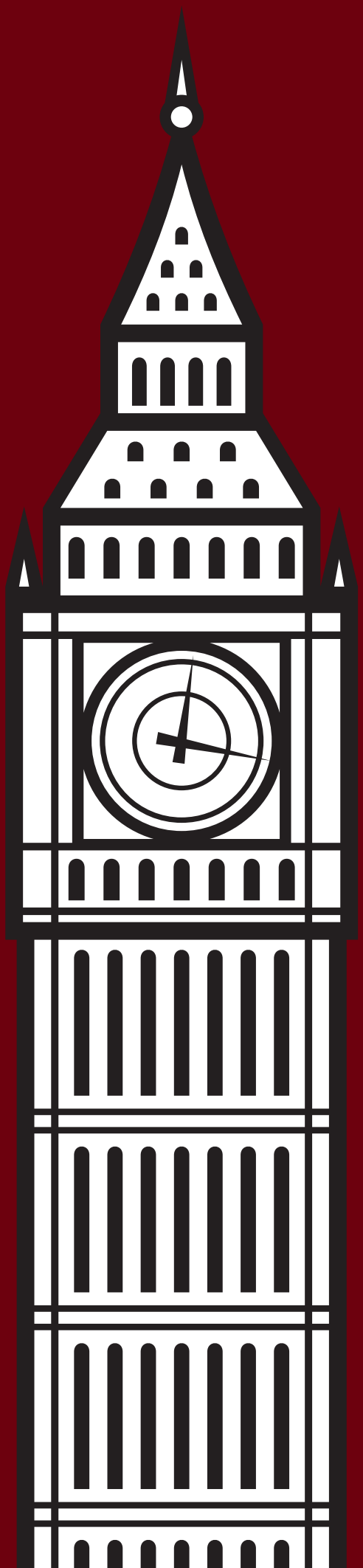


XML LONDON 2017

CONFERENCE PROCEEDINGS

UNIVERSITY COLLEGE LONDON,
LONDON, UNITED KINGDOM

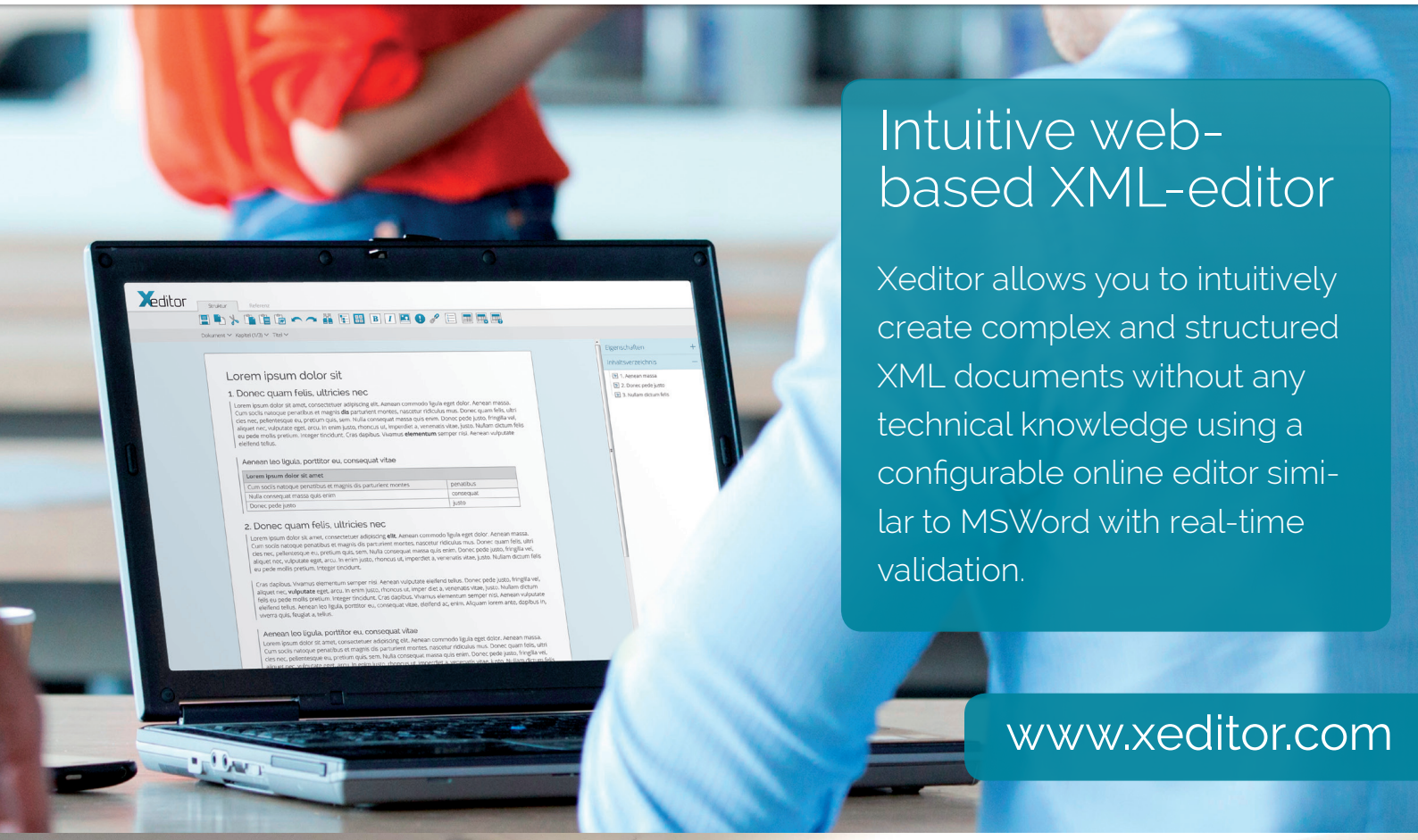
JUNE 10-11, 2017

XML London 2017 – Conference Proceedings
Published by XML London

ISBN 978-0-9926471-4-8

Table of Contents

Xeditor
professional authoring



Intuitive web-based XML-editor

Xeditor allows you to intuitively create complex and structured XML documents without any technical knowledge using a configurable online editor similar to MSWord with real-time validation.

www.xeditor.com

General Information

Date

Saturday, June 10th, 2017
Sunday, June 11th, 2017

Location

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

Organising Committee

Charles Foster, Socionics Limited
Dr. Stephen Foster, Socionics Limited
Geert Bormans, C-Moria
Ari Nordström, Creative Words
Andrew Sales, Andrew Sales Digital Publishing Limited
Tomos Hillman, eXpertML Limited

Programme Committee

Abel Braaksma, AbraSoft
Adam Retter, Evolved Binary
Andrew Sales, Andrew Sales Digital Publishing Limited
Ari Nordström, Creative Words
Charles Foster (chair)
Eric van der Vlist, Dyomedea
Geert Bormans, C-Moria
Norman Walsh, MarkLogic
Philip Fennell, MarkLogic

Produced By

XML London (http://xmllondon.com)

Sponsors

Xeditor - http://www.xeditor.com

Intuitive, user-friendly interface similar to MSWord

With Xeditor, the web-based XML-Editor, authors can easily create and edit structured documents and files in XML data format, providing media-neutral data storage. The user-friendly frontend leads authors intuitively through the defined document structure (XSD/DTD). The Editor is web-based and operates completely in the browser.

Saxonica - http://www.saxonica.com

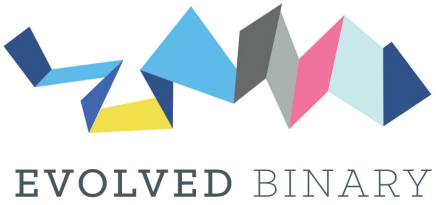
Developers of the Saxon processor for XSLT, XQuery, and XML Schema, including the only XSLT 3.0 conformant toolset.

Saxonica was founded by Dr Michael Kay in 2004. Saxonica now have more than 500 clients world-wide.

Evolved Binary - http://www.evolvedbinary.com

Since 2014, Evolved Binary has been engaged in Research and Development to create the next generation document database platform, Project "Granite". Granite was initially envisaged as an Enterprise scalable replacement for eXist-db while staying true to Open Source and it is now shaping up to be much more! Although still operating predominantly in "stealth mode", Evolved Binary's customers are already trialling Granite. A public beta is scheduled for Q3 2017.

Evolved Binary was founded by Adam Retter in 2014. Evolved Binary focuses on R&D in the areas of Information Storage, Concurrent Processing, and Transactions.



Preface

This publication contains the papers presented during the XML London 2017 conference.

This is the fifth international XML conference to be held in London for XML Developers, Semantic Web and Linked Data enthusiasts as well as Managers, Decision Makers and Markup Enthusiasts. It is a platform for attendees to discuss and share their experiences with other W3C technology users, while discovering the latest innovations and finding out what others are doing in the industry.

This Conference also hosts an Expert Panel Session where invited Industry Leaders will share their thoughts around challenges and successes found in electronic publishing projects.

The conference is taking place on the 10th and 11th June 2017 at the Faculty of Engineering Sciences (Roberts Building) which is part of University College London (UCL). The conference dinner and the XML London 2017 DemoJam is being held in the Jeremy Bentham Room also situated at UCL, London.

— Charles Foster
Chairman, XML London

Distributing XSLT Processing between Client and Server

O'Neil Delpratt

Saxonica

<oneil@saxonica.com>

Debbie Lockett

Saxonica

<debbie@saxonica.com>

Abstract

In this paper we present work on improving an existing in-house License Tool application. The current tool is a server-side web application, using XForms in the front end. The tool generates licenses for the Saxon commercial products using server-side XSLT processing. Our main focus is to move parts of the tool's architecture client-side, by using "interactive" XSLT 3.0 with Saxon-JS. A beneficial outcome of this redesign is that we have produced a truly XML end-to-end application.

Keywords: XSLT, Client, Server

1. Introduction

For a long time now browsers have only supported XSLT 1.0, whereas on the server-side there are a number of implementations for XSLT 2.0 and 3.0 available. For applications using XSLT processing, the client/server distribution of this processsing is governed by the implementations available in these environments. As a result, many applications, including our in-house "License Tool" web application, rely heavily on server-side processing for XSLT 2.0/3.0 components.

The current License Tool web application is built using the Servlex framework [1] [2], and principally consists of a number of XSLT stylesheets. The HTML front end uses XForms, and the form submission creates HTTP requests which are handled by Servlex. We use XSLTForms [3] to handle the form processing in the browser, an implementation of XForms in XSLT 1.0 and JavaScript.

The main motivation for this project is to improve our License Tool webapp by moving parts of the server-side XSLT processing into the client-side. This can only be made possible by a client-side implementation of XSLT 2.0/3.0. We would like to see which components of the application's architecture can now be done using client-side interactive XSLT.

Interactive XSLT is a set of extension elements, functions and modes, to allow rich interactive client-side applications to be written directly in XSLT, without the need to write any JavaScript. (For information on the beginnings of interactive XSLT, see [4], and for the current list of `ixsl` extensions available see [5].) Stylesheets can contain event handling rules to respond to user input (such as clicking on buttons, filling in form fields, or hovering the mouse), where the result may be to read additional data and modify the content of the HTML page. The suggested idea for building such interactive XSLT applications is to use one skeleton HTML page, and dynamically generate page content using XSLT. Event handling template rules are those which match on user interaction events for elements in the HTML DOM. For instance, the template rule

```
<xsl:template match="button[id='submit']"
              mode="ixsl:onclick"/>
```

handles a click event on a specific HTML button element. When the corresponding event occurs, this causes a new transformation to take place, with this as the initial template, and the match element (in the HTML DOM) as the initial context item. The content of the template defines the action. For example, a fragment of HTML can be generated and inserted into a specific target element in the HTML page using a call such as

```
<xsl:result-document select="div[id='target']"
                     mode="ixsl:replace-content"/>
```


doi:10.14337/XMLLondon17.Lockett01

In order to use client-side interactive XSLT 3.0 within our License Tool, we use Saxon-JS [6] - a run-time XSLT 3.0 processor written in pure JavaScript, which runs in the browser and implements interactive XSLT. We still maintain some server-side XSLT processing, as required. But by using XSLT 3.0 [7], with the interactive extensions, we are able to do much more of the tool's processing client-side, which means that we can achieve our objective. The redesign means that the tool is now XML end-to-end, without any environment specific glue, which minimises the need to translate between objects.

One benefit of moving the processing client-side is that more of it is brought directly under our control, so we should then be in a better position to resolve and in places avoid incompatibilities between the technologies and environments. For instance, in the current tool, we are aware of some data encoding issues for non-ASCII characters [8]. In the current License Tool the data is sent by XSLTForms encoded in a certain format, but this encoding is not what Servlex expects. The problem is made more complicated by the multiple layers of technologies in use, and of course the internal XSLTForms and Servlex processing is out of our control.

The reluctance of browser vendors to upgrade XSLT support has meant that tools such as XSLTForms are stuck with using XSLT 1.0. Many mobile browsers do not even support XSLT 1.0. By using Saxon-JS in the browser, we are freed from this restriction, and so can replace our use of XSLTForms in the License Tool. We have worked towards a new implementation of XForms using XSLT 3.0 and interactive XSLT, and produced a prototype partial implementation.

Along with making improvements to the tool, we were also interested to see how the experience gained from this real world example may initiate further developments for Saxon-JS itself. In particular, one major challenge is how to handle the communications between client and server using HTTP, within our interactive XSLT framework.

In the following sections we will introduce what the application actually does, how it originally worked, and the changes we have made. We will focus on how we have used XSLT 3.0 and interactive XSLT in the redesign, the benefits of this change, and how it has improved the application.

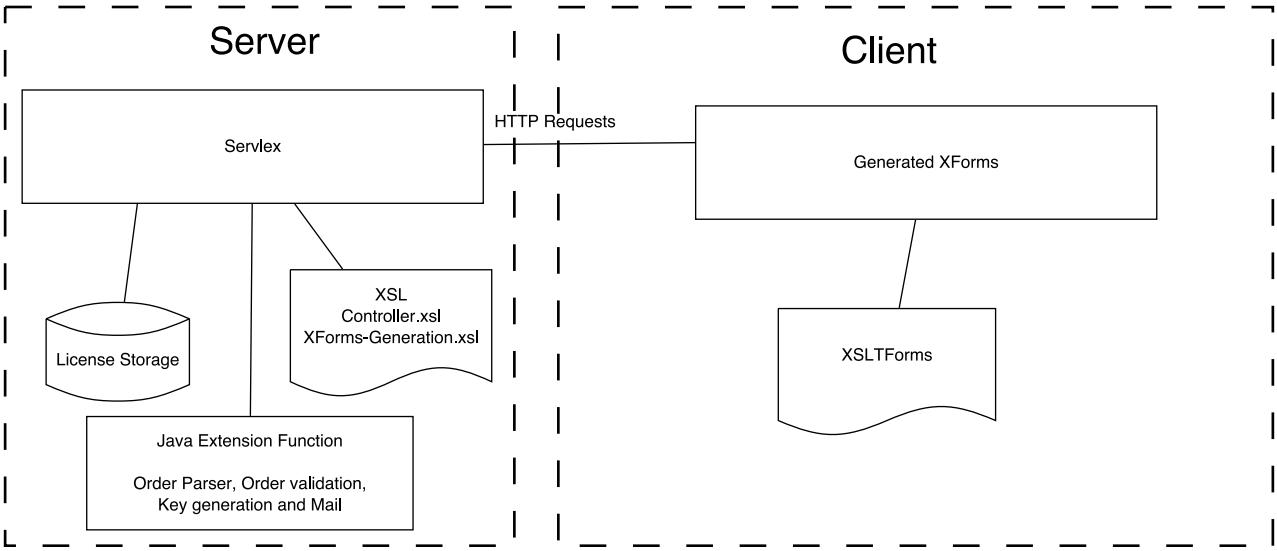
2. License Tool application: what it does, and how it currently works

The License Tool processes license orders (i.e. purchase or registration infomation) for the Saxon commercial products, and then generates and issues license files (which contain an encrypted key used to authorize the commercial features of the product) to the customer. The License Tool also maintains a history of orders and licenses (as a set of XML log files) and provides simple reporting functions based on this history.

The application is built using the Servlex framework. Servlex is a container for EXPath Web Applications [9], using the EXPath Packaging System [10], whose functionality comes from XML technologies. Servlex handles all the networking and server-side XML processing, and connects the XML components to the HTTP layer.

For our current tool, this means using Servlex on the server-side to parse the license order and convert to a custom XML format, process the XML instance data received from the XForms form, and send feedback to the user in the browser. This is all driven by XSLT stylesheets on the server. On the client-side, we use XSLTForms to handle the XForms processing. An overview of the architecture of the License Tool is shown in [Figure 1]. This architecture diagram shows the main components, and technologies used, client and server side.

Figure 1. Old License Tool architecture diagram



In more detail, the tool works as follows:

1. When purchasing or registering for a license, a customer completes a web form to provide certain order information: contact details, the name of the purchased product, etc.
2. This license order information is sent to us by email, as structured text (it would be nice if it were XML or JSON, but this is the real world). See [Figure 2] and [Figure 3] for examples.
3. We input the license order text from the email into the License Tool via an XForms form in the "Main Form" HTML page of the webapp, and use the form submit to send this data to the server.
4. All communication with the server-side of the License Tool is done using HTTP requests, which are picked up within the Servlex webapp by an XSLT controller stylesheet which then processes the license order. The first steps are to parse the text and convert it into a custom XML format, which is then validated. See [Figure 4] for an example of the order XML.
5. The application then returns the license order to the user as the XML instance data of another XForms form, the "Edit Form". At this point the order may be manually edited. (This page can also be used to edit existing licenses before reissuing, for instance for upgrades and renewals.)
6. Next, when the "Edit Form" is submitted, the customer's license file is created. This processing is done using reflexive extension functions written in Java within the XSLT on the server.
7. The application then reports to its user the outcome of generating the license, for final confirmation. If there has been a problem with the license generation, there is again the option to manually modify the license order. Otherwise, when the user confims the order, the license is issued (again using a server-side Java extension function).
8. The data model of the application is XML driven, with the exception of the email text for a license order used as the initial input.

Figure 2. Example license order text for an evaluation license.

```
First Name: Tom

Last Name: Bloggs

Company: Bloggs XML

Country: United Kingdom

Email Address: tom@bloggs.com

Phone:

Agree to Terms: checked
```

Figure 3. Example license order text for a purchased license.

```
Order #9999 has just been placed

Email: tom@bloggs.com

Comments: ZZZ-9999

==== Items ====

item_name: Saxon-EE (Enterprise Edition),
           initial license (ref: EE001)
item_ID: EE001
item_options:
item_quantity: 1
item_price: £360.00

item_name: Saxon-EE (Enterprise Edition),
           additional licenses (ref: EE002)
item_ID: EE002
item_options:
item_quantity: 2
item_price: £180.00


==== Order Totals ====

Items: £720.00
Shipping: £0.00
Tax: £0.00
TOTAL: £720.00

-- Billing address --

company: Bloggs XML

billing_name: Tom Bloggs
billing_street: 123 Fake St
billing_city: Somewhere
billing_state: Nowhere
billing_postalCode: A1 1XY
billing_countryName: United Kingdom
billing_phone:
```

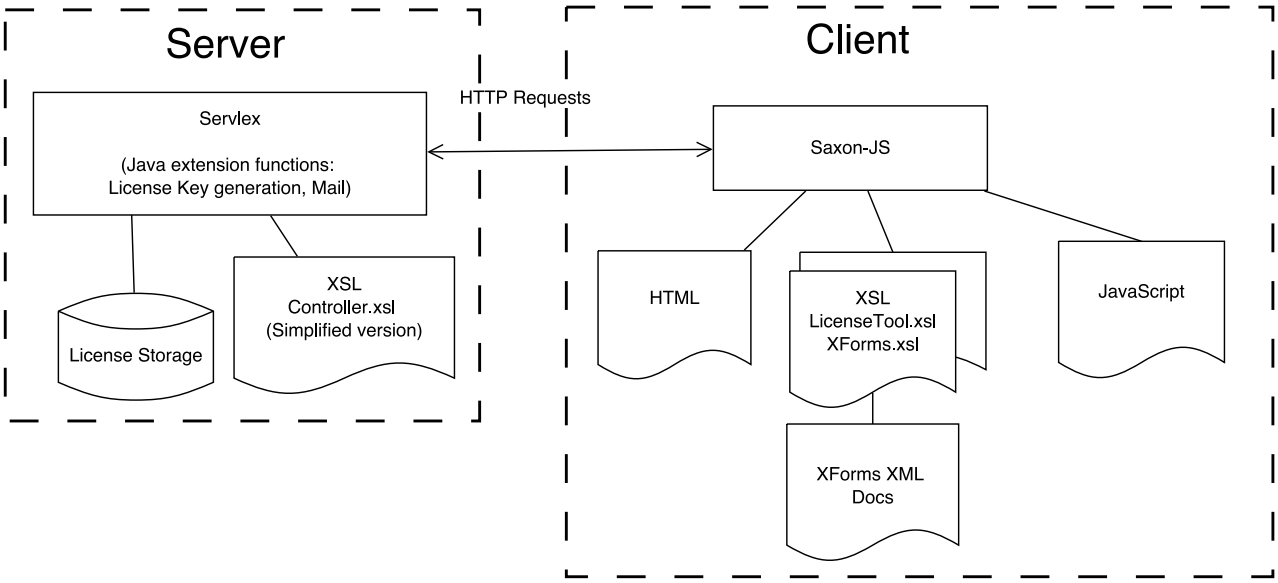
Figure 4. Example of an order in XML format (the result of converting the example license order text in [Figure 3])

```
<Order>
   <OrderRef>#9999</OrderRef>
   <DatePlaced>2017-05-05</DatePlaced>
   <DateOfExpiry>never</DateOfExpiry>
   <First>Tom</First>
   <Last>Bloggs</Last>
   <Company>Bloggs XML</Company>
   <Address1>123 Fake St</Address1>
   <Address2/>
   <Town>Somewhere</Town>
   <County>Nowhere</County>
   <Postcode>A1 1XY</Postcode>
   <Country>United Kingdom</Country>
   <Email>tom@bloggs.com</Email>
   <Phone/>
   <UpgradeDays>366</UpgradeDays>
   <MaintenanceDays>366</MaintenanceDays>
   <Online>true</Online>
   <OrderPart>
      <ProductCode>EE001</ProductCode>
      <Edition>EE</Edition>
      <Platform>J</Platform>
      <Features>TQV</Features>
      <Quantity>1</Quantity>
      <Value>360</Value>
      <Domain/>
   </OrderPart>
   <OrderPart>
      <ProductCode>EE002</ProductCode>
      <Edition>EE</Edition>
      <Platform>J</Platform>
      <Features>TQV</Features>
      <Quantity>2</Quantity>
      <Value>360</Value>
      <Domain/>
   </OrderPart>
</Order>
```

3. Application redesign

The main aim of this project is to move more components of the License Tool's processing architecture client-side, by using interactive XSLT 3.0. We have achieved this by building a new interactive front end for our tool, written in interactive XSLT 3.0. This stylesheet is compiled using Saxon-EE to produce a stylesheet export file (SEF) which the Saxon-JS run-time executes in the browser.

Figure 5. New License Tool architecture diagram



This redesign to the application means that the client-side processing can now handle the initial parsing of the license order text, and convert to the order XML format; before the need for any server-side processing. We have also produced a new partial prototype implementation for XForms using interactive XSLT 3.0, as an improvement to using XSLTForms, which is included in the front end process. An overview of how the main processing components and technologies are now distributed, client and server side, is shown in the architecture diagram for the new License Tool in [Figure 5].

The redesign has also introduced some changes to the tool's processing pipeline. The flow diagram in [Figure 6] illustrates the design for the new tool. It shows the steps of the process - user interactions with the application, the processing actions client and server side - and the flow between all of these steps. As can be seen, we have indeed moved much of the processing client-side: parsing license order text; converting to XML; validating; generating and rendering the XForms "Edit Form"; and handling the submit button click event. We currently still rely on server-side processing for some final stage components of the pipeline - namely generating the license (which includes the encrypted key), issuing it via email, and storing the license order.

In the following sections, we will describe in more detail the three main areas of development in the License Tool's redesign:

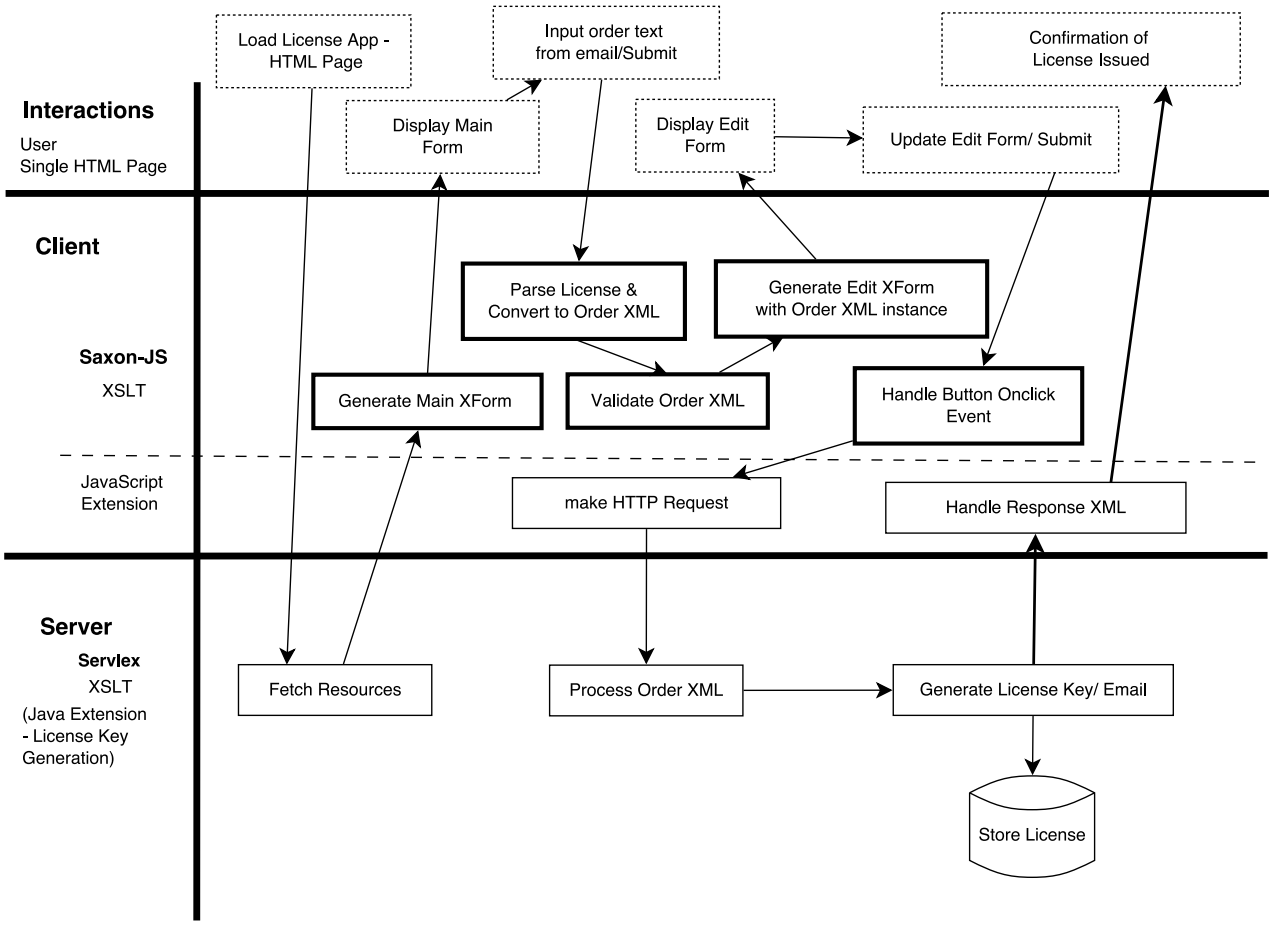
1. Client-side XSLT processing, to replace the use of Java extension functions.
2. A prototype for a new XForms implementation using XSLT 3.0 and interactive XSLT, to run client-side.
3. Handling HTTP communication between client and server.

4. Client-side XSLT processing

There is great potential to simplify the tool's architecture by using XML end-to-end. Throughout the pipeline, the main object we are dealing with is an "order" - which contains information about the customer, the products ordered, the date of the order, etc. Ideally we would be handling this order in our custom XML format throughout the processing pipeline. So the order information is captured directly into XML format, which can be modified, passed between client and server, and stored server-side, without the need to be converted to any other different formats along the way.

The original legacy version of the License Tool was a Java application. The current tool makes much use of Java extension functions in the webapp's XSLT stylesheets in order to reuse the original code. For instance, the process of parsing the license order text and converting it into XML format is done by a Java extension function. So there is a Java class with methods to parse the input license order text, and produce a Java `Order` object. This `Order` object is then converted into XML using Java methods. In fact, many such Java components of the webapp could be rewritten in XSLT,

Figure 6. New License Tool flow diagram



and this is clearly more straightforward, since we then avoid converting between XML and Java objects and back. Making the change to do such processing directly in XSLT could have been done already within the current server-side webapp, but there was perhaps little incentive - "if it ain't broke, don't fix it". However, now we are looking to bring the processing client-side, it does make sense to write XSLT solutions to replace the Java code. As well as being able to move such components to the client-side, the tool's architecture is generally simplified by replacing the Java code.

