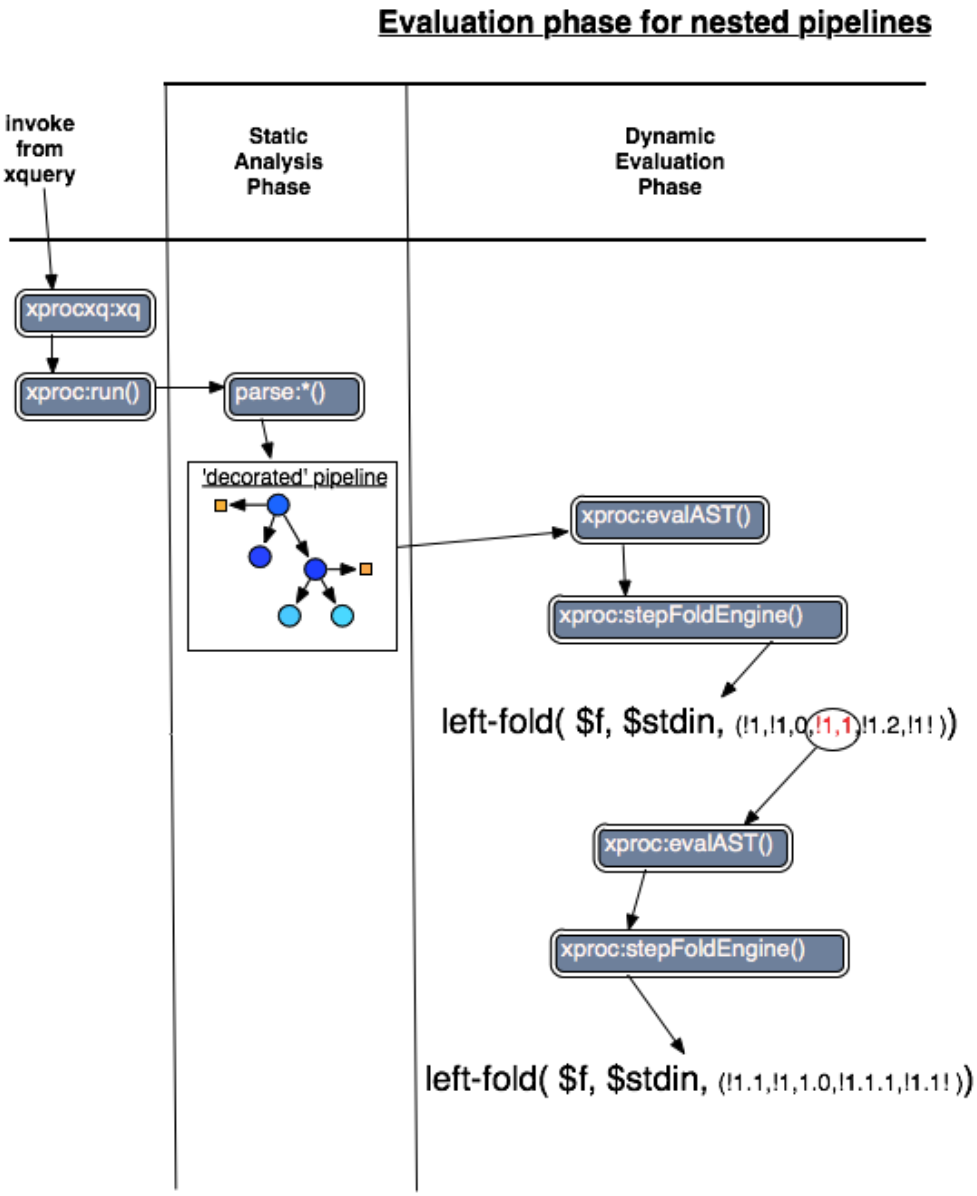
The next diagram shows a complex pipelines, containing a `p:choose` that switches to one subpipeline or the other, based on some condition. At the top level, we view each step atomically, that is we have only two steps `p:choose` and `p:store`.



During runtime, `p:choose` invokes a new instance of the evaluator of the chosen subpipeline. The subpipelines are modeled as separate ordered sequences, with only one of them ever actually selected to process during runtime.



We do not spawn or clone instances of the evaluator, execution still occurs on a single code path. The next diagram shows how nested functional composition provides the mechanism for achieving this.



Reading the diagram from top to bottom and left to right illustrates programmatic execution flow.

Once we get into dynamic evaluation, we see that the main engine is the novel application of an XQuery 3.0 `left-fold()` function. The decorated pipeline steps are represented by a an ordered sequence of steps identified by their `@xproc:default-name`. This solves the problem of how to reduce complex process sequences into a single process sequence, with itself reducing to a single final output ('turtles all the way down').

Example 9. xproc.xq evaluation stage 'engine'

```
left-fold(
  $xproc:eval-step-func,
  $starting-input, (!1, !1.0, !1.1, !1.2, !1!)
)
```

If any of the steps in the process sequence represent another process sequence (e.g. a subpipeline) our ReduceReduce algorithm naturally 'reduces' up the value as if the original step was atomic.
The parameters of the `left-fold()` function are described below;

- `$xproc:eval-step-func` - step evaluator function defined and passed in at runtime. This function is responsible for running each steps function.
- `$starting-input` - every evaluation process sequence has a starting input
- sequence of `@xproc:default-name`-. Names are unique to a pipeline and while we could have passed in the step's functions themselves it seemed to make more sense to pass around the step's ID.

You may recall that decorated pipelines defines an `@xproc:func` attribute and that is looked up for execution by `$xproc:eval-step-func` function (which itself is defined as `xproc:evalstep#4` by default). Defining at runtime, a function that runs each step function, proves to be a powerful and flexible idiom that opens up a range of interesting possibilities for redefining and enhancing step processing without amending xproc.xq codebase. (we have some fun doing just this later in the paper).

The other important responsibility of the dynamic evaluation stage is to ensure that with each step processed that its inputs and outputs are placed within the resource manager, making them available in a consistent and easy manner for other steps to use. The resource manager is vendor specific and in the case of xproc.xq running on MarkLogic [2], we take advantage of the database to store this data. With other vendors we may have to push to disk or keep process results within in memory data structures (such as `map:map` extensions in many of the XQuery processors).

Today, the resource manager just serves up and manages internal IRI's which are constructed from the XProc episode system property[1] and `@xproc:default-name` but the plan is to use the resource manager to override URI to provide xml catalog[2] like functionality.

4.4. Serialisation

xproc.xq implements a naive serialisation strategy, as all it does today is looks up from the resource manager the output result port for the step named !1!, which in the XProc naming convention is always the top level (and last) result port.

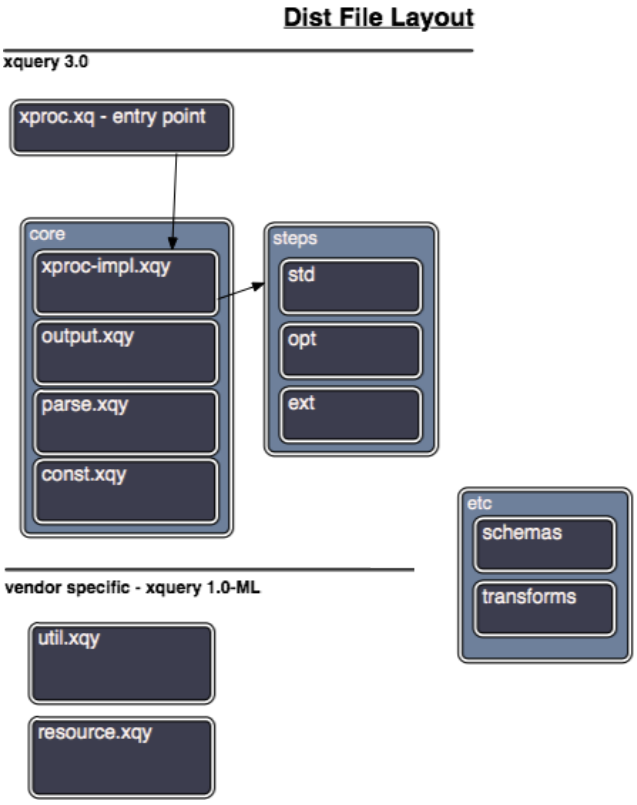
The only other responsibility of the serialisation stage is to output debug information (previously shown) which contains the decorated pipeline and a dump of all port values.

As work progresses on xproc.xq, the serialisation layer will need to become more sophisticated, especially in the area of performance as we want to be able to persist documents to the underlying database efficiently.

4.5. Code Layout

XQuery being a relatively youthful language with lightweight reuse mechanisms, means it is important to put some thought into how to design a project's code layout.

In the case of xproc.xq it was important to provide an entry point 'documented' module (xproc.xq) which is the XQuery Library module developers would import into their own XQuery projects. Having a rigid interface encapsulates the implementation, just a fancy way of saying we can make changes to xproc-impl.xqy without changing the entrypoint that other modules may invoke.



Isolating all vendor specific code within the util.xqy and resource.xqy modules assists in making it easier to provide support for other XQuery vendors going forward into the future.

4.6. A word about testing in xproc.xq

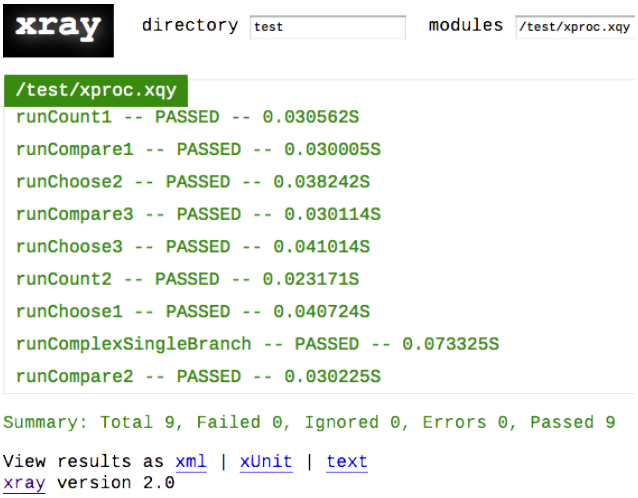
Tests for xproc.xq are contained within the `src/tests` directory of the distribution and provide coverage for individual steps, as well as end to end testing from the `xprocxq:xq()` entrypoint.
Tests are identified using a `%test:case` annotation with most tests employing the `assert:equal()` function to test for equivalence between two inputs. The following code listing shows how a typical test looks like;

[1] XProc system properties - http://www.w3.org/TR/xproc/#f.system-property
[2] XML catalog - http://en.wikipedia.org/wiki/XML_Catalog

Example 10. test:testIdentity

```
declare %test:case function test:testIdentity() {
  let $actual := std:identity(<test/>, (), (), ())
  return
    assert:equal($actual, document{<test></test>})
};
```

The test suite runner used by xproc.xq is based on the Rob Whitby's excellent XRay [11] which has both a web and command line GUI.



XRay also defines the annotations `%test:setup` and `%test:teardown` for setting up the test environment.

Overall, XRay's set of features, robustness and speed make it the perfect choice for XQuery testing in MarkLogic [2] and can highly recommend its usage.

One gap in automation revolves around running the current W3C XProc test suite which are not yet covered under the XRay system and uses a custom test runner built in XQuery. Time permitting these tests will also find their way under XRay automation.

Another known deficiency is mock testing within XProc. which could be addressed with implementation of a Resource Manager.

5. Some Design Decisions

There have been many design decisions over the past few years of xproc.xq development, but some have been more impactful then others.

5.1. XQuery 3.0 to the rescue

Introducing XQuery 3.0 into the xproc.xq project represented a significant and positive turning point in its development. In previous incarnations, using XQuery v1.0, xproc.xq had serious issues as there were subtle and hard to debug issues arising due to the usage of eval functions, which were employed to achieve flexible execution of pipelines and steps.

Problems with performance were appearing, especially where we had nested evaluation. Each invoke of the eval function raised the overall computing resources 'cost' through the need to clone an entire execution environment.

As Michael Kay observed in his 2009 Balisage 'You Pull, I'll Push: on the Polarity of Pipelines' paper [12], XQuery can be used to create 'pull' pipelines which take advantage of using function calls as the primary composition mechanism, in the case of xproc.xq we achieve this using `fn:left-fold()` to encapsulate nested function calls. The cited work also demonstrates that pull pipelines are not particularly good at broadcasting to multiple execution streams of execution, but are good at merging multiple inputs. As all branching mechanisms in XProc naturally resolve to only one execution path at any point in runtime, using a 'pull' style pipeline seems to represent a good match.

Here are just a few reasons why XQuery 3.0 is

- Using a Reducer, such as `left-fold()`, in combination with dynamic function calls underpin the heart of xproc.xq dynamic evaluation engine. It means we have no problems with performance or any of the aforementioned issues with spawned environments for evaluation.
- XQuery 3.0 annotations feature is employed to identify in the codebase step functions. As we can query which functions have this annotation at runtime it vastly simplifies xproc.xq extensibility mechanism making it straightforward to author new steps in pure XQuery.
- The choice of the 'flow' work flow model is a perfect match for a functional programming language which has functions as first class citizens. All step inputs and outputs are written once and never mutated thereafter. Changing state 'in-place' is destructive and can represent a loss of fidelity, as xproc.xq has a complete trace of every steps input and outputs it is very easy to diagnose and test.

5.2. Steps with XSLT & XQuery

As explained in the xproc.xq history section, I had embarked on large pipelining projects in early 2000's with XSLT and even had gone so far as to implement a primitive version of xproc.xq using XSLT v2.0.

As I had originally targeted xproc.xq to run in the context of an XML database, it seemed a reasonable choice to use XQuery as the implementation language (it being the stored proc language for most XML databases).

The collory to this decision was to banish XSLT v2.0 which turned out to be the wrong approach. My original concerns had revolved around using XSLT v2.0 within the dynamic evaluation phase, but I in so deciding I also opted out of using XSLT v2.0 anywhere in xproc.xq development. XSLT's polymorphism and dynamic dispatch makes static analysis hard enough if you are creating an XSLT processor and even more difficult if the static analysis is being performed at a higher application level.

XProc dependency on XSLT match patterns combined with the fact that many of the steps lent themselves to implementation using XSLT v2.0, I had inadvertently created a more difficult development path. Another aggravating factor was the gap between maturity of XSLT 2.0 and XQuery v1.0, which usually resulted in some hodge podge of XQuery v1.0 with vendor extensions.

Finally, during the transition from using Saxon [5] to Marklogic [2] XQuery processor, I changed my decision and decided to use XSLT v2.0 in the implementation of steps.

This was enormously useful as it had an immediate effect of simplifying many of the buggy step functions and enabling the development of some functions which had proved complicated to even start developing.

5.3. Nascent Resource Manager

xproc.xq implements a simple resource manager that provides durable storage of every input and output that is generated throughout the dynamic evaluation phase. Today, this resource manager is used for internal lookup of port values and is useful when debugging pipelines.

As it exists today in the codebase, xproc.xq is well placed to expose resource manager functionality directly to XProc developers, in the form of switches, options or extension steps. An enhanced resource manager could provide;

- pluggable storage api - could mean that storage backends could be easily swapped with no change to the XProc code. Absolute URI's could be overriden or new scheme's could be contemplated (ex. `resource://`).
- mock testing - being able to 'override' URI's that refer to real resources would make testing easier and provide greater coverage

There has been much discussion on the W3C XML Processing Working Group about the idea of formalising the concept of a resource manager within XProc. I am not personally convinced that the resource manager needs to be a sanctioned feature of XProc, so far its felt like an implementation detail, though this is one of those features where experience in the field should be gathered before rushing to specification.

6. Having some fun

Now that we've dived into the innards of xproc.xq and explained some of the more significant design decisions, its time to corroborate the impact of these decisions by directly experiencing xproc.xq.

6.1. Run step function in XQuery

Why let xproc.xq have all the fun ? We can run individual steps from XQuery, as long as we know what to pass to the step function.

Each step function has the same functional signature;

- primary input - input sequence of XML document(s)
- secondary input sequence of XML document(s)
- options - you will need to use specific `xproc:options` syntax for specifying
- variables - n/a

The example shows how we can apply the `p:add-attribute` step operation to an XML document;

Example 11. invoking a step in XQuery

```
xquery version "3.0";

import module namespace std =
  "http://xproc.net/xproc/std"
  at "/xquery/steps/std.xqy";

declare namespace p = "http://www.w3.org/ns/xproc";
declare namespace xproc = "http://xproc.net/xproc";

std:add-attribute(
  <test/>,
  (),
  <xproc:options>
    <p:with-option name="match"
                   select="*"/>
    <p:with-option name="attribute-name"
                   select="id"/>
    <p:with-option name="attribute-value"
                   select="'test'"/>
  </xproc:options>,
  ()
)
```

6.2. Extending xproc.xq with pure xquery steps

Creating an extension step using pure XQuery is straightforward in xproc.xq. You just add your new step function into the `src/steps/ext.xqy`, ensuring to mark it with the `%xproc:step` annotation

Example 12. new ext:mynewstep function

```
declare %xproc:step function ext:mynewstep(
  $primary, $secondary,
  $options, $variables)
{
  <my-new-step>my new step</my-new-step>
};
```

All that is required is to add this new step's definition into the internal XProc library which defines extensions, at `src/etc/pipeline-extensions.xml`.

Example 13. ext step library

```
<p:library xmlns:p="http://www.w3.org/ns/xproc" xmlns:ext="http://xproc.net/xproc/ext"
           xmlns:xproc="http://xproc.net/xproc" name="xprocxq-extension-library">

  <p:declare-step type="ext:pre" xproc:step="true" xproc:bindings="all" xproc:support="true"
                  xproc:func="ext:pre#4">
    <p:input port="source" primary="true" sequence="true" select="/"/>
    <p:output port="result" primary="true" sequence="true" select="/"/>
  </p:declare-step>

  <p:declare-step type="ext:post" xproc:step="true" xproc:func="ext:post#4" xproc:support="true">
    <p:input port="source" primary="true" sequence="true" select="/"/>
    <p:output port="result" primary="true" sequence="true" select="/"/>
  </p:declare-step>

  <p:declare-step type="ext:xproc" xproc:step="true" xproc:func="ext:xproc#4" xproc:support="true">
    <p:input port="source" primary="true" select="/"/>
    <p:input port="pipeline" primary="false" select="/"/>
    <p:input port="bindings" primary="false" select="/"/>
    <p:output port="result" primary="true"/>
    <p:option name="dflag" select="0"/>
    <p:option name="tflag" select="0"/>
  </p:declare-step>

  <p:declare-step type="ext:xsltforms" xproc:step="true" xproc:func="ext:xsltforms" xproc:support="true">
    <p:input port="source" sequence="true" primary="true" select="/"/>
    <p:output port="result" primary="true" select="/"/>
    <p:option name="xsltformsURI"/>
    <p:option name="debug"/>
  </p:declare-step>

  <p:declare-step type="ext:mynewstep" xproc:step="true" xproc:func="ext:mynewstep" xproc:support="true">
    <p:input port="source" sequence="true" primary="true" select="/"/>
    <p:output port="result" primary="true" select="/"/>
  </p:declare-step>

</p:library>
```

The library markup contains some extra attributes, which assist the dynamic evaluation stage validate a step's signature.
You will never need to use a `p:import` statement to use the extension steps as they library is loaded automatically.
The standard and optional steps are implemented in a similar manner, but these libraries should only contain steps defined in the XProc specification.

6.3. BYOSR (bring your own step runner)

The following code listing shows the overloaded version of the `xprocxq:xq()` function where you can pass it a function that evaluates each step function (the default for this is `$xproc:eval-step-func`). During dynamic evaluation, this step runner is responsible for executing each step's function.

Example 14. step runner

```
xquery version "3.0";

import module namespace xprocxq =
  "http://xproc.net/xprocxq"
  at "/xquery/xproc.xq";
import module namespace xproc =
  "http://xproc.net/xproc"
  at "/xquery/core/xproc-impl.xqy";
import module namespace u =
  "http://xproc.net/xproc/util"
  at "/xquery/core/util.xqy";

declare namespace p="http://www.w3.org/ns/xproc";

let $pipeline :=
  <p:declare-step version='1.0'>
    <p:input port="source" sequence="true"/>
    <p:output port="result"/>
    <p:count/>
  </p:declare-step>

let $stdin := (
  <document>
    <doc xmlns=""/>
  </document>,
  <document>
    <doc xmlns=""/>
  </document>,
  <document>
    <doc xmlns=""/>
  </document>)

  let $dflag    := 0
  return xprocxq:xq($pipeline,$stdin, (), (), (),
                    $dflag, 0,
                    $xproc:eval-step-func)
```

Looking up `$xproc:eval-step-func()` in `src/core/xproc-impl.xqy` we see it has 4 parameters in its signature, so we could easily rewrite as an anonymous function, as shown in bold in the amended code listing

Example 15. anonymous function as step runner

```
xquery version "3.0";

import module namespace xprocxq =
  "http://xproc.net/xprocxq"
  at "/xquery/xproc.xq";
import module namespace xproc =
  "http://xproc.net/xproc"
  at "/xquery/core/xproc-impl.xqy";
import module namespace u =
  "http://xproc.net/xproc/util"
  at "/xquery/core/util.xqy";

declare namespace p="http://www.w3.org/ns/xproc";

let $pipeline :=
  <p:declare-step version='1.0'>
    <p:input port="source" sequence="true"/>
    <p:output port="result"/>
    <p:count/>
  </p:declare-step>

let $stdin := (
  <document>
    <doc xmlns=""/>
  </document>,
  <document>
    <doc xmlns=""/>
  </document>,
  <document>
    <doc xmlns=""/>
  </document>)

let $dflag    := 0
let $tflag    := 0
let $bindings := ()
let $options  := ()
let $outputs  := ()
return
xprocxq:xq(
  $pipeline, $stdin, (), (), (),
  $dflag, 0,
  function($step, $namespaces, $input, $ast) {
    $xproc:eval-step-func(
      $step, $namespaces, $input, $ast
    )
  }
)
```

Everything works as it does before, all we've done is provide an anonymous function wrapper around our dynamic invocation.

- `$step` - contains the `xproc:default-name` for the step
- `$namespaces` - list of declared namespaces
- `$input` - usually contains primary input port value for a step
- `$ast` - the decorated pipeline is passed in and used as a kind of 'lookup' table

What if we wanted to do some additional processing with each step or preprocess any of the parameters ? We can use this anonymous function idiom to trivially insert new runtime behaviors without having to amend a line of the core xproc.xq code itself.

Example 16. enhancing step runner with anonymous function

```
function($step, $namespaces, $input, $ast) {
  let $log := u:log("processing step: " ||
                    $step || " at " ||
                    fn:current-dateTime())
  return
    $xproc:eval-step-func(
      $step, $namespaces, $input, $ast)
}
```

The example uses `u:log()` function which is a wrapper to the vendors own log function.
When we run the pipeline now and observe MarkLogic's `ErrorLog.txt` file, we see a trace of each step's name and the timestamp when processed.
Alternately, we could have opted to implement a step that does this logging, but this would be presumably a step that does nothing with the input and worst does not output anything (or just copies input to output like `p:identity` step). Just like functions with no input or return no output, steps that do nothing with the flowing XML document(s) indicate that some 'impure' processing with side effects is going on. This idiom allows you to mitigate the impact 'out of band' processing.

7. Summary

XProc provides the foundation which to create a facade over the bewildering array of XML technologies in existence today. In this sense, XProc can be the the layer which exposes a component architecture designed for provisioning reusable software.

The short term future for xproc.xq is;

- continue enhancing quality and adherence to the XProc specification
- document performance within the MarkLogic database and enhance serialisation/persistence to the data layer
- With help from the community, I would like to see xproc.xq running on other vendor XQuery 3.0 processors.
- replace parsing with solution based on Gunther Rademacher's REx Parser Generator[1]

With xproc.xq release, it now transitions from my own personal 'workbench' to something I hope others find as useful as I've found it instructive in developing.

A. Getting and installing xproc.xq

The only dependency for xproc.xq is an installed and available MarkLogic Server version 6.03 or later,

Download xproc.xq from https://github.com/xquery/xproc.xq and follow the up-to-date installation instructions contained in the `README`.

[1] http://www.bottlecaps.de/rex/

Bibliography

[1] XProc: An XML Pipeline Language, W3C Recommendation 11 May 2010
http://www.w3.org/TR/xproc/

[2] MarkLogic
http://www.marklogic.com

[3] Finite-state machine. (2013, September 10). In Wikipedia, The Free Encyclopedia. Retrieved 10:11, September 12, 2013, from http://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=572362215

[4] Zergaoui, Mohamed. “Memory management in streaming: Buffering, lookahead, or none. Which to choose?” Presented at International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth, Montréal, Canada, August 10, 2009. In Proceedings of the International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth. Balisage Series on Markup Technologies, vol. 4 (2009). doi:10.4242/BalisageVol4.Zergaoui02.

[5] Michael Kay's XSLT & XQuery Processor
http://www.saxonica.com

[6] Kay, Michael. “A Streaming XSLT Processor.” Presented at Balisage: The Markup Conference 2010, Montréal, Canada, August 3 - 6, 2010. In Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, vol. 5 (2010). doi:10.4242/BalisageVol5.Kay01.

[7] eXist XML Database
http://www.exist-db.org

[8] The xproc.xq project hosted at GitHub
https://github.com/xquery/xproc.xq

[9] XSL Transformations (XSLT) Version 2.0. Michael Kay, editor. W3C Recommendation. 23 January 2007.
http://www.w3.org/TR/xslt20/

[10] Transform.xq - A Transformation Library for XQuery 3.0. John Snelson. XML Prague 2012.
http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[11] Rob Whitby's XRay
https://github.com/robwhitby/xray

[12] 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.

Lazy processing of XML in XSLT for big data

Abel Braaksma

Abrasoft

<abel@abrasoft.net>

Abstract

In recent years we've come to see more and more reports on processing big XML data with XSLT, mainly targeted at streaming XML.This has several disadvantages, mainly because streaming is often implemented as forward-only processing, which limits the expressive power of XSLT and XPath.

In this paper I present an alternative which processes XML in a lazy manner by not fully loading the whole XML document in memory and by timely dismissing XML fragments.

We will find that this solves many document-centric XML use-cases for large datasets, while leaving the full power of XSLT at your fingertips. In addition, the small memory footprint of this method makes it ideal for scenarios such as mobile devices where memory is limited.