So, in the new tool, the first step of parsing the license order text, and converting to the custom order XML format, is done directly in XSLT. In fact, rather than parsing the structured text and creating the order XML directly, as an intermediate stage it is convenient to use a representation of the order as an XPath 3.0 map item. Having split the input license order text by line, if a line looks like an order category/value pair, then it is added to the order map as a key/value pair. This order map is then used to add text content to an order XML skeleton. For example,

```
<First>
  <xsl:value-of select="$orderMap?first"/>
</First>
```

There may be other stages where it is more convenient to use the XPath map representation for a license order, rather than the XML format. It actually makes a lot of sense to handle the order as an XPath map item, which is easier to modify, and use XSLT functions to convert these to the custom XML format and back. However currently we generally stick to the order XML format.

There may still be times when we need to serialize to a string, and reparse to XML. Infact also, in the new tool, at certain stages we convert to JSON and back, as well as to XPath map and back. But at least these can all now be handled directly within XSLT 3.0, and there is no need to use other objects outside of the XDM model.

5. XForms implementation in interactive XSLT 3.0

XSLTForms is based on XSLT 1.0 to compile XForms to (X)HTML and JavaScript in the browser. As previously discussed, support for XSLT in browsers is limited to XSLT 1.0, and vendors are not inclined to move this forward. Saxon-JS now provides us with XSLT 3.0 processing in the browser, and so we can write a new XForms implementation using XSLT 3.0 and interactive XSLT, to replace the use of XSLTForms. We now describe our proof-of-concept XForms implementation exploring the possibilities in (interactive) XSLT 3.0.

The XForms model, instance data and form controls which provide the user interactions are written as XML in accordance with the XForms specification. The XForms processor is entirely written using interactive XSLT 3.0 using the XSLTForms implementation as a starting point. In the main entry template rule we use `xsl:params` to supply the XForms form (as an XML document), and optionally corresponding XML instance data, to the stylesheet. The main process of the implementation stylesheet is to convert the XForms form controls elements into equivalent (X)HTML form controls elements (inputs, drop-down lists, textareas, etc.). At the same time the forms controls are populated with any bound data from the XML instance data.

For the XForms controls we specify the binding references to the XML instance data as an XPath expression. For example

```
<xforms:input incremental="true"
              ref="Shipment/Order/DatePlaced" />
```

In the template rule which matches the `xforms:input` control we get the string value from the `ref` attribute, and use this XPath in two ways. Firstly, we call the XSLT 3.0 instruction `<xsl:evaluate>` to dynamically evaluate the XPath expression, which obtains the relevant data value from the XML instance data [10b]. This will be used to populate the HTML form `input` element which we convert to. Secondly, the XPath from the `ref` attribute is copied into the `id` attribute of the `input` element, to preserve the binding to the XML instance data (so that upon form submission, we will be able to copy the changes made within the HTML form back into the instance data, as described later). The result in this example is the following:

```
<input type="text" id="Shipment/Order/First/text()"
       value="Tom">
```

The conversion, and binding preservation, of other XForms controls elements are achieved in a similar way. The full code example is shown in [Figure 7].

Figure 7. Example template from the new XForms implementation, for converting an `xforms:input` element

```
<xsl:template match="xforms:input">
  <xsl:param name="instance1" as="node()?"
             select="()"/>
  <xsl:param name="bindings"
             as="map(xs:string, xs:QName)"
             select="map{}"/>
  <xsl:variable name="in-node" as="node()?">
    <xsl:evaluate xpath="@ref"
      context-item="$instance1/*:document"/>
  </xsl:variable>

  <input>
    <xsl:choose>
      <xsl:when test="
        map:get($bindings, generate-id($in-node)) =
                          xs:QName('xs:date')">
        <xsl:attribute name="type"
                       select="'date'"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:attribute name="type"
                       select="'text'"/>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:attribute name="id" select="@ref"/>
    <xsl:attribute name="value">
      <xsl:if test="exists($instance1) and
                    exists(@ref)">
        <xsl:evaluate xpath="@ref"
          context-item="$instance1/*:document"/>
      </xsl:if>
    </xsl:attribute>
  </input>
</xsl:template>
```

As well as the conversion from XForms form elements to HTML form elements, the second main purpose of the XForms implementation stylesheet is to handle form interaction. We now describe the process of capturing HTML form data, updating the data in the XForms instance data, and making a form submit.

In order to ensure that the complete XML instance data structure is sent when a user submits a form, we

hold the XML instance data as a JSON object on the page. We use JSON rather than XML for two reasons:

1. Browser inconsistencies and complications in embedding XML islands within HTML, see section 2.4 of [11].
2. The XSLT representation of maps and arrays allows for efficient modifications of immutable data structures (typically `map:put()` does not require the whole tree to be physically copied; the Saxon implementation uses an immutable trie structure to achieve this). It is much harder to achieve efficient small local changes to an XML tree, because an XML tree has node identity and parent pointers which a JSON data structure typically doesn't. Making a small change to an XML document typically involves copying the whole tree. (See [12].)

The purpose of holding the instance data is to ensure that its structure is maintained, so that when a user submits a form, the complete XML representation of the instance data is sent. When the form is submitted (e.g by HTTP post request), the JSON instance data is converted back to its XML format, and then updated with any changes that have been made in the HTML form, before being sent. (Note that it is not necessarily possible to rebuild the XML instance data from the HTML form from scratch - for example, using the XPath paths in the `id` attributes - since the HTML form may of course not be a direct mapping to the instance data. So we need to hold the instance data structure somewhere in the page.) This is achieved by a number of steps.

Firstly, the JSON object is created using the XPath 3.1 function `xml-to-json()` and added to the page in a script element. The code below shows how this is done, to add to the script element with `id="{$xforms-instance-id}"` on the HTML page. Since there is no direct mapping of the XML instance data format to JSON we first have to convert the instance data to an intermediate form - i.e. the XML representation of JSON which is accepted by the `xml-to-json()` function [13] - using our stylesheet function `convert-xml-to-intermediate()`.

```
<xsl:result-document href="{$xforms-instance-id}"
                     method="ixsl:replace-content">
  <xsl:value-of select="xml-to-json(
      local:convert-xml-to-intermediate(
        $instance-doc
      ))"/>
</xsl:result-document>
```

Secondly, the submission process is implemented using an interactive XSLT event handling template. The submit control element has been converted to an HTML button which includes generated `data-*` attributes which match the XForms submission specific attributes, such as `action`. The click event is handled by an event handling template for a button with a `data-action` attribute. At the current stage of development of the license tool, we actually override this event handling template with one which is specific to our tool (as will be described in Section 6).

Within the event handling template rule we convert the instance data held as JSON back to the XML format (going via the intermediate XML format using the XPath 3.1 function `json-to-xml()`), and from this build new updated XML instance data. As we build the new XML instance data we update with any new data from the form, by using the `id` attributes with the XPath paths. This is achieved by using an XSLT apply-templates (with `mode="form-check"`) on the XML instance data. The matching template rules keep track of the path to the matched node within the XML instance data. The template which matches text nodes then uses its path to look within the HTML form for an element whose `id` attribute value is this path. If such an element is found, and a change has been made to the form data, then the new XML instance data is updated correspondingly. See below for the full template:

```
<xsl:template match="text()" mode="form-check">
  <xsl:param name="curPath" select="''"/>
  <xsl:variable name="updatedPath"
    select="concat($curPath,
                   local-name(parent::node()),
                   '/text()')"/>

  <xsl:variable name="control">
    <xsl:apply-templates
      select="ixsl:page()//*[@id=$updatedPath]"
      mode="get-control"/>
  </xsl:variable>

  <xsl:choose>
    <xsl:when test="$control=.">
      <xsl:copy-of select="."/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="$control"/>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>
```

Some basic validation is included in our partial XForms implementation (for instance, at the loading of the page,

and the submission of the form). There is much more that needs to be implemented for full validation.

As an exploration exercise, we have certainly shown the capabilities of an XForms implementation in interactive XSLT 3.0.

6. HTTP client-server communication

Within the License Tool application, communications between client and server are made using HTTP methods. As well as when the application is initially opened in a browser, the other main point of communication (in both the old and new versions of the tool), is when a user clicks the "submit order" button in the "Edit Form" page. Clicking the button sends the form data via HTTP to the server-side Servlex webapp, and a response is then returned in the browser. However, the details of how this works in the old and new versions of the tool is quite different.

1. In the old version of the License Tool, the form is an XForms form in an (X)HTML page. This (X)HTML page is itself the response from a previous HTTP request. The (X)HTML page calls the XSLTForms implementation for XForms (which uses XSLT 1.0 and JavaScript in the browser) to render the form, and handle form interaction - e.g. making changes to the form entries, and the submit click. As defined in the XForms form controls (using an `xforms:submission` element), clicking the submit button sends the form data (held in the `xforms:instance`, in the custom XML order format) to the URI specified (in the `action` attribute) using an HTTP post request. We use the attribute `method="xml-urlencoding-post"`, an XSLTForms-specific `method` attribute option, which means that the HTTP post request contains the form data XML serialized as a string in a parameter called "postdata".

   The server-side webapp receives this request, and the relevant XSLT component picks up the value of the postdata parameter, parses it back into XML, and processes it. If the processing is successful (i.e. the order is allowed and a license is issued), then the HTTP response sent back is a newly generated HTML page (in fact, a new "Main Form" page) which contains some "success" paragraph saying that the license has been sent.
2. In the redesigned tool, the form is again an XForms form in an (X)HTML page. This time, the content of this (X)HTML page has been generated using an interactive XSLT stylesheet processed by Saxon-JS in JavaScript in the browser. The stylesheet includes interactive XSLT to dynamically insert generated fragments of HTML into the page. The XForms form is one of these generated (X)HTML fragments. The form was generated in XSLT using the prototype implementation of XForms discussed in the previous section (this implementation is an XSLT 3.0 stylesheet using interactive XSLT, which is imported into the main client-side XSLT stylesheet).

   Again, clicking the submit button sends the form data using an HTTP post request. However, this time the click event is handled by Saxon-JS. The interactive XSLT stylesheet contains an event handling template rule, which is called on click events for the submit button. The template's action is to call a user-defined JavaScript function, using the instruction

```
<xsl:sequence select="js:makeHTTPrequest(
                        serialize($orderXML))"/>
```

   This JavaScript function creates an asynchronous HTTP request (using an `XMLHttpRequest` object), with the desired URI destination, and with the data (serialized order XML) sent as content of the request, in plain text type (rather than as a parameter which would force URL encoding). (It may seem preferable to send the order XML in the request directly using the content type "application/xml". However, more work is required to find the best way to do this in JavaScript.)

   The redesigned server-side webapp receives this request, picks up the body of the request, parses it back into XML, and processes it. If the processing is successful (i.e. the order is allowed and a license is issued), then the HTTP response sent back is a piece of XML which contains some "success" data. Having used an asynchronous HTTP request, it is not possible to return the response XML directly from the `makeHTTPrequest` JavaScript function to the XSLT stylesheet; but we may produce some output in the HTML page in another way. As defined in the `makeHTTPrequest` JavaScript function, when the response XML is received by the client, it is supplied to a new call on `SaxonJS.transform` as the source XML, using a different initial template. This named template generates a fragment of HTML, containing a "success" paragraph, which is inserted into the original HTML page.

On the surface, it may not be apparent that our new version is actually an improvement. It certainly didn't seem any less complicated to explain; previously we were just using XForms, but now we're using XSLT and

JavaScript as well as XForms. One main benefit is that we now have much more freedom to define the HTTP request ourselves. Previously, because we were using XSLTForms, we were very constrained by having to send the data with a post request using the "postdata" parameter. Using this method, the content is restricted to being URL encoded, and we have not controlled the encoding used. This is one place where potentially our encoding issue arises. Parameter values are URL encoded in the browser before being sent, and we may not be dealing with this correctly on the server-side. We have now eliminated the issue at this point by controlling the HTTP request, and specifically the content (and its type), ourselves.

Actually, our new solution is only a step towards what we would really like to do, so this is one reason why it is still quite complicated. As discussed in the next paragraph, we would like to produce a solution without the need to use a locally defined JavaScript function, by providing this functionality in interactive XSLT implemented in Saxon-JS. Such a solution which only uses XForms and interactive XSLT would clearly be simpler.

Rather than using a locally defined JavaScript function to create the HTTP request (as we have done currently), it would be nice to implement this functionality directly in Saxon-JS. For instance, we could implement the HTTP Client Module [14], which provides a specification for the `http:send-request()` function. This function allows the details of the request to be specified using a custom XML format: the `<http:request>` element defined in the specification. However, the function is defined to return the content of the HTTP response. In order to return the HTTP response, we would need to use a synchronous request; but it is considered better practice and preferable to use asynchronous requests. So we would rather be able to define an extension which takes as input the request information, as well as information specifying what to do when the response is returned. Compare this proposal to the existing `ixsl:schedule-action` instruction, which makes an asynchronous call to a named template, either after waiting a specified time, or after fetching a specified document. We could add a new version which makes the call to the named template once a response from a specified HTTP request has returned. We could use the HTTP Client Module XML format for defining an HTTP request using a `http:request` element, though it may be more natural (and convenient) to use an XPath 3.0 map. The details of how best to do this are still being developed; but working on this License Tool project has been very useful as an exercise to get started, learn about the relevant technologies, and begin getting ideas to work towards a solution.

7. Conclusion

In this paper we have presented a redesign of our License Tool Web application which utilises interactive XSLT 3.0 to allow more of the processing to be done client-side, with minimum server-side processing. The interactive XSLT extensions broaden the benefits of using XSLT in the browser. To use these technologies we use the XSLT run-time engine Saxon-JS. This processor also provides the capability to call global JavaScript functions, which makes it possible to define HTTP requests and handle the responses in the browser.

The main interface of the License Tool is XForms driven. We have implemented a new XForms prototype implementation using interactive XSLT 3.0 for use in the browser. This proof-of-concept shows that it would be possible to implement the full XForms specification using interactive XSLT 3.0. We are no longer reliant on the XSLTForms implementation, which was limiting because it is an XSLT 1.0 implementation. Unfortunately even XSLT 1.0 is not well supported by all browsers - in particular many mobile browsers simply not do implement XSLT. We get around this by using the Saxon-JS XSLT processor that runs within a browser's JavaScript engine.

Can we eradicate the use of XSLT or other processing on the server-side? Possibly not as we still use Servlex to do some XSLT processing on the server. And would it be desirable? No, because for such an application it is paramount to maintain the security of sensitive data and keep data centralised. But we have certainly achieved our aim of improving our tool, so that it now processes and moves around XML data from end-to-end, and does this processing mostly on the client-side, having moved most of the processsing from the server-side environment. Removing the need for translations between so many different third-party tools and languages outside of the XDM model minimises possible failures and incompatibilies, such as encoding issues, which can only be good in the long run.

With the increase of XML data on the web, and continual demands for speed improvements, the option of using client-server distributed XSLT processing is surely attractive, though of course there may be trade-offs. While it may not become a phenomena, certainly we have showcased the innovative possibilities.

Bibliography

[1] *Servlex*. Florent Georges.
http://servlex.net

[2] *CXAN: a case-study for Servlex, an XML web framework*. Florent Georges. XML Prague. March, 2011. Prague, Czech Republic. .
http://archive.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf#page=49

[3] *XSLTForms*. Alain Couthures.
http://www.agencexml.com/xsltform

[4] *Interactive XSLT in the browser*. O'Neil Delpratt and Michael Kay. Balisage. 2013.
doi:10.4242/BalisageVol10.Delpratt01

[5] *Interactive XSLT extensions specification*. Saxonica.
http://www.saxonica.com/saxon-js/documentation/index.html#!ixsl-extension

[6] *Saxon-JS: XSLT 3.0 in the Browser*. Debbie Lockett and Michael Kay. Balisage. 2016.
doi:10.4242/BalisageVol17.Lockett01

[7] *XSL Transformations (XSLT) Version 3.0*. Michael Kay. W3C. 7 February 2017.
https://www.w3.org/TR/xslt-30

[8] *Experiences with XSLTForms and Servlex*. O'Neil Delpratt. 8 March 2013.
http://dev.saxonica.com/blog/oneil/2013/03/experiences-with-client-side-xsltforms-and-server-side-servlex.html

[9] *Web Applications*. EXPath Candidate Module. Florent Georges. W3C. 1 April 2013.
http://expath.org/spec/webapp

[10] *Packaging System*. EXPath Candidate Module. Florent Georges. W3C. 9 May 2012.
http://expath.org/spec/pkg

[10b] *XPath 3.1 in the Browser*. John Lumley, Debbie Lockett and Michael Kay. XML Prague. February, 2017. Prague, Czech Republic.
http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=13

[11] *HTML/XML Task Force Report*. W3C Working Group Note. Norman Walsh. W3C. 9 February 2012.
https://www.w3.org/TR/html-xml-tf-report/

[12] *Transforming JSON using XSLT 3.0*. Michael Kay. XML Prague. February, 2016. Prague, Czech Republic.
http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#page=179

[13] *XML Representation of JSON*. XSL Transformations (XSLT) Version 3.0, W3C Recommendation. Michael Kay. W3C. 7 February 2017.
https://www.w3.org/TR/xslt-30/#json-to-xml-mapping

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

Location trees enable XSD based tool development

Hans-Jürgen Rennau

Traveltainment GmbH

<hrennau@yahoo.de>

Abstract

Conventional use of XSD documents is mostly limited to validation, documentation and the generation of data bindings. The possibility of additional uses is little considered. This is probably due to the difficulty of processing XSD, caused by its arcane graph structure. An effective solution might be a generic transformation of XSD documents into a tree-structured representation, capturing the model contents in a transformation-friendly way. Such a tree-structured schema derivative is offered by location trees, a format defined in this paper and generated by an open-source tool. The intended use of location trees is an intermediate to be transformed into interesting artifacts. Using a chemical image, location trees can play the role of a catalyst, dramatically lowering the activation energy required to transform XSD into valuable substances. Apart from this capability, location trees are composed of a novel kind of model components inviting the attachment of metadata. The resulting metadata trees enable innovative tools, including source code generators. A few examples illustrate the new possibilities, tentatively summarized as XSD based tool development.

Keywords: XML Schema, XML tool chain

1. Introduction

Well-written XML schema documents (XSD) contain a wealth of information. Its potential value is by no means exhausted by conventional schema uses, which are validation, documentation and data binding (e.g. [3]). Why are there hardly any tools available for unlocking the treasure? A major reason is probably the sheer difficulty to write code which evaluates XSD documents reliably - coping with the complete range of the XSD language, rather than being limited to XSD documents written in a particular style.

The difficulty of processing XSD has a key reason: the semantics of XSD are graph-structured, not tree-structured like the instance documents which XSD describes. To realize this, think of the many relationships between schema components, like uses-type, extends-type, restricts-type, uses-group, uses-attributegroup, memberof-substitutiongroup. A clear perception of the problem suggests a straightforward solution: we need a generic and freely accessible transformation of XSD into a tree-structured equivalent. This can be used as a *processing-friendly intermediate* which is readily transformed into useful artifacts. The new intermediate should play a role similar to a catalyst in chemical processes: it is neither input nor output, but it reduces the energy required to turn input into output. Availability of this catalyst might turn the development of XSD processing tools into a fascinating and rewarding challenge.

Departing from an analysis of the main problems besetting XSD processing, this paper proposes a solution based on a tree-structured representation of XSD contents. The new format is explained in detail, and the new possibilities of XSD processing are illustrated by several examples.

2. Problem definition

XSDs describe the structure and constrain the contents of XML documents ([5], [6], [7], [8]). These documents are tree-structured. The schema itself, however, is a peculiar mixture of tree and graph structure. It is a graph whose nodes are trees (e.g. type definitions) as well as nodes within trees (e.g. element declarations within type definitions). The edges of this graph connect trees (e.g. a type references its base type) as well as tree nodes and trees (e.g. an element references its type definition). The data model of XSD is justified by the conflicting needs to describe tree structure and to support reusable components. (If not for the second, XSD would probably consist of straightforward representations of the desired tree structures!) Nevertheless, the model as it is creates several problems which deserve attention.

doi:10.14337/XMLLondon17.Rennau01

Ignoring the problems, we would limit ourselves to the services of mature and completed tools. Thus we would miss the chance to discover innovative uses of the precious information content locked up in XSD documents.

2.1. The problem of understanding

To understand a schema means, in the first place, to understand the tree structure of its instance documents. However, this is often difficult - or even virtually impossible - when studying the schema contents in their raw form.

Fortunately, several commercial IDEs offer graphical representation of schemas. At first sight, they give us everything we need in order to understand the schema: the graphical representation is very clear and intuitive. But when working with large schemas, a serious shortcoming becomes obvious. The graphical representation is very space consuming, rendering fast browsing of larger chunks of the schema impossible. The screen is completely filled by a small number of tree nodes; a *sliding* glance at more than a small piece of a large whole requires a "bumpy" navigation involving much scrolling mixed with a series of clicks for alternately collapsing and expanding nodes.

2.2. The query problem

A second problem is closely related to the problem of understanding. Managing a large schema requires more than clear pictures of pieces of tree structure: it requires *querying* the schema. Example questions: Which data paths will be affected if I change this element declaration? (There may be hundreds of them, but also none at all!) Do all `<Foo>` elements have the same type? What are the elements on which a @Bar attribute may appear? Compared to the previous schema version, what data paths have been added or removed? If the schema is large, the number of tree nodes is too large for finding the answers by visual inspection. The IDEs give us icon trees, but no data trees, no data sets for queries which give us the answers.

2.3. The transformation problem

Schemas describe the tree structure of instance documents, and in doing so they define crucial information about relevant domain entities. Questions arise: What are the key entities, what are their properties, what are the data types of those properties, what are their cardinalities? We should have tools to reorganize such information into comprehensive and intuitively structured representations. But the required schema transformation is too difficult for a developer who is not a schema specialist.

2.4. The metadata problem

XML schema components can be associated with annotations. Special schema elements are available for this purpose (`xs:annotation`, `xs:documentation`, `xs:appinfo`). The schema author may also enhance any schema component with attributes which are not defined by the schema language. The names of these additional attributes are only limited to belong to a non-XSD namespace. This possibility to add arbitrary name/value pairs means language support for the addition of metadata.

When dealing with data-oriented XML, elements and attributes describe real-world entities and metadata can extend these descriptions, e.g. adding processing hints. Metadata of an element declaration might, for instance, specify the data source from where to retrieve the value of the element, as well as the data target where to store it:

```
<xs:element name="loyaltyNumber" type="xs:string"
x:dataSource="msgs.travelInfo#//Passenger/@loyalty"
x:dataTarget="dbs.someDb.someTab.someCol"/>
```

Such metadata could be put to many uses, including code generation and system consistency checks (are the schema types of data source and data target compatible?). But the possibilites are much more limited than they appear at first sight, due to the *reuse* of schema components (like type definitions) in different contexts: such metadata would typically apply in one context, but not in the other. We come to realize that many kinds of interesting metadata cannot be attached to the schema components themselves: they should be attached to a novel kind of entity representing the use of a particular element or attribute declaration at a particular place within a particular document type. Such *component use* is an abstraction lying midway between a schema component and the items of an instance document. This paper attempts to capture component use conceptually, proposing the notion of an *info location*. It introduces the means to materialize component uses, turning them into the elements of a new kind of resource called a *location tree*.

3. Location trees

An info location tree (**location tree**, for short) is an XML document which represents the tree structure of documents described by an XML schema. More precisely, a location tree captures the tree structure implied by a complex type definition or a group definition. As a rough approximation, think of location trees as XML documents capturing the information visualized by the familiar graphical views of schema documents.

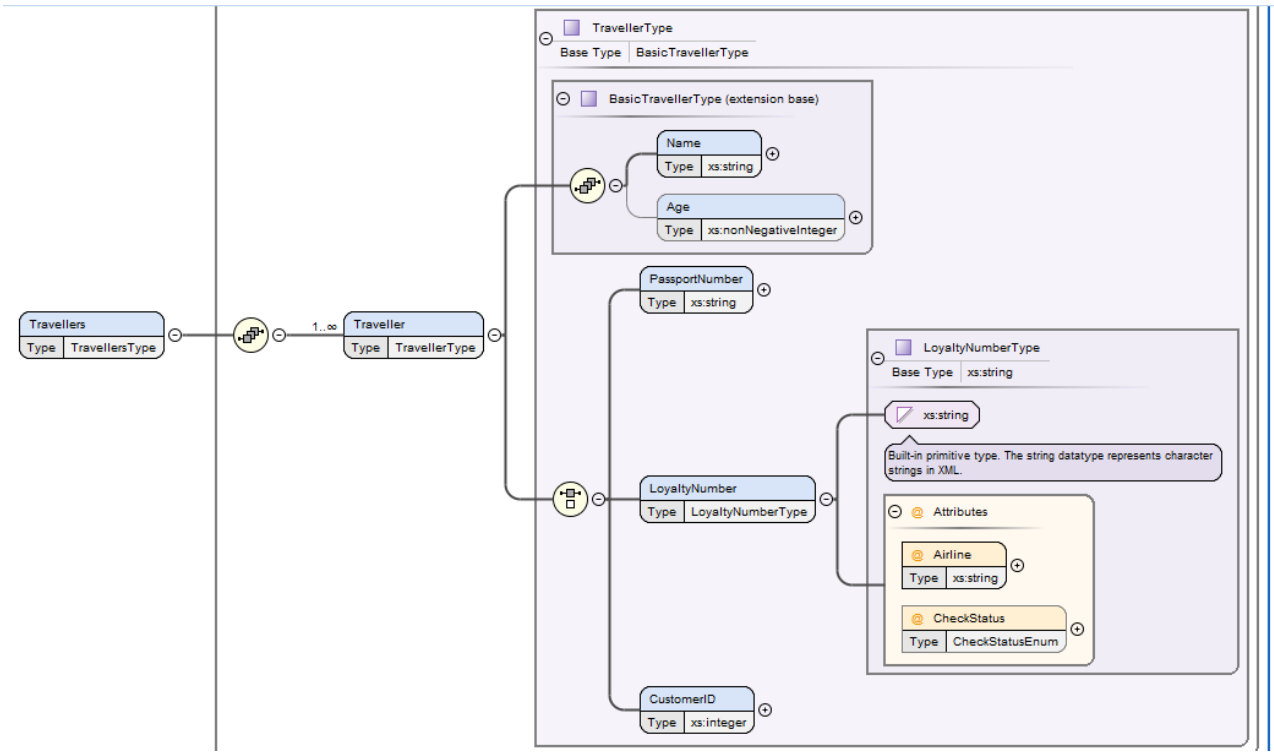
3.1. A simple example

Let us look at a simple example. The following schema defines documents rooted in a `<Travellers>` element:

```
<xs:schema xmlns="http://example.com/ns"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.com/ns"
  elementFormDefault="qualified">
  <xs:element name="Travellers"
    type="TravellersType"/>
  <xs:complexType name="TravellersType">
    <xs:sequence>
      <xs:element name="Traveller"
                  type="TravellerType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="BasicTravellerType">
    <xs:sequence>
      <xs:element name="Name"
                  type="xs:string"/>
      <xs:element name="Age"
                  type="xs:nonNegativeInteger"
                  minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="TravellerType">
    <xs:complexContent>
      <xs:extension base="BasicTravellerType">
        <xs:choice>
          <xs:element name="PassportNumber"
                      type="xs:string"/>
          <xs:element name="LoyaltyNumber"
                      type="LoyaltyNumberType"/>
          <xs:element name="CustomerID"
                      type="xs:integer"/>
        </xs:choice>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="LoyaltyNumberType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="Airline"
                      type="xs:string"
                      use="required"/>
        <xs:attribute name="CheckStatus"
                      type="CheckStatusEnum"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType name="CheckStatusEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="NoCheck"/>
      <xs:enumeration value="Ok"/>
      <xs:enumeration value="NotOk"/>
      <xs:enumeration value="Unknown"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Though this is a very small and simple schema, the tree structure of a `Travellers` document is not immediately obvious. Figure 1 offers a graphical representation of this tree structure, created using the XML IDE Oxygen 18.1.

Figure 1. Graphical XSD representation (Oxygen 18.1)



Graphical representation of the `Travellers` schema, created by Oxygen 18.1

Such a representation is tremendously helpful, but its use is restricted to human inspection, as it cannot be used as processing input. A processing-friendly alternative is offered by a location tree. The following listing shows a pruned version, leaving out items (including type information) for the sake of readibility:

```
<a:Travellers xmlns:a="http://example.com/ns">
   <a:Traveller z:occ="+">
      <a:Name/>
      <a:Age z:occ="?"/>
      <z:_choice_>
         <a:PassportNumber/>
         <a:LoyaltyNumber>
            <z:_attributes_>
               <Airline/>
               <CheckStatus occ="?">
            </z:_attributes_>
         </a:LoyaltyNumber>
         <a:CustomerID/>
      </z:_choice_>
   </a:Traveller>
</a:Travellers>
```

The meaning of this representation is easy to grasp intuitively. Instance documents have a `<Travellers>` root element which contains one or more `<Traveller>` elements. Note the @z:occ attribute on `<Traveller>` which indicates that the `<Traveller>` element of the location tree represents a sequence of one or more elements in the instance document. We understand that the first child of `<Traveller>` is a `<Name>` element, which is followed by an optional `<Age>` element. The last element in `<Traveller>` is one of three possibilities: either a `<PassportNumber>` element, or a `<LoyaltyNumber>` element, or a `<CustomerID>` element. Note that the `<z:_choice_>` element does not represent any node in an instance document; rather, it is an auxilliary element indicating that at this point of the structure the contents of instance documents is given by exactly one of several possibilities. Finally we note that `<LoyaltyNumber>` elements have a couple of attributes, @Airline (required) and @CheckStatus (optional).

Roughly speaking, each location tree element which is not in the z-namespace represents a set of XML elements or attributes - a set comprising all items found within instance documents "at the same location". How to define the location? All items found at a particular location have the same data path (e.g. `/Travellers/Traveller/Age`). Note that the inverse is not always true (though it is always true in the example): two elements

may have the same data path but occupy different locations. This would for example be the case if an element's content model were the sequence of child elements <a>, <b>, <c>, <a>. All <a> elements would have the same datapath, but the first and last siblings among them would occupy two distinct locations. This matches the intuitive expectation of a location tree or a graphical schema view: the siblings <a>, <b>, <c>, <a> should certainly be represented by four location nodes or graph icons, rather than three.

3.2. Info locations

In spite of the obvious intuitive meaning of location tree nodes, a formal definition of their semantics is not trivial. The key abstraction is an **info location**, which is akin to an RDF class whose members are XML nodes. An info location is a class of XML nodes belonging to a document type's instance documents, where class membership depends on the node's validation path. The validation path is the sequence of schema components used to validate the item itself, its ancestors and their preceding siblings, when validated in document order. Two items are said to occupy the same location if their validation paths are equal according to an algorithm of comparison. (Note. The comparison of validation paths must be performed according to an algorithm, rather than as a straightforward comparison, in order to ignore the irrelevant variability introduced by optional and multiple occurrences.)

The following listing shows the complete location tree of <Travellers> elements. **Element locations** are represented by elements not in the z-namespace, excluding child elements of <z:_attributes_> elements. **Attribute locations** are represented by child elements of <z:_attributes_> elements. The properties of the info locations are exposed by attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<z:locationTrees xmlns:a="http://example.com/ns"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:z="http://www.xsdplus.org/ns/structure"
  count="1">
  <z:locationTree compKind="elem"
    z:name="a:Travellers"
    z:loc="element(a:Travellers)"
    z:type="a:TravellersType"
    z:typeLoc="complexType(a:TravellersType)">
    <z:nsMap>
      <z:ns prefix="a"
        uri="http://example.com/ns"/>
      <z:ns prefix="xs"
        uri="http://www.w3.org/2001/XMLSchema"/>
      <z:ns prefix="z"
        uri="http://www.xsdplus.org/ns/structure"/>
    </z:nsMap>
    <a:Travellers
      z:name="a:Travellers"
      z:type="a:TravellersType"
      z:typeVariant="cc"
      z:typeLoc="complexType(a:TravellersType)">
      <a:Traveller
        z:name="a:Traveller"
        z:occ="+"
        z:type="a:TravellerType"
        z:typeVariant="cc"
        z:baseType="a:BasicTravellerType"
        z:derivationKind="extension"
        z:typeLoc="complexType(a:TravellerType)"
        z:loc="complexType(a:TravellersType)/
        xs:sequence/a:Traveller"
        name="Traveller"
        type="TravellerType"
        maxOccurs="unbounded">
        <a:Name
          z:name="a:Name"
          z:type="xs:string"
          z:typeVariant="sb"
          z:typeDef="string"
          z:loc="complexType(a:BasicTravellerType)/
          xs:sequence/a:Name"
          name="Name"
          type="xs:string"/>
        <a:Age z:name="a:Age"
          z:occ="?"
          z:type="xs:nonNegativeInteger"
          z:typeVariant="sb"
          z:typeDef="nonNegativeInteger"
          z:loc="complexType(a:BasicTravellerType)/
          xs:sequence/a:Age"
          name="Age"
          type="xs:nonNegativeInteger"
          minOccurs="0"/>
        <z:_choice_>
          <a:PassportNumber
            z:name="a:PassportNumber"
            z:type="xs:string"
            z:typeVariant="sb"
            z:typeDef="string"
            z:loc="complexType(a:TravellerType)/
            xs:choice/a:PassportNumber"
            name="PassportNumber"
            type="xs:string"/>
          <a:LoyaltyNumber
```

```
z:name="a:LoyaltyNumber"
z:type="a:LoyaltyNumberType"
z:typeVariant="cs"
z:baseType="xs:string"
z:derivationKind="extension"
z:builtinBaseType="xs:string"
z:contentType="xs:string"
z:contentTypeVariant="sb"
z:contentTypeDef="string"
z:typeLoc="
  complexType(a:LoyaltyNumberType)"
z:loc="complexType(a:TravellerType)/
xs:choice/a:LoyaltyNumber"
name="LoyaltyNumber"
type="LoyaltyNumberType">
<z:_attributes_>
  <Airline
    z:name="Airline"
    z:type="xs:string"
    z:typeVariant="sb"
    z:typeDef="string"
    z:loc="
    complexType(a:LoyaltyNumberType)/
    @Airline"
    name="Airline"
    type="xs:string"
    use="required"/>
  <CheckStatus
    z:name="CheckStatus"
    z:occ="?"
    z:type="a:CheckStatusEnum"
    z:typeVariant="sa"
    z:typeDef="string:
       enum=(NoCheck|NotOk|Ok|Unknown)"
    z:baseType="xs:string"
    z:derivationKind="restriction"
    z:builtinBaseType="xs:string"
    z:typeLoc="
      simpleType(a:CheckStatusEnum)"
    z:loc="
      complexType(a:LoyaltyNumberType)/
      @CheckStatus"
    name="CheckStatus"
    type="CheckStatusEnum">
    <z:_stypeTree_
      z:name="a:CheckStatusEnum">
      <z:_builtinType_
        z:name="xs:string"/>
      <z:_restriction_
        z:name="a:CheckStatusEnum">
        <z:_enumeration_
          value="NoCheck"/>
        <z:_enumeration_
          value="Ok"/>
        <z:_enumeration_
          value="NotOk"/>
        <z:_enumeration_
          value="Unknown"/>
      </z:_restriction_>
    </z:_stypeTree_>
  </CheckStatus>
</z:_attributes_>
</a:LoyaltyNumber>
<a:CustomerID
  z:name="a:CustomerID"
  z:type="xs:integer"
  z:typeVariant="sb"
  z:loc="complexType(a:TravellerType)/
  xs:choice/a:CustomerID"
  name="CustomerID"
  type="xs:integer"/>
</z:_choice_>
</a:Traveller>
</a:Travellers>
</z:locationTree>
</z:locationTrees>
```

The next two sections give an overview of the elements and attributes used by location trees.

3.3. Location tree structure

Now we take a closer look at the structure of location trees. The following table summarizes the elements of which a location tree is composed. (Descendants of `<z:_stypeTree_>` elements are omitted.)

Table 1. Location tree elements

Location tree element	Meaning	XSD element
Any element not in the z-namespace	Represents an element or attribute location	xs:element, xs:attribute
z:_attributes_	Wraps the representations of all attribute locations belonging to an element location	-
z:_sequence_	Represents a `sequence` compositor	xs:sequence
z:_choice_	Represents a `choice` compositor	xs:choice
z:_all_	Represents an `all` compositor	xs:all
z:_any_	Represents an element wildcard	xs:any
z:_anyAttribute_	Represents an attribute wildcard	xs:anyAttribute
z:_sgroup_	Contains the element locations corresponding to a substitution group	-
z:_groupContent_	Signals a group reference which *remains unresolved* as it constitutes a cyclic definition (the reference occurs in the content of a member of the referenced group or in its descendant)	xs:group (with a @ref attribute)
z:_stypeTree_	Structured representation of a simple type definition	xs:simpleType
z:_annotation_	Reports a schema annotation	xs:annotation
z:_documentation_	Reports a schema documentation	xs:documentation
z:_appinfo_	Reports a schema appinfo	xs:appinfo

Note that group definitions (`<xs:group>`) are not represented by distinct location tree elements, as a group reference is represented by the content of the referenced group. Similarly, attribute group references are always resolved and attribute group definitions (`<xs:attributeGroup>` do not appear as distinct entities.

An important aspect of location tree structure is the representation of element composition (sequence, choice and all group). Representation rules aim at simplification: the number of group compositor elements is reduced, nested grouping is replaced by flattened grouping if possible, and irrelevant variability of XSD content is removed by normalized representation. Simplification is achieved by the definition of default composition and rules of group normalization.

Sequence is the **default composition**: any sequence of items (element locations and or compositors) which are children of an element location represents a sequence of child elements. The location tree uses only such `<z:_sequence_>` elements as cannot be left out without loss of information. This is the case when the sequence has a minimum or maximum occurrence unequal 1, or when the sequence is child of a choice. **Group normalization** is achieved by content rewriting. It replaces the use of compositors as found in the XSDs by a simpler model which is equivalent with respect to the instance documents described. Normalization rules effect the removal of "pseudo groups" containing a single item, the flattening of nested choices and the flattening of nested sequences. Note that the removal of a compositor element may change the occurrence constraints of its child elements. As an example, consider a pseudo group occurring one or more times and containing an optional element (occurrence "zero or one"). After unwrapping this element, its occurrence will be "zero or more".

It is important to understand the handling of type derivation. Remember the general goal to capture the structure of instance documents by a model structure which is as similar and straightforward as possible. Element content defined by a derived complex type is always represented in resolved form, expressing the integration of base type content and derived type content as defined by the XML schema specification. Consider a

type extending another type with complex content. The derived type is treated like a non-derived type with equivalent content: its attributes comprise all attributes of derived type definition and the (recursively resolved) base type definition; and its content is a sequence containing the contents of the (recursively resolved) base type, followed by the explicit content model of the derived type. Subsequent group normalization ensures that derivation does not introduce any structural complexity.

3.4. Location tree attributes

The elements of a location tree (element and attribute locations, group compositors, wildcards and substitution groups) represent model components – the building blocks of which the model is composed. The **properties** of these components are represented by info attributes. Example properties are the element or attribute name, type name and cardinality constraints. The following table compiles the most important attributes used to express component properties.

Table 2. Property attributes

Location tree attribute	Property	Remarks
z:name	Element name, attribute name	As normalized qualified name
z:occ	Cardinality	Examples: *?, *, +, 1-4, 2*
z:type	Type name	As normalized qualified name
z:typeVariant	Type variant	sb: builtin type
		sa: simple type, atomic
		sl: simple type, list
		su: simple type, union
		cs: complex type, simple content
		cc: complex type, complex content
		cc: complex type, empty content
z:typeDef	A human-readible string representation of a simple type definition	Examples: *string:enum=(NotOk|Ok) string:maxLen=64*
z:baseType	Base type name	As normalized qualified name
z:builtinBaseType	Name of the builtin base type (if type variant = sa)	As normalized qualified name
z:contentType	Name of the content type (if type variant = cs)	As normalized qualified name
z:contentTypeVariant	Content type variant (if type variant = cs)	See z:typeVariant; value is one of: sb, sa, sl, su
z:contentTypeDef	A human-readible string representation of the content type (if type variant = cs)	Examples: *decimal:range=[0.901,1.101] string:pattern=#[A-Z]{2}#*
z:itemType	Name of the list item type (if type variant = sl)	As normalized qualified name
z:itemTypeVariant	Item type variant (if type variant = sl)	See z:typeVariant; value is one of: sb, sa, su
z:typeLoc	XSD type location	If item has a user-defined type: identifies the location of the type definition within the XSDs
z:loc	XSD component location	Identifies the location of the component within the XSDs
any name without namespace	XSD component property (e.g. {min occurs})	The attribute is a copy of the XSD attribute found on the XSD element represented by the location tree element (e.g. @minOccurs)
any name in a namespace which is not the z-namespace	XSD annotation property (e.g. {myorg:lastUpdate})	The attribute is a copy of the annotation attribute found on the XSD element represented by the location tree element (e.g. @myorg:lastUpate)

Normalized qualified names use prefixes which are derived from the target namespaces (and possibly also annotation namespaces) encountered in the set of XSDs currently considered. The mapping of namespace URI to prefix is achieved by assigning to the *sorted* set of namespace URIs the prefixes a – y, a2 – y2 etc. The prefix z is reserved for the URI http://www.xsdplus.de/ns/structure, and the prefix xs is used for the XSD namespace. All URI to prefix mappings are documented by a `<z:nsMap>` element which is child of the `<z:locationTree>` element.

The location attributes (@z:loc, @z:typeLoc) identify an individual XSD component and thus document the alignment between location tree and XSD components. The XSD component is identified by a simple path syntax consisting of a first step identifying a top-level component (e.g. `complexType(a:TravellersType)`) followed by an optional logical path (e.g. `xs:choice/a:LoyaltyNumber`) drilling down into the top-level component. The @z:loc attribute of an element location (attribute location) identifies the element declaration (attribute declaration) aligned with the location. Similarly, @z:typeLoc identifies the type definition referenced by an element or attribute location.

Two further attributes are used in order to flag an element location or a group reference as "recursion point" (see Table 3. Recursion point attributes).

3.5. Open source tool for creating location trees

An open source tool for transforming arbitrary XSD into location trees is available [10]. The tool is written in the XQuery language [9]. It can be used as a command-line tool, or as an XQuery library ready for import by XQuery programs. Execution requires an XQuery processor supporting XQuery version 3.1 or higher. See [10] for a documentation of tool usage. Command-line invocation requires two mandatory parameters:

- A FOXpath expression [2] selecting one or more XSD documents
- A name pattern selecting the schema components for which to construct the location trees; the name pattern can alternatively be used to select element declarations, complex type definitions or group definitions

Tool output is an XML document with a `<z:locationTrees>` root element containing `<z:locationTree>` elements representing location trees obtained for the selected schema components. Here comes an example call, using the XQuery processor BaseX [1]:

```
basex -b request=ltrees?xsd=/projects/ota//*.xsd,
        enames=*RQ xsdp.xq
```

It creates location trees for each element declaration whose name ends with RQ.

4. XSD based tool development

Location trees are ordinary XML documents. The semantic graph structure of XSD documents has been replaced by semantic tree structure, so that processing location trees is a straightforward operation. Initial transformation of XSD into location trees makes valuable information, hitherto virtually out of reach from a developer's point of view, readily accessible. Location trees thus encourage the development of innovative XSD processing tools. This section illustrates the new possibilites by several examples ranging from very simple to fairly complex tools.

4.1. Getting your feet wet - first schema queries

The simplest way of using location trees consists in translating questions about a schema into queries of its location trees. As an example, consider this question about a schema: *at which places (expressed as data paths)*

Table 3. Recursion point attributes

Location tree attribute	Meaning	Value
z:recursionType	The type of the parent element location is equal to the type of an ancestor element location	Type name, as normalized qualified name
z:recursionGroup	Flags a group reference found within the contents of an element which is a member of the referenced group, or is a descendant of such a member	Group name, as normalized qualified name

do instance documents with a `<Travellers>` root contain elements with the name "CustomerID"?

Our starting point is the location tree obtained for the top-level element declaration with name "Travellers". The following XQuery query, which expects as context node a `z:locationTrees` document, gives the answer:

```
declare namespace z =
  "http://www.xsdplus.org/ns/structure";
declare namespace f =
  "http://www.xsdplus.org/ns/xquery-functions";

declare function f:path(
  $e as element()) as xs:string {
  '/' ||
  string-join((
    $e/ancestor::*[not(self::z:*)]/local-name(),
    $e/concat(parent::z:_attributes_/'@',
    local-name())), '/')
};

//*:CustomerID/f:path(.)
```

Extending this adhoc query into a veritable tool is easy: we introduce two external variable which expect name patterns for the document root element and the items of interest:

```
declare namespace z =
  "http://www.xsdplus.org/ns/structure";
declare namespace f =
  "http://www.xsdplus.org/ns/xquery-functions";
declare variable $root external := '*';
declare variable $item external := '*';

declare function f:path(
  $e as element()) as xs:string {
  '/' ||
  string-join((
    $e/ancestor::*[not(self::z:*)]/local-name(),
    $e/concat(parent::z:_attributes_/'@',
    local-name())), '/')
};

declare function f:regex(
  $name as xs:string) as xs:string {
  concat(
    '^',
    replace(replace($name, '\*', '.*'), '\?', '.'),
    '$'
  )
};
/*/z:locationTree/(* except z:*)[
      matches(local-name(.), f:regex($root), 'i')]
//(* except z:*)[
  matches(local-name(),
          f:regex($item), 'i')]/f:path(.)
```

After storing this code in a file `locationPaths.xq` and writing the location trees of all top-level elements of the Niem 3.0 XSDs (downloaded from [4]) into a file `ltrees-niem30.xml`:

```
basex -b "request=ltree?xsd=/xsdbase/niem-3.0//
          *.xsd,enames=*,global"
          xsdp.xq > ltrees-niem30.xml
```

the following invocation

```
basex -i ltrees-niem30.xml -b item=*xml*
      locationPath.xq
```

yields this output:

```
/EDXLDistribution/contentObject/nonXMLContent
/EDXLDistribution/contentObject/xmlContent
/EDXLDistribution/contentObject/xmlContent/keyXMLContent
/EDXLDistribution/contentObject/xmlContent/embeddedXMLContent
```

These four data paths leading to items with a name containing "xml" have been selected from 90763 data paths which can be obtained from the location trees in a straightforward way (see function `f:path` above). The example shows how simple queries applied to location trees can give valuable insight which would be very difficult to obtain by processing the XSD documents themselves. The next section discusses more complex uses of location trees.

4.2. Schema reporting - treesheets

Location trees are not meant for human inspection – they are intermediates serving as input to the query or transformation producing an artifact of interest. An important category of such artifacts are various representations of the schema-defined tree structures and information associated with their main building blocks, the elements and attributes.

A **treesheet** is a text document which combines an indentation-based representation of the tree structure (on the left-hand side) with information about the info locations of which the tree is composed (on the right-hand side). The following treesheet, for example

```
Element: Travellers;  Type: TravellersType
==========================================

Travellers     ~    ~    ctype: a:TravellersType
. Traveller+   ~    ~    ctype: a:TravellerType
. . Name  ...  ...  ...  type:  string
. . Age?  ...  ...  ...  type:  nonNegativeInteger
. . choice{
. . 1 PassportNumber  .  type:  string
. . 2 LoyaltyNumber  ..  type:  string
. . 2   @Airline .  ...  type:  string
. . 2   @CheckStatus? .  type:  string: enum=(NoCheck|NotOk|Ok|Unknown)
. . 3 CustomerID    ...  type:  integer
. . }
```

displays type information, whereas the next one

```
Element: Travellers;  Type: TravellersType
==========================================

Travellers     ~    ~    anno: A travel party.
. Traveller+   ~    ~    anno: Represents an individual traveller.
. . Name  ...  ...  ...  anno: The surname, as used in the passport.
. . Age?  ...  ...  ...  anno: The age at checkin date.
. . choice{
. . 1 PassportNumber  .  anno: The passport number in latin letters.
. . 2 LoyaltyNumber  ..  anno: The Loyalty number, assigned by the airline.
. . 2   @Airline .  ...  anno: Airline, identified by IATA code.
. . 2   @CheckStatus? .  anno: Specifies if checked and the check result.
. . 3 CustomerID    ...  anno: The customer ID, assigned by the back office.
. . }
```

displays the contents of schema annotations, retrieved from `<xs:documentation>` elements of the schema. These examples align the tree items with information originating from the XSD (type information and schema annotation). The information might however also be metadata externally provided, or facts based on instance

document data. (See the following sections for more about facts and metadata.) Each type of treesheet can be viewed as a different facet of the schema-defined tree. Opening different treesheets of the same document type in an editor and switching between their tabs can offer an interesting experience of contemplating complex information trees from different sides. The open source tool [10] provides a `treesheet` operation transforming XSD components into treesheets of various kinds and controlled by various representational options.

4.3. Fact trees

A location tree can be regarded as a model of item use: given a document type, *where* (data path) to use *what* (element/attribute name) and *how* (number of occurrences, data type or content structure). Any set of instance documents, on the other hand, generates facts about how items are actually used. Example facts about an info location are the frequency of documents containing items in this location, as well as the value strings used in these items. Interesting reports can be created by merging these facts into a pruned version of the location tree. Recipe:

- Starting point: location tree
- Remove all elements which do not represent compositors, elements or attributes
- Remove all or a selection of attributes
- Add attributes conveying facts

A simple example uses several attributes telling us how info locations are actually used:

- @dcount – the number of documents observed (attribute at the root element only)
- @dfreq – the relative frequency of documents containing an item at this location
- @ifreq – the mean/minimum/maximum number of items at this location, ignoring documents without items in this location (attribute omitted if the schema does not allow for multiple items)

The resulting fact tree may reveal interesting details, like choice branches never used, or optional elements never or virtually always used:

```
<a:Travellers xmlns:a="http://example.com/ns"
              dcount="78925">
  <a:Traveller z:occ="+" dfreq="1.00"
               ifreq="3.01 (1-8)">
    <a:Name dfreq="1.00"/>
    <a:Age z:occ="?" dfreq="0.99"/>
    <z:_choice_>
      <a:PassportNumber dfreq="0.64"/>
      <a:LoyaltyNumber dfreq="0.36">
        <z:_attributes_>
          <Airline dfreq="0.35"/>
          <CheckStatus dfreq="0.09"/>
        </z:_attributes_>
      </a:LoyaltyNumber>
      <a:CustomerID dfreq="0"/>
    </z:_choice_>
  </a:Traveller>
</a:Travellers>
```

If the fact tree has been obtained for a set of request messages belonging to test suites, attributes like @dfreq can express test coverage in a meaningful way.

A fact tree can be transformed into a treesheet:

```
Element: Travellers;  Type: TravellersType
==========================================

Travellers     ~    ~    count: 78925
. Traveller+   ~    ~    dfreq: ********** (1.00, ifreq=3.01 (1-8)
. . Name  ...  ...  ...  dfreq: ********** (1.00)
. . Age?  ...  ...  ...  dfreq: ********** (0.99)
. . choice{
. . 1 PassportNumber  .  dfreq: ******     (0.64)
. . 2 LoyaltyNumber  ..  dfreq: ****       (0.36)
. . 2   @Airline .   ...  dfreq: ****       (0.35)
. . 2   @CheckStatus ..  dfreq: *          (0.09)
. . 3 CustomerID     ...  dfreq:           (0)
. . }
```

Note that the concept of "facts about locations" is not limited to usage statistics. The next section discusses the enhancement of locations by metadata. Such metadata may imply new kinds of facts which can be represented by specialized fact trees. Assume, for example, location metadata which specify from where to retrieve item data. If those "places" can also be expressed as info locations (e.g. belonging to the location tree of some web service message), *derived facts* of data consistency emerge. If the data source of an enumerated string is an unconstrained string, a data type inconsistency emerges as a fact.

The effort required to transform location trees into treesheets and fact trees is moderate, although it is substantially greater than the simple queries shown earlier. The final section about the use of location trees presents a complex application performing code generation. It is based on metadata trees which are created by a semi-automatic procedure: automated transformation of location trees into an initial version of a metadata tree, followed by manual editing which replaces the generated dummy values of metadata attributes with real values.

4.4. Metadata trees and code generation

Location trees are composed of nodes representing the elements and attributes which can be used by instance documents. Adding to the nodes of a location tree metadata, we obtain a metadata tree. Recipe:

- Starting point: location tree
- Remove all elements which do not represent compositors, elements or attributes
- Remove all or a selection of attributes
- Add attributes specifying metadata

Like a location tree, a metadata tree captures the structure of instance documents. Unlike location tree nodes, however, metadata tree nodes describe the items of instance documents *beyond XSD-defined information*. If metadata provide information about how to process the items, the metadata tree may be transformed into source code which implements the processing.

This section discusses an example of source code generation based on metadata trees. Our goal is a generator of program code which implements the transformation of source documents with a particular document type (e.g. "TravelInfo") into target documents with a different document type (e.g. "Travellers"). The solution involves the following main steps:

1. Design a metadata model
2. Create a metadata tree generator
3. Create a metadata tree transformator

4.4.1. Design a metadata model

Our approach is to use a metadata tree derived from the location tree of the *target* document type. The metadata model must define a set of metadata which suffices to determine every detail of the code to be generated. Note that the required types of metadata may be different for different nodes of the location tree: they will depend on node properties like the content type variant (simple versus complex) or occurrence constraints. Each type of metadata is represented by attributes with a particular name. Metadata values are XPath expressions to be evaluated in the context of source document nodes. The following table provides an overview of the more important metadata types and how their usage depends on node properties.

Table 4. Metadata model for doc2doc transformation

Attribute name	Node condition	Meaning	Example
alt	Location of a simple content element or attribute, which is optional	XPath expression; evaluated if @src yields the empty sequence; if @alt yields a non-empty value, a target node is created and the value is used	`'#UNKNOWN'`
case	Child of a `<z:_choice_>`	XPath expression; the selected choice branch is the first child of the `choice` compositor whose @case has a `true` effective boolean value	`@Success eq "false"`
ctxt	Complex element location	XPath expression; its value is used as the new data source context	`Services/Hotel`
default	Location of a simple content element or attribute, which is mandatory	XPath expression; its value is used if @src yields the empty sequence	`'?'`
for-each	Element location (or compositor) with maxOccurs > 1	XPath expression; for each item of the expression value an element node is created (or compositor contents are evaluated)	`Contacts/Contact`
if	Complex element location (or compositor) which is optional	XPath expression; if its effective boolean value is `true`, an element node is created (or compositor contents are evaluated)	`Hotel/Properties`
src	Location of a simple content element or attribute	XPath expression; its value is used as element content or attribute value	`@Currency`

4.4.2. Create a metadata tree generator

Writing the metadata tree generator is a fairly straightforward task: the generator transforms location trees into a modified copy in which most location tree attributes are removed and attributes representing the metadata are added. The values of metadata attributes are placeholders, to be replaced by hand-editing. For each location tree node an appropriate selection of metadata attributes is made, according to the "Node condition" column in table Table 4, “Metadata model for doc2doc transformation”. Conditions are checked by evaluating the attributes of the location tree node. The condition "Location of a simple content element or attribute", for instance, is true if the location tree attribute @z:typeVariant has a value starting with "s" or equal to "cs". If the condition also requires that the location is optional, the @z:occ attribute is also checked – and so forth.

Applying our metadata tree generator to the `Travellers` schema, we obtain this initial metadata tree:

```
<a:Travellers xmlns:a="http://example.com/ns"
                        ctxt="###">
   <a:Traveller         for-each="###">
      <a:Name           src="###"
                        default=""/>
      <a:Age            src="###"
                        alt=""/>
      <z:_choice_>
         <a:PassportNumber
                        case="###"
                        src="###"
                        default=""/>
         <a:LoyaltyNumber case="###"
                        ctxt="###"
                        src="###"
                        default="">
            <z:_attributes_>
               <Airline
                        src="###"
                        default=""/>
               <CheckStatus
                        src="###"
                        alt=""/>
            </z:_attributes_>
         </a:LoyaltyNumber>
         <a:CustomerID  case="###"
                        src="###"
                        default=""/>
      </z:_choice_>
   </a:Traveller>
</a:Travellers>
```

4.4.3. Create a metadata tree transformator

A metadata tree transformator transforms a metadata tree into a useful artifact. In our present context, the metadata tree transformator is a code generator. It transforms the metadata tree into code implementing the transformation of a source document into a target document. If code in several programming languages is required, we can create several transformators, one for each required language. Here, we limit ourselves to writing a code generator creating XQuery code. The task is neither trivial, nor exceedingly difficult. See [10], module `xmap2xq.xqm` for a solution.