Keywords: XML, XSLT, XPath, big-data, lazy-processing, lazy-loading

1. Disclaimer

This paper is based on the publicly available versions of XPath 3.0, XSLT 3.0 and XDM 3.0 as of January 10, 2013 [XSLWD][XPCR][XDM3]. Since neither of these specifications is final, it is possible that references and details change before the final specification has received Recommendation status. Some items discussed here have not (yet) made it into the public Working Draft.

2. Introduction

Since the dawn of XML there have been many attempts to load XML efficiently. In this paper we are only interested in those methods that enable XML processing of large datasets for scenarios where available main memory is not enough to contain the XML in memory at once. These methods typically fall in one of the following categories:

- Streaming;
- Lazy loading;
- Succinct data structures;
- Compression.

Streaming is arguably the best known method for processing large XML documents. Its main features are high speed of parsing and constant low memory footprint. With most streaming approaches, each node is visited once in a depth-first left-to-right traversal. This means that its biggest limitation that it cannot accomodate for free-ranging traversal through the tree, for instance, a normal DOM tree would be out of the question. Several types of streaming XML exist, from SAX in Java [SAX] and XmlReader in .NET [XRDR] to the highly versatile but complex STX engine [STX]. In the XSLT domain, streaming is available as an extension to Saxon [SSTRM] and since XSLT 3.0 it is also available for compliant processors that implement this optional feature, like Saxon [Saxon] and the upcoming Exselt [Exselt] processors.

Lazy loading is a method where nodes that are not required are not loaded from the storage medium, henceforth resulting in a smaller memory footprint and, depending on the scenario, faster processing and loading times. Typically, this method works best from an XML database, but can also applied from disk. One such example is Oracle's XDB [XDB], but they call it Lazy Manifestation. I'm not aware of XSLT processors supporting this directly, but we will see that by combining streaming and classical approaches, it is possible to mimic this behavior and to get similar benefits.

doi:10.14337/XMLLondon13.Braaksma01

Succinct data structures [Jacobsen] are a way of minimizing storage required by the data structures that store references to the objects in a DOM tree. Some of these methods are briefly described below. This is a way of compressing the data structure without having to decompress them when traversing the data tree. This method was presented at XML Prague 2013 on a poster by [Joannou] and the VTD parser [VTD] is a practical available example.

Compression can be applied in many forms. A compressed tree requires uncompressing prior to processing it, which makes it less suitable for XML processing with XSLT. However, partial compression, or on-the-fly compression, for instance of text nodes only, may have a significant benefit to make the overall memory footprint smaller.

Combinations of these methods exist, again the poster by Joannou showed that on-the-fly compression of text-nodes, possibly combined with lazy-loading, can yield a memory footprint for the whole XDM or DOM to be less than 100%, in certain cases even as low as 50% of the on-disk size, without compromising on functionality.

This paper focuses on lazy loading of XML as an approach to optimizing XML processing with XSLT. We will find that current standards of XSLT, namely XSLT 1.0 and XSLT 2.0, do not provide any means for lazy loading, but with the upcoming feature of streaming in XSLT 3.0, combined with the traditional XDM approach of processing, we have powerful tools at hand to simulate lazy loading without the need for a specific XML parser that does so. However, we will also see that it depends on implementations to make use of the available information in the stylesheet.

3. The challenge

With streaming, it is possible to process virtually any size of input XML with XSLT. However, streaming XSLT is severely limited. While the XSL Working Group has gone to great lengths to make streaming as useful as possible, it is by its very nature forward-only, which makes it very hard to process a large document where the requirement is to use free-ranging expressions. In other words, when you need to look back and forward through the document from any given point in the process. References and index building are examples of these requirements.

The approach presented here will make it possible to process XML documents that do not fit in memory at once, but would fit in memory if only the nodes that are needed for the processing were loaded. This approach I call lazy processing of XML.

4. The method

With classical XSLT 1.0 and 2.0, the processor will build an XDM tree of the whole XML source document. With streaming, the XML source document is not maintained in memory but instead is loaded on the fly with only the current node, without its siblings or children, is typically kept in memory. Streaming helps primarily in those cases where the XML document does not fit in memory as a whole, or when you are reading from a source that cannot give the whole document at once, for instance when processing a Twitter or news feed.

We consider streaming a whole document without the need of processing a given node, skipping that node. Skipping a node gives the XSLT processor a hint that it does not need to load that node in memory, simply because it is not needed. This is only partially true, of course, as the position of subsequent nodes relies on the skipped node, and thus certain properties must always be loaded, but the node itself and its contents and children, are not required. In its simplest form, the following example can skip the whole document except for the root element:

```
<xsl:mode streamable="yes" />
<xsl:template match="/root">
  <xsl:value-of select="@creation-date" />
</xsl:template>
```

This example only reads out an attribute of the root node and doesn't do any further processing. We will use this example in the timings to find out the difference between a processor skipping nodes and a processor actually processing individual nodes.

A counter-example that requires all nodes to be processed is the following:

```
<xsl:strip-space elements="*" />
<xsl:mode streamable="yes" />
```

This example merely processes all nodes through the default template rules and will output it as text, skipping white-space-only text nodes.

For measuring the difference in processing speed between those two methods, skipping all, and processing all, I added a small template to remove all text nodes from the output stream, to make sure writing the output stream does not tamper with the timings. This found a remarkable difference with streaming processors, where the first example was processed in 4.5 seconds and the second in 6.1 seconds. Similar differences in timings were seen with different sizes of the input.

This observation is the basis of selectively processing nodes, which we'll examine in the next sections.

5. Filtering nodes

The essence of lazy processing XML is to not load any nodes that are not used. In its most simple form, consider that you want to filter out certain elements from an XML document. If you are not interested in these elements, you do not need to load them.

Suppose you are only interested in the chapters of a book, but not in the contents, you could've previously written a stylesheet as follows:

```
<xsl:template match="chapter">
  <chapter>
    <xsl:value-of select="@title" />
  </chapter>
</xsl:template>
<xsl:template match="paragraph" />
```

Assuming the structure of your input is as follows:

```
<chapter title="some title" />
<paragraph> some paragraph</paragraph>
<chapter title="some title" />
<paragraph> some paragraph</paragraph>
<paragraph> some paragraph</paragraph>
<chapter title="some title" />
<paragraph> some paragraph</paragraph>
```

This will effectively only output the titles. But what you are essentially doing here is telling the processor to process the paragraph nodes, but then to do nothing. This has become easier in XSLT 3.0, where we can explicitly tell the processor to ignore unmatched nodes when they are encountered:

```
<xsl:mode on-no-match="deep-skip" />
```

This declaration means that all nodes that are not specifically matched will be skipped, including all its children. This is mainly a simplified way of writing what was already possible, but it also tells the processor it can quickly skip over unmatched nodes, and ignore all its children.

However, in the classical approach, this helps us little with lazy loading, because as long as we still have to load the whole document in the XDM, the skipped nodes are still parsed. For large documents, this can be quite a performance hit. In cases where the document still fitted into memory, there was no noticable performance gain of using deep-skip.

In trivial scenario's, a processor could ignore those nodes while loading the XML into the XDM, but if a processor were indeed to do so, it will have to take extra care of the positions of nodes, because positions don't change, even when a node is skipped.

An alternative approach, where the programmer could explicitly tell the processor what nodes to skip, was aptly called xsl:filter, i.e:

```
<xsl:filter operation="ignore" select="paragraph"/>
```

However, this approach violates the node identities and the node traversal. Parsing the same document with or without a filter, would yield different node identies, because of the missing nodes. It could've been a shortcut for a micro-pipeline, but that technique is already well-known and programmers have more expressive power capturing the result of a transformation in a variable and re-processing it.

Some processors, namely Saxon, have an extension function that allows a certain kind of filtering. In the case of Saxon it is called Burst Mode Streaming [Burst] and it allows you to create a single select statement that only selects those nodes from an input stream that you require. However, the returned nodes are parentless nodes and it is not possible to traverse between the disconnected nodes.

6. Lazy loading of XML

The limitations of skipping nodes as a means for programmer-induced lazy loading do not fit its purpose. The idea of lazy loading is not to skip nodes, but to be able to process a node when we need it, and to only load it when we need it.

However, the nature of XML is that it does not maintain any information on node length, so unless the source of the XML is a system that does have this information, like a database, we will still require, whatever approach we'll take, to load each node at least once, if not only just to know where the next node will start.

There are few systems around that apply a form of lazy loading in the traditional sense. Taking the previous paragraph, it may come as no surprise that all of these systems are in fact databases. In a way, they cheat, because any XML database must have at least read the XML already. What they do is, when you create a query of the data, that they only provide you with the actual data of that query when you actually process that particular node, of element, which typically maps to a certain row in a table. Oracle takes this approach in xDB for instance.

Given the above, what can we consider lazy loading of XML if we apply the term to other systems than databases? When XML is loaded into a DOM or an XDM, a lot of time is spent on constructing the data model. Skipping constructing the data model, or not loading the node information in memory, is lazy loading XML.

7. Streaming processing of XML using XSLT

At the core of the new features of XSLT 3.0 lies one simple declaration, xsl:mode, which sets the features a particular mode must operate under. The feature of interest here is turning streaming off or on for any particular mode:

```
<!-- turn streaming on for the default,
     unnamed mode -->
<xsl:mode streamable="yes" />

<!-- turn streaming on for mode with
     the name "streaming" -->
<xsl:mode name="streaming" streamable="yes" />
```

When streaming is turned on, all template rules for this mode must be guaranteed streamable. Whether a template rule is guaranteed streamable of not is beyond the scope of this paper, but bottom line is that each template rule must have at most one downward expression and no upward epxressions or otherwise free-ranging expressions.

According to the XSLT 3.0 specification, streaming means the following [Streaming]:

> The term streaming refers to a manner of processing in which documents (such as source and result documents) are not represented by a complete tree of nodes occupying memory proportional to document size, but instead are processed "on the fly" as a sequence of events, similar in concept to the stream of events notified by an XML parser to represent markup in lexical XML.

In other words, the memory will remain constant.

8. Using streaming to achieve lazy loading

In the previous filtering example, we were trying to simply output all the chapter titles. Let's rewrite that example using streaming:

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

<xsl:template match="chapter">
  <chapter>
    <xsl:value-of select="@title" />
  </chapter>
</xsl:template>
```

Now that was easy, wasn't it? We simply mark the default unnamed mode as streamable and leave the rest as it was. This works, because the only template rule in this stylesheet has one downward expression only (actually, the attribute axis is not considered a downward expression, it is possible to access multiple attributes in a single sequence constructor).

What has changed is the mode of operation. The processor will not load the document into an XDM in the tradition way. Instead, it will now load it streamed, that is, it will not keep a memory of the tree of nodes that are being processed.

In this example, we have applied lazy loading to the whole document. One could argue, that the only node that was necessarily loaded completely, is the title attribute node, because it was needed to create the text node with xsl:value-of. It was not needed to load the whole chapter element, nor was it needed to load the paragraph elements, let alone the children of these elements. Of course, the XML parser is required to process these elements and all their attributes to at least the minimal extend required to determine the well-formedness and, in the case of schema-awareness, the validity. But it was not required, in fact it wasn't even allowed, to create an XDM tree of any of these nodes.

Let us see how this performs in our original extreme example:

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

<xsl:template match="/root">
  <xsl:value-of select="@creation-date" />
  <!-- continue processing to see
       on-no-match in practice -->
  <xsl:xsl-apply-templates />
</xsl:template>
```

The difference with the original example is that we now force the processor to process all nodes. However, we also tell the processor to deep-skip any nodes that are not matched. The result of timing this is similar to the original example in The Method section above, it is only marginally slower. This shows us that this approach gives us a strong handle on how to lazily load elements only when we need them.

Now that we know how to not load the nodes, how would we go about loading the nodes once we actually need them? How can we break out the streaming mode and load one element, and all its children, or perhaps a more fine-grained selection, into memory?

XSLT 3.0 comes with several aides to achieve this. The main approaches are the following:

- fn:snapshot
- fn:copy-of
- micro-pipelining

One might think that simply switching modes would be enough, but switching modes is not allowed. Once you are in a guaranteed streamable mode, you cannot simply break out by applying a non-streaming mode.

8.1. fn:snapshot and fn:copy

You can take a snapshot of a node and its children at any time, which copies all the nodes and makes them available for normal processing. This is the easiest way of fully loading a particular node that you are interested in. In the following example, this is demonstrated by loading only the third employee and all its children:

```
<xsl:mode streamable="yes" />
<xsl:mode streamable="no" name="non-streaming" />
<xsl:template match="employees">
  <xsl:for-each select="employee">
    <xsl:if test="position() = 3">
      <xsl:apply-templates select="fn:snapshot(.)"
                           mode="non-streaming" />
    </xsl:if>
  </xsl:for-each>
</xsl:template>
<xsl:template match="employee"
              mode="non-streaming">
  ....
</xsl:template>
```

Note the switching of modes. In the current state of the standard, this is not allowed by the streamability rules that are in the current internal draft of the spec. The previous, and current public Working Draft does allow this type of processing though. It is not sure whether this will become possible or not in the final specification.

There's currently an easy way out of this. The trick is to change the context in the for-each loop to be the context of the result of the snapshot. For instance, you can rewrite the above example as follows:

```
<xsl:mode streamable="yes" />
<xsl:mode streamable="no" name="non-streaming" />
<xsl:template match="employees">
  <xsl:for-each
       select="employee[position() = 3]/copy-of()">
    <xsl:apply-templates select="."
                         mode="non-streaming" />
  </xsl:for-each>
</xsl:template>
<xsl:template match="employee"
              mode="non-streaming">
  ....
</xsl:template>
```

Note that we moved the if-statement inside the XPath expression. If we did not do so, the benefits of selectively copying only the needed elements would be void.

8.2. Micro-pipelining

Micro-pipelines have been possible since XSLT 2.0. In its simplest form, they take a (modified) copy of data into a variable and process it again. In the case of streaming, the possibilities of micro-pipelining get a new dimension and make it possible to effectively create very fine-grained lazy loading XSLT transformations.

An example of micro-pipelining:

```
<xsl:variable name="filtered">
  <xsl:stream href="feed.xml">
    <xsl:for-each select="news/news-item[
                            @date > '2013-06-10']">
      <xsl:copy-of select="." />
    </xsl:for-each>
  </xsl:stream>
</xsl:variable>
<xsl:apply-templates select="$filtered/news-item" />
```

This example lazily loads all news items, and then creates an XDM of only the items after June 10, 2013.

9. More advanced lazy loading

A prime example where lazy loading can be applied is when the requirements clearly state that we do not need to look at all elements in the source document, and when the selection of these skippable elements form a significant part of it. For instance, consider that you are creating a document containing only the abstracts and titles of all papers, and all papers are inside one document. If the XML is similar to DocBook, you can make use of the deep-skip method examples like provided above.

If we expand the requirements a little further and only want to show the abstracts of papers that have been written by a certain author, simply using deep-skip will not be possible, because we cannot create a pattern that tests the contents of text nodes (this is considered a free-ranging expression, because the whole text node must be loaded to know whether it matches a certain string). In this scenario, the copy-of function comes to our help, as it allows a free-ranging expression and the expression can be used by the processor to return only a limited set of nodes. For instance:

```
<xsl:copy-of
  select="article/info/copy-of()[
            author/personname = 'John'
          ]/(abstract | title)" />
```

The predicate, as above, without the fn:copy-of function, would be consider free-ranging because it requires processing the personname elements and its children. By adding the fn:copy-of function, this works, as the function creates a copy of the current node which can then be processed in the normal, non-streaming way.

10. Always lazy loading XML with XSLT

It would be nice if we had a way to process any XSLT using streaming and lazy loading automatically. However, the complexity of the possible scenarios makes it impossible for an automated process to determine statically how the best lazy loading approach should work. Even more, this is the main reason why you cannot simply turn every stylesheet into a streaming stylesheet.

For instance, if your requirement is to visit the preceding-sibling axis often, or, worse, you need to use the preceding axis in a predicate inside a pattern, it is unlikely that streaming or lazy loading will help. Similarly, if you need to find a string in any node, there's no way lazy loading will help, because you need to visit every node anyway.

When dealing with large documents, one needs to limit oneself. Using the lazy loading technique, you can achieve the best of both worlds. Instead of using free-ranging expressions on the whole tree, you need to consider filtering the input tree to a smaller subset, or if that is not possible, use streaming and lazy loading to specifically load the parts only when you need them.

11. Limits of lazy loading XML

While there is no limit to the size of the input XML when using streaming, lazy loading has its limits. In fact, you have to be really careful. In particular, you need to consider the following rules:

- Do not use snapshots when your stream should run indefinitely, unless you've confirmed that your processor discards the used memory correctly;
- The size of the snapshots multiplied by the maximum amount of taken snapshots should not exceed available memory;
- Minimize the size of the snapshots by micro-pipelining them and removing all unnecesary data from the stream
- Do not use snapshots in inner loops

While the first rule is obvious (indefinitely running input stream will yield indefinite required size for snapshots), the second one can be hard to measure. Depending on the processor and the chosen XML parser, as a rule of thumb, take the size on disk and multiply that by 2.5 to 4.0. Multiply the result with the expected amount of snapshots to be taken and you should have a fair guess of the memory needed.

If your processor has an extension function to discard the memory used, or to flush the output stream to disk, your reach is virtually unlimited.

12. Lazy processing performance

To determine the performance of lazy loading, I've created several scenario's with a large input file, which I processed in one of four ways:

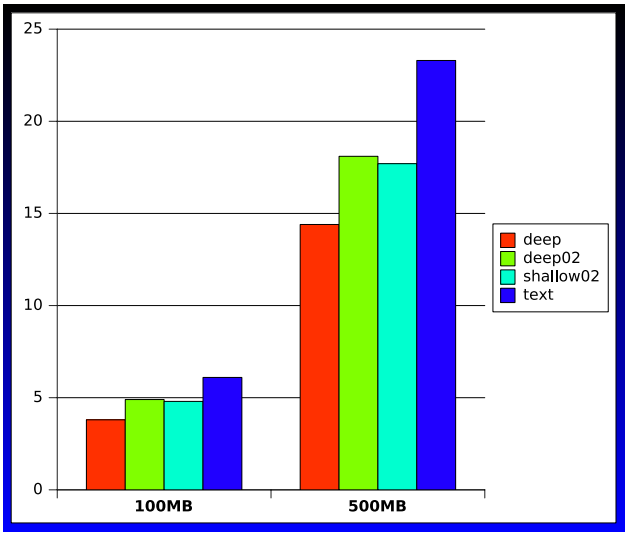
- Deep: use deep-skip, only read an attribute of the root element of the source document, this is the minimum possible transformation.
- Deep02: deep-skip, processing 2% of the elements of the source document, a typical use-case where only a small part of the original document needs to be processed.
- Shallow02: shallow-skip, processing 2% of the elements of the source document, same as previous, but forces the processor to go over each node to find a match.
- Text: the same transform as above, but this time the non-matching nodes are matched and a one-character string is output to prevent the processor from optimizing the match away.

All transformations are equal, except for the last one, where an addition match="node()" is added. The other differ only in the setting of the on-no-match attribute of the current mode. The first one however uses a deliberate non-selecting apply-templates and only matches against the root node. This we use to have a baseline to compare the measurements to: it is the simplest transformation possible, and it measures how fast the processor dismisses the input document once it knows that the select statement selects nothing.

As input for the transformation I used an XML corpus of all Shakespeare's plays. The combined plays where copied in one XML file, which surmounted to about 10MB and then copied again until the sizes 100MB, 500MB and 5GB were reached. This file is relatively document-centric, which is a requirement for good tests, because using a vanilla XML file gives the processor a lot of chances to read-ahead, which does not give very meaningful results.

The processor used for all tests was Saxon 9.5.0.2 for Java and it was run on a 2x 3.3GHz Xeon 48GB Dell Precision Workstation with a Windows 7 64 bit operating system. To eliminate timer errors, each test was run 5 times, the lowest and highest values were removed and the other three arithmetically averaged. All results are shown in seconds.

Let us look at some graphs. The first graph shows processing a 100MB and a 500MB input file using traditional in-memory non-streaming processing. The deep02 and shallow02 bars are noticeably of the same size. This means that the processor doesn't optimize the scenario of deep-skip, or, perhaps more likely, because the document is already loaded in memory anyway, there is no benefit for deep-skip versus shallow-skip. The test of the 5GB document is not included, because it didn't load in the available memory. The largest measured memory size was for the shallow02 test, the processor took about 4.1GB.

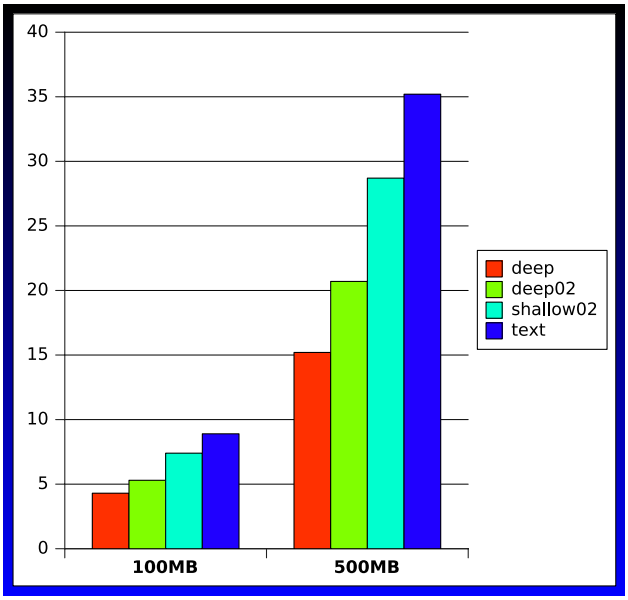


This gives us a good starting point to compare the in-memory processing to the streaming processing. I ran exactly the same tests, except that the modes were switched to streaming. Three things became immediately apparent:

- Processing with streaming takes longer, on average about 20-25%
- Memory remains constant on about 200MB (though still quite high, I didn't try to force the Java VM to force less available memory)
- On the longer runs, memory slowly dropped during the process to about 140MB on the longest run. I have not been able to find a satisfying explanation for this behavior.

That streaming takes longer may come as a surprise. However, streaming, and more particularly streaming using the XSLT 3.0 standard, is fairly new. In fact, the previous version of Saxon, 9.4 didn't yes support the streaming feature (it did support streaming extensions though). Many features of streaming, in particular the ones related to guaranteed streamability analysis, are still heavily under development. As Michael Kay mentions himself in [Kay], features first, performance later. In theory, streaming can be much faster than in-memory processing, especially when the stylesheet is designed with streaming in mind.

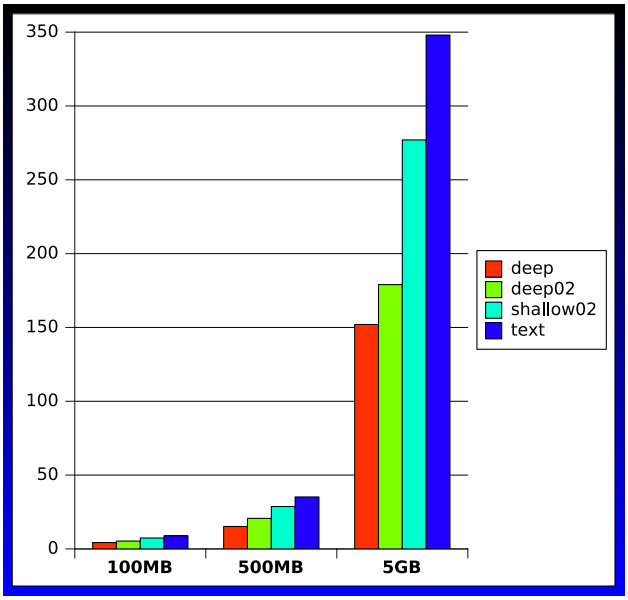
The main reason for looking at the streaming timings, is to find the difference of lazy processing or loading XML nodes. What happens when we skip nodes? What happens when we only load a subset of the nodes? The deep02 and shallow02 only touch a subset of the nodes and skip the rest:



The graph shows a very clear performance gain when deep-skipping nodes in a streaming process. This stylesheet mimics the behavior that you find when you deliberately use fn:copy-of or fn:snapshot, but I wasn't able to use these functions correctly, the implementation still treats too many scenarios as not streamable. Hence I stuck to using the more straightforward approach of skipping nodes. We can conclude three things here:

- Processing all nodes versus processing one node takes double or more time (note that we only measure the "touching" of nodes by matching them, we don't actually do anything complex with the nodes, which would clearly make this distinction even bigger)
- There's an expected and noticeable performance difference between using deep-skip and shallow-skip. Even in a scenario where only 2% of the input document is actually matched, deep vs shallow skipping shows a big difference. It is more challenging to write a stylesheet for deep-skipping, but it is worth the effort when processing large documents.
- Lazy loading has a clear advantage over processing all nodes. Thinking carefully about the design of an XSLT stylesheet becomes more and more important, because a naive implementation may easily load too many nodes, which costs the processor too much time to process. This difference is much more apparent than in in-memory processing.

Let us look at one more graph to see what happens when we try to load a really large document of 5GB:



The document processed here was exactly 10x the size of the 500MB document. The timings are almost exactly 10x as long as well. In streaming environments, this lineair performance behavior is very expected, because of the way streaming works. With in-memory processing, many performance comparisons that have been done over the years, have shown a non-lineair performance graph [Zavoral].

13. Further improvements using succinct data models

Beyond the XSLT 3.0 standard, more improvements are possible. These improvements must be done on the XML data model. Several researches and implementation exist currently, that apply the succinct data model principles to XML loading.

During the International Symposium on Information Processing 2009 in Huangshan, China, Yunsong Zhang, Lei Zhao and Jiwen Yang presented a method for reusing the parsed XML tree to optimize repetitive processing [Zhang]. They called this R-NEMXML and the essential idea they presented was to encode the node information in a 64 bit integer and store the actual data of the tree elsewhere. The integer contains type information and an offset to the actual node in another storage medium. There paper was a follow up on NEMXML, presented in June the same year, which showed a non-extractive method of processing XML.