4.4.4. Using the code generator

Now we are ready to use our code generator. Let the transformation source be documents similar to this:

```
<travelInfo xmlns="http://example2.com/msgs">
   <time startDate="2017-07-31"
         endDate="2017-08-08"/>
   <players>
      <passengers>
         <passenger surName="Boateng"
                    firstName="Rachel"
                    bookingAge="32"
                    loyalty="KLM:1234567"
                    loyaltyCheck="Ok"/>
         <passenger surName="Boateng"
                    firstName="Belinda"
                    documentNr="9293949596"/>
      </passengers>
   </players>
</travelInfo>
```

We edit the metadata tree appropriately:

```
<a:Travellers xmlns:a="http://example.com/ns"
  ctxt="/s:travelInfo/s:players/s:passengers">
  <a:Traveller for-each="s:passenger">
    <a:Name src="@surName" default=""/>
    <a:Age src="@bookingAge" alt=""/>
    <z:_choice_>
      <a:PassportNumber case="@documentNr"
                        src="@documentNr"
                        default=""/>
      <a:LoyaltyNumber case="@loyalty" ctxt="."
        src="@loyalty/substring-after(., ':')"
        default="'0'">
        <z:_attributes_>
          <Airline src="@loyalty/
                        substring-before(., ':')"
            default="'#UNKNOWN-AIRLINE'"/>
          <CheckStatus src="@loyaltyCheck"
                       alt="'Unknown'"/>
        </z:_attributes_>
      </a:LoyaltyNumber>
      <a:CustomerID case="@custId" src="@custId"
                    default=""/>
    </z:_choice_>
  </a:Traveller>
</a:Travellers>
```

Passing this metadata tree to the code generator, we obtain the following *generated* XQuery program:

```
let $c := /s:travelInfo/s:players/s:passengers
return if (empty($c)) then () else

<a:Travellers xmlns:a="http://example.com/ns">{
  for $c in $c/s:passenger
  return
  <a:Traveller>{
    let $v := $c/@surName
    return <a:Name>{string($v)}</a:Name>,
    let $v := $c/@bookingAge
    return
    if (empty($v)) then ()
    else
      <a:Age>{string($v)}</a:Age>,
    if ($c/@documentNr) then
      let $v := $c/@documentNr
      return <a:PassportNumber>{
        string($v)
      }</a:PassportNumber>
    else if ($c/@loyalty) then
      let $contentV := $c/@loyalty/
                       substring-after(., ':')
      let $contentV := if ($contentV) then
                         $contentV else '0'
      return
      <a:LoyaltyNumber>{
        let $v := $c/@loyalty/
                  substring-before(., ':')
        let $v := if ($v) then $v
                  else '#UNKNOWN-AIRLINE'
        return
        attribute Airline {$v},
        let $v := $c/@loyaltyCheck
        let $v := if ($v) then $v
                  else 'Unknown'
        return attribute CheckStatus {$v},
        $contentV
      }</a:LoyaltyNumber>
    else if ($c/@custId) then
      let $v := $c/@custId
      return
      <a:CustomerID>{
        string($v)
      }</a:CustomerID>
    else ()
    }</a:Traveller>
  }</a:Travellers>
```

5. Discussion

The main building block of a location tree constitutes an interesting new concept. A location is obviously a *model entity*, as it depends on schema contents, never on instance document contents. It encapsulates model information about a well-defined class of real world data: the items occurring in a particular document type at a particular place. The limitation and precision of the modelling target make a location particularly well-suited for novel kinds of metadata, as well as a suitable entity for defining and collecting related facts. Locations, formally defined and readily available in materialized form, might influence the way how one thinks and speaks about structured information.

Appreciation of location trees entails additional appreciation for XSD, as location trees are derived from XSD. Location trees unite "the best of two worlds", XSD's advanced features of model design and component reuse, and a clear and straightforward relationship between model entity and real world data, which enables intense use of the model beyond validation and documentation.

Bibliography

[1] *BaseX - an open source XML database*. BaseX.
http://basex.org

[2] *FOXpath navigation of physical, virtual and literal file systems*. Hans-Jürgen Rennau. XML Prague. 2017.
http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=173

[3] *Java Architecture for XML Binding (JAXB)*. Java Community Process.
https://jcp.org/en/jsr/detail?id=222

[4] *NIEM 3.0*. Georgia Tech Research Institute, Inc. (US).
https://release.niem.gov/niem/3.0/

[5] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 5 April 2012. Shudi Gao, C. M. Sperberg-McQueen, and Henry S Thompson. World Wide Web Consortium (W3C).
https://www.w3.org/TR/xmlschema11-1/

[6] *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. 5 April 2012. World Wide Web Consortium (W3C). David Peterson, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen, and Henry S Thompson.
https://www.w3.org/TR/xmlschema11-2/

[7] *XML Schema Part 1: Structures Second Edition*. 28 October 2004. World Wide Web Consortium (W3C). Henry S Thompson, David Beech, Murray Maloney, and Noah Mendelsohn.
https://www.w3.org/TR/xmlschema-1/

[8] *XML Schema Part 2: Datatypes Second Edition*. 28 October 2004. World Wide Web Consortium (W3C). Paul V Biron and Ashok Malhotra.
https://www.w3.org/TR/xmlschema-2/

[9] *XQuery 3.1: An XML Query Language*. 21 March 2017. World Wide Web Consortium (W3C). Jonathan Robie, Michael Dyck, and Josh Spiegel.
https://www.w3.org/TR/xquery-31/

[10] *xsdplus - a toolkit for XSD based tool development*. Hans-Jürgen Rennau.
https://github.com/hrennau/xsdplus

An Architecture for Unified Access to the Internet of Things

Jack Jansen

CWI, Amsterdam

<jack.jansen@cwi.nl>

Steven Pemberton

CWI, Amsterdam

<steven.pemberton@cwi.nl>

Abstract

The Internet of Things is driven by many tiny low-powered processors that produce data in a variety of different formats, and produce the data in different ways, sometimes on demand (such as thermostats), sometimes by pushing (such as presence detectors). Traditionally, applications have to be a mash up of accesses to devices and formats. To use the data in a cohesive application, the data has to be collected and integrated; this allows very low demands to be put on the devices themselves.

The architecture described places a thin layer around a diverse collection of Internet of Things devices, hiding the data-format and data-access differences, unifying the actual data in a single XML repository, and updating the devices automatically as needed; this then allows a REST-style declarative interface to access and control the devices without having to worry about the variety of device-interfaces and formats.

Keywords: Internet of things, iot, REST, XML, Software Architecture

1. Internet of Things

Moore's Law is about three properties of integrated circuits: the number of components on them, the price, and the size. Hold two of these constant, and the other displays its full Moore's Law effect: after one cycle, you can get the same thing for half the price, the same thing for the same price at half the size, or the same thing at the same size with twice the number of components.

So while computers have been getting more powerful at one end, and people have been optimising price/size/power in the middle, computers have been getting smaller, and cheaper at the other end. While a dozen years ago, you could count the number of CPUs in your house on one hand, now they are being added to everything, very often with networking built in.

The result is that there are now millions of cheap, tiny devices, with low processing power, embedded in devices everywhere.

The internet of things is not necessarily a new concept: twenty years ago a typical petrol station would have had embedded devices in the pumps, the storage tanks, the tills, the vending machines, and these would all have been centrally accessible and controllable. What is new is the ubiquity, and the diversity.

A problem that accompanies this diversity is a lack of standardisation. For instance, there are a number of different access methods, such as on demand, where you go to the device to access the data, push, where the device sends the data out at some point, and you had better be listening if you want to access it, and storage in the cloud, where the device either sends the data to some central point, or the central point polls the device for its values. Similarly, there are various data formats used, including XML, JSON, and a number of other text formats.

This all means that creating applications that combine data from different devices involves programming and dealing with lots of fiddly detail.

2. Social Problems

Apart from the technical problems, there are also some social problems involved, particularly since companies producing the devices like to keep control, with the possibility of monetising the data they have access to.

doi:10.14337/XMLLondon17.Pemberton01

The problems include ownership: since the data may be stored on a device in the cloud, who owns the data? Do you as owner of the device even have unrestricted access to your own data? There is also privacy: whether or not the data is stored on the device itself, who can access and see the data, and what are the access mechanisms for ensuring the data is not publicly visible? And then you have the issue of control: who is allowed to do anything with the data or device?

3. An Architecture

This paper describes an architecture and system based on it, that addresses these issues and that permits:

- retainment of ownership of the data,
- an access control mechanism to keep control over who may see and modify the data and devices,
- hiding the data-format and data-access differences by placing a thin layer around the diverse collection of devices,
- integration of the data into homogeneous collections,
- keeping the data and devices updated automatically as needed, without intervention.

The resulting architecure allows a REST-style declarative interface to access and control the devices without having to worry about the variety of device-interfaces and formats.

4. Design

The central element of the design is an XML repository that stores the incoming data. XML gives the advantage of data homogeneity, and an advanced existing toolchain, and the separation of elements and attributes facilitates the separation of data from metadata.

The essence of how the system works is that this repository is kept up-to-date with the devices bi-directionally: if the device changes, the repository is updated to match, and if the data in the repository is changed, the device is updated. This has been referred to previously, in contradistinction to WYSIWYG (What You See Is What You Get) as TAXATA (Things Are eXactly As They Appear) [1].

To achieve this, there is a thin functional layer around the repository that communicates with the devices. Plug-ins for devices and formats are responsible for knowing how to access the data from the devices, obtaining data from them, converting as necessary to XML, and for sending data back to the devices in their native format should the data change in the repository.

In support of this there are events that can be listened for and reacted on within the repository. The fundamental event is *value changed*, that signals when a data value changes, and allows event listeners to react; however, other events include timer events, and events signalling changes in the structure of a sub-tree, such as insertions and deletions.

Using the DOM model of events [2], events are allowed to bubble up the tree, so that listeners can respond at the value itself, or higher up the tree for a group of values.

Finally there are constraints and relationships, that specify how values relate to each other, and ensure that values that depend on others are automatically updated (in the process possibly changing the state of the related devices).

5. Some (necessarily simple) Examples

For instance there is a single bit: `lights`. When the lights are on, that bit is 1, and when the lights are off it is 0.

However, it works both ways: to turn the lights off, you just set the bit to 0; if anything changes the value to 1 they go on again.

There is another single bit: Is Jack home? (Which is not two-way ;-))

There are two ways to influence a value in the repository. One is equality "=", which ensures that the equality is *always* true.

For instance, if we said

```
lights = jack-home
```

this would mean that whenever Jack is home the lights are on, and whenever he isn't home, the lights are off. However, this would be upsetting if he wanted to sleep.

Consequently, we use the other method of influencing a value: "←". This only changes the value when the value of the expression changes. So, if we say:

```
lights ← jack-home
```

this would ensure the lights are on when he arrives home (they may have already been on), and ensures they are off when he leaves (they may already have been off).

Since this only happens when changes happen, it allows an override. For instance a switch on the wall also

has a bit in the database, and this can be bound to the value of the lights:

```
lights ← switch
```

(Note how switches are no longer hard-wired to their function.)

You probably don't want the lights to come on when it is already light, but you can have a sensor that detects whether it is dark or not, with an affiliated bit in the store:

```
lights ← jack-home and dark
```

This switches the lights on if Jack comes home in the dark; it switches the lights on if Jack is already home and it gets dark; and it also ensures that the lights are off when Jack leaves (whether they were on or not already, and whether or not it is dark). Note that this also ensures that the lights go off when it gets light.

Note that you could combine these statements into a single one using equality:

```
lights = (jack-home and dark) or switch
```

However, the separate statements allow a certain degree of modularity, since, for instance, if you decide to reassign the switch to another purpose, the other statements continue to work.

6. Is Jack Home?

How do we know if Jack is home?

Well he carries a mobile phone, that connects to the wifi, maybe a bluetooth watch that a sensor can pick up, and he has a laptop that also connects. These all get recorded in the repository (along with other details such as IP address assigned). So we could say

```
jack-home = jack-phone and jack-watch
            and jack-laptop
```

However, sometimes he switches his laptop off. How about:

```
jack-home = jack-phone or jack-watch or jack-laptop
```

Well, he might accidently or deliberately leave his phone or watch at home. Then we use a heuristic:

```
jack-home =
     count(jack-phone, jack-watch, jack-laptop) > 1
```

This is not absolutely failsafe, but likely to be satisfactory.

(For the purpose of exposition, we have treated `jack-home` as if it were a single standalone value, but in reality it will be part of a structured value, such as `person[name="jack"]/present`).

7. Living together

Jack doesn't live alone though. So there is a bit that records if anyone we know is home:

```
anyone-home = jack-home or jill-home or jim-home
```

and we would use that in preference to just `jack-home` in the above examples.

We can let the central heating automatically activate depending on whether someone is home or not:

```
heating = anyone-home
```

Of course, the required temperature of the heating is also a value in the database, as well as the actual temperature, so unlike the lights example, we don't need an extra override, since that is already taken care of.

8. Lock

One of the devices we have built is a door lock that is openable with any RFID device (a phone, a bank card, a dongle, etc) that has been registered with the lock.

Opening the lock is easy. If you swipe the RFID by the reader, the identification gets stored in the repository at `lock/request`, so the lock may be opened if that identification is in the list of allowed values:

```
lock/unlocked ← lock/request in lock/allowed
```

However, this only opens the lock. There are two options for relocking. One is if the lock is intended to be opened, and left open until it is locked again. Then you swipe a second time to lock it, and replace the above statement with:

```
lock/unlocked ← lock/unlocked xor
                (lock/request in lock/allowed)
```

Then a swipe just toggles the locked/unlocked state.

The other option is if the lock is opened with a swipe and then locks itself shortly after. For this we use timer events:

```
lock/unlocked ← lock/request in lock/allowed

changed(lock/unlock):
   dispatch(init, lock, 2sec)

init(lock):
   lock/request ← ""
   lock/unlock ← 0
```

Here we see a listener for the `value-changed` event on the lock. This dispatches an *init* event to the lock after 2 seconds. The listener for the init event relocks the lock.

9. Mesters's Law

One principle that we have applied in the project is Mesters's Law, named after its instigator:

"A Smart anything must do at minimum the things that the non-smart version does"

So, for instance, a thermostat that doesn't allow you to change the desired temperature at the thermostat itself, but requires you first to find the thermostat remote control does not fulfil Mester's Law.

To this end, the individual devices must have as few dependencies on the general infrastructure as possible. Clearly, there is nothing much you can do if there is a powercut and you have no backup power supply, but you don't want to depend on the wifi to be running, or the domain name server to be up, in order to be able to get in to your house.

What this means is that our system runs on the local devices as well, so that there are several copies of the system distributed, and communicating with each other: there is no dependency on a central version of the system being up and running.

10. Privacy

A basic principle is that none of the data is visible outside the system, unless explicitly revealed to someone.

This allows Jack, should he wish, to expose that he is home, without exposing details such as his phone identity.

For instance, he can reveal to the janitor of the building whether he is home, or reveal whether anyone is home to other inhabitants of the building without revealing any other details, such as his phone MAC address. This means the janitor can't also determine if Jack is at the bar down the road.

Since the architecture is primarily state-based, with events a side-effect, in effect it is the reverse of IFTTT-style systems that are quite common nowadays for IoT solutions [3].

The advantage of the state-based paradigm, together with the hierarchical containment that XML gives us, is that it supplies the scaffolding for the necessary security and privacy mechanisms. Doing fine-grained access control in an event-based system like IFTTT would be more difficult, because there would basically have to be access rules for every event/trigger, but with this system it can be done on the basis of subtrees.

As mentioned earlier, a design decision was to store data in XML elements and use attributes for storing meta-data. Part of that meta data is information about access.

While this part of the system is not yet implemented, we are investigating two possible access mechanisms: one, based on ACLs (access control lists) [4] mirrors the system that is used in hierarchical filestores, such as Unix, that are based on user-indentities. However, the one that has our current preference is a system based on Capabilities [5], where to access a part of the structure you have to be in possession of a token.

11. Communication

Since the system is state-based, the ideal communication method is REST.

REST (REpresentational State Transfer) is the architectural basis of the web. As Wikipedia points out:

"REST's coordinated set of constraints, applied to the design of components in a distributed hypermedia system, can lead to a higher-performing and more maintainable software architecture."

In other projects we actually have proof of this claim: we have seen it save around an order of magnitude in time and costs.

Therefore communication with the system, and between instances of the system is HTTP/HTTPS.

12. User Interface

Since all actions are now controlled by changing data, all we need is a straightforward way to access and change data.

Luckily we have XForms [6], which has more or less the same structure as the system: a collection of (XML) data, events, and constraints.

On top of that, XForms has a method of binding user interface controls to the data for displaying and updating the data.

This has been treated in some details in an earlier paper "XML Interfaces to the Internet of Things" [7].

13. Conclusion

We have a system that insulates us from the details of the different devices, how to drive them and the format of the data. It offers a powerful security mechanism, and straightforward access protocol.

It gives us a very simple yet powerful mechanism for reading and controlling devices.

The system is at an early stage of development at present: we currently have a system running at two locations.

Bibliography

[1] *The ergonomics of computer interfaces*. designing a system for human use. Lambert Meertens and Steven Pemberton. 1992. Centrum Wiskunde and Informatica (CWI).
http://www.kestrel.edu/home/people/meertens/publications/papers/Ergonomics_of_computer_interfaces.pdf

[2] *Document Object Model (DOM) Level 2 Events Specification*. Tom Pixley. World Wide Web Consortium (W3C). 13 November 2000.
http://www.w3.org/TR/DOM-Level-2-Event

[3] *IFTTT (If This Then That)*.
https://en.wikipedia.org/wiki/IFTTT
Wikipedia. Accessed: 13 May 2017.

[4] *Access Control List*. Wikipedia. Accessed :13 May 2017.
https://en.wikipedia.org/wiki/Access_control_list

[5] *Capability-based Security*. Wikipedia. Accessed: 13 May 2017.
https://en.wikipedia.org/wiki/Capability-based_security

[6] *XForms 2.0*. Erik Bruchez, Steven Pemberton, and Nick Van den Bleeken. World Wide Web Consortium (W3C).
https://www.w3.org/community/xformsusers/wiki/XForms_2.0

[7] *XML Interfaces to the Internet of Things*. Steven Pemberton. XML London 2015. June 2015. 163-168.
doi:10.14337/XMLLondon15.Pemberton01

Migrating journals content using Ant

A case study

Mark Dunn

Oxford University Press

<mark.dunn@oup.com>

Shani Chachamu

Oxford University Press

<shani.chachamu@oup.com>

Abstract

This paper is a case study of a project to migrate several thousand problematic articles (out of a collection of over 2 million) published online in academic journals to a new platform. The text of the articles is captured as JATS XML [1]. Articles are loaded to the new platform in zip packages representing journal issues. Each package consists of an XML file for each article in the issue, together with associated assets (images, PDF versions, etc.). Most of the 2 million articles in our collection were migrated without a problem. But several thousand articles were not loaded successfully on the first pass. We describe the creation of an Ant [2] pipeline to automatically fix the problems we found, which are typical of large historic data sets and included missing DOIs (Digital Object Identifiers) [3], invalid subject taxonomic codes, badly-formatted publication dates, corrupt PDFs, and missing images.

Keywords: Ant, JATS, oXygen XML Editor, XSLT

1. Background

In late 2015 OUP announced a partnership with Silverchair Information Systems to host its 400 journals, with the launch of a new Oxford Academic platform due to take place in late 2016.

The new platform accepts journal articles in a flavour of the industry-standard JATS (Journal Article Tag Suite) data model [1]. This variant is referred to here as "SC JATS" and is essentially a set of restrictions of JATS 1.1, of the kind that could be enforced using Schematron. For example, controlled vocabularies for certain attribute values and element content, and making certain elements compulsory which are optional in standard JATS. OUP's backfile had to be converted to this format using an XSLT transformation.

The 2 million articles in our collection were delivered to Silverchair, who performed an analysis, identifying every element used, in every context. This analysis was compared to the SC JATS data model. For each structure disallowed by SC JATS, we had to make a decision on how to handle it. In most cases an SC JATS equivalent was found, and code written into the XSLT transformation to map the invalid structure to the agreed SC JATS structure. In a few cases an SC JATS restriction was loosened.

This process was over 99% successful, in terms of content passing through the XSLT transformation and loading successfully to the new platform. But several thousand articles failed to load on the first pass, for various reasons, described below. OUP's Content Architects (CAs) were approached to provide a quick, tactical, automated solution to get the remaining files published on the new platform.

2. Problems to solve

OUP has been capturing journals content as XML (and previously SGML) for over 20 years, to the evolving standard now called JATS, formerly NLM. We also regularly acquire journals from other publishers, which means taking on their archive of content and republishing it on our platform.

As a result, OUP's backfile contains a number of variations of the basic JATS data model. The transformation to SC JATS handled much of this variation, but not all. We now had to account for the remaining problems, for example:

doi:10.14337/XMLLondon17.Dunn01

- Missing DOIs (Digital Object Identifiers)
- Missing article titles
- Invalid subject taxonomic codes
- Badly formatted publication dates
- Image callouts with missing or incorrect file extension

The source material for this effort was a data export from the legacy platform.

As well as XML errors, we also encountered non-XML problems in this data, including:

- Corrupt PDFs
- Missing or misplaced images
- Multiple versions of images

In broad outline, our solution needed to automate the following steps:

1. Unwrap each package of content (a zip file containing the articles in a single journal issue along with their assets).
2. Check for corrupt PDF files.
3. Fix image callouts in the XML and select corresponding image assets to keep for loading.
4. Run the XSLT transformation to SC JATS over the content XML.
5. Run additional XSLT to handle identified XML errors.
6. Validate and QA the output XML.
7. Move and/or rename certain asset files.
8. Quarantine articles containing problems we were unable to fix automatically.
9. Repackage successfully transformed content XML and assets in the folder structure required by the platform loader.
10. FTP the output packages to the loader.

3. Choosing the technology for the solution

There are many languages that could be used to implement a pipeline of tasks. Because of the need for a rapid solution, we considered just a few options we were familiar with.

We have in the past written pipelines in Perl, but this is not a skill all OUP's Content Architects have. Because of the tight deadline for this work, we needed to use a technology that enabled more than one CA to work on the script.

We have also dabbled with XProc, but found it a language with a steep learning curve. At the time we last tried to use it, XProc did not have an easy method of handling some of the tasks, e.g. zip/unzip.

Most recently we have had success employing Ant [2] [4] for pipelines. Ant was designed as a build tool for Java programs, but it has been extended with a library (Ant-Contrib) [5] of additional features, and hits a sweet spot for our needs, covering all the steps that form part of the solution. Ant pipelines are written in XML, so it is easy to learn for people who are familiar with XML and XSLT. Ant is also well supported in oXygen XML Editor [6], which is a standard tool for OUP's Content Architects, and is easy to deploy to a Jenkins continuous integration environment [7], to enable other users to run the pipeline.

4. Applying Ant to the solution

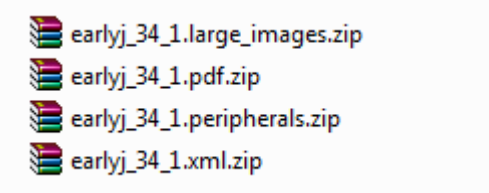
An Ant script is an XML file (usually called `build.xml`) which consists of several tasks, or groups of tasks, called "targets", performed in a sequence determined by `@depends` attributes on the `<target>` elements.

Breaking the pipeline down into discrete targets allows an individual part of the script to be run, without having to execute the entire pipeline. This is handy for unit testing and debugging new code.

4.1. Unzip source files

Typically there were four source zip files per journal issue, one each for the XML versions of the articles, the PDF versions, the full-size image assets (figures etc.), and "peripherals" (e.g. a cover image for the issue).

Figure 1. Source files for "Early Music" volume 34 issue 1



Example 1. Ant target for unzipping the source files

```
<target name="unzipInputFiles"
        depends="untarInputFiles">
  <echo message="unzip input files"/>
  <for param="file" parallel="true"
       threadcount="${threadcount}">
    <fileset dir="${input.dir}" includes="*.zip"/>
    <sequential>
      <propertyregex
          override="yes"
          property="foldername"
          input="@{file}"
          replace="\1"
          regexp="^.+?[\\/]([^\\/]+)\.zip$" />
      <unzip src="@{file}"
             dest="${input.dir}/${foldername}"/>
      <delete file="@{file}"/>
    </sequential>
   </for>
 </target>
```

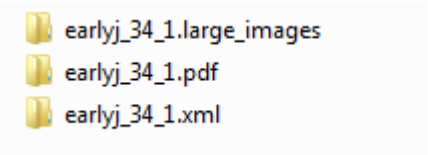
The Ant target follows a couple of similar targets which handle the cases where the source files are in `.tar` or `.tar.gz` format.

The task in this target uses the Ant-Contrib `<for>` loop to iterate over each zip file. The `<fileset>` element identifies the paths to each zip file (using the "*" wildcard). The `file` parameter is set to each path in turn.

For each zip file (minus the peripherals, which are not required in the output), a regular expression is used to set a local property `foldername` whose value is the name of the zip file, minus the `.zip` extension. The regular expression allows for the path separator to be either "/" or "\", since the script was written and tested on a Windows PC but run by users on a Linux server.

The `<unzip>` element extracts the files into a new folder within the working directory, named after the `foldername` property.

Figure 2. Unzipped source files for "Early Music" volume 34 issue 1



4.2. Reorganize files and folders

The next step was to consolidate the XML, PDF, and image assets under a single folder representing the journal issue. This made it easier to check assets within a single issue when running the script over dozens of issues at a time, and also got us closer to the desired output format for loading to the new platform.

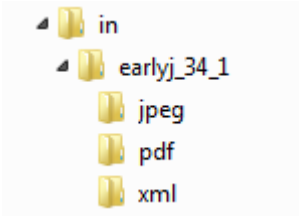
Example 2. Ant target for moving the XML files

```
<target name="moveXML" depends="unzipInputFiles">
  <echo message="move XML files"/>
  <for param="file" parallel="true"
                   threadcount="${threadcount}">
    <fileset dir="${input.dir}"
             includes="*.xml/*"/>
    <sequential>
      <propertyregex
        override="yes"
        property="foldername"
        input="@{file}"
        regexp="^(.+?)[\\/]([^\\/]+)\.(xml|XML)
                [\\/]([^\\/]+)\.(xml|XML)$"
        replace="\2"/>
      <move file="@{file}"
          todir="${input.dir}/${foldername}/xml/"/>
    </sequential>
  </for>
 </target>
```

This Ant target works in a similar way to the `unzipInputFiles` target, iterating over each XML file and using a regular expression to extract from its path the string representing the journal issue name. The `<move>` element moves the XML file to its new location.

Similar Ant targets were used for the images and PDFs.

Figure 3. Reorganized folder structure for "Early Music" volume 34 issue 1



4.3. Check for corrupt PDFs

There are (probably) many ways in which a PDF file can be corrupt, but one indication we found we could check for was a lack of an EOF (end of file) marker at the end of the document. Investigation showed that this method correctly identified the problem PDF files that would

otherwise be rejected by the platform. The Ant targets that performed this check contained a number of tasks.

First, a copy was made of each PDF in a journal issue (represented by the `folder` parameter). We renamed the copy (using the `<mapper>` element) to pretend it was a text file and change the file extension:

```
<copy todir="@{folder}/pdftext">
  <fileset dir="@{folder}/pdf">
    <include name="*.pdf"/>
    <include name="*.PDF"/>
  </fileset>
  <mapper type="glob" from="*.(pdf|PDF)"
          to="*.txt"/>
</copy>
```

We tried to use the XSLT function `unparsed-text()` to read this file, but it produced an error message:

```
build.xml:251: Fatal error during transformation
               using
  \tools\pdf-checker\checkPDF.xsl:
Failed to read input file
  /in/restud82-4/pdftext/rdv013.txt
  (java.nio.charset.MalformedInputException);
Caused by: net.sf.saxon.trans.XPathException
```

To get round this, we stripped out non-word characters from the file:

```
<replaceregexp match="\W" replace="" flags="g"
               byline="true">
  <fileset dir="@{folder}/pdftext"
           includes="*.txt"/>
</replaceregexp>
```

The XSLT script read the text file using the `unparsed-text()` function and produced a report (`<xsl:result-document>`) in either an "OK" or a "broken" folder, depending on whether the EOF character was found:

```
<xsl:variable name="pdfContent"
  select="normalize-space(
    unparsed-text(
      concat($PDFissueFolder,'/pdftext/',
             $PDFfileName)
    )
  )"/>

<xsl:choose>
  <xsl:when test="ends-with($pdfContent,'EOF')">
    <xsl:message>PDF file
      <xsl:value-of select="$PDFfileName"/>
      is OK </xsl:message>
    <xsl:result-document
      href="{concat($PDFissueFolder,'/pdftext/OK/',
                    $PDFfileName)}">
      <pdf><xsl:value-of select="."/></pdf>
    </xsl:result-document>
  </xsl:when>
  <xsl:otherwise>
    <xsl:message>PDF file
      <xsl:value-of select="$PDFfileName"/>
      is broken</xsl:message>
    <xsl:result-document  href="{
      concat($PDFissueFolder,'/pdftext/broken/',
             $PDFfileName)}">
      <pdf><xsl:value-of select="."/></pdf>
    </xsl:result-document>
  </xsl:otherwise>
</xsl:choose>
```

Ant has an `<xslt>` element to handle running this transformation:

```
<xslt style="tools/pdf-checker/checkPDF.xsl"
    in="${input.dir}/${PDF.issuefolder}/
        pdftext/${PDF.fileName}in.xml"
    out="${input.dir}/${PDF.issuefolder}/
        pdftext/${PDF.fileName}out.xml"
    reloadstylesheet="true">
  <factory
    name="net.sf.saxon.TransformerFactoryImpl"/>
  <param name="PDFfileName"
    expression="${PDF.fileName}"/>
  <param name="PDFissueFolder_base"
    expression="${input.dir}/${PDF.issuefolder}"/>
</xslt>
```

The Ant XSLT task requires either a single input and output file, or a specified input and output folder. We were only interested in the report generated as a

byproduct of reading the text, so we created dummy input and output files, which were deleted (using Ant's `<delete>` element) once they had served their purpose.