In XML Prague in February 2013, Stelios Joannou, Andreas Poyias and Rajeev Raman of the University of Leicester presented a poster about their SiXML (Succinct Indexing XML) parser [Joannou], which took the NEMXML idea one level further to provide space-efficient data structures for in-memory parsing of large XML. They did so by storing the data structures in a pointerless data structure as parenthesized strings, known as succinct data structures [SiXML]. SixDOM and SiXML was pioneered in 2009 by O'Neil Delpratt in his thesis at the University of Leicester [Delpratt]. Earlier work similar to this, but using a different approach can be seen in the VTD XML parser as mentioned in the introduction, which came forth from the 2003 concept presented by XimpleWare. VTD stands for Virtual Token Descriptor [VTD] and stores the location of data of nodes using an offset, as opposed to using objects to represent the data, which has a relative overhead that bloats the DOM model.

For our processor [Exselt] we are currently investigating the SiXML model as an updateable model (it is currently read-only) and we try to expand that using lazy loading of the data content of nodes, dependent on the structure of the XSLT, as described in the previous sections in this paper.

14. Conclusion

We have seen that XSLT 3.0 provides several ways to lazily load elements, or to at least lazily process them. While certain environments provide lazy loading out of the box, especially in certain database systems, when it comes down to processing input from a memory stream, a local or a remote file, lazy loading was not yet a feature. With the addition of streaming to XSLT 3.0 came a few other enhancements to the language that facilitate handling large datasets, that make it possible without using extensions to load a node, with all its children no demand or to skip nodes by using filtering based on the available matching templates and the settings in the current mode.

By testing these assumptions using a real-world scenario, we've shown that lazy loading in streaming processing has significant benefits. While the processors still need to optimize this kind of processing, it is very encouraging that in a simple comparison of different approaches, we see already gains of 50% and more.

If these approaches can be combined with an XML parser that can serve content of nodes lazily, the resulting processing speed and required memory may go even further down.

Bibliography

[Burst] *Burst Mode Streaming*. http://saxonica.com/documentation9.4-demo/html/sourcedocs/streaming/burst-mode-streaming.html.

[Delpratt] *Space efficient in-memory representation of XML documents*. PhD. thesis at University of Leicester https://lra.le.ac.uk/handle/2381/4805. O'Neil Davion Delpratt. 2009.

[Exselt] *Exselt, a concurrent streaming XSLT 3.0 processor for .NET*. http://exselt.net. Abel Braaksma.

[Jacobsen] *Succinct static data structures*. Ph.D. thesis, Pitssburgh, PA, USA. G.J. Jacobsen. 1988.

[Joannou] *In-Memory Representations of XML Documents with Low Memory Footprint*. Poster XML Prague 2013, part of SiXML. Stelios Joannou, Andrieas Poyias, and Rayeev Raman. 2013.

[Kay] *Streaming the identity, confusing running times*. http://saxon-xslt-and-xquery-processor.13853.n7.nabble.com/Streaming-the-identity-confusing-running-times-td11839.html.

[SAX] *Simple API for XML*. http://www.saxproject.org/.

[Saxon] *Saxon by Saxonica*. http://saxonica.com. Michael Kay.

[SiXML] *Succinct indexable XML*. http://www.cs.le.ac.uk/SiXML/.

[SSTRM] *Streaming in Saxon using saxon:stream*. http://saxonica.com/documentation9.4-demo/html/sourcedocs/streaming/.

[Streaming] *Streaming definition of XSLT 3.0*. http://www.w3.org/TR/xslt-30/#streaming-concepts.

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

[VTD] *VTD-XML: Virtual Token Descriptor*. http://vtd-xml.sourceforge.net/.

[XDB] *Lazy Manifestation in Oracle's XDB*. http://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdb10pls.htm.

[XDM3] *XQuery and XPath Data Model 3.0, W3C Candidate Recommendation 08 January 2013*. http://www.w3.org/TR/2013/CR-xpath-datamodel-30-20130108/. Norman Walsh, Anders Berglund, and John Snelson.

[XP3] *XML Path Language (XPath) 3.0, Latest Version*. http://www.w3.org/TR/xpath-30/. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[XPCR] *XML Path Language (XPath) 3.0, W3C Candidate Recommendation 08 January 2013*. http://www.w3.org/TR/2013/CR-xpath-30-20130108/. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[XRDR] *XmlReader .NET BCL class*. http://msdn.microsoft.com/en-us/library/system.xml.xmlreader.aspx.

[XSLT3] *XSL Transformations (XSLT) Version 3.0, Latest Version*. http://www.w3.org/TR/xslt-30/. Michael Kay.

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

[Zavoral] *Perfomance of XSLT Processors on Large Data Sets*. ICADIWT '09. Second International Conference on theApplications of Digital Information and Web Technologies http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5273945. Filip Zavoral. Jana Dvorakova.

[Zhang] *R-NEMXML: A Reusable Solution for NEM- XML Parser*. International Symposium on Information Processing 2009 in Huangshan, China http://www.academypublisher.com/proc/isip09/papers/isip09p155.pdf. Yunsong Zhang. Lei Zhao. Jiwen Yang.

Using Distributed Version Control Systems

Enabling enterprise scale, XML based information development

Dr. Adrian R. Warman

IBM United Kingdom Limited

<Adrian.Warman@uk.ibm.com>

1. Disclaimer

Any views or opinions expressed in this paper are those of the author, and do not necessarily represent official positions, strategies or opinions of International Business Machines (IBM) Corporation.

No guarantees are offered as to the timeliness, accuracy or validity of information presented.

2. Introduction

Enterprise scale technical writing, or "information development", typically requires many people, working on many aspects of content, across many time zones. At this level, traditional "word processor" tooling simply cannot cope, particularly where more than one person must work on the same or closely related content. By contrast, XML-based information development, where content is divided into discrete information units, solves many of the problems. But not all of them.

In particular, where several people must work on the same files concurrently, or where the timescale for the required work might conflict with other delivery obligations, technical writers can find themselves in situations where updates might be lost or broken.

These are classic problems for software developers, too, and it turns out that some software development techniques can usefully be applied to the problem of managing XML-based documentation at an enterprise scale. Specifically, Distributed Version Control System (DVCS) tooling can be applied to everyday documentation tasks. In this paper, we explain how a DVCS might be used by a product documentation team to help deliver diverse documentation sets, created using XML, for large scale parallel delivery.

3. Definitions

It is helpful to clarify exactly what is meant by specific terms and concepts mentioned within this paper. We prefer straightforward, pragmatic definitions to more formal, rigorous definitions.

- **Information Development** refers to the combination of resources and processes used to create content that is readily usable by the intended audience. So, if a technical writer uses a word processor to create some notes about how to install a software package, that is information development. If a software developer creates some extended comments within application source code, and those files are run through an automatic documentation generation tool (such as `javadoc`) to create HTML reference guides, that too is information development. If a marketing team creates some visually impressive video files that cannot be used by someone with a visual impairment, that is *not* information development.
- An **Information Set** is a logically-distinct collection of content that has been created or assembled to address a specific concept, task or purpose. A manual for a product running on a specific platform is an information set. A chapter on installing a product is an information set. A collection of documentation source files containing all the materials necessary to produce an information set is an **information stream**.
- **DITA** is the Darwinian Information Typing Architecture, and is a form of **XML markup**, where well-formed and valid XML files are used to hold human-readable documentation source. This enables enterprise scale work, such as reuse of content, while applying strict controls to help ensure consistency.
- A **Distributed Version Control System**, or DVCS, is a peer-to-peer file management tool. It helps track and manage changes to files, with no single repository being the definitive storage point. The distributed nature means that the ability to merge, and manage merges, of source files is essential. Each DVCS user has an entire copy of the source repository, making it possible to carry out substantial work 'offline'. Traditionally aimed at software development, the application of DVCS to information development provides some interesting and significant benefits, as suggested by this paper.

doi:10.14337/XMLLondon13.Warman01

4. A simple example of workflow

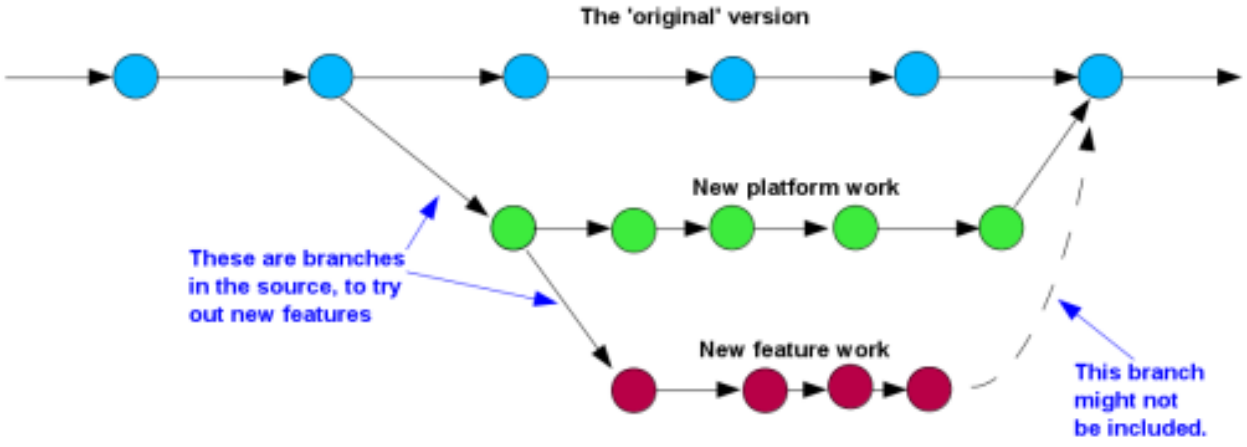
Workflow in enterprise scale information development is remarkably similar to software development. Assume that a new feature is being implemented within a product. This feature will be the subject of specification and design documents. These describe details such as implementation constraints, performance targets, test cases, and so on. Similarly, if the feature must be documented because it is usable by customers, there will be usage considerations such as the target audience, the level of detail required, whether to include examples and to what level of detail, what output format to use, and so on. Other essential considerations include translation and accessibility compliance, but these are not discussed further in this paper.

As a simple example, let us assume that an established product, called 'ProductEx', is about to be released on an additional hardware platform. This requires substantial changes to the product documentation, such as platform-specific installation and troubleshooting details. At the same time, a new feature called 'FastMemEx' is being introduced and will be available on the established and new platforms. The feature modifies the behavior of the software product to make it run faster, but a trade-off is that the software requires more memory to operate. It is possible that the performance testing carried out during development might conclude that FastMemEx is not ready to be included with the next release of ProductEx, and would be deferred to a later date. Waiting until the FastMemEx"go or no-go" decision is made would not leave enough time for the documentation to be written, therefore it must be possible to remove the FastMemEx documentation quickly if necessary.

Even this minimal example makes it clear that there are often substantial information development tasks in a project. It represents a typical scenario for information development. In addition to any other development or maintenance work, the example identifies two distinct new information streams, each of which has implications for the other stream. Ideally, both these new streams will be delivered, but it is possible that the product component associated with one stream might be deferred, and therefore the updates for that component must be held back.

We can represent this scenario diagrammatically, as shown in Figure 1, "Flow of streams and merging".

Figure 1. Flow of streams and merging



The problem for technical writers is how to manage each of these challenges.

5. Basic XML content creation using DITA

Part of the solution is to use an appropriate documentation source format. As mentioned above, DITA is a well-formed XML based content markup language. It is easy to find more details about DITA, but for the purposes of this paper, the following two example files adapted from samples provided with the DITA Open Toolkit implementation are helpful.

quickstart.ditamap

```
<?xml version="1.0"
  encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN"
  "map.dtd">
<map xml:lang="en-us">
  <title>Getting started</title>
  <topicref
    href="quickstartguide/exploring-the-dita-ot.dita"
    collection-type="sequence">
    <topicref
      href="readme/installing-full-easy.dita"/>
    <topicref
      href="quickstartguide/rundemo.dita"/>
    <topicref
      href="quickstartguide/runmore.dita"/>
  </topicref>
</map>
```

exploring-the-dita-ot.dita

```
<?xml version="1.0"
  encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN"
  "task.dtd">
<task id="exploring-the-dita-ot">
  <title>
    Getting Started with the DITA Open Toolkit
  </title>
  <shortdesc>The
    <ph><cite>Getting Started Guide</cite></ph> is
    designed to provide a guided exploration of the
    DITA Open Toolkit. It is geared for an audience
    that has little or no knowledge of build scripts
    or DITA-OT parameters. It walks the novice user
    through installing the full-easy-install version
    of the toolkit and running a prompted build.
  </shortdesc>
</task>
```

The ditamap in `quickstart.ditamap` provides navigational and structural information. It determines which 'topic' files appear in the final publication, and in what order.

The dita content in `exploring-the-dita-ot.dita` is actual 'product' documentation. Even without knowing any DITA, you should be able to recognize content such as the topic title, a simple paragraph, and a citation reference to another publication.

These files can be created and modified using any text editor. Ideally, you would use a tool that is XML-aware, so that the proper checks are in place to ensure that the files are well-formed and valid. There are also some more advanced tools that go some way towards WYSIWYG presentation, although in practice these can be more frustrating than helpful because many enterprise scale documentation projects make use of attribute tagging to conditionalize the build according to a specific product, platform, audience, and so on.

The key point about these examples is that they show 'simple' text files as the basis of extremely large, enterprise scale, documentation. The use of XML as the 'hosting' structure helps ensure consistency and reliability in creating output deliverables.