Finally, an entire journal issue was quarantined if any corrupt PDFs were found within it. Because of the way the platform loader works, it was simpler to quarantine an entire issue until the problem was fixed rather than load a partial issue to the platform and add the missing material later on.

Example 3. Ant target for quarantining issues containing corrupt PDFs

```
<target name="catchBrokenPDFs"
        depends="checkPDFText">
  <for param="folder" parallel="true"
       threadcount="${threadcount}">
    <dirset dir="${input.dir}/" includes="*" />
    <sequential>
      <var name="brokenPDFs" unset="true"/>
      <propertyregex override="yes"
                     property="issuefolder"
                     input="@{folder}"
                     regexp="^(.+?)[\\/]([^\\/]+)$"
                     replace="\2"/>

      <condition property="brokenPDFs">
        <available file="@{folder}/pdftext/broken"
                   type="dir"/>
      </condition>
      <move todir="${output.dir}/errors/
                  error5-brokenPDFs/${issuefolder}"
        if:set="brokenPDFs">
        <fileset dir="@{folder}"/>
      </move>
    </sequential>
  </for>
</target>
```

This target uses the Ant-Contrib `<for>` loop again to iterate over each issue. A property `brokenPDFs` is set using the `<condition>` element, if the PDF "broken" folder exists within the issue (the `<available>` element checks for the folder's existence).

The entire issue is quarantined to an `errors` folder using `<move>`, if the `brokenPDFs` property is set (`@if:set`). Users can then check the `broken` folder to see which PDFs caused the problem.

4.4. Transform article XML to SC JATS

An Ant target was used to perform the XSLT transformation to SC JATS. (The Ant features used in this target have already been covered in this paper, so are not repeated here.)

4.5. Handle images

A journal issue cannot be loaded to the new platform unless all the images called out in the article XML can be found in the package.

Example 4. Image callout in JATS XML

```
<graphic xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:href="cah188f1.jpg"/>
```

The source material exhibited a number of problems with images:

- Callout in JATS XML is incomplete, lacking a file extension. The incomplete callout may have multiple possible matches in the source files (e.g. `cah188f1.jpg`, `cah188f1.jpeg`, `cah188f1.png`)
- The file named in the JATS `@xlink:href` attribute includes a file extension, however there is no such file present among the source files for the issue.
- The file named in the JATS `@xlink:href` attribute is empty (zero bytes).

The first step in this part of the pipeline was to run a pre-existing script which generates a report on images in a folder.

Example 5. Ant code to run an external script

```
<if>
  <os family="windows"/>
  <then>
    <echo>Running getImageSizes.exe
          (Windows)</echo>
    <exec executable="
           tools/image-file-ext/getImageSizes.exe">
      <!-- input folder -->
      <arg value="@{folder}/jpeg"/>
      <!-- output folder (for output.xml) -->
      <arg value="@{folder}"/>
    </exec>
  </then>
  <else>
    <echo>Running getImageSizes.php</echo>
    <exec executable="php">
      <arg value="
          tools/image-file-ext/getImageSizes.php"/>
      <!-- input folder -->
      <arg value="@{folder}/jpeg"/>
      <!-- output folder (for output.xml) -->
      <arg value="@{folder}"/>
    </exec>
  </else>
</if>
```

We used the Ant-Contrib `<if>` structure to run different versions of the script, depending on which environment it was running in. The `<os>` element checks the operating system environment. The `<exec>` element calls the script, with arguments passed using the `<arg>` element.

Example 6. Output of getImageSizes script

```
<images>
  <image width="909" height="689"
    file="in/earlyj_34_1/jpeg/cah188f1.jpeg"/>
  <image width="1280" height="1173"
    file="in/earlyj_34_1/jpeg/cah188f10.jpeg"/>
  <image width="819" height="710"
    file="in/earlyj_34_1/jpeg/cah188f11.jpeg"/>
  <image width="1800" height="767"
    file="in/earlyj_34_1/jpeg/cah188f12.jpeg"/>
  ...
</images>
```

The next step was to run an XSLT transformation over each content XML file to check and fix the image callouts.

The XSLT code contains a template for the `@xlink:href` attribute in the JATS XML. Its value is compared with the `@file` attributes in the output of the getImageSizes script, ignoring any `<image>` elements without a `@width` attribute (these are the zero-byte images). The XSLT ignores any image extension information provided in the `@xlink:href`, as it is not reliable, and also checks for case-insensitive matches.

If there is more than one match (e.g. if the same file name is found with different image formats), then the XSLT code chooses a file extension using a defined order of preference (`.tif`, `.jpg`, `.png`, `.gif`).

If there is no match, the XSLT code generates a report (`<xsl:result-document>`) which is used in a later step to quarantine the journal issue.

When this is complete, all XML files (which have not been quarantined) must contain only valid image callouts, but the `images` folder may contain files which are not called out in the XML and so cannot be loaded to the platform.

To keep only the required images, the Ant pipeline performs another XSLT transformation, which runs over each content XML file and generates a `<xsl:result-document>` for each image callout, in a folder called `keepimages`. This `<xsl:result-document>` has the same name as the valid image asset it refers to, e.g. `cah188f1.jpeg`, although it is empty.

This `keepimages` directory is then used as a reference list on which image assets to move into the final package, using a `<for>` loop.

Example 7. Ant target to keep only images referenced in JATS XML

```
<target name="moveImageFiles">
  <!--
    for each fake image file in keepimages,
    move the equivalent real image asset
  -->
  <for param="files" parallel="true"
       threadcount="${threadcount}">
   <path>
    <fileset dir="${input.dir}/"
             includes="*/*/keepimages/*"/>
   </path>
   <sequential>
    <propertyregex
      override="yes"
      property="issuefolder"
      input="@{files}"
      regexp="^(.+?)(build[\\/]in[\\/])
              (.+?)([\\/]xml4[\\/]
              keepimages[\\/])(.+?)$"
      replace="\3"/>
    <propertyregex
      override="yes"
      property="imagefilename"
      input="@{files}"
      regexp="^(.+?)(build[\\/]in[\\/])
              (.+?)([\\/]xml4[\\/]
              keepimages[\\/])(.+?)$"
      replace="\5"/>
    <copy todir="${output.dir}/
                 ${issuefolder}/Assets">
      <file file="${input.dir}/${issuefolder}/
                  jpeg/${imagefilename}"/>
    </copy>
   </sequential>
  </for>
</target>
```

4.6. Tidy up the article XML

An Ant target was used to perform another XSLT transformation to fix certain problems in the article XML, any of which would prevent the issue from loading successfully to the platform:

- Missing DOI (Digital Object Identifier).
- Missing article title.
- Invalid subject taxonomic code.
- Badly formatted publication date.

4.6.1. Insert missing DOIs

We had assigned DOIs [3] to the articles, but not (yet) in the article XML itself. Our content holdings were summarized in XML files which were used as lookup files for the DOIs.

Example 8. Holdings file `earlyj.xml`

```
<journal journalCode="earlyj"
         xmlns="http://xmlns.oup.com/journals">
  <volume volumeNumber="38">
    <issue issueNumber="1">
      <article doi="10.1093/em/caq003"
        title="Editorial"/>
      <article doi="10.1093/em/cap131"
        title="Early wind music"/>
      <article doi="10.1093/em/cap128"
        title="Viva Biber!"/>
      <article doi="10.1093/em/cap133"
        title="Bach keyboard music"/>
      <article doi="10.1093/em/cap130"
        title="A century of tragédie en musique"/>
      <article doi="10.1093/em/cap132"
        title="Old friends and new discoveries"/>
      ...
    </issue>
  </volume>
  ...
</journal>
```

The attribute values `@journalCode`, `@volumeNumber`, `@issueNumber`, and `@title` in this file correspond to values held in the article XML, enabling us to identify the DOI.

Example 9. JATS metadata elements used to look up the article DOI

```
<article>
  <front>
    <journal-meta>
      <journal-id journal-id-type="publisher-id">
        earlyj
      </journal-id>
      ...
    </journal-meta>
    <article-meta>
      <title-group>
        <article-title>
          Early wind music
        </article-title>
      </title-group>
      ...
      <volume>38</volume>
      <issue>1</issue>
      ...
    </article-meta>
  </front>
  ...
</article>
```

The XSLT script populates a variable with the matching `<article>` element, from which we obtain its DOI (`@doi` attribute value) which is inserted into the XML.

```
<xsl:variable
    name="doi-lookup-article"
    as="element()*"
    select="
      if ($doi-lookup-document)
      then $doi-lookup-document/j:journal
        /j:volume[@volumeNumber = $vol]
        /j:issue[@issueNumber = $iss]
        /j:article[@title = $article-title]
      else () "/>
```

4.6.2. Insert (or report) missing article title

This problem occurred occasionally in older print journals, when a number of separate short items in the journal issue's front matter were captured under a single heading, e.g. “In this issue”, “Calendar of events”. The heading was present in the article XML but in a different element, and it was approved by the business stakeholders that certain heading values would simply be copied to become the article title.

A report (`<xsl:result-document>`) was generated for other missing titles, and the presence of these report files was used to quarantine an issue which exhibited this problem.

4.6.3. Remove invalid subject taxonomic code

Some journals use particular taxonomies to classify articles. For example, economics journals use an industry standard taxonomy called “JEL”, named after the Journal of Economic Literature, for which it was developed.

Example 10. JATS structure for capturing subject taxonomy codes

```
<subj-group
  subj-group-type="category-journal-collection">
  <subject>JEL/D81</subject>
  <subject>JEL/G11</subject>
</subj-group>
```

We hold copies of these taxonomies in the SKOS (Simple Knowledge Organization System) RDF format [8].

Example 11. SKOS fragment for a JEL code

```
<rdf:Description rdf:about="http://data.oup.com/taxonomy/JEL/D81">
  <rdf:type rdf:resource="http://www.w3.org/2004/02/skos/core#Concept"/>
  <rdfs:label xml:lang="en">D81 - Criteria for Decision-Making under
    Risk and Uncertainty</rdfs:label>
  <skos:prefLabel xml:lang="en">D81 - Criteria for Decision-Making under
    Risk and Uncertainty</skos:prefLabel>
  <skos:broader rdf:resource="http://data.oup.com/taxonomy/JEL/D8"/>
  <skos:inScheme rdf:resource="http://data.oup.com/taxonomy/JEL/ConceptScheme"/>
  <skos:definition xml:lang="en">Covers mostly theoretical studies about
  issues related to criteria for decision making under risk and uncertainty.
  Empirical studies that are of general interest are also classified
  here.</skos:definition>
  <skos:scopeNote xml:lang="en">Studies that incorporate risk or uncertainty
  in a simple or a conventional manner should not be classified here, for
  example studies simply using the well-known expected utility paradigm.
  Studies about insurance and insurance companies (which should be classified
  under G22) should be cross-classified here only if they are also of general
  relevance to the subject matter treated in this category.</skos:scopeNote>
</rdf:Description>
```

The XSLT code contains a template for the `<subject>` element. Its value is compared with the `@rdf:about` attributes in the SKOS file. A non-matching subject code is removed from the article XML.

4.6.4. Fix publication dates

Publication dates recorded in article XML are not always complete. For example, a date in the XML may consist of just a month and a year, where a day is also required for a complete and valid format.

Example 12. JATS structure for capturing the print publication date of an article

```
<pub-date pub-type="ppub">
  <month>February</month>
  <year>2006</year>
</pub-date>
```

The XSLT code contains a template to add `<day>01</day>` when the publication date was missing this element.

Example 13. JATS print publication date following the tidying-up step

```
<pub-date pub-type="ppub">
  <day>01</day>
  <month>February</month>
  <year>2006</year>
</pub-date>
```

4.7. Validate the article XML

The JATS data model is highly flexible, and the content XML in our collection is very varied. The transformation to SC JATS standardized the structures, but some unusual structures in the source XML resulted in invalid output. An Ant target was used to validate the final XML against the SC JATS schema.

Example 14. Ant code for validating XML against a schema

```
<trycatch property="err">
  <try>
    <schemavalidate file="@{file}">
      <schema
        namespace="http://specifications.silverchair.com/xsd/article/1/0/SCJATS-journalpublishing1-0.xsd"
        file="tools/SCJATS_conversion/xmlspecs/SCJATS-journalpublishing1-0.xsd"/>
    </schemavalidate>
  </try>
  <catch>
    <echo file="@{folder}/xml5/QAerrors/${basename}-invalid.xml">
      &lt;invalid>${err} - see console log for details&lt;/invalid>
    </echo>
  </catch>
</trycatch>
```

We could not find a way of adding the error messages to a report document, but they appeared in the console log generated when the Ant pipeline was run:

```
[schemavalidate] Validating cvt159.xml...
[schemavalidate] cvc-complex-type.2.4.a: Invalid
  content was found starting with
  element '{issue-title}'.
One of '{fpage, elocation-id, product,
  supplementary-material,  history, permissions,
  self-uri, related-article, related-object,
  abstract, trans-abstract, kwd-group,
  funding-group, conference,  counts,
  custom-meta-group}' is expected.
```

When it was possible to automate a fix for errors reported by this step, we put the necessary code into the XSLT script in the previous tidying-up step.

4.8. Create output package for loading to the platform

A sequence of Ant targets, using elements previously described, performed a few final steps to prepare a package for loading to the platform. For each journal issue that was not quarantined because of an unfixable error:

- Create a folder for the journal issue in the output area
- Move article XML files into an `XML` subfolder.
- Move images and PDFs into an `Assets` subfolder.
- Filter out unwanted files (unreferenced images, but also others, e.g. some source folders contained `Thumbs.db` files).
- Zip up the journal issue folder.

4.9. Load the journal issues

The last step in the pipeline was to be a call to SCP to copy the files to the Silverchair loader, where they would be automatically ingested. We did not get this step to work, but ultimately it was not a problem, as we managed to load the files manually.

Example 15. Ant code for transferring files by SFTP

```
<scp sftp="true" trust="true"
  todir="oupuser@
        filespace.silverchair.com:ProductionFolder"
  password="****">
  <fileset dir="${output.dir}">
    <include name="*.zip"/>
  </fileset>
</scp>
```

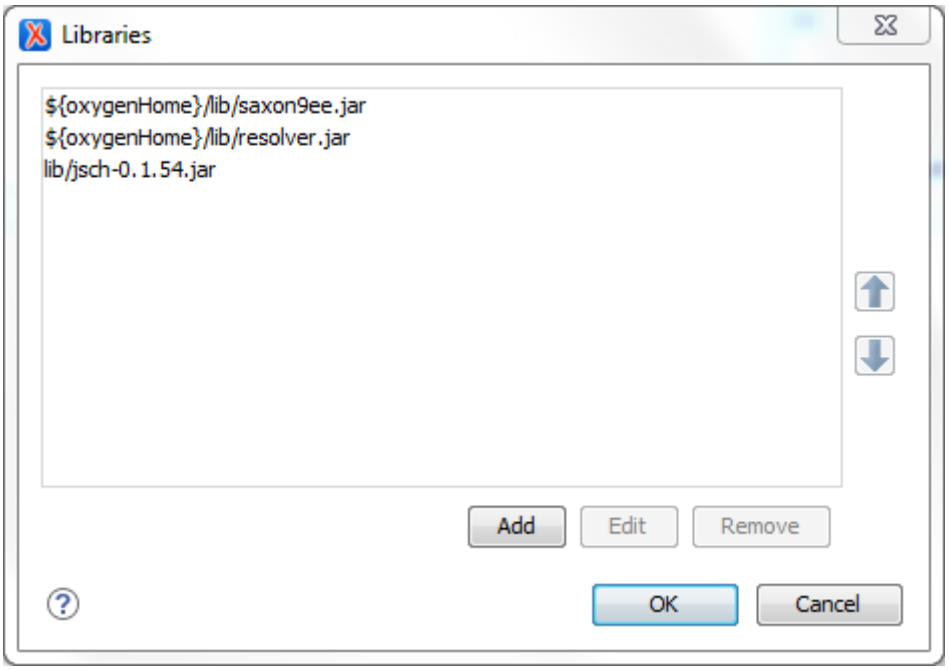
This step failed to work because the password contained an ampersand, and we could find no way of escaping the character in the Ant script for the system to recognize it. We raised a ticket for the password to be changed, but were able to complete the manual load of the files before it could be addressed.

5. Running the pipeline (Windows)

We developed the pipeline on Windows PCs running oXygen XML Editor 18.1.

oXygen [6] includes a default transformation scenario for running Ant pipelines. We created a duplicate of this scenario and added Java libraries to it to support the XSLT transformations in the pipeline (Saxon and an XML catalog resolver) and to support the (aborted)

Figure 4. Java libraries configured for running Ant in oXygen



loading step (Java Secure Channel [9], an implementation of SSH2).

6. Running the pipeline (Linux)

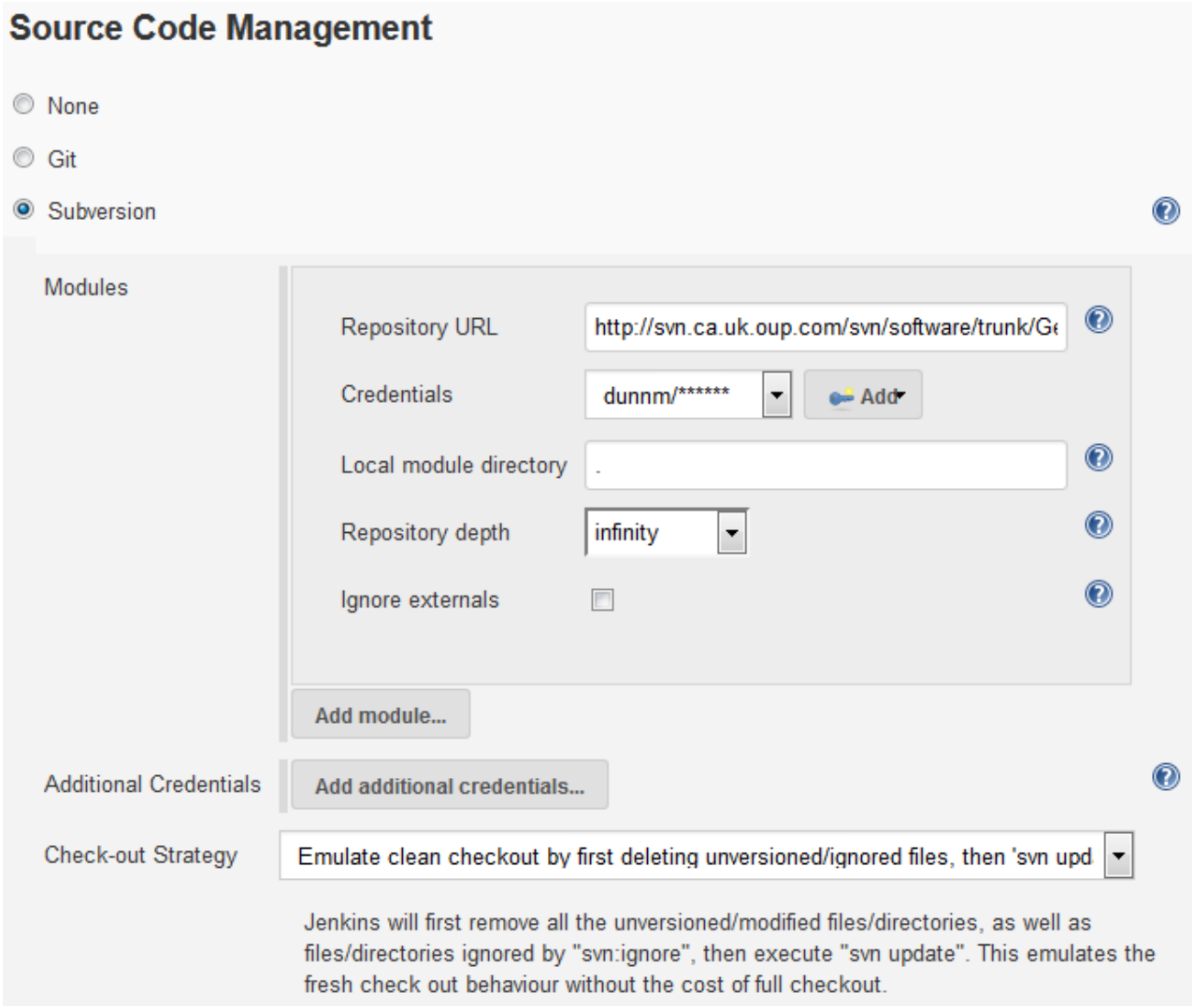
We have access to a Linux server running an instance of the Jenkins continuous integration environment. This is what we wanted our end users to work in. This ensured they were always using the current code, because it was updated from our SVN repository each time the job was run. It also meant that the process was not hogging resources on their PCs, and gave us access to the output of each job, including log files, so we could easily troubleshoot issues and use the output to improve the code.

We set up a project on Jenkins to run the Ant pipeline. We have a few reservations about Jenkins: it doesn't always connect successfully to our SVN repository, resulting in failed builds; it uses a lot of disk space if you want to keep multiple builds; documentation is not comprehensive. But it met our needs for this task.

The project configuration included:

- Settings for certain parameters in the Ant script.
- Instructions to pull in the pipeline and associated XSLT scripts, DTD and XML Schema files, etc. from our code repository.
- Paths to Java libraries for Saxon, the Apache catalog resolver, and Ant-Contrib.

Figure 5. Jenkins configuration for SVN connection



7. Dealing with "quarantined" journal issues

We developed the pipeline to the point where it could automatically fix most of the problems it encountered. This required some pragmatic business decision-making, e.g. when a taxonomic code was invalid, we could not spend time identifying the correct code; we just deleted the invalid code.

At certain points in the pipeline, a journal issue would be quarantined if an automated fix could not be made, e.g. when the output XML was invalid or an image could not be found in the source data. The issues which had come through the process successfully could now be loaded to the new platform, and the quarantined issues were dealt with manually.

In some cases where the poor structure of the source XML meant that the output of the SC JATS transformation was invalid, there were not enough examples of the problem to justify writing a fix in XSLT, so we fixed those cases manually.

Where an image could not be traced in our source data, or a PDF was reported as corrupt, we pulled the relevant files from the legacy platform by visiting the article page.

But overall we were left with dozens rather than hundreds of manual fixes, which was a manageable amount in the time available to us.

8. Regression testing in Ant

The development of this pipeline was iterative. When a new loader error was seen, we built an automated fix into the pipeline where this was possible.

Although this pipeline was a one off, unit testing and regression testing is something we've been building into our processes over the years.

We discovered a neat way to do this in Ant, namely creating a "regression test" task to run the pipeline as many times as necessary with different data and parameters, and then compare the output with a benchmark previous build. We used oXygen's "Compare Directories" tool to make the comparison. On projects where we have implemented this process, testing time has been significantly reduced and we have caught errors that might otherwise have been missed.

```
<target name="regressiontest">
  <echo message="Regression testing"/>
  <antcall target="all" inheritall="false">
   <param name="data.dir"
          location="data/regressiontest/OHO"/>
   <param name="dtd" value="OxEncyclML"/>
   <param name="pipeline" value="OHO"/>
   <param name="zip.output" value="false"/>
   <param name="DEBUG" value="true"/>
   <param name="build.dir"
          value="build_regressiontest/OHO"/>
  </antcall>
  <antcall target="all" inheritall="false">
   <param name="data.dir"
          location="data/regressiontest/OSO"/>
   <param name="dtd" value="OxChapML"/>
   <param name="pipeline" value="OSO"/>
   <param name="zip.output" value="false"/>
   <param name="DEBUG" value="true"/>
   <param name="build.dir"
          value="build_regressiontest/OSO"/>
  </antcall>
</target>
```

Bibliography

[1] *JATS: Journal Article Tag Suite (ANSI/NISO Z39.96 Version 1.1)*. 06 January 2016. NISO. http://www.niso.org/standards/z39-96-2015/

[2] *The Apache Ant Project*. Apache Software Foundation. http://ant.apache.org

[3] *Digital Object Identifier System (DOI)*. International DOI Foundation. https://www.doi.org

[4] *Ant: The Definitive Guide*. 2nd edition. Steve Holzner. O'Reilly. 2005.

[5] *ANT Contrib*. Curt Arnold and Matt Inger. https://sourceforge.net/projects/ant-contrib/

[6] *oXygen XML Editor*. SyncRO Soft SRL. http://www.oxygenxml.com/

[7] *Jenkins CI*. http://jenkins-ci.org

[8] *Simple Knowledge Organization System (SKOS)*. World Wide Web Consortium (W3C). https://www.w3.org/2004/02/skos/

[9] *Java Secure Channel*. JCraft. http://www.jcraft.com/jsch/

Improving validation of structured text

Jirka Kosek
<jirka@kosek.cz>

Abstract

XML schema languages are mature and well understood tool for validation of XML content. However the main focus of schema languages is on validation of document structure and values adhering to few relative simple standard data types. Controlling order and cardinality of elements and attributes is very easy in all of DTD, W3C XML Schema and RELAX NG. Checking that element/attribute values is number, date or string of particular length is also very easy in both W3C XML Schema and RELAX NG with XSD datatypes.

However there are situations when traditional schema languages can not offer much validation. Many vocabularies are using structured text inside elements and attributes because it is more practical or concise than to express the same structure using additional elements and attributes. In many cases structured text can be described by grammar. In this paper/presentation I will show how to mix classical validation against XML schema with invocation of grammar based validation for elements/attributes containing structured text from Schematron schema. Proposed solution uses XSLT parsing function generated by REx from grammars and shows how to invoke code generated by REx from Schematron rules. Other integration techniques will be explored as well.

Keywords: text validation, grammars, XML schema, text parsing

1. Introduction

XML schema languages are mature and well understood tool for validation of XML content. However the main focus of schema languages is on validation of document structure and values adhering to few relative simple standard data types. Controlling order and cardinality of elements and attributes is very easy in all of DTD, W3C XML Schema and RELAX NG. Checking that element/attribute values is number, date or string of particular length is also very easy in both W3C XML Schema and RELAX NG with XSD datatypes.

However there are situations when traditional schema languages can not offer much validation. Many vocabularies are using structured text inside elements and attributes because it is more practical or concise than to express the same structure using additional elements and attributes. For example in SVG path to be drawn is expressed by very concise syntax:

```
<path d="M100,200 C100,100 250,100
         250,200 S400,300 400,200" />
```

where `d` attribute contains string where each letter starts command (e.g. *M* means "move to", *C* means "draw curve", …) and numbers are expressing coordinates or other measures. In traditional schema languages we can not do much about validation of such path. We can try to invent very convoluted regular expression checking if path syntax is correct or not. But creating and debugging such regular expression is a nightmare. And even if we will succeed error reporting will not be very user-friendly as checking value against regular expression yields only matches/non-matches result. If the value is not matching then there is no indication what has been wrong – missing comma, wrong letter used, …

Microsyntaxes like above mentioned draw path are relatively common in a real world XML documents, for example:

- HTML derived vocabularies often support `style` element and attribute containing complete cascading stylesheet or at least declaration part of CSS rule;
- syntax of citations in many scientific and legal documents is quite often beyond what can be reasonably validated using regular expressions;
- some XML vocabularies are badly designed and expect that numbers and dates are written using locale specific syntax instead requiring lexical form of standard types like xs:decimal or xs:date.

2. Current status of text validation support

Idea behind XML is that complex structures are represented by corresponding structure of elements and attributes. Text content of elements or attribute values should then contain only simple values like numbers, dates or string values. However in reality the distinction is not that sharp and many vocabularies are using strings

doi:10.14337/XMLLondon17.Kosek01

for representing more or less complex structures. Lets look how current schema languages can cope with validation of such structured text content.

2.1. Custom data types based on regular expressions

Regular expression is well known tool for pattern matching on text strings. Designing and understanding regular expressions is quite easy as long as a text structure we want to validate is not very complex. For example if we would like to validate `VAT` element containing VAT identification number [1] we can describe this very easily with regular expression. VAT number starts with two letter country code and then with number which can have 2 to 13 digits and in some countries there can be not only digits but also letters. This is very easy to declare in W3C XML Schema:

```
<xs:element name="VAT">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z]{2}[0-9A-Z]{2,13}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

RELAX NG also supports regular expressions so we can apply similar approach here as well:

```
element VAT {
  xsd:string { pattern = "[A-Z]{2}[0-9A-Z]{2,13}" }
}
```

This regular expression is simple and anyone with basic knowledge of regular expressions can quickly understand what values will be matching the regular expression. But we might want to make regular expression more stringent, for example by explicitly listing country codes:

```
(AT|BE|BG|HR|CY|CZ|DK|EE|FI|FR|DE|EL|HU|IE|IT|LV|
LT|LU|MT|NL|PL|PT|RO|SK|SI|ES|SE|GB)[0-9A-Z]{2,13}
```

We can be even ambitious and check for national specifics. For example in Austria VAT number is always `ATU` followed by 8 characters. In Belgium VAT number always has 10 digits and first digit must be 0 or 1. We can modify our regular expression accordingly:

```
(ATU[0-9A-Z]{8}|BE[01][0-9]{9}|(BG|HR|CY|CZ|DK|EE|
FI|FR|DE|EL|HU|IE|IT|LV|LT|LU|MT|NL|PL|PT|RO|SK|
SI|ES|SE|GB)[0-9A-Z]{2,13})
```

It is still possible to decipher what is the aim of such regular expression but it takes more time. Now try to imagine how such expression will look like if we would use national specific patterns for each country similarly to Austria and Belgium.

Lets try something more complex – create regular expression for checking SVG paths. With little bit of simplification a path consist of sequence of drawing commands. Each command strarts with letter and then follows different number of coordinates or other numbers. Lets start with simple move command. It starts with `M` letter (case insensitive) and it is followed by a sequence of coordinates. Each coordinate is two numbers usually separated by space or comma:

```
[Mm]\s*([+-]?([0-9]+|[0-9]*\.[0-9]+(e[+-][0-9]+)?)
(,|\s*)[+-]?([0-9]+|[0-9]*\.[0-9]+(e[+-][0-9]+)?)
\s*)+
```

This is of course not very readable. Main reason for bad readability is that there is a lot of repetition. For example regular sub-expression for number (`[+-]?([0-9]+|[0-9]*\.[0-9]+(e[+-][0-9]+)?)`) is repeated twice here. And if we would like expand this regular expression to cover all other SVG path commands we will get unreadable monster.

It would be possible to improve readability if we could use something like variables in regular expression. However this is supported neither by W3C XML Schema nor by RELAX NG. However we can use this approach in Schematron:

```
<sch:pattern>
  <sch:let name="number" value=
   "'[+-]?([0-9]+|[0-9]*\.[0-9]+(e[+-][0-9]+)?)'"/>
  <sch:rule context="svg:path">
    <sch:assert
      test="matches(@d,
              concat('[Mm]\s*(',
                     $number,
                     '(,|\s*)',
                     $number,
                     ')+'
              )
            )">Invalid path</sch:assert>
  </sch:rule>
</sch:pattern>
```

It is questionable if this is really more readable. We are not repeating pattern for number in the regular expression. But the regular expression itself is synthesized from several smaller pieces using `concat()` function – approach where special attention must be given to proper escaping and quoting of all pieces.

2.2. ISO/IEC 19757-5

Interesting standard ISO/IEC 19757-5 (Extensible Datatypes, formerly know also as Datatype Library Language) has been developed couple of years ago although never implemented to my knowledge. Work on Extensible Datatypes has been motivated by the lack of datatype support in RELAX NG. Unless we want stick to datatypes from W3C XML Schema or implement new datatypes by using validator API there are just two types `string` and `token` available.

Extensible Datatypes allowed definition of new data types. Lexical representation of new data type was usually declared by using regular expression. In order to improve readability of expression it is possible to ignore whitespaces in expression and split it to several lines making it more readable. Also it is possible to capture text matching group in a regular expression into named variable and define additional constraints on such variable using XPath. This could help with writing more complex expressions and also provide better error messages [2].

We can only speculate why Extensible Datatypes has not succeeded. But it would be nice to have some of their flexibility around regular expressions available in W3C XML Schema and RELAX NG.

2.3. Limitations of using regular expressions for validation

As we have seen on previous examples regular expressions are very powerful mechanism for creating constraints for various microsyntaxes. However there are two big issues with using regular expressions. First of them is maintainability and sustainability of regular expressions. For non-trivial tasks regular expressions are becoming more and more complex. Some people might characterize regular expressions as a "write-only" language as it is quite difficult to read and understand already existing regular expressions.

From user point of view regular expression are not very friendly mechanism either. If value is not matching regular expression then validation fails. Many if not all validators will simply say that value is not valid against regular expression. But you will not get any hint about possible problems in your file like: is the number used on place where there should be only letter, unknown country has been used, etc. Outcome of checking against regular expression is simply binary yes/no answer.

3. Describing syntax using grammars

The question is whether we can do better than using regular expressions. Of course we can. One of foundations of computer science is a formal language theory. In this theory you can use grammars to describe valid syntax of formal language. A typical example of formal language are programming languages. For each programing language there is usually grammar describing its syntax. Grammar is consisting of rules that describe how individual characters can be combined together to form syntactically correct source code.

There are also existing grammars for data formats like XML or JSON. For very specific microsyntaxes we can create our own grammar. For example VAT number grammar can be as simple as:

```
VATNumber ::= country number
country ::= letter letter
number ::= (digit | letter)+
letter ::= [A-Z]
digit ::= [0-9]
```

First rule `VATNumber` says that VAT number consist from `country` code followed by `number`. Then next rule says that country code is two letters. Third rule says that `number` is sequence of arbitrary digits and letters. Two last lines define what is letter and digit.

For our SVG path example we don't have to create grammars ourselves. Grammar is part of SVG specification [3], see Example 1, "Grammar for SVG path from SVG 2 specification".

Example 1. Grammar for SVG path from SVG 2 specification[1]

```
svg_path::= wsp* moveto? (moveto drawto_command*)?

drawto_command::=
  moveto
  | closepath
  | lineto
  | horizontal_lineto
  | vertical_lineto
  | curveto
  | smooth_curveto
  | quadratic_bezier_curveto
  | smooth_quadratic_bezier_curveto
  | elliptical_arc
  | bearing
```

[1] Source: https://www.w3.org/TR/SVG2/paths.html#PathDataBNF

```
moveto::=
  ( "M" | "m" ) wsp* coordinate_pair_sequence
                wsp* closepath?

closepath::=
  ("Z" | "z")

lineto::=
  ("L"|"l") wsp* (coordinate_pair_sequence |
                  closepath)

horizontal_lineto::=
  ("H"|"h") wsp* coordinate_sequence

vertical_lineto::=
  ("V"|"v") wsp* coordinate_sequence

curveto::=
  ("C"|"c") wsp* (curveto_coordinate_sequence |
                  (coordinate_pair_sequence?
                   closepath))

curveto_coordinate_sequence::=
  coordinate_pair_triplet
  | (coordinate_pair_triplet comma_wsp?
     curveto_coordinate_sequence)

smooth_curveto::=
  ("S"|"s") wsp*
  (smooth_curveto_coordinate_sequence
  | (coordinate_pair_sequence? closepath))

smooth_curveto_coordinate_sequence::=
  coordinate_pair_double
  | (coordinate_pair_double comma_wsp?
     smooth_curveto_coordinate_sequence)

quadratic_bezier_curveto::=
  ("Q"|"q") wsp*
  (quadratic_bezier_curveto_coordinate_sequence |
    (coordinate_pair_sequence? closepath))

quadratic_bezier_curveto_coordinate_sequence::=
  coordinate_pair_double
  | (coordinate_pair_double comma_wsp?
     quadratic_bezier_curveto_coordinate_sequence)

smooth_quadratic_bezier_curveto::=
  ("T"|"t") wsp* (coordinate_pair_sequence |
                  closepath)

elliptical_arc::=
  ( "A" | "a" ) wsp*
  (elliptical_arc_argument_sequence
  | (elliptical_arc_argument_sequence?
     elliptical_arc_closing_argument))

elliptical_arc_argument_sequence::=
  elliptical_arc_argument
  | (elliptical_arc_argument comma_wsp?
     elliptical_arc_argument_sequence)

elliptical_arc_argument::=
  number comma_wsp? number comma_wsp? number
  comma_wsp
  flag comma_wsp? flag comma_wsp? coordinate_pair

elliptical_arc_closing_argument::=
  number comma_wsp? number comma_wsp? number
  comma_wsp
  flag comma_wsp? flag comma_wsp? closepath

bearing::=
  ( "B" | "b" ) wsp* bearing_argument_sequence

bearing_argument_sequence::=
  number | (number comma_wsp?
            bearing_argument_sequence)

coordinate_pair_double::=
  coordinate_pair comma_wsp? coordinate_pair

coordinate_pair_triplet::=
  coordinate_pair comma_wsp? coordinate_pair
  comma_wsp? coordinate_pair

coordinate_pair_sequence::=
  coordinate_pair | (coordinate_pair comma_wsp?
                     coordinate_pair_sequence)

coordinate_sequence::=
  coordinate | (coordinate comma_wsp?
                coordinate_sequence)

coordinate_pair::= coordinate comma_wsp? coordinate

coordinate::= sign? number

sign::= "+"|"-"
number ::= ([0-9])+
flag::=("0"|"1")
comma_wsp::=(wsp+ ","? wsp*) | ("," wsp*)
wsp ::= (#x9 | #x20 | #xA | #xC | #xD)
```

Several competing syntaxes are in use for writing down grammars like BNF, ABNF, EBNF, … Also there are several different types of grammars like context-free or regular which affects what we can express with a grammar.

As we can see grammar is concise and readable way how to describe allowed syntax. Because grammar is written in a formal way it can be processed by computer. It is easy to take grammar and check whether some text is adhering to grammar. Also parser can be constructed from grammar and then used for parsing of texts adhering to grammar. Result of parsing is usually tree which represents input text broken into smaller pieces in a structure corresponding to the grammar.

Because parsing tree can be easily represented as XML there are several projects that are using grammars to parse non-XML formats into virtul XML trees that can be then processed using well known XML tools [4] [5].

Problem we are trying to solve is little bit more complex because we need to validate full XML document which can contain complex text structures in few elements and attributes. And only those selected values should be validated against respective grammar. So we need to integrate classical XML validation with grammar based validation of selected text fragments of input XML file.

Grammars should feel familiar to XML users because schema languages like DTD, W3C XML Schema and RELAX NG are also grammar based. What is different is input on which grammar operates – XML schemas are grammars that operate on trees while classical grammars are operating on string sequences. But many principles are very similar between tree (sometimes also called hedge) grammars and classical grammars [6].

Similarity is so big that there even exists standardized language Data Format Description Language (DFDL) [7] which is basically W3C XML schema with embedded annotations which define how to parse text or binary data into XML structure corresponding to the schema.

4. Using grammars for validation of texts inside XML documents

As we have shown grammars are effective tool how to describe microsyntaxes appearing inside elements and attributes. But now we need to integrate this into existing XML validating tools. We need to generate parser from the grammar and then invoke this parser during validation for checking values. We can not use W3C XML Schema or RELAX NG for such task as those languages can not directly invoke external code during validation. However Schematron with proper binding can invoke code written in other languages like XPath, XQuery or XSLT. For example we can invoke parsing function compiled into XSLT language directly from Schematron.

4.1. Generating parsing code from grammar

There are many tools that can use grammars and generate parsers from them and do other fancy things as well. However number of available tools that can generate parser code in XSLT is much more limited. One such popular tool is REx parser generator from Gunther Rademacher.

REx can take EBNF grammar as an input and generate corresponding parsing code in many languages including XSLT and XQuery. This is reason why this tool is so popular in XML community. We will be using it through the rest of this paper.

We can control in what language parser will be generated by option, in our case we will be using `-xslt` option. If we will try to compile grammar taken directly from SVG2 spec we will get error messages like:

```
rex -xslt svg-path-2-w3c.ebnf

svg-path-2-w3c.ebnf: syntax error: line 97,
                                   column 13:
   96 sign::= "+"|"-"
   97 number ::= ([0-9])+
   ---------------^
   98 flag::=("0"|"1")
 expected:
   Whitespace
   Name
   StringLiteral
   '('
   ')'
   '/'
   '&lt;?'
   '|'
```

This is caused by the fact that there are small differences in EBNF syntax supported by different tools. We need to make few cosmetical changes in grammar. Updated grammar is available in GitHub repository https://github.com/kosek/xmllondon-2017.[1] We can try to generate parser from this updated grammar again:

```
strong-LL(3) conflict #12 in 2nd alternative of
  choice operator of production
  coordinate_pair_sequence:
  conflicting lookahead token sequences:
    sign number sign
    number sign number
    number comma_wsp number
    sign number comma_wsp
```

This means that there are ambiguities in grammar and it is not possible to generate very efficient parser that will read just few additional tokens in advance. But we can instruct REx to overcome this limitation by using `-backtrack` option. The generated code will then use backtracking to decide which branch to use when there is ambiguity:

```
rex -xslt -backtrack svg-path.ebnf

12 strong-LL(3)-conflicts handled by backtracking
```

This finally generates parsing code in XSLT. New file `svg-path.xslt` has been generated. It contains a lot of functions the most important is `p:parse-svg-path()`. This function can be called with SVG path as an input. If it's called with path that corresponds to grammar empty sequence is returned. `ERROR` element with an error message is returned instead if function is called on an incorrect SVG path . For example calling `p:parse-svg-path('L 100,100')` would return the following value:

```
<ERROR b="1" e="1" s="14">lexical analysis failed
while expecting [eof, wsp, 'M', 'm']
at line 1, column 1:
...L 100,100...</ERROR>
```

The parser correctly refuses this string as valid SVG path as path must start with moveto command (denoted by `M` letter). Error message says that first character of SVG path must be either end of path (ie. empty path, denoted by `eof` production rule in grammar), whitespace (denoted by `wsp` production rule) or `M` letter (in any case). This is much better error message then simple yes/no verdict from regular expression matching.

All functions in generated XSLT file are by default belonging to namespace that corresponds to top-level prouction rule name. In our case namespace binding for prefix `p` would be `xmlns:p="svg-path"`. This is not completely correct as namespaces should be using URIs. We can use `-name` parameter and instruct REx to generate all functions in a specific namespace:

```
rex -xslt -backtrack -name \
http://example.com/parser/svg-path svg-path.ebnf

12 strong-LL(3)-conflicts handled by backtracking
```

4.2. Integration of parsing code into Schematron

Schematron is very flexible validation language which can use many different languages to write validation expression in. The most common language used is XPath. Other languages including XSLT and XQuery can be used with right implementation that supports `queryBinding` attribute. As typical Schematron implementations are XSLT based it is very common to use XSLT 2.0 as a language for writing all conditions inside Schematron. By using XSLT we can use some additional functions over plain XPath. Also XSLT specific functionality like keys or user-defined functions might be used.

We can use REx to generate XSLT code containing parsing functions for our grammar. But now we need to find way how to make these functions available inside Schematron schema. Schematron offeres two elements which can be used for including one schema into another.

First and very commonly used is `sch:include`. This element is very similar to XInclude – basically it will be replaced by the content of included file. It is used for modularizing schema into more manageable smaller files, typically each pattern is stored as an individual file. Unfortunately this functionality is not approapriate in our scenario because including complete XSLT transformation into Schematron will not produce valid Schematron schema.

However we can use less known `sch:extends` element. It includes only child elements of referenced file. So if we will point this element to document with XSLT transformation only top level components of transformation like function and variable declarations will be included. There is one small glitch – `sch:extends` can reference only Schematron file not XSLT transformation. So we need to modify generated

[1] During work on the paper bugs has been found in grammar and reported. Unfortunately this happens if no one is using grammar for its real purpose.

transformation little bit. We have to change root element from

```
<xsl:stylesheet version="2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:p="http://example.com/parser/svg-path">
```

to Schematron `schema` element and add `queryBinding` attribute:

```
<sch:schema queryBinding="xslt2"
  xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  xmlns:p="http://example.com/parser/svg-path"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

It is easy and mechanical change so we can easily automate this step. There is also another solution. REx supports special processing instructions that can be used inside grammar and can be used for generating custom root element. If we will enclose original grammar with the following processing instructions:

```
<?schematron
  <sch:schema queryBinding="xslt2"
  xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:p="http://example.com/parser/svg-path">
?>

... original grammar ...

<?ENCORE?>
<?schematron
  </sch:schema>
?>
```

then we can invoke REx with additional `-a` option as

```
rex -xslt -backtrack -a schematron svg-path.ebnf
```

and generated code will be enclosed in `sch:schema` element instead of `xsl:stylesheet` element and could be directly embeded into Schematron file by using `sch:extends` element:

```
<sch:extends href="svg-path.sch"/>
<sch:ns uri="http://example.com/parser/svg-path"
        prefix="p"/>
```

In the example above we are not only including file with parsing code but also declaring prefix that can be used for invoking parsing function.

4.3. Invoking parsing functions from Schematron

REx generates separate parsing function for each production rule in a grammar. In our example complete SVG path is described be the rule `svg_path` and there is correspoding function `p:parse-svg_path()` with the following signature:

```
<xsl:function name="p:parse-svg_path" as="item()*">
  <xsl:param name="s" as="xs:string"/>
```

Input to the function is SVG path and output is normally empty sequence which means that path can be successfully parsed and is valid according to the grammar. If path is not matching grammar then `ERROR` element is generated instead:

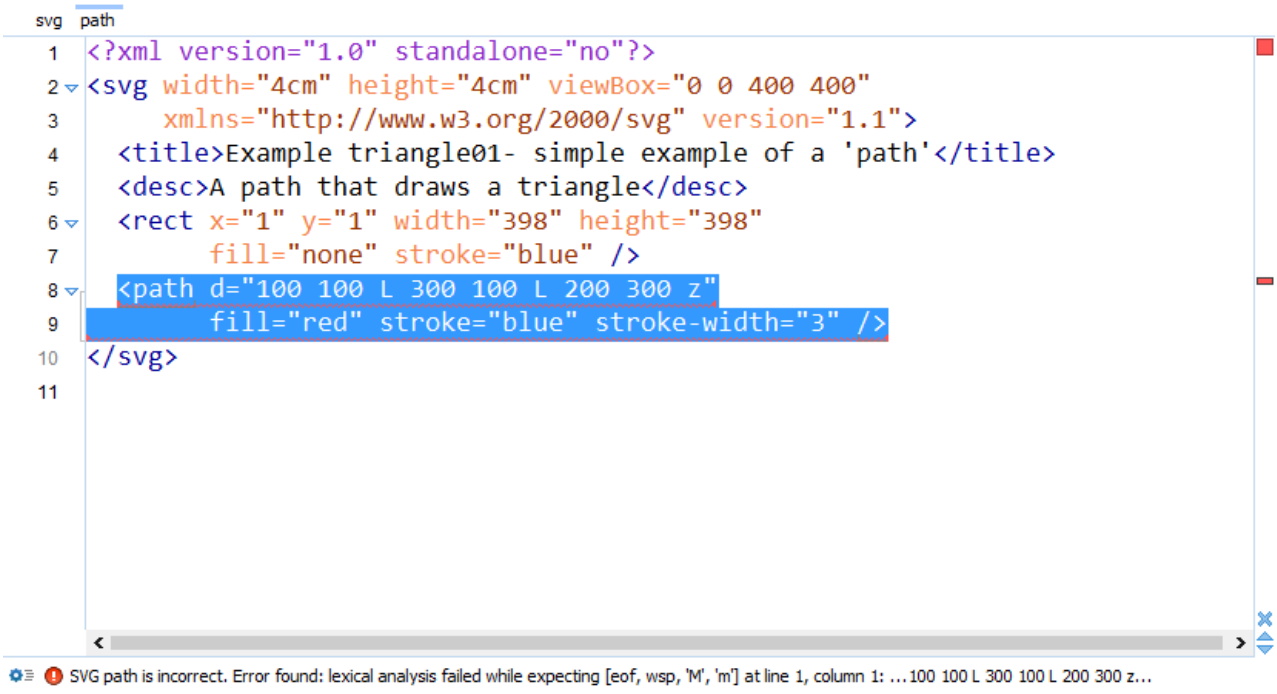
```
<ERROR b="1" e="1" s="13">lexical analysis failed
while expecting [eof, wsp, 'M', 'm']
at line 1, column 1:
...100,200 C100,100 250,100 250,200S400,300 400,200
...</ERROR>
```

We can use such function very easily in Schematron – cases when function returns `ERROR` element will be treated as errors. The following example shows how to invoke checking of SVG path syntax for `d` attribute on each `svg:path` element:

```
<sch:rule context="svg:path">
  <sch:report test="p:parse-svg_path(@d)/
                    self::ERROR">
    SVG path is incorrect. Error found:
    <sch:value-of select="p:parse-svg_path(@d)"/>
  </sch:report>
</sch:rule>
```

We can use such schema for checking SVG paths in all Schematron enabled validators that support XSLT 2.0 query binding and foreign elements. Figure 1, “Error in path reported in oXygen XML Editor user interface” shows how is such error reported in oXygen XML Editor.

Figure 1. Error in path reported in oXygen XML Editor user interface



4.4. Deep checking

As we already mentioned grammar can be used not only for checking syntax but it can be also used for building tree structure that represents parsed value in accordance with production rules. If we want to use this functionality we must use `-tree` option with REx:

```
rex -tree -xslt -backtrack -name \
http://example.com/parser/svg-path svg-path.ebnf

12 strong-LL(3)-conflicts handled by backtracking
```

With this option parsing functions are now returning trees with parsed result instead of empty sequence for strings that are matching grammar. For example result of calling `p:parse-svg_path('M100,100 L200,200')` is

```
<svg_path>
  <moveto>
    <TOKEN>M</TOKEN>
    <coordinate_pair_sequence>
      <coordinate_pair>
        <coordinate>
          <unsigned-coordinate>
            <number>100</number>
          </unsigned-coordinate>
        </coordinate>
        <comma_wsp>,</comma_wsp>
          <unsigned-coordinate>
            <number>100</number>
          </unsigned-coordinate>
        </coordinate_pair>
      </coordinate_pair_sequence>
      <wsp> </wsp>
    </moveto>
    <drawto_command>
      <lineto>
        <TOKEN>L</TOKEN>
        <coordinate_pair_sequence>
          <coordinate_pair>
            <coordinate>
              <unsigned-coordinate>
                <number>200</number>
              </unsigned-coordinate>
            </coordinate>
            <comma_wsp>,</comma_wsp>
            <unsigned-coordinate>
              <number>200</number>
            </unsigned-coordinate>
          </coordinate_pair>
        </coordinate_pair_sequence>
      </lineto>
    </drawto_command>
    <eof/>
</svg_path>
```

It is quite verbose piece of XML because each production rule matched is represented as one element. In order to generate more modest trees grammar can be augment with additional information defining which rules should contribute to tree building. Reduction of generated tree is discussed in more detail in Invisible XML [8].

We can access any information in this generated XML tree and check it further using XPath. For example we can easily check that all coordinates are within specified range as shown on the following example:

```
<sch:rule context="svg:path">
  <sch:let name="path"
           value="p:parse-svg_path(@d)"/>
  <sch:assert
    test="every $c in $path//(signed-coordinate |
                      unsigned-coordinate)/number
          satisfies abs(number) le 1000">
     All coordinates must be within [-1000, 1000]
     range.
  </sch:assert>
</sch:rule>
```

Accessing parsing result as XML gives us infinite opportunities for writing more advanced checks. We can use grammar to split complex microsyntax structure into smaller chunks and check them individually.

4.5. Comparision with regular expressions

As we have seen parsing code generated from a grammar offers much better diagnostic messages and allows to easily define additional checks on individual parts of validated values. Still the biggest advantage over regular expressions is readability of grammar.

But we should be as well aware of grammar based approach drawbacks. Regular expressions can be used directly as there is no need to compile grammar and invoke it from Schematron. In some situations it might be little bit tricky to write a correct grammar – there should not be ambiguities in grammar. But developers of XML schemas usually know how to overcome this as similar limitations exist in majority of XML schema languages.

We could also compare approaches in terms of performance. Proper benchmarks would be needed for this but practical experience shown that REx generated parser has similar speed compared to regular expression engine and can be even significantly faster for very complex regular expressions. Also complex regular expressions usually have quite high memory consumption.

5. More use cases

It is not very difficult to write grammars after learning some basics. Also for many computer languages there are existing grammars which can be reused. So improving validation of structured text inside XML content is quite straightforward. There are many use-cases for using such improved validation as many XML vocabularies are using various microsyntaxes. Lets briefly explore some of them.

Schematron is very powerful but assert or report is triggered only if XPath condition is true. Schematron validator will issue error if there is syntax error in XPath expression. But if the element name is just misspelled error will be silently ignored. It is not rare for complex Schematron schemas to contain constraints that are never triggered because there are mistyped element/attribute names. It is very hard to spot such problems unless unit testing with a good coverage is applied or schema-aware XPath is being used.

But we can very easily improve situation here. We can use grammar for XPath to parse all XPath expressions in Schematron schema and then try to lookup all elements/attributes used in node tests in a schema for vocabulary that Schematron schema is checking. If lookup fails for some name it means that we are accessing non-existent element/attribute which usually corresponds to typo in expression.

Another example – many computer languages are hybrid – they can embed fragments of code in different languages to support some special functionality. The most typical example is probably HTML and CSS. You can embed CSS stylesheets into HTML `style` element and similarly style declarations can be directly specified in HTML `style` attribute. We can integrate CSS grammar into validation workflow to easily check that CSS property names are not mistyped etc.

Another possible use-case is “proof reading” of textbooks about computer languages. It is very embarrassing to find syntactically incorrect examples of code in books about programming. As there are already existing grammars for many programming languages it is not hard to integrate this into processing workflow and run syntax checks on all embedded source code examples. Same grammars can be used for syntax highlighting of source code.

6. Better integration of grammar-based checking into existing schema languages

Compiling grammar into XSLT code and then invoking it from Schematron is workable solution for validation of microsyntaxes but it feels little bit as a hack. If such approach should be used more commonly then something better and more tightly integrated into existing schema languages will be needed.

Schematron is usually not used alone it just complements traditional validation agains W3C XML Schema or RELAX NG. If we want to be sure that both checks against classical schema and Schematron doing microsyntax checking are invoked we can use NVDL [9] to couple both schemas together:

```
<rules xmlns=
 "http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/2000/svg">
    <validate schema="svg.xsd"/>
    <validate schema="svg-path.sch"/>
  </namespace>
</rules>
```

It is even possible to integrate XML validation and microsyntax grammars directly into RELAX NG. This language already supports datatype libraries – user can supply implementation of few methods that are required for validation and then new datatype can be used in a schema in the same way as other normal datatypes. It would be possible to create preprocessor that would take grammar and transform it into the code that implements validation interfaces. Then we will be able to use this custom datatype library directly inside code, something like this:

```
<element name="path">
  <attribute name="d">
    <data datatypeLibrary=
            "http://example.com/SVG-datatypes"
          type="path"/>
  </attribute>
  ...
</element>
```

This is workable solution but it requires preprocessing step which will update configuration of validator. So in practice it would be more complex then previously mentioned Schematron based solution.

We can improve this solution and create grammar based datatype library which will compile grammars into parsing code on-the-fly. We can use datatype parameters to specify location of grammar:

```
<element name="path">
  <attribute name="d">
    <data datatypeLibrary=
      "http://example.com/grammar-based-validation"
          type="string">
      <param name="grammar">
        http://..../location/of/grammar.ebnf
      </param>
      <param name="start-rule">svg_path</rule>
    </data>
  </attribute>
  ...
</element>
```

Advantage of this solution is that once validator is extended by datatype library that understands arbitrary grammar we can reference any grammar from a schema and use it easily for validation.

Because of a high entry barrier of implementing validation interface RELAX NG datatype libraries are not as widely used as one would wish. But famous validator.nu validator which is used for validation of HTML5 content is using such custom datatype library. However definition of individual datatypes is not generated from grammars but it is written directly in Java. There is even implementation of SVG path checking code but reading it is not big fun compared to reading corresponding grammar.

Another integration option is again NVDL. Schemas referenced from NVDL can be of any type and it is implementation dependant whether some technology is supported or not. For example JNVDL validator can invoke XSLT transformation directly from NVDL to perform more algorithmically difficult checks. If we would implement direct grammar validation support in NVDL validator we could directly reference grammar from the schema. We will need to use `context` element to

specify content of which element should be validated against supplied schema.

```
<rules startMode="root" xmlns=
  "http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
  >
  <mode name="root">
    <namespace ns="http://example.com/invoice">
      <validate schema="invoice.rng">
        <context path="VAT"
                 useMode="VATNumber"/>
      </validate>
    </namespace>
  </mode>
  <mode name="VATNumber">
    <namespace ns="http://example.com/invoice">
      <validate schema="vat-number.ebnf"/>
    </namespace>
  </mode>
</rules>
```

We are using `context` element here to isolate one element and validate it not against classical XML schema but against grammar. Unfortunately NVDL does not provide similar functionality for single attribute.

7. Conclusions

We have shown that XSLT parsing code generated from a grammar can be invoked from Schematron to perform better checks and provide more useful error messages than regular expressions on a structured text content. With little bit of tooling Schematron proved itself again as a very flexible validation framework. There are several possible ways how to integrate grammar based checks more tightly into existing XML schema languages but this area needs more exploration.