6. DVCS principles

As we saw in the earlier diagram, when writing enterprise scale documentation, you are very likely to encounter situations where two or more people are working on the same documentation set. Often, they will work on different files, but from time-to-time they need to update the same file at the same time. Anecdotally, but not surprisingly, many of the changes tend to be within files that are in close 'proximity' to each other in the overall structure, rather than being distributed 'randomly' throughout the content.

This need to have several people working on a common set of files is not unique to technical writers. Software developers encounter exactly the same challenges. They have tackled the problem by extending the concept of a Version Control System (VCS) into a Distributed Version Control System (DVCS).

In its simplest form, a VCS stores historical records of files in a repository. Each time a file is modified, a new copy of the updated files is stored or "checked in" to the repository. The main objectives are to make it easy to backup all the work, and to be able to "roll back" to earlier versions of the files.

A VCS traditionally has a single, central server to host the repository. This makes it easy to backup the content at regular intervals. However, it means that people working on the files must first copy them to their local machine ("check out"), make any changes, then upload the changed files ("check in") back to the central server.

The VCS model does have limitations. In particular:

1. You must have effectively continuous network connectivity to check out or check in to the server.
2. It is complicated to deal with different threads of development on your local machine.
3. There is a risk that your changes to a file will be overwritten by changes made to the same file by another developer, who checks in their changes after you have checked in yours.

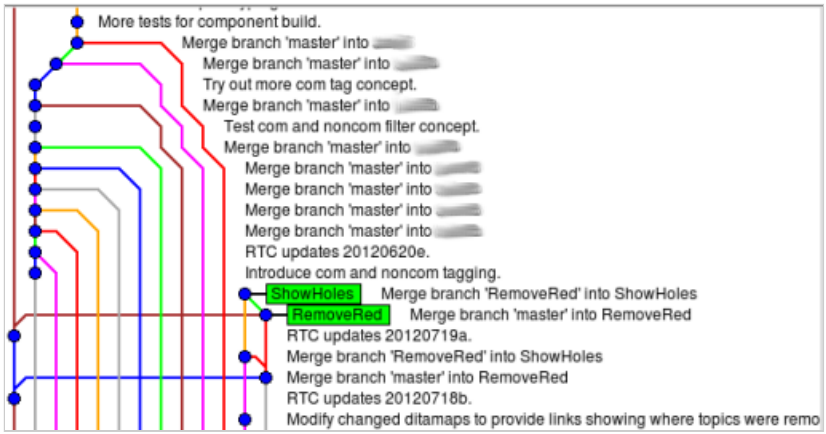
One solution to these problems is to use a DVCS. A key characteristic of a DVCS is that *everyone* has a *complete copy* of the entire repository. Surprisingly, this can be more cost-effective in terms of storage space than a traditional VCS, for reasons explained in Appendix A. Any changes made to files locally must be available to everyone else who has a copy of the repository. This means that a DVCS is intentionally designed to share file updates between the repositories, and to accommodate changes made to the same file by different developers. The sharing works in a way that:

- Is as quick and efficient as possible.
- Makes the merging of separate changes to the same file automatic.
- Ensures that any problem merging the changes results in a "fail gracefully" report.

The last point is especially important. If something does go wrong during the merge, none of the changes are lost, and you are able to work on the file directly to resolve the problems.

In Figure 2, "Merging of documentation streams", you can see different threads of development work from a real documentation project using a DVCS called **git**. At frequent intervals, the changes in a given thread can be seen merging back into other threads.

Figure 2. Merging of documentation streams



7. Using a DVCS for multi-branch content

Note

In this paper, we do not have the space to describe DVCS operation in detail. For more information about typical DVCS tools and tasks, see either git or Mercurial.

In a DVCS such as **git**, each of the work streams is created as a 'branch', where the current collection of files is duplicated in a separate development strand. This branch can be edited independently of the original files. At any time, you can save the current state of the files in a branch, and then switch to any other branch. There is no specific limit to the number of branches you might create; in practice, creating too many branches would make it difficult to keep track of them yourself.

Branching means that each task can be split off into its own stream of activity. You can safely work on all the files within the branch, knowing that any changes you make can be re-integrated into the main branch later.

In **git**, the main branch is called `master`. To start working on the task of documenting a new feature, you might create a new branch as follows:

```
git branch MyNewFeature
```

However, nothing obvious happens until you 'switch to', or "check out" the new branch:

```
git checkout MyNewFeature
```

You have now taken copies of all the files in your project for use in the new branch. You can proceed to create and update the various files associated with the new feature.

Note

A DVCS does not normally duplicate files when a new branch is created. A duplicate file is created only when the original file is modified. This makes branch creation and switching quick and easy.

Every so often, you save your work by checking in the changes. In **git**, this is called making a "commit". A typical commit action might look something like this:

```
... various file editing tasks
git add <list of new or modified files>
git commit -m "Corrected the syntax explanation for the new feature."
... continue editing files.
```

Similarly, you will want to bring in the other changes that colleagues have made to documentation files in their branches. Assuming they have checked in their changes to the `master` branch, you can bring those changes into your files by using the command:

```
git merge master
```

All being well, the external changes are merged safely into your files. Eventually, however, a problem is likely to occur, where you and your colleagues have been trying to change the same section of the same file. When this happens, the DVCS might not be able to resolve the discrepancy. Instead, it reports that the merge has failed, and tells you details of the affected file or files. For the merge to complete, you must edit the broken files and fix the problem. For any broken file, **git** makes all the necessary information available to you, including:

- The file contents as they were before you or your colleague made any changes.
- The changes *you* made to the file.
- The changes *your colleague* made to the file.

Using these three details, you should have everything you need to solve the merging problem. At worst, you know who else has been editing the file, and so could talk to them about the specific changes and how you might come to an agreement on what the final content should be.

After going through several cycles of editing files and merging in changes from colleagues, you are ready to merge your changes back into the `master` branch. To do this, you switch back to the master branch, then merge in the changes from your working branch:

```
git checkout master
git merge MyNewFeature
```

It is important to realize that documentation files might be correct or incorrect according to three precise measures:

1. Whether the file is well-formed.
2. Whether the file is valid.
3. Whether the file is semantically meaningful.

A DVCS cannot assist you with the third measure, not least because a (comparatively) simple tool like a DVCS cannot interpret the vagaries of human communication. However, the way in which a DVCS can support the first and second measures suggests a possible future development that might one enable support for semantic assessment.

Most DVCS systems can be extended and modified in many ways; customization was an important principle in developing **git**. For example, when you check in your updates using the **git commit** command, some extra tasks can be run using an extension called a "hook". A very useful application of this is to run some tests on the files as they are prepared for check in. In the following code segment, the command-line utility **xmllint** is applied to each DITA file just before it is checked in. The script ensures that only DITA files are tested in this way.

```
for FILE in `exec git diff-index --cached \
  --name-status $against | egrep '^(A|M)' | awk \
  '{print $2;}'` ; do
if ( echo $FILE | egrep -q '.+\.(dita|ditamap)$' )
# if the filename ends in .dita or .ditamap
then
  xmllint --valid --noout $FILE > /dev/null 2>&1
  RESULT=$?
  # echo "Checking $FILE, return code: $RESULT"
    if [ $RESULT -ne 0 ]
    then
      EXIT_STATUS=1
      echo "Invalid DITA markup: $FILE"
    fi
    # echo "EXIT_STATUS now $EXIT_STATUS"
# else
#   echo "Ignoring $FILE"
fi
```

The **xmllint** command checks that a given file is both well-formed and valid according to the XML catalog defined on the system. In this case, the DITA DTDs have been included in the XML catalog, which means that every time you check in your DITA files, they are automatically tested to be well-formed and valid. A potential enhancement would be to modify the **git** hook so that semantic analysis might also be performed.

8. Using a DVCS to manage multiple deliveries

A common reason for using DITA and similar markup tools for enterprise scale documentation tasks is that they support conditional or filtered building of content. This is an exceptionally useful capability, and is based on the use of XSL-style pattern matching to include or exclude content, according to precise conditions.

A good example is where a content file is created that describes a task in general terms, but also includes platform- or version-specific details. One option for the final document delivery would be to build a single documentation set that includes all of the platform- or version-specific information. Within this large collection, each instance of (say) platform dependent content might be clearly identified using a small icon or logo. However, given that markup must be included anyway to identify content specifics, a second option is to have entirely separate builds: one for each of the specific variants. The result is a much larger number of documents, but each one is specifically tailored for a precise audience.

What we are describing here is another way of viewing the documentation streams. Previously, we have identified a main documentation stream (`master`), with separate branches established for clearly defined update tasks. These branches are then merged or reintegrated back into `master` as required.

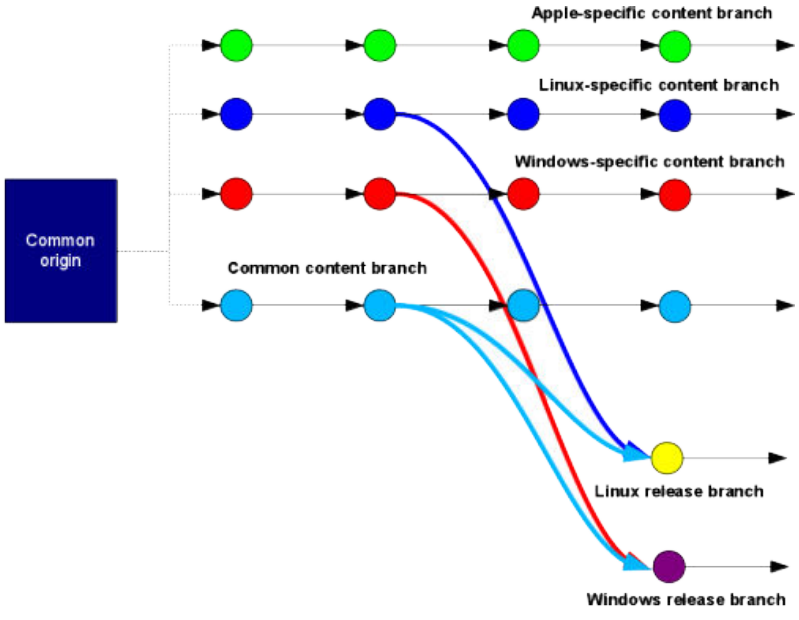
But the branching model of a DVCS allows us to adopt a different model, where content that is common to all documentation is present in one branch, and content that is unique to a given platform or product version can be isolated off in a separate branch. This concept can be seen diagrammatically in Figure 3, "Use of DVCS for multiple deliveries".

It is important to note that this diagram is *not* illustrating a deployment model; indeed some DVCS specialists actively discourage the view that a DVCS should be used for deployment purposes, where the existence of branch represents the live content on a production system. Rather, the diagram shows that content may be assembled to produce some release-ready material, in much the same way that ingredients are assembled prior to working through a recipe to create a meal.

Using a DVCS for multiple deliveries offers several advantages in comparison to the traditional single-stream content model, over and above those already described. For example, re-use of content is much each easier to achieve, promoting consistency and simplicity. The assembly-on-demand approach to multiple deliveries has some similarities to a Just-In-Time delivery model; there is no need to 'retire' a documentation stream if it is no longer required - instead you simply omit it from the final build process.

At the same time, it must be admitted that using a DVCS in this way is pushing the boundaries of what was originally intended. In a small scale, real project experiment, where content for multiple deliveries was assembled prior to deployment on a hosting system, the mechanism worked well. However, the difference in comparison to the earlier description of using DVCS is that there is no requirement to store the resulting merged content. Therefore, if there had been problems during the merge, it is possible that the effort required to resolve the problems might have exceeded the benefit of separating out the deliverable streams. It will be interesting to see whether larger scale tests of DVCS-based multi-delivery remain viable.

Figure 3. Use of DVCS for multiple deliveries



9. Summary

In this paper, we have reviewed the basic principles of enterprise-level documentation tasks. We have outline some of the challenges faced when producing documentation that comprises different strands of work, such differing content or timescales. We have shown how some of these challenges reflect similar problems encountered when managing large scale software development concept. We have introduced a software-oriented solution: Distributed Version Control Systems. Using these building blocks, we have explained how DVCS technology might be used to help enable enterprise scale, XML-based information development.

A. Storage requirements for a local repository

A common example of a VCS is **subversion**. This tool provides a central server that hosts a repository for all the working files of a project. At any time, a developer might use a client program (**svn**) to check out a 'snapshot' of the project files from a precise point in time. To switch to an older or more recent snapshot, the developer uses the **svn** command again to request the correct files from the server. This means that a **subversion** user only ever has a single set of files, and must communicate with the server to switch to a new set. If there is no network access, no switching is possible.

By contrast, a DVCS user has a complete set of all files in the repository. It is a quick and easy task to switch to any other snapshot or indeed branch within the repository, for the simple reason that all the files are available locally.