Source code of examples used in this article are available at https://github.com/kosek/xmllondon-2017.

I would like to thank to Gunther Rademacher for creating REx parser generator and for helping me to discover some of its less known features.

Bibliography

[1] *VAT identification number*. Wikipedia. Accessed: 20 May 2017.
https://en.wikipedia.org/wiki/VAT_identification_number

[2] *ISO/IEC 19757-5:2011 – Information technology – Document Schema Definition Languages (DSDL) – Part 5: Extensible Datatypes.* [1]
https://www.iso.org/obp/ui/#iso:std:52118:en

[3] *Scalable Vector Graphics (SVG) 2*. World Wide Web Consortium (W3C). 15 September 2016. Nikos Andronikos, Rossen Atanassov, Tavmjong Bah, Amelia Bellamy-Royds, Brian Birtles, Bogdan Brinza, Cyril Concolato, Erik Dahlström, Chris Lilley, Cameron McCormack, Doug Schepers, Dirk Schulze, Richard Schwerdtfeger, Satoru Takagi, and Jonathan Watt.
https://www.w3.org/TR/SVG2/

[4] *Invisible XML*. Steven Pemberton. Montréal, Canada. Balisage. August 2013.
doi:10.4242/BalisageVol10.Pemberton01

[5] *Automated Tree Drawing: XSLT and SVG*. Jirka Kosek. O'Reilly Media. 8 September 2004.
http://www.xml.com/pub/a/2004/09/08/tree.html

[6] *Hedge automata: a formal model for XML schemata*. Murata Makoto. 1999.
http://www.horobi.com/Projects/RELAX/Archive/hedge_nice.html

[7] *Data Format Description Language (DFDL) v1.0 Specification*. Michael J Beckerle and Stephen M Hanson. September 2014. OGF 2014.
https://www.ogf.org/documents/GFD.207.pdf

[8] *On the Descriptions of Data The Usability of Notations*. Steven Pemberton. XML Prague. 2017.
http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=155

[9] *ISO/IEC 19757-4:2006 – Information technology – Document Schema Definition Languages (DSDL) – Part 4: Namespace-based Validation Dispatching Language (NVDL)*. 1 June 2006.
http://standards.iso.org/ittf/PubliclyAvailableStandards/c038615_ISO_IEC_19757-4_2006(E).zip

[1] FCD version of draft standard is available at
http://web.archive.org/web/20110515005051/http://www.itscj.ipsj.or.jp/sc34/open/1130.pdf

XSpec v0.5.0

Sandro Cirulli

XSpec and Oxford University Press

<sandro.cirulli@oup.com>

Abstract

XSpec is an open source unit test and behaviour driven development framework for XSLT and XQuery. XSpec v0.5.0 was released in January 2017 and included new features such as XSLT 3.0 support and JUnit report for integration with continuous integration tools. The new release also fixed long standing bugs, provided feature parity between the Windows and MacOS/Linux scripts, integrated an automated test suite, and updated the documentation. XSpec v0.5.0 is currently included in oXygen 19.0.

This paper highlights the importance of testing, provides a brief history of the XSpec project, describes the new features available in XSpec v0.5.0 and the work currently under development for future releases, and reports the effort of the XML community to revive this open source project.

Keywords: XSpec, Unit Testing, Behaviour Driven Development, Continuous Integration

1. The Importance of Testing

Testing is a fundamental part of writing software that aims to be robust, reliable, and maintainable. In fact, testing can be considered as a promise made to customers and users that the code behaves as intended. Writing tests regularly also improves the code base as it forces developers to write smaller units of code that can be more easily tested, debugged, and maintained. Finally, testing acts as self-documentation and can help other developers to understand and modify existing code.

Testing plays a central role in software development practices such as extreme programming (XP) and test-driven development (TDD) as well as in agile methodologies like Scrum and Kanban. For example, in test-driven development, unit tests (i.e. tests for individual units of code such as a function or a method) are usually written by developers as they write their code in order to make sure that new features work according to specifications and bug fixes do not break other parts of the code base. Unit tests increase the overall quality and maintainability of the code and it has been estimated that unit tests alone contribute to removing an average of 30% of defects present in software [1].

Although testing is important for any serious software developer, there aren't many testing tools for XSLT and XQuery when compared to other programming languages. Furthermore, their use is not yet very widespread. Back in 2009 Tony Graham [2] made an inventory of all the available testing frameworks available for XML technologies - most of which are unfortunately not actively developed any more. XSpec aims to fill this gap by offering a testing framework and raising awareness about testing in the XML community.

While any piece of XSLT and XQuery code can be tested using XSpec, the greatest return on investment occurs when testing code that gets called and reused frequently. Functions and named scenarios are perfect fits for unit testing as they are self-contained pieces of code upon which other parts of the code base may rely.

Integration with other testing and automation tools is also a key part of testing frameworks as unit tests are typically triggered automatically on events such as commits to the code base or software builds. Software development practices such as continuous integration (CI) popularized the importance of integrating development work into the code base on a daily basis, running test suites automatically, and providing developers with fast feedback when new code breaks tests or software builds. As a result, debugging and fixing bugs at the early stages of development improves the productivity of software developers and reduces the overall risk and cost of code releases and software maintenance.

2. A Brief History of XSpec

XSpec was created by Jeni Tennison in 2008 and is inspired by the RSpec testing framework for Ruby. Jeni Tennison presented XSpec at XML Prague 2009 [3] and released it as open source under the MIT license on Google Code. XSpec v0.3.0 was integrated in oXygen 14.0 in 2012 and this helped to spread its use and raise awareness about testing among XSLT developers.

doi:10.14337/XMLLondon17.Cirulli01

The project was maintained and expanded by Florent Georges who released v0.4.0-RC in 2012. Unfortunately active development stagnated between 2012 and 2015 with no further releases. The code base was moved from Google Code (now defunct) to GitHub in 2015.

I started contributing actively to XSpec in 2016 in order to fix an old bug and add a new feature I implemented at work. I forked the project and after few months I transferred my fork to the XSpec organisation repository under https://github.com/xspec/xspec. To my surprise, several people started contributing by raising issues and sending pull requests. This recreated an XSpec community that will hopefully sustain the project in the long term. This process culminated in release v0.5.0 in January 2017.

XSpec is under active development and new features and bug fixes are regularly merged into the master branch as soon as they are available and pass the test suite. For those who prefer a more stable version, the latest release is available as a zip file and can be retrieved from the official release page on GitHub [4].

3. New Features

A selection of the new features released in XSpec v0.5.0 is presented here. The full list of new features is available in the official release notes [4]. New features come with a test that makes sure that the feature behaves according to the specifications. Tests also act as documentation to show how the feature is implemented and are often used as examples in the documentation available on the official wiki [5].

3.1. XSLT 3.0 Support

XSpec now supports XSLT 3.0 [6]. This patch was provided by oXygen which first integrated it in its XML editor.

To illustrate XSLT 3.0 support, Example 1, "XSLT 3.0 Example" shows an example of XSLT that makes use of the inline function expression available in XPath 3.0 [7]:

Example 1. XSLT 3.0 Example

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs" version="3.0">

  <xsl:template name="supportXPath3">
    <root>
      <question>
        <xsl:text>Does XSpec
                  support XPath 3.0?</xsl:text>
      </question>
      <answer>
        <xsl:value-of select="
          let $answer := 'Yes it does'
          return $answer"/>
      </answer>
    </root>
  </xsl:template>

</xsl:stylesheet>
```

The template can be tested using the XSpec test in Example 2, "XSpec Test for XSLT 3.0". Note the use of `@xslt-version` specifying the version of XSLT (when `@xslt-version` is not provided, XSpec uses XSLT 2.0 by default).

Example 2. XSpec Test for XSLT 3.0

```
<x:description
  xmlns:x="http://www.jenitennison.com/xslt/xspec"
  stylesheet="xspec-xslt3.xsl" xslt-version="3.0">

  <x:scenario label="When testing the inline
                     function expression in XPath 3">

    <x:call template="supportXPath3"/>

    <x:expect label="it returns the expected answer">
      <root>
        <question>Does XSpec
                  support XPath 3.0?</question>
        <answer>Yes it does</answer>
      </root>
    </x:expect>

  </x:scenario>
</x:description>
```

3.2. JUnit Support

JUnit [8] is a popular unit testing framework for Java. JUnit reports are XML-based and are understood natively by popular continuous integration servers such as Jenkins.

In the past XSpec reports where only available in XML and HTML. XSpec v0.5.0 introduced JUnit reports which can be easily generated with the `-j` option from the command line as illustrated in Example 3, "Run XSpec with JUnit Option" (the sample file `escape-for-regex.xspec` is available in the tutorial folder on GitHub):

Example 3. Run XSpec with JUnit Option

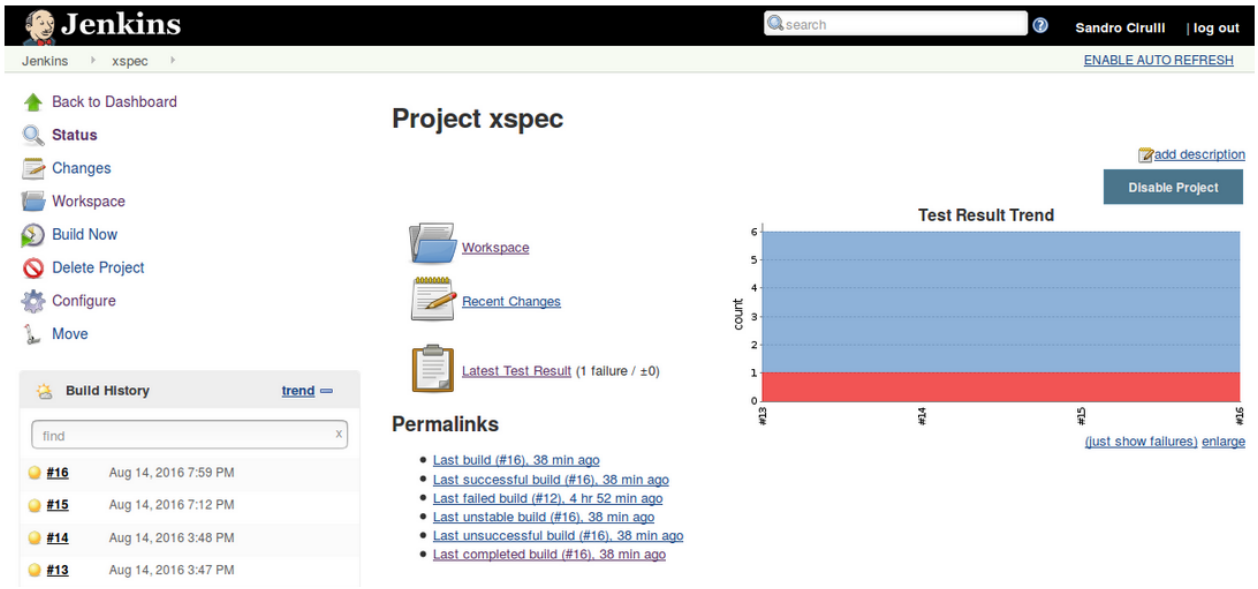
```
/bin/xspec.sh -j tutorial/escape-for-regex.xspec
```

Example 4, "JUnit Report" shows an example of the generated JUnit report with a successful and a failing test:

Example 4. JUnit Report

```
<testsuites>
  <testsuite name="When processing a list of phrases" tests="2" failures="1">
    <testcase name="All phrase elements should remain" status="passed"/>
    <testcase name="Strings should be escaped and status attributes should
                    be added" status="failed">
      <failure message="expect assertion failed">&lt;x:expect
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:test="http://www.jenitennison.com/xslt/unit-test"
        xmlns:x="http://www.jenitennison.com/xslt/xspec"
        xmlns:functx="http://www.functx.com"&gt;
            &lt;phrases&gt;
               &lt;phrase status="same"&gt;Hello!&lt;/phrase&gt;
               &lt;phrase status="same"&gt;Goodbye!&lt;/phrase&gt;
               &lt;phrase status="changed"&gt;\(So long!\)&lt;/phrase&gt;
            &lt;/phrases&gt;
         &lt;/x:expect&gt;
      </failure>
    </testcase>
  </testsuite>
</testsuites>
```

Figure 1. Test Result Trend in Jenkins



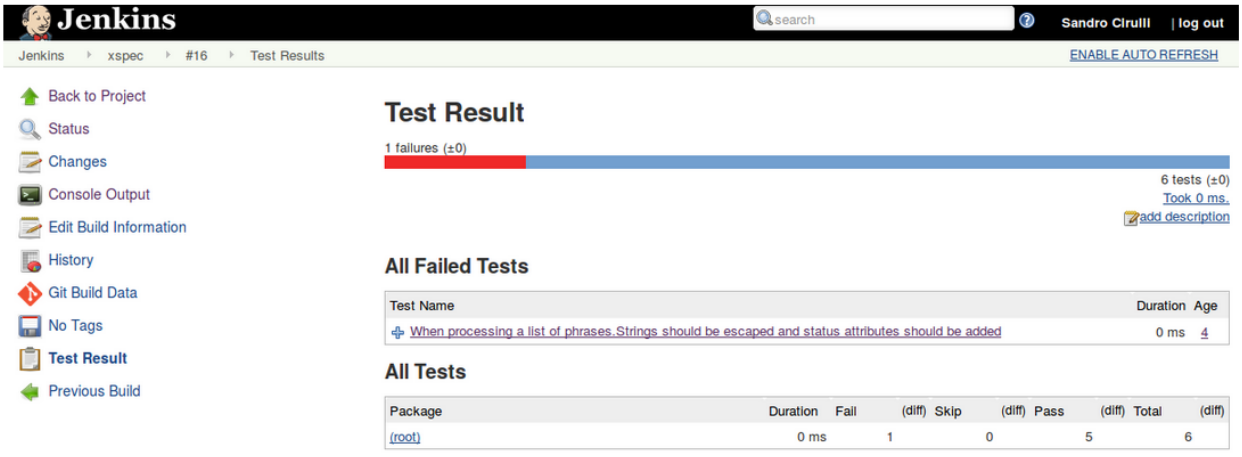
Note that the generation of JUnit reports requires Saxon 9 EE or Saxon 9 PE as the implementation makes use of Saxon extension functions.

JUnit reports can be easily plugged into continuous integration tools that understand JUnit natively such as Jenkins. The XSpec documentation describes how to configure Jenkins to run XSpec tests and generate JUnit reports [9]. Figure 1, "Test Result Trend in Jenkins" and Figure 2, "Test Result in Jenkins" show a test result trend and details of a failing test auto-generated by Jenkins from JUnit reports.

3.3. Testing XSpec

XSpec itself is tested using a mix of XSpec tests and shell and batch scripts. The test suite is executed automatically on online continuous integration servers (Travis for Linux and AppVeyor for Windows) every time a pull request or a code merge are initiated. This allows to spot regression bugs as soon as they appear and makes code reviews and approval of pull requests quicker and safer. In addition, testing XSpec with continuous integration tools such as Travis and AppVeyor provides example configuration and documentation for other projects that

Figure 2. Test Result in Jenkins



wish to use XSpec to run tests in continuous integration workflows.

3.4. Feature Parity between Windows and MacOS/Linux

XSpec can be executed from the command line using a batch script in Windows or a shell script in MacOS/Linux. Historically, the batch script lagged behind and did not provide all the options available in the shell script. XSpec 0.5.0 ships with a brand new version of the batch script that fully supports existing and new command line options available in the shell script. In addition, a test suite for the batch script is now executed on every new commit thus providing the same level of testing available for the shell script.

4. Bug Fixes

The full list of bug fixes is available in the official release notes [4]. Most bug fixes come with a test that makes sure that future code changes do not introduce regression bugs.

As example of defects fixed in this release, it is worth mentioning the code coverage bug. XSpec relies on Saxon extension functions for the implementation of the code coverage option that allows to check which parts of the XSLT code are covered by XSpec tests. This functionality was broken for several years due to a change in the implementation of the `TraceListener` interface between Saxon 9.2 and 9.3 and the bug was flagged by several users [10] [11].

This long standing issue has been fixed in v0.5.0 and the code coverage now works with the recent versions of Saxon EE and PE (extension functions are only available with these two versions of Saxon). Documentation on how to use the code coverage functionality is now available on the official wiki page [12].

5. Future Work

XSpec users can raise issues and feature requests on the official issue tracker on GitHub and contribute with pull requests. Some of the work that is currently under development or scheduled for the future releases of XSpec includes:

- **Schematron support**: A feature request was raised in order to have Schematron support in XSpec. This included use cases such as writing XSpec tests for Schematron rules and schemas. Vincent Lizzi provided a pull request for Schematron support in XSpec and demoed it during an open session at JATS-Con in April 2017. As I write these lines, the pull request has just been merged into the main XSpec code base and documentation will soon be available on the wiki. Schematron users are invited to test this new functionality and provide feedback.
- **Full XQuery support**: although XSpec allows to test both XSLT and XQuery, XQuery support is often lagging behind or untested. This work aims to bring full feature parity between XSLT and XQuery and to provide tests and documentation covering XQuery. A tutorial on how to write XSpec tests for XQuery is in the process of being written and will soon be available in the official documentation.
- **Harmonisation with oXygen**: XSpec is integrated by default in oXygen but some features such as the output report and the ant configuration are implemented differently. This work aims to harmonize XSpec so that the version provided in XSpec is the same version available on GitHub.

6. Conclusion

Testing is a crucial part of software development and XSpec aims to provide XSLT, XQuery, and Schematron developers with a testing framework for making their code more robust, reliable, and maintainable. After few years of stagnation, active development of XSpec restarted and culminated in the release of v0.5.0 in January 2017. This new release included several new features and fixed long standing bugs. Being an open source project, XSpec is developed and maintained by an active community gathering around the GitHub repository at https://github.com/xspec/xspec and welcomes new and existing users to contribute with issues, questions, and pull requests.

7. Acknowledgements

I would like to thank Jeni Tennison for creating XSpec back in 2008 and releasing it under an open source licence. I'm also deeply indebted to Florent Georges for maintaining the project in the past years and to Tony Graham for his support during the migration to GitHub. I own my deepest gratitude to the XSpec community who contributed to this release and provided me with encouragement and support, their GitHub user names are listed in the official release notes. A special thank you

to AirQuick - I ignore his real name - whose many pull requests, comments, and code reviews have been extremely valuable for the development of XSpec.

Bibliography

[1] Steve McConnell. 2006. *Software Estimation: Demystifying the Black Art*. Microsoft Press. Redmond, Washington.
ISBN 978-0735605350.

[2] Tony Graham. *Testing XSLT*. In Conference Proceedings of XML Prague 2009. March 21-22, 2009.
http://archive.xmlprague.cz/2009/presentations/XMLPrague_2009_proceedings.pdf#page=83

[3] Jeni Tennison. *Testing XSLT with XSpec*. In Conference Proceedings of XML Prague 2009. March 21-22, 2009.
http://archive.xmlprague.cz/2009/presentations/XMLPrague_2009_proceedings.pdf#page=105

[4] XSpec. *XSpec v0.5.0*.
https://github.com/xspec/xspec/releases/tag/v0.5.0
Accessed: 5 May 2017.

[5] XSpec. *XSpec Documentation Wiki*.
https://github.com/xspec/xspec/wiki
Accessed: 5 May 2017.

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

[7] World Wide Web Consortium (W3C). *XML Path Language (XPath) 3.0*. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. 8 April 2014.
http://www.w3.org/TR/xpath-30/

[8] JUnit. *JUnit*.
http://junit.org
Accessed: 5 May 2017.

[9] XSpec. *Integration with Jenkins*.
https://github.com/xspec/xspec/wiki/Integration-with-Jenkins
Accessed: 5 May 2017.

[10] XSpec Users Google Group. *XSpec 0.3.0*.
https://groups.google.com/forum/#!topic/xspec-users/0BIzNfFv4-Y
Accessed: 5 May 2017.

[11] Sandro Cirulli. *Continuous Integration for XML and RDF Data*. In Conference Proceedings of XML London 2015. June 6-7, 2015.
doi:10.14337/XMLLondon15.Cirulli01

[12] XSpec. *XSpec Code Coverage*.
https://github.com/xspec/xspec/wiki/Code-Coverage
Accessed: 5 May 2017.

Bridging the gap between knowledge modelling and technical documentation

Engage subject-matter experts to contribute to knowledge management and help them write accurate & correct documentation.

Bert Willems

FontoXML

<bert.willems@fontoxml.com>

Abstract

This paper describes an architecture which allows subject-matter experts and the systems to co-create both structured content and knowledge models. The proposed architecture creates a knowledge model from structured content which, in turn, is queried to validate and improve the accuracy and correctness of structured content leveraging the expertise of the subject-matter expert. The proposed architecture effectively describes a feedback loop.

1. Introduction

Writing content is hard. One has to understand the subject and the intended target audience and one must be able to express oneself in written word.

Fortunately, one does not usually stand alone. There is software to support one's writing endeavors. A well-known piece of software integrated into virtually any text editor out there is the spell checker. A spell checker typically works on individual words without looking at the meaning: as long as the word is spelled correctly, it is happy. More advanced are grammar checkers, which typically work by looking at sentences as a whole. They help authors write sentences that are correct from a grammatical perspective as prescribed by a particular language.

Readability checkers help authors write sentences that are easy to read. You don't want to squander your exquisite phrasing if your target audience is 6 years old. Several industries have defined a standardized subset of languages in order to improve readability, like ASD STE-100 Simplified Technical English.

However, none of the above prevents authors from writing complete and utter nonsense as long as it is spelled well, grammatically correct, and easy to read. Although this may seem like a benefit in some cases, in technical documentation it is not. An important use of documentation, whether online or printed, is to help users to do their work as efficiently as possible.

This paper proposes a solution to help authors to write *accurate* and *correct* documentation by bridging the gap between knowledge modelling and technical documentation. The first part of this paper describes a general architecture. In the second part a practical implementation is explored to prove the prosoed architecture can be built. The final part holds the conclusions and future work.

2. Structured Content feedback loop architecture

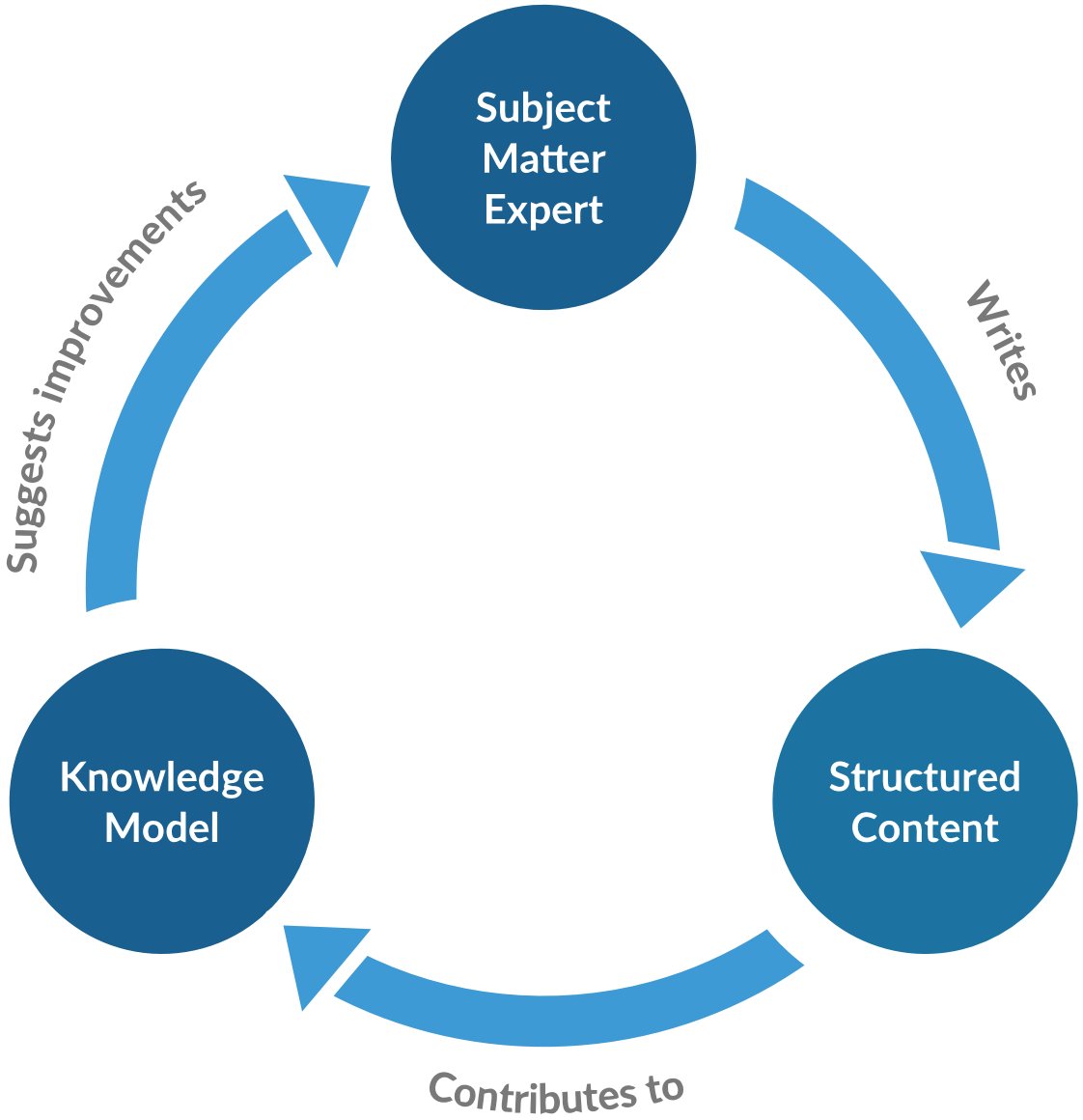
Technical documentation, or more generally speaking body of knowledge, contains valuable information from which knowledge models can be build. There is a lot of research in the area of automated extraction of facts to build knowledge models. Most of them rely on training sets put together by domain experts.

Structured content is usually written by subject-matter experts, who are domain experts themselves or act as proxies to experts and therefore are qualified to contribute to those knowledge models. This means that structured content is an excellent source of knowledge to build knowledge models from. Extracting facts from content, structured or not, is a well studied field.

There are numerous papers published which describe how information can be mined from content and how that information can be queried. However, none of those approaches are a 100% accurate, just like humans.

doi:10.14337/XMLLondon17.Willems01

Figure 1. Loop overview



Although the lines are blurring, in general computers are better at repetitive tasks while humans are better at unstructured problem-solving and empathy. This creates an interesting opportunity: *allow the subject-matter expert and the system to co-create both structured content and the knowledge model at the same time*.

Figure 1, "Loop overview" illustrates how ideas and knowledge are exchanged between subject-matter experts and systems:

We propose an architecture where the system ingests structured content in the form of XML from which a knowledge model is created. From this created knowledge model the system starts to suggest improvements to the subject-matter expert. The subject-matter expert evaluates the provided suggestions and, once accepted, improves the structured content. The improved structured content will in turn improve the created knowledge model, effectively closing a feedback loop.

The type of suggestions given by the system, dependents on the structure of the knowledge model and the algorithms used. In the problem space of technical documentation suggestions may include missing

prerequisites, opportunities for reuse and of course missing markup.

Even if the subject-matter expert decides to reject a suggestion, it is valuable. Consider the subject-matter expert rejecting a suggestion for a spelling correction: it might be the case that the word is missing from the dictionary or it should've been in the taxonomy. Some algorithms take counter examples as input to optimize their output. This means it is worthwhile to ask the subject-matter expert to provide feedback and update the knowledge graph accordingly, hence a secondary feedback loop.

3. Example implementation

This section describes an example implementation of the feedback architecture proposed in the previous section. The implementation is intentionally simple and straightforward but proves the loop can be built.

3.1. Problem example

Have a look at the following abbreviated example, taken from a procedure in a manual of our fictional ACME router:

```
Procedure: List all the files in the current
           working directory
Step 1: Execute the command 'dir'.
Result: An enumeration of all the files in the
        current directory.
```

An IT professional can, even without intimate knowledge of the ACME router, name at least a few (potential) errors in the seemingly simple snippet:

1. The procedure requires a terminal to be opened, which should have been encoded as the first step or as prerequisite.
2. It is unlikely the command is called 'dir' since that is Windows specific, it is more likely to be called 'ls' since it is more likely that the router is based on Linux.

In order to reason in the same way an IT pro can, the system requires the following knowledge to be available:

1. The `dir` command requires a terminal to be opened.
2. The `dir' and `ls` commands are directory listing commands.
3. The `dir` command is Windows specific.
4. The `ls` command is Linux specific.
5. The ACME routers run Linux.

3.2. Implementation

In order to create the feedback loop, the implementation works in two stages. The first stage creates the knowledge model from reference content. The second stage validates task-based content against the reference content. Where XML is given it is based on OASIS DITA (Oasis DITA 1.3).

3.2.1. Input reference documents

The following documents are used as reference documents from which the knowledge graph will be created. The areas that are relevant for creating the domain model are italicized.