In most cases, many of the files are unchanged from snapshot to snapshot. Further, software or documentation source files are typically in a text format. Both these aspects mean that DVCS software can apply compression techniques extremely effectively. Here are some numbers from an actual project:

Table A.1. Storage requirements compared: VCS and DVCS

Aspect	VCS	DVCS
Server copy of entire repository	800+ MB	331 MB
Local copy of repository and snapshot	1.6 GB	1 GB

B. Useful resources

Choosing a Distributed Version Control System
- Choosing an XML Schema: DocBook or DITA?
- Distributed Revision Control
- DITA Open Toolkit

A complete schema definition language for the Text Encoding Initiative

Lou Burnard

Lou Burnard Consulting

<lou.burnard@retired.ox.ac.uk>

Sebastian Rahtz

IT Services, University of Oxford

<sebastian.rahtz@it.ox.ac.uk>

Abstract

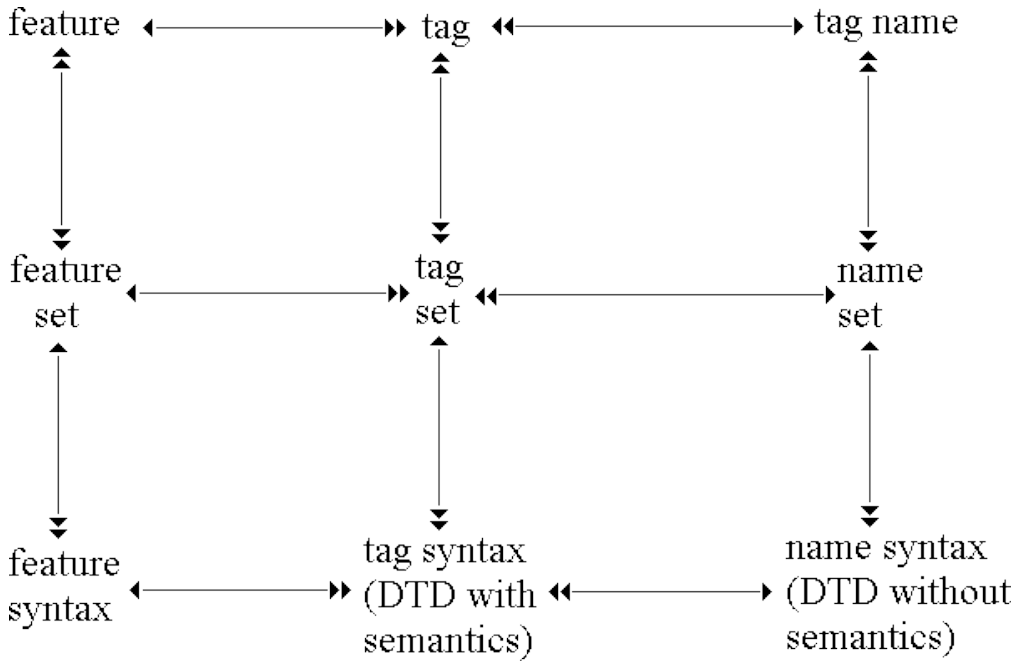
For many years the Text Encoding Initiative (TEI) has used a specialised high-level XML vocabulary known as ODD in the "literate programming" paradigm to define its influential Guidelines, from which schemas or DTDs in other schema languages are derived. This paper describes a minor but significant modification to the TEI ODD language and explores some of its implications. In the current ODD language, the detailed content model of an element is expressed in RELAX NG, embedded inside TEI markup. We define a set of additional elements which permit the ODD language to cut its ties with existing schema languages, making it an integrated and independent whole rather than an uneasy hybrid restricted in its features to the intersection of the three current schema languages. This may pave the way for future developments in the management of structured text beyond the XML paradigm. We describe the additional features, and discuss the problems of both implementing them, and of migrating existing TEI definitions.

1. Introduction

The Text Encoding Initiative (TEI) began in the late 1980s as a conscious attempt to *model* existing and future markup systems. The original TEI editors, Lou Burnard and Michael Sperberg-McQueen, had spent much of their careers trying to find satisfactory ways of expressing the rugosities of typical humanities datasets using the database modelling techniques common in the IT industry at that time. They naturally turned to the same techniques to help draw up a formal model of textual features, and their representations in different markup schemes. The following figure, taken from an early paper on the topic, typifies the approach: distinguishing sharply between the features perceived in a text, their representation by the application of tags, and the names that might be used for those tags.

doi:10.14337/XMLLondon13.Rahtz01

Figure 1. Abstract model from TEI EDW05, 1989



This exercise in modelling started to become more than theoretical quite early on in the life of the TEI, notably during 1991, when the TEI's initial workgroups started to send in their proposals for textual features which they felt really had to be distinguished in any sensible encoding project. It rapidly became apparent that something better than a Hypercard stack or relational database would be needed to keep track of the tags they were busy inventing, and the meanings associated with them. In particular, something able to *combine* text and formal specifications in a single SGML document was needed. Fortunately Donald Knuth had been here before us, with his concept of "literate programming".[1].

In the autumn of 1991, Michael Sperberg-McQueen and Lou Burnard started seriously thinking about ways of implementing the idea of a single DTD which could support both the documentation of an encoding scheme and its expression as a formal language. Our thoughts were necessarily constrained to some extent by the SGML technology at our disposal, but we made a considered effort to abstract away from that in the true spirit of literate programming as Knuth eloquently defines it elsewhere: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." [2] The documentation for each element in the proposed system thus needed to provide informal English language expressions about its intended function, its name and why it was so called, the other elements it was associated with in the SGML structure, usage examples and cross-references to places where it was discussed along with formal SGML declarations for it and its attribute list. Relevant portions of these tag documents could then be extracted into the running text, and the whole could be reprocessed to provide reference documentation as well as to generate document type declarations for the use of an SGML parser. The following figure shows a typical example of such a tag document

[1] "Literate programming is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language" (http://www-cs-faculty.stanford.edu/~uno/lp.html)

[2] Donald Knuth, *Literate Programming* (1984)

Figure 2. Tagdoc for <resp> element in P2 (colorized version)

```
<tagdoc usage=rwa id="resp"><gi>resp</gi>
<name>statement of responsibility</name>
<desc>supplies information about someone other than an author,
sponsor, funder or principal researcher responsible for the
intellectual content of a text, edition, recording, or
series.</desc>
<attlist></attlist>
<exemplum><eg><![ CDATA [
  <resp><role>transcribed from original ms</role>
        <name>Claus Huitfeldt</name>
  </resp>
]]>
</eg></exemplum>
<exemplum><eg><![ CDATA [
  <resp><role>converted to SGML encoding</role>
        <name>Alan Morrison</name>
  </resp>
]]>
</eg></exemplum>
<remarks></remarks>
<part>auxiliary tag set for TEI headers</part>
<classes>
<files names='teihdr2'>
<datadesc></datadesc>
<parents>editionStmt recording seriesStmt titleStmt</parents>
<children>role name</children>
<elemdecl>
<![ CDATA [
<!ELEMENT resp          - o  ((role & name)+)                  >
]]>
</elemdecl>
<attldecl>
<![ CDATA [
<!ATTLIST resp              %a.global;                         >
]]>
</attldecl>
<xref target='hd21'>
</tagdoc>
```

Note here how the SGML declarations are embedded as floating CDATA marked sections, effectively isolating them from the rest of the document, and thus making it impossible to process them in any way other than by simple inclusion. Such refinements as, for example, checking that every element referenced in a content model has its own specification are hard or impossible. There is also ample scope for error when the structural relationships amongst elements are redundantly expressed both in DTD syntax, and in human readable form using the `<parents>` and `<children>` elements. Nevertheless, this system [1]

[1] The first full specification for an ODD system is to be found in TEI working paper ED W29, available from the TEI archive at http://www.tei-c.org/Vault/ED/edw29.tar. It defines a set of extensions to the existing "tiny.dtd" (an early version of a simple TEI-compliant authoring schema, not unlike TEI Lite), which adds new elements for documenting SGML fragments, elements and entities. It also specifies the processing model which the markup was intended to support. An ODD processor was required to

- extract SGML DTD fragments
- generate reference documentation (REF) form
- generate running prose (P2X)

A processor to carry out the reverse operation (that is, generate template ODD specifications from existing DTD fragments) is also described. Although intended for the use of TEI Workgroups, in practice ODD processors built to this model were used only by the TEI editors.

At the fifth major revision of the guidelines (P5, released in 2007 after 6 years of development), the TEI switched to using RELAX NG as the primary means of declared its content models, both within the element specifications which had replaced the old tag documents as input, and as output from the schema generation process. As a separate processing step, XML DTDs are also generated from this same source, while W3C schema is generated from the RELAX NG outputs using James Clark's trang processor. Another major change at TEI P5 was the introduction and extensive use of model classes as means of implementing greater flexibility than had been achievable by using SGML parameter entities. Both of these changes are reflected in the part of the TEI P5 specification for the `<respStmt>` element shown in the following figure:

Figure 3. Parts of <respStmt> element in P5 (XML)

```
  <desc versionDate="2007-01-21" xml:lang="it">fornisce una dichiarazione di r
    responsabile del contenuto intelletuale di un testo, curatela, registrazio
    in cui gli elementi specifici per autore, curatore ecc. non sono sufficien
  <classes>
    <memberOf key="att.global"/>
    <memberOf key="model.respLike"/>
    <memberOf key="model.recordingPart"/>
  </classes>
  <content>
    <choice xmlns="http://relaxng.org/ns/structure/1.0">
      <group>
        <oneOrMore>
          <ref name="resp"/>
        </oneOrMore>
        <oneOrMore>
          <ref name="model.nameLike.agent"/>
        </oneOrMore>
      </group>
      <group>
        <oneOrMore>
          <ref name="model.nameLike.agent"/>
        </oneOrMore>
        <oneOrMore>
          <ref name="resp"/>
        </oneOrMore>
      </group>
    </choice>
  </content>
  <exemplum xml:lang="en">
    <egXML xmlns="http://www.tei-c.org/ns/Examples">
      <respStmt>
        <resp>transcribed from original ms</resp>
        <persName>Claus Huitfeldt</persName>
      </respStmt>
    </egXML>
  </exemplum>
  <exemplum versionDate="2008-04-06" xml:lang="fr">
    <egXML xmlns="http://www.tei-c.org/ns/Examples">
      <respStmt>
        <resp>Nouvelle édition originale</resp>
        <persName>Geneviève Hasenohr</persName>
      </respStmt>
    </egXML>
```

Where TEI P2 had used embedded DTD language to express content models, TEI P4 had expressed them using string fragments still recognisably derived from SGML DTD language. In TEI P5, we moved to embedding RELAX NG in its own namespace, thus placing that schema language in a privileged position, and inviting the question expressed internally as the Durand Conundrum.[1] No contrainst is placed on editors as to the features of RELAX NG they can use, so it is easy to make something which cannot be converted to W3C Schema by trang (eg interleave), or which is not covered by the conversion to DTD, or which duplicates work being done elsewhere. The last of these is particularly worrisome, as attributes are managed separately in the ODD language language, but the RELAX NG content model fragment may add attribute (or child element) declarations.

2. What's not ODD?

In the current source of TEI P5, there is extensive use of several different XML vocabularies:

- Examples in TEI P5 are presented as if they belonged to some other "TEI Example Namespace"; this however is merely an ingenious processing trick to facilitate their validation;
- Element content models are expressed using a subset of RELAX NG, as discussed in the previous section;
- Datatypes are expressed in a variety of ways, mapping either to built-in W3C datatypes (as defined in the W3C Schema Language) or to RELAX NG constructs;
- Some additional semantic constraints (for example, co-dependence of attributes and element content) are expressed using ISO Schematron rules.
- Specialist vocabularies such as XInclude, MathML and SVG are used where appropriate.

Everything else in a TEI-conformant ODD specification uses only constructs from the TEI namespace. In this paper, we describe a further extension of the ODD language to replace at least some of the cases listed above.

2.1. Element content models

ODD is intended to support the **intersection** of what is possible using three different schema languages. In practice, this reduces our modelling requirements quite significantly. Support for DTD schema language in particular imposes many limitations on what would otherwise be possible, while the many additional facilities provided by W3C Schema and RELAX NG for content validation are hardly used at all (though some equivalent facilities are now provided by the `<constraintSpec>` element). A few years ago, the demise of DTDs was confidently expected; in 2013 however the patient remains in rude health, and it seems likely that support for DTDs will continue to be an ongoing requirement. We therefore assume that whatever mechanism we use to specify content models will need to have the following characteristics:

- the model permits alternation, repetition, and sequencing of individual elements, element classes, or sub-models (groups of elements)
- only one kind of mixed content model — the classic `(#PCDATA | foo | bar)*` — is permitted
- the SGML ampersand connector — `(a & b)` as a shortcut for `((a,b) | (b,a))` is not permitted
- a parser or validator is not required to do look ahead and consequently the model must be deterministic; that is, when applying the model to a document instance, there must be only one possible matching label in the model for each point in the document

We think these requirements can easily be met by the following small incremental changes to the ODD language:

[1] The **Durand Conundrum** is a jokey name for a serious question first raised by David Durand when the current TEI ODD XML format was being finalised at a meeting of the TEI Technical Council held in Gent in May 2004. David pointed out that the TEI's mixed model was a compromise solution: like other XML vocabularies, the TEI was perfectly hospitable to other namespaces, so we could equally well embed our TEI additions within a natively RELAX NG document. A similar suggestion is made in Eric Van der Vlist's *RELAX NG* (O'Reilly, 2011), which proposes a hybrid language called "Examplotron" in which the documentation is expressed using the XHTML vocabulary, the document grammar is expressed using RELAX NG, and additional constraints are expressed using Schematron. See further http://examplotron.org

Specification

At present, references to content model components use the generic <rng:ref> element. As a consequence, naming conventions have been invented to distinguish, for example references to an element or attribute class (name starts with "model." or "att.") from references to a predefined macro (name starts with "macro.") or from references to an element (name starts with something other than "model." or "macro."). Although these name changes are purely a matter of convenience, we suggest that it would be better to use the existing TEI ODD elements <elementRef>, <classRef> and <macroRef> elements.

For example,

```
<rng:ref name="model.pLike"/>
```

becomes

```
<classRef key="model.pLike"/>
```

Repeatability

In RELAX NG, this is indicated by special purpose grouping elements <rng:oneOrMore> and <rng:zeroOrMore>. We propose to replace these by the use of attributes @minOccurs and @maxOccurs, which are currently defined locally on the <datatype> element. Making these also available on <elementRef>, <classRef> and <macroRef> elements, gives more delicate and consistent control over what is possible within the components of a content model.

Sequence and alternation

Sequencing and alternation are currently indicated by elements defined in the RELAX NG namespace (<rng:choose>, <rng:group>, etc.) We replace these by similar but more constrained TEI equivalents <sequence> which operates like <rng:group> to indicate that its children form a sequence within a content model, and <alternate> which operates like <rng:choose> to supply a number of alternatives.

For handling character data, we follow the W3C Schema approach and define an attribute @mixed for each container element.The two simple cases of empty content, and of pure text content, will be covered by an empty <content> element.

We now provide some simple examples, showing how some imaginary content models expressed using RELAX NG compact syntax might be re-expressed with these elements.

In this example ((a, (b|c)*, d+), e?) we have a sequence containing a single element, followed by a repeated alternation, a repeated element, and an optional element. This would be expressed as follows:

```
<sequence>
  <sequence>
    <elementRef key="a"/>
    <alternate minOccurs="0" maxOccurs="unbounded">
      <elementRef key="b"/>
      <elementRef key="c"/>
    </alternate>
    <elementRef key="d" maxOccurs="unbounded"/>
  </sequence>
  <elementRef key="e" minOccurs="0"/>
</sequence>
```

Repetition can be applied at any level.
In ((a, (b*|c*))+, for example, we have a repeated sequence. This would be expressed as follows:

```
<sequence maxOccurs="unbounded">
  <elementRef key="a"/>
  <alternate>
    <elementRef key="b" minOccurs="0"
                maxOccurs="unbounded"/>
    <elementRef key="c" minOccurs="0"
                maxOccurs="unbounded"/>
  </alternate>
</sequence>
```

A mixed content model such as (#PCDATA | a | model.b)* might be expressed as follows:

```
<alternate minOccurs="0" maxOccurs="unbounded"
           mixed="true">
  <elementRef key="a"/>
  <classRef key="model.b"/>
</alternate>
```

References to model classes within content models pose a particular problem of underspecification in the current ODD system. In the simple case, a reference to a model class may be understood as meaning any one member of the class, as assumed above. Hence, supposing that the members of class model.ab are <a> and <b>, a content model

```
<classRef key="model.ab" maxOccurs="unbounded"/>
```

is exactly equivalent to

```
<alternate maxOccurs="unbounded">
  <elementRef key="a"/>
  <elementRef key="b"/>
</alternate>
```

However, sometimes we may wish to expand model references in a different way. We may wish to say that a reference to the class `model.ab` is not a reference to any of its members, but to a sequence of all of its members, or to a sequence in which any of its members may appear, and so forth. This requirement is handled in the current ODD system by over-generating all the possibilities, again using a set of naming convention to distinguish amongst them. We propose instead to control this behaviour by means of a new `@expand` attribute on `<classRef>` (modelled on an existing `@generate` on `<classSpec>`), but with the advantage of being usable at the instance level.

For example,

```
<classRef key="model.ab" expand="sequence"/>
```

is interpreted as `a,b` while

```
<classRef key="model.ab"
          expand="sequenceOptional"/>
```

is interpreted as `a?,b?`,

```
<classRef key="model.ab"
          expand="sequenceRepeatable"/>
```

is interpreted as `a+,b+` and

```
<classRef key="model.ab"
          expand="sequenceOptionalRepeatable"/>
```

is interpreted as `a*,b*` Note that the ability to specify repetition at the individual class level gives a further level of control not currently possible. For example, a model containing no more than two consecutive sequences of all members of the class `model.ab` could be expressed quite straightforwardly:

```
<classRef key="model.ab" maxOccurs="2"
          expand="sequence"/>
```

2.2. Datatyping and other forms of validation

Validation of an element's content model is but one of many different layers of validation that a TEI user may wish to express in their ODD specification. The current system also provides mechanisms to constrain the possible values of attributes by means of datatyping and also, increasingly, by explicit constraints expressed using languages such as ISO Schematron. It seems reasonable to ask how many of these additional layers may be incorporated into our proposed new vocabulary.

The vast majority of TEI attributes currently define their possible values by reference to a datatype macro which is defined within the ODD system, where it is mapped either to a native RELAX NG datatype or to an expression in RELAX NG syntax. This indirection allows the schema builder to add a small amount of extra semantics to an underlying "bare" datatype. For example `data.duration.iso`, `data.outputMeasurement`, `data.pattern`, `data.point`, `data.version`, and `data.word` all map to the same datatype (`token` as far as a RELAX NG schema is concerned; `CDATA` for an XML DTD). As their names suggest, however, each of these TEI datatypes has a subtly different intended application, which an ODD processor may use in deciding how to present the corresponding information, even though the mapping to a formal schema language is identical in each case.

Given the existence of this TEI abstraction layer, it seems unnecessary to propose further change to the way attribute values are constrained in the ODD system. At the time of writing, there are still a few attributes whose values are expressed directly in RELAX NG syntax, but that is a corrigible error in the Guidelines source code.

The most commonly used datatype macro is `data.enumerated`, which maps to another frequently used datatype `data.name`, and thence to the underlying RELAX NG datatype for an XML Name. The difference between an enumeration and a name is, of course, that a (possibly closed) list of possible values can be provided for the former but not for the latter. In the ODD system, for every datatype declared as `data.enumerated`, a sibling `<valList>` element should be provided to enumerate and document all or some of the possible values for this attribute. This ability to constrain and document attribute values is of particular interest because it permits TEI schema-specifiers to define project-specific restrictions and semantics considerably beyond those available to all schema languages.

A further layer of constraint specification is provided by the <constraintSpec> element which may be used to express any kind of semantic constraint, using any suitable language. In the current TEI specifications, the ISO-defined Schematron language is deployed to replace as many as possible of the informally expressed rules for good practice which have always lurked in the Guidelines prose. This facility allows us to specify, for example, the co-occurrence constraint mentioned in the previous paragraph (that the specification for an attribute with a declared datatype of data.enumerated should also contain a <valList>). It also allows an ODD to make more explicit rules such as “a relatedItem element must have either a @target attribute or a child element” or “the element indicated by the @spanTo attribute must follow the element carrying it in document sequence”, which are hard to express in most schema languages.
For our present purposes, it is important to note that the TEI <constraintSpec> element was designed to support any available constraints language. Although the current generation of ODD processors assume the use of ISO Schematron, there is no reason why future versions should not switch to using different such languages as they become available without affecting the rest of the ODD processing workflow or the ODD language itself. As such, we see no need to modify our proposals to take this level of validation into account.

3. Discussion

The ideas presented here were first sketched out in the summer of 2012, and greeted positively at the ODD Workshop held following the DH 2012 conference in Hamburg. An earlier version of this paper was presented at the TEI Conference in Texas in November 2012. In this section we briefly summarize some of the comments received.

At first blush, our proposals seem to flout the TEI philosophy of not re-inventing the wheel. The TEI does not and should not take on itself the task of inventing a new XML vocabulary for such matters as mathematics or musical notation where perfectly acceptable and well established proposals are already in place. However the TEI has arguably already gone down the road of defining some aspects its own schema language, (for example, by providing constructs for representing element and attribute classes, and for associating attribute lists and value lists with element declarations) and this proposal simply continues along the same path. It should also be noted that there are three competing standards for schema language in the marketplace (DTD, RELAX NG, W3C Schema) each with its own advantages. By making ODD independent of all three, we make it easier to profit from the particular benefits of each, as well as providing the ability to document intentions not necessarily expressible using any of them.

Resolving the Durand conundrum in this way, rather than taking the alternative approach of embedding TEI documentation elements in the RELAX NG namespace, is clearly a compatible expansion of the current scheme rather than an incompatible change of direction which would not break existing systems or documents.

As a concrete example, consider the occasionally expressed desire to constrain an element's content to be a sequence of single specified elements appearing in any order, that is, to define a content model such as (a,b,c,d) but with the added proviso that the child elements may appear in any order. In SGML, the ampersand operator allowed something like this; in RELAX NG the <interleave> element may be used to provide it, but there is no equivalent feature in W3C Schema or DTD languages, and we have not therefore proposed it in our list of requirements above.

Suppose however that the Technical Council of the TEI decided this facility was of such importance to the TEI community that it should be representable in TEI ODD. It would be easy enough to add a new grouping element such as `<interleave>` (or add an attribute `@preserveOrder` taking values TRUE or FALSE to our existing proposed `<sequence>` element) to represent it. Generating a RELAX NG schema from such an ODD would be simple; for the other two schema languages one could envisage a range of possible outcomes:

- an ODD processor might simply reject the construct as infeasible;
- an ODD processor might over-generate; that is, it will produce code which validates everything that is valid according to the ODD, but also other constructs that are not so valid;
- an ODD processor might over-generate in that way, but in addition produce schematron code to remove "false positives".

For example, consider the following hypothetical ODD

```
<interleave>
  <elementRef key="a"/>
  <elementRef key="b" maxOccurs="2"/>
  <elementRef key="c"/>
</interleave>
```

In XML DTD or W3C schema languages (which lack the `<rng:interleave>` feature), an ODD processor can represent these constraints by generating a content model such as

```
(a|b|c)+
```

and at the same time generating additional Schematron constraints to require the presence of no more than one `<a>` or `<c>` and up to two `<b>`s. An extra twist, in this case, is that if there are more than two `<b>` elements, they must follow each other.

As a second example, consider the need for contextual variation in a content model. For example, a `<name>` or `<persName>` appearing inside a "data-centric" situation, such as a `<listPerson>` element, is unlikely to contain elements such as `<del>` or `<corr>` which are however entirely appropriate (and very useful) when identifying names within a textual transcription. In a linguistic corpus, it is very likely that the child elements permitted for `<p>` elements within the corpus texts will be quite different from those within the corpus header — the latter are rather unlikely to include any part of speech tagging for example.

At present only ISO schematron rules allow us to define such contextual rules, although something analogous to them is provided by the XSD notion of base types. It is not hard however to imagine a further extension to the ODD language, permitting (say) an XPath-valued `@context` attribute on any `<elementRef>`, `<macroRef>`, or `<classRef>` restricting its applicability. Thus, the content model for `<p>` might say something like

```
<elementRef
  key="s"
  context="ancestor::text"
  maxOccurs="unbounded"
  minOccurs="1"/>
<macroRef key="macro.limitedContent"
          context="ancestor::teiHeader"/>
```

to indicate that a `<p>` within a `<text>` element must contain one or more `<s>` elements only, whereas one within a TEI Header must use the existing macro definition `limitedContent`.

However, before embarking on such speculative exercises, it is clear that further experimentation is needed, to see how easily existing content models may be re-expressed, and what the consequences of such a conversion would be for existing processing tools.

4. Implementation

In order to prove the language additions proposed above can adequately replace the current embedded RELAX NG, we have completed four pieces of work:

- Formal definition of the new elements in the TEI ODD language; this presents no problems.
- Implementation, in the ODD processor, of the conversion from the new language elements to the target schema languages (DTD, RELAX NG, W3C Schema), each according to its abilities
- Conversion of the existing content models in RELAX NG to the new format
- Testing of the resulting generated schemas to ensure that they are at least as permissive as the old ones.

The last of these tasks is covered by the already extensive testing undertaken after each change to the TEI source; a suite of test schemas and input files is checked,[1] and it is expected that these will already catch most errors.

The second task is not problematic. The TEI ODD processing engine is written in XSLT and already works through a complex process to arrive at a generated DTD or RELAX NG schema (XSD is generated by trang); the work of adding in the extra processing is simply an extension of existing code.

[1] The TEI Guidelines themselves are also, of course, a very large TEI document which is extensively checked against itself.

This leaves the problem of converting the existing 573 models in the TEI Guidelines. These fall into four groups:

- Reference to a macro, predefined patterns of which the TEI provides 8 (which is a small enough number to be translated by hand if necessary)
- Simple reference to a members of a class of elements
- A content model of plain text, `<empty>`, or a data pattern
- Complex hand-crafted model with branches of choices and nesting

An analysis shows that the final group covers 269 of the content models:

Table 1. Types of content models in TEI

Type	Number
class members	28
empty, data and text	62
macro	194
simple name	20
other	269

In practice, the majority of these are amenable to relatively simple automated conversion.

5. Next steps

The extended schema language for the TEI as described here is implemented in a preliminary release (May 2013), and will be completed in time for the Members Meeting of the TEI Consortium in October 2013. It is expected to be made an optional part of the TEI language by the end of 2013, even if the TEI Guidelines themselves do not convert to using it internally.

Charles Foster

XML London 2013
Conference Proceedings

Published by
XML London

103 High Street
Evesham
WR11 4DN
UK

This document was created by tranforming original DocBook XML sources into an XHTML document which was subsequently rendered into a PDF by **Antenna House Formatter**.

1st edition

London 2013

ISBN 978-0-9926471-0-0