```
<reference id="router">
  <title>ACME Router</title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Linux</platform></emphasis>
    </prodinfo>
  </prolog>
</reference>

<reference id="ls">
  <title>
    <emphasis><cmdname>ls</cmdname></emphasis>
  </title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Linux</platform></emphasis>
    </prodinfo>
  </prolog>
  <refbody>
    <p>The
         <emphasis><cmdname>ls</cmdname></emphasis>
       is a command used for <emphasis>
         <systemoutput>directory
            listing</systemoutput></emphasis>.
       It must run in the <emphasis>
         <uicontrol>terminal</uicontrol>
       </emphasis>.</p>
  </refbody>
</reference>

<reference id="dir">
  <title>
```

```
    <emphasis><cmdname>dir</cmdname></emphasis>
  </title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Windows</platform></emphasis>
    </prodinfo>
  </prolog>
  <refbody>
    <p>The
      <emphasis>
        <cmdname>dir</cmdname>
      </emphasis> is a command used
      for
      <emphasis>
        <systemoutput>directory
                        listing</systemoutput>
      </emphasis>.
      It must run in the
      <emphasis>
        <uicontrol>terminal</uicontrol>
      </emphasis>.
    </p>
  </refbody>
</reference>
```

Note

Some elements were removed from the documents for brevity.

Note

The examples given do not explicitly encode the "must run in a terminal" relation. This cannot be expressed well using standard DITA. Either specialization is needed or advanced text analysis software.

3.2.2. Knowledge graph

In order to validate the correctness according to the case described in the example section, the following knowledge must be captured by the model:

1. product to platform (used to determine the platform)
2. command to platform (used to determine whether a command is available)
3. command to result (used to infer the correct command based on platform)
4. command to required uicontrol (used to infer that a terminal is needed in order to perform the command)

The knowledge model is defined as an RDF graph. The graph is stored in a triple store, which allows semantic queries.

Constructing the knowledge graph

Using a simple extraction method the following triples can be extracted:

```
<#acme-router>
  runs <#linux> .
<#linux>
  a <#operating-system> .
<#windows>
  a <#operating-system> .
<#ls>
  a <#command> ;
  runsOn <#linux> ;
  requiresUI <#terminal> ;
  outputs <#directory-listing> .
<#dir>
  a <#command> ;
  runsOn <#windows> ;
  requiresUI <#terminal> ;
  outputs <#directory-listing> .
```

3.2.3. Task document

The following document is validated against the constructed knowledge graph:

```
<task>
<title>List all files and folder.</title>
  <prolog>
    <prodinfo>
      <prodname>ACME Router</prodname>
    </prodinfo>
  </prolog>
  <taskBody>
    <steps>
      <step>
        <cmd>Execute <cmdname>dir</cmdname>.</cmd>
      </step>
    </steps>
    <result>An enumeration of all the files
    in the current directory.</result>
  </taskBody>
</task>
```

3.2.4. Validating the task

Based on the given knowledge model and the task document, the system is able to find two inaccuracies:

1. The 'ls' command should've been used instead of the 'dir' command.
2. A terminal is required in order to execute the command.

Finding the correct command name

In order to find the correct command name the following traversals need to be made in the knowledge graph:

Input:

1. Product 'ACME Router'.
2. Command 'dir'.

Traversals:

1. Infer the 'ACME Router' runs on 'Linux'.
2. Infer the 'dir' command 'is available on' 'Windows'.
3. Infer the 'dir' command 'outputs' a 'directory listing'.
4. Infer the 'ls' command also 'outputs' a 'directory listing' AND 'is available on' 'Linux'.

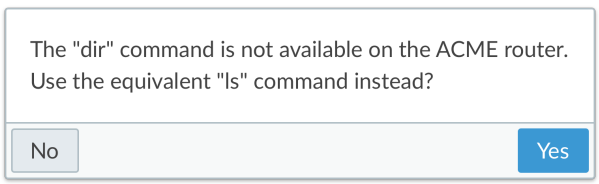
Based on the inputs and travels of the knowledge graph the system can return two facts to the user:

1. The command 'dir' is *not* available on 'Linux' and therefore *cannot* be correct.
2. The command 'ls' is a *substitute of* 'dir' and is available on 'Linux' and therefore *is* a suitable alternative.

Based on the inputs and traversal of the knowledge graph, the system can return the fact that the `dir` command should have been `ls` command. See Figure 2, "Replace command suggestions" for the suggestion. Approving the suggestion will change the XML into:

```
<cmd>Execute <cmdname>ls</cmdname>.</cmd>
```

Figure 2. Replace command suggestions



In order to find the missing prerequisite of the 'ls' command, the presence of a 'terminal', the following traversals need to be made in the knowledge graph:

Input:

1. Command 'ls'.

Traversals:

1. Infer the 'ls' requires a 'terminal'.

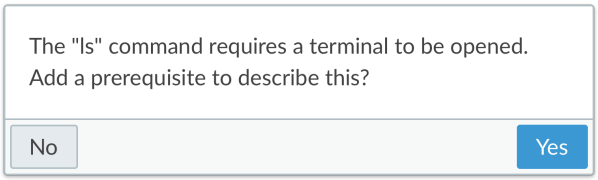
Check:

1. Check whether 'terminal' is referenced as a uicontrol.

Based on the inputs and traversal of the knowledge graph, the system can return the fact that a terminal is required. See Figure 3, "Replace command suggestions" for the suggestion. Approving the suggestion will insert the following XML snippet:

```
<prereq>Open a
          <uicontrol>terminal</uicontrol>.
</prereq>
```

Figure 3. Replace command suggestions



Resulting task

Now the resulting task is correct according to the available knowledge in the graph:

```
<task>
  <title>List all files and folder.</title>
  <prolog>
    <prodinfo>
      <prodname>ACME Router</prodname>
    </prodinfo>
  </prolog>
  <taskBody>
    <emphasis><prereq>Open a
      <uicontrol>terminal</uicontrol>.</prereq>
    </emphasis>
    <steps>
      <step>
        <cmd>Execute
          <emphasis>
            <cmdname>ls</cmdname>
          </emphasis>.</cmd>
      </step>
    </steps>
    <result>An enumeration of all the files
    in the current directory.</result>
  </taskBody>
</task>
```

Adding validation to the <result> of the task is left as an exercise to the reader of this paper.

4. Conclusions & Observations

As shown by the simple implementation presented in the previous section it is straightforward to derive a knowledge model based on reference content. Equally straightforward is the use of that knowledge model to validate task-based content for accurateness and completeness. This proves that the structured content feedback loop can be created.

However the implementation does not scale well: both the mapping to the knowledge model and the definition of the queries are done by hand. It is essentially an hard-coded rule engine.

The XML vocabulary used in the implementation example, does not support encoding all the relations out of the box. The DITA vocabulary does have a formal extension mechanism but, extending the vocabulary to support an evolving knowledge model does not scale either.

5. Future work

There is quite some work to be done in all the parts which make up the proposed system.

The first step is to remove the need for hard-coded rules by leveraging Information Extraction, a field of study which includes Part-of-speech (POS) tagging, phrase identification and word classification [1] and PATTY as described in Nakashole's 2012 PhD thesis [2] . Furthermore the mapping can be enhanced with the knowledge that can be mined from the XML schema as described in "Semi-Automatic Ontology Development" [3] and GRDDL [4].

The second step is to use Relational Machine Learning to query the knowledge model and provide useful suggestions and corrections to the subject-matter expert. The paper "A Review of Relational Machine Learning for Knowledge Graphs" [5] provides an excellent overview of that field of study.

Another step is to develop the software architecture based on OASIS Unstructured Information Management Architecture. This allows us to leverage and integrate existing components that are developed by third parties rather than developing everything ourselves.

The last, and perhaps the most important, step is to design and build an easy-to-use user interface. The suggestions must be easily understood and displayed in a relevant context to allow subject-matter experts to make quick and accurate decisions.

Bibliography

[1] *Information Extraction and Named Entity Recognition*. Christopher Manning. Stanford University.
https://web.stanford.edu/class/cs124/lec/Information_Extraction_and_Named_Entity_Recognition.pdf

[2] *Automatic Extraction of Facts, Relations, and Entities for Web-Scale Knowledge Base Population*. Ndapandula T Nakashole.
http://nakashole.com/papers/2012-phd-thesis.pdf

[3] *Semi-Automatic Ontology Development*. Processes and Resources. Maria Teresa Pazienza and Armando Stellato.
doi:10.4018/978-1-46660-188-8

[4] *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*. Dan Connolly. 11 September 2007. World Wide Web Consortium (W3C).
http://www.w3.org/TR/grddl/

[5] *A Review of Relational Machine Learning for Knowledge Graphs*. Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 25 September 2015.
https://arxiv.org/pdf/1503.00759.pdf

DataDock

Using GitHub to Publish Linked Open Data

Khalil Ahmed

Networked Planet Limited

<kal@networkedplanet.com>

Abstract

DataDock (http://datadock.io/) is a new service that aims to make it easy for anyone to publish Linked Open Data. It consists of two main parts, a data conversion service that turns CSV into RDF and creates a GitHub pages site from the data; and a gateway that performs the necessary redirects to make the published data work as Linked Data.

Although a number of other projects already use GitHub and GitHub Pages as a way to manage and publish (Linked) Open Data, DataDock has a unique way of managing the raw RDF data that makes it possible to use Git commands to determine the change history of a dataset.

This paper will describe the technical implementation of the DataDock service and our approach to storing RDF data in Git. It also proposes a method for making use of our storage approach to support distributed SPARQL querying of DataDock repositories.

Keywords: Git, RDF, LinkedData

1. Introducing DataDock

DataDock is a free data publishing service that enables anyone with data in CSV format to easily step up to publishing 5-star Linked Open Data. DataDock connects to a user's GitHub account and uses a public repository under their account to store the static pages for their Linked Data site. The pages themselves are served from GitHub Pages via a DataDock proxy that performs the necessary redirects to make the static pages work as linked data for both human beings and machines.

1.1. Background for DataDock

We built DataDock to address a perceived need for an easy way for small organisations and individuals to move from publishing open data as CSV to publishing open data as Linked Data. Our targets are charities and voluntary sector organisations; as well as data-for-good projects and "citizen data" projects. The goal of DataDock is to not only provide a simple CSV to RDF conversion service but to also provide the basic web server infrastructure required to serve Linked Data.

In our experience most organisations are relatively comfortable with working with data in tabular format - sometimes in relational databases, but more frequently in the form of spreadsheets. As a result, once the decision to publish open data has been taken the natural next step is to convert the data to CSV and publish a collection of CSV files. This makes the data available, but it is not as discoverable, or easily reusable in CSV form as it would be in Linked Data form. This is especially the case with data from smaller organisations as such data tends to be tightly focussed on one specific theme and geographical area and by itself may lack the scope for real insights to be derived from the standalone dataset. We believe that there is huge potential in supporting the many thousands of small organisations that are working across the UK to publish Linked Open Data that can be automatically located and integrated into other datasets from other organisations. However, it has to be recognised that two big barriers remain in the jump from CSV Open Data to 5-star Linked Open Data

The first barrier is in understanding and managing the mapping between the CSV data that the organisation wants to publish and useful Linked Data. A number of tools for transforming CSV to RDF already exist and although any of these could have been used in DataDock we took the decision to implement the recent W3C Recommendation on CSV to RDF conversion [1]. This decision was taken for two reasons. Firstly, the work done in the W3C has produced a specification that provides a great deal of scope for a flexible data mapping; and secondly from a technical perspective we wanted a native .NET solution that would enable us to use Microsoft Azure to host our conversion service. However, it is not just the technical issue of how the mapping is performed that is a problem, there is also a problem of choosing the ontology to map to. At this stage in the development of DataDock this remains an open issue which we think can really best be addressed by working

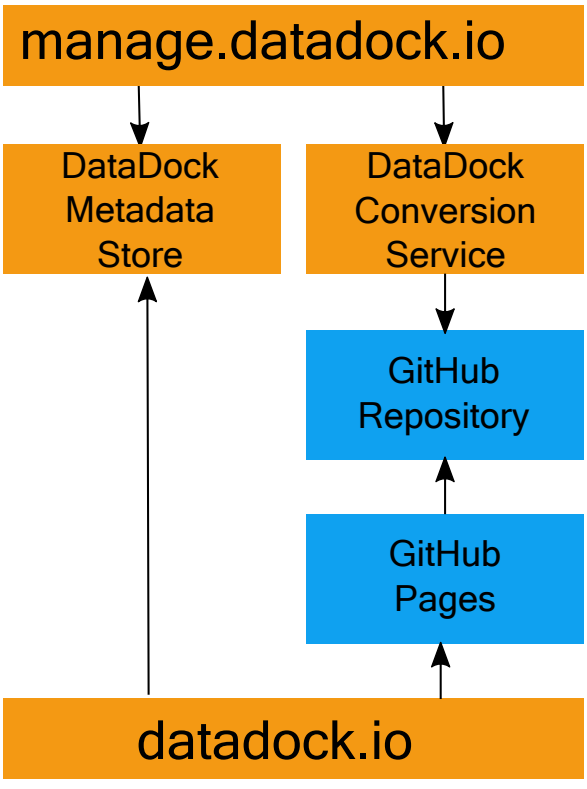
doi:10.14337/XMLLondon17.Ahmed01

closely with organisations to create recommendations on suitable ontologies to use for different types of data. We plan in future to use analysis of column names and datatypes as a way to drive intelligent suggestions for possible ontologies.

The second barrier for small organisations and for individuals is that hosting Linked Open Data requires a technical infrastructure and configuration that can be hard for non-technical (and even for many technical) users to set up. DataDock leverages GitHub Pages that provides a scalable, static web publishing infrastructure with our own proxy service that implements the additional functionality required of a Linked Open Data server (such as content negotiation for RDF data). The DataDock proxy also provides very basic directory services that enables search-based discovery of published data sets.

1.2. Architecture

The diagram below shows the high-level architecture of the solution:



Orange boxes represent the DataDock services. Blue boxes represent the services provided by GitHub. Arrows represent service dependencies rather than data flow.

The solution consists of two principle sites, a management portal (manage.datadock.io) and a proxy service (datadock.io).

The management portal allows users to sign in with their GitHub account, register a GitHub repository with DataDock, and to upload CSV data that is then converted into RDF and stored in their repository. The data conversion process converts the uploaded CSV to RDF and merges that RDF into the repository; it then generates a set of static HTML and RDF (NQuads) [2] files that will be served up via GitHub Pages. The RDF data is managed in such a way that a single GitHub repository can contain any number of separate datasets, but the RDF data in those datasets can be navigated and viewed as a single merged dataset. In addition to generating RDF data from the uploaded CSV, DataDock also generates VoID metadata for each of the datasets and for the repository as a whole. Finally at the end of the conversion process, the Git repository is tagged and a release is added which has the RDF data for the uploaded dataset attached to it as gzipped n-quads, providing a data download link that will be stable across time even as the dataset is modified in future updates.

The proxy service is a simple web proxy that is currently implemented using nginx. The proxy implements redirect rules that enable an identifier of the form `http://datadock.io/{user}/{repo}/id/` to be redirected to a GitHub pages address of the form `http://{user}.github.io/{repo}/`. The redirect rules take into account the Accept header of the request, redirecting the request either to the static HTML page generated for a resource, or to the static RDF file generated for that resource. These static files are generated from the merged RDF data in the repository and so render all of the data about a given resource that is contained across all of the datasets in that repository.

The DataDock metadata store is a NoSQL document database (Azure DocumentDB) that keeps a record of the VoID metadata and the conversion parameters for each dataset that a user uploads. This store is used to provide simple search facilities enabling users of either site to find relevant datasets by metadata such as title, description and keywords. In future iterations we plan to also use this store to enable sharing of vocabularies and social aspects of data publishing such as followers, likes and comments on datasets and repositories.

1.3. DataDock CSV Conversion

The CSV conversion process used by DataDock implements a subset of the conversion process described

in [1]. The goal of DataDock is to make the whole process of going from open CSV to Linked Open Data as straightforward as possible and it is our experience that one of the main intellectual stumbling blocks for users is the concept of ontology and ontology management. For this reason the first iteration of the conversion tool makes most of the decisions about the mapping to RDF for the user. The column names are used to generate predicate identifiers, and a single resource is generated per row using a combination of the source file name and the row number as the resource identifier. As the generated predicate identifiers are not scoped to the source file, by using consistent column names across their CSV files users can benefit from common predicate identifiers across all of their datasets. As an option, the user can select a column from their CSV to be used as the basis for generating an identifier for the row resource, in this case the generated identifier is also not scoped by the source CSV file, enabling resource identifiers to also be reused across multiple datasets. For other columns, the upload tool attempts to detect the column datatype, but the user is free to override the datatype as necessary. All configuration options selected by the user, plus any metadata that they provide such as dataset title, description and keywords are managed as a JSON object conforming to the Metadata Vocabulary for Tabular Data specification [3].

Conversion takes place as a batch process. The conversion service receives the source CSV file and a CSV Metadata JSON file that contains the conversion settings specified by the user. The result of the conversion process is a pair of RDF graphs, one graph contains the VoID metadata for the dataset and the other contains the RDF triples extracted from the CSV. Both graphs have a graph identifier that is based on the source CSV file name, so uploading a file with the same name as a previously uploaded file will replace the existing data in that graph - giving users a simple mechanism for continually updating their data.

1.4. Ontology Mapping

By default the column names in the uploaded CSV file are used to generate RDF property IRIs. The generated IRI uses the form `http://datadock.io/{user}/{repo}/id/definition/{column_name}`. This simplistic approach ensures that by default an ontology resource is created under the repository's datadock.io namespace for each column. It also means that when uploading more CSV data to the same repository with consistently named columns the ontology is automatically shared for each uploaded dataset.

The upload script also makes an attempt to determine the data-type for each column, but the user is also allowed to override the detected data-type and to suppress columns from the conversion process entirely.

For more advanced users the upload process offers the ability to manually modify the assigned property IRIs. This enables the use of external ontologies and we plan to extend this feature to provide suggestions from common ontologies. The column to IRI mapping is stored as part of the CSV Metadata JSON file in the GitHub repository, and is automatically reloaded when the user uploads a new version of the same CSV file to their repository.

The listings below show a sample CSV file and its associated CSV metadata. In this case the only change that the user made was to identify the "library" column as providing a value suitable for use as a resource identifier.

```
Library,Type,Computer Provision,No of PCs,Wi-Fi Provision,Latitude,Longitude

Blakelaw,Library,Public Computer Access,8,Public Wi-Fi,54.994164,-1.673143
```

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "url": "http://datadock.io/kal/data_dev/id/dataset/Newcastle%20Libraries%20List.csv",
  "dc:title": "Newcastle Libraries List.csv",
  "dc:description": "",
  "dcat:keyword": "",
  "dc:license": "https://creativecommons.org/publicdomain/zero/1.0/",
  "tableSchema": {"columns": [
    {
      "name": "library",
      "titles": ["Library"],
```

```
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/library",
      "required": true,
      "datatype": "string"
    },
    {
      "name": "type",
      "titles": ["Type"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/type",
      "required": true,
      "datatype": "string"
    },
    {
      "name": "computer_provision",
      "titles": ["Computer Provision"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/computer_provision",
      "required": true,
      "datatype": "string"
    },
    {
      "name": "no_of_pcs",
      "titles": ["No of PCs"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/no_of_pcs",
      "required": true,
      "datatype": "integer"
    },
    {
      "name": "wi-fi_provision",
      "titles": ["Wi-Fi Provision"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/wi-fi_provision",
      "required": true,
      "datatype": "string"
    },
    {
      "name": "latitude",
      "titles": ["Latitude"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/latitude",
      "required": true,
      "datatype": "decimal"
    },
    {
      "name": "longitude",
      "titles": ["Longitude"],
      "propertyUrl": "http://datadock.io/kal/data_dev/id/definition/longitude",
      "required": true,
      "datatype": "decimal"
    }
  ]},
  "aboutUrl": "http://datadock.io/kal/data_dev/id/resource/library/{library}"
}
```

This CSV input and mapping results in the following quads being generated (for clarity line-breaks and additional whitespace has been inserted into the generated NQuads output):

```
<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
 <http://datadock.io/kal/data_dev/id/definition/computer_provision>
 "Public Computer Access"^^<http://www.w3.org/2001/XMLSchema#string>
 <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/latitude>
  "54.994164"^^<http://www.w3.org/2001/XMLSchema#decimal>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/library>
  "Blakelaw"^^<http://www.w3.org/2001/XMLSchema#string>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/longitude>
  "-1.673143"^^<http://www.w3.org/2001/XMLSchema#decimal>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/no_of_pcs>
  "8"^^<http://www.w3.org/2001/XMLSchema#integer>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/type>
  "Library"^^<http://www.w3.org/2001/XMLSchema#string>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.

<http://datadock.io/kal/data_dev/id/resource/library/Blakelaw>
  <http://datadock.io/kal/data_dev/id/definition/wi-fi_provision>
  "Public Wi-Fi"^^<http://www.w3.org/2001/XMLSchema#string>
  <http://datadock.io/kal/data_dev/id/dataset/libraries_ncc-libraries-current_csv.csv>.
```

Currently the mapping process is limited to defining a single level of properties for a row-based entity. It is possible to choose which column provides the entity identifier and to map the IRI for that identifier; but it is not currently possible to define a nested structure for the properties of that entity. We plan to extend the mapping functionality in future releases to support mapping common CSV structures such as time-series data with minimal effort as well as providing a more complex UI to enable arbitrary nesting of generated entity properties to the extent supported by [1].

2. DataDock Repository Internals

Each DataDock GitHub repository has a four top-level folders.

- csv – contains the most recently uploaded CSV for each dataset along with a CSV metadata JSON file that uses the structure defined by [3].
- data - contains the generated static RDF files for each linkable resource in the repository's RDF graph. A linkable resource is one that is contained in the id namespace of the repository - i.e. one whose IRI is under `http://datadock.io/{user}/{repo}/id/`
- page - contains the generated static HTML files for each linkable resource in the repository's RDF graph.
- quince - contains the Quince repository that stores the complete RDF graphs for each dataset in a merged, searchable and diff-able format.

2.1. CSV

The csv directory in the repository contains the uploaded CSV files that have been converted to produce the data served by the repository. As these files are part of the public GitHub repository, this means that the source CSV files are always available for download, and that changes to the files over time are tracked by Git. Alongside each CSV file is a JSON file that contains the Tabular Metadata (as described in [3]) that describes how the CSV is mapped to RDF as well as the additional dataset metadata provided by the user on upload.

Each CSV file uploaded to a DataDock repository maps to its own named graph using the name of the CSV file to generate the graph IRI (as `http://datadock.io/{user}/{repo}/id/dataset/{filename}`). Keeping the RDF statements generated by each CSV file in a separate named graph makes later update of that data much easier to manage.

We take a very simple approach to managing updates to CSV files - when a file is uploaded with the same name as an existing file in the repository, the new file is treated as a new version of the existing file and replaces it in the repository. As a result, the named graph for that CSV file is simply dropped and replaced with the RDF generated from the new file.

2.2. Data

The data directory contains the static RDF files which are generated only for those RDF resources which can be accessed under the datadock.io namespace for the repository (i.e. those with identifiers that start `http://datadock.io/{user}/{repo}/id/`). Each resource contains all of the quads where the resource with that identifier is the subject of the quad.

Presently only NQuads format is supported to keep the file generation process lean.

2.3. Page

The page directory contains the static HTML files for the Linked Data site. As with the RDF, files are generated only for those resources accessible under the datadock.io namespace for the repository. Files are simple HTML5 with RDFa.

Currently these pages follow a fixed internal page template using the Liquid templating language with some extensions to more easily handle sorting and filtering collections of RDF quads. This opens up the scope for a future update allowing users to build their own page templates and have some level of control over the way that these HTML pages present their data.

2.4. Quince

Quince (QUads IN Calculable Storage) stores RDF data as NQuads files. NQuads is great because it is a line-oriented format and because Git reports diffs at a line level, we can directly use Git diff reports as a quad-level diff report. To do this effectively however we need to address the following issues:

- Very large files are not efficiently handled in Git so we need some means to split the RDF data across multiple files.
- On the other hand lots of very small files causes IO problems even when using solid-state storage so a balance needs to be struck somewhere between the naive options of using one NQuads file for all data at one end of the scale versus using a separate file for each distinct subject at the other.
- When a quad is inserted into the store, it should not duplicate an existing quad - so we need an efficient way to check for duplicates.
- We need some sort of consistent ordering for quads so that Git's diff reports don't (too often) get thrown out by a reordering of the lines in a file.
- We need to be able to support efficient lookup both by subject IRI and by object IRI/Literal. Object-to-subject traversal will be particularly important for doing user-friendly diff and for rendering incoming links on the static HTML pages.

2.4.1. Quince Algortihms

To address these issues we follow this algorithm for performing an insert:

1. Insert the quad into two separate files, one file has a file name generated from the subject of the quad, the other has a name generated from the object of the quad.
2. Generate a node ID using the SHA1 hash of the NQuads string representation of the subject/object (with the hash value formatted as a hex string), prefixed with "_s" if the ID is generated from the subject and "_o" if the ID is generated from the object.
3. The target file to be updated is determined by taking pairs of characters from the ID starting with the first pair (which will be "_s" or "_o"). If a directory is found matching that name, enter that directory and look for a subdirectory that matches the next pair of characters in the ID and so on until no matching directory is found. At that point the last (non-matching) pair of characters is used to instead generate a file name (simply the two characters of the pair with the file extension .nq added). If that file is not found it is created, otherwise the file is read in. These NQuads files are kept in neutral (code-point based) string sort order. The lines of the file are search for the quad and if the quad is found then no update is performed; if the quad is not found then it is inserted into the correct point of the file to maintain its sort order.
4. When the file update is ready to be written to disk, check the output file size (currently using a simple count of the number of lines in the file as a threshold). If the file size exceeds the threshold size, create a new directory using the file name (without the extension); re-insert each of the quads in the file (they will now go into files inside the newly created sub-directory); and then delete the file. In this way a file gradually grows until it reaches a target size and at that point it spawns another 256 (two hex chars) smaller files in a sub-directory.

Deletion of a single quad follows a similar algorithm for locating the quad in both the _s and _o sub-trees. Similarly lookup by subject or object IRI is trivially handled by using the same file location algorithm.

Exporting an entire named graph is handled by the traversal of the entire subtree rooted at _s. Deletion of an entire named graph requires a traversal and update of both the _s and _o subtrees.

2.4.2. Reporting Diffs

Git's default line-based differencing engine enables us to quickly generate a report of which files in a Quince repository have been modified. NQuads is a line-oriented format, so this report effectively gives us a list of quads added and deleted.

However when a file is split (and sometimes even when files have not split) Git will report some lines as being both inserted and deleted. The solution to this over-reporting of changes is to simply treat the lines inserted and the lines deleted as sets (I and D) and then report the effective insertions as the set difference I\D and the effective deletions as the set difference D\I.

It should be noted that this report deliberately does not specify which file(s) in the quince repository were modified as this information is irrelevant to applications which treat the repository as an RDF data-store. Applications which require file-level diff information can simply use the default Git diff reports.

2.4.3. Merging

For DataDock, merging is generally not an issue as we process the updates on a repository in sequence rather than in parallel. However in the more general case Quince needs a way to handle merge and merge conflicts.

With fixed file paths, merge conflicts would be simple to handle by accepting both sets of changes and then re-sorting the lines in each modified file and removing any duplicate rows that occurred due to both sides adding the same quad. However, the dynamic approach to the file store prevents this approach from working effectively as we must now handle the possibility that we are merging a change that has split a file with a change that modified the file without splitting it. As a result we cannot rely on Git's default conflict resolution.

When a conflict occurs as the result of a merge we instead take the effective diffs of each branch from their common ancestor commit (as described in the section above), union the insert and difference sets and apply the same set algebra to determine the overall effective inserts and deletions and apply those changes to the store state at the ancestor commit. This has the effect of preserving meaningful merge semantics but the down-side is that the Git repository does not accurately reflect the merge state of individual branches. Future work will look at combining this approach with Git's default merge operation so that merges can be effectively tracked.

2.4.4. Future Work: Quince and Triple Pattern Fragments

The current layout of a Quince repository is particularly suited to serving Linked Data Fragments [4] using the Triple Pattern Fragments interface. The two top-level directories (_s and _o) provide effective indexes for matching triple pattern fragment queries where either the subject or the object (or both) are bound. The current layout does not provide efficient support for handling the case where only the predicate is bound. We are currently considering what form of on-disk file-based index would be suitable to support this use case. One simple approach would be to add a parallel _p directory where the file path is generated from a hash of the predicate node, but this must then take account of the likelihood that for any given predicate there could be a very large number of quads requiring an extension to our file splitting algorithm that allows such large sets to be split over multiple files even though they all share the same predicate value.

However, with this one problem solved the potential is then opened up for building a Linked Data Fragments server over a Quince store, and from there leveraging the existing SPARQL [5] implementations that use the Triple Pattern Fragment interface (such as https://github.com/LinkedDataFragments/Client.js).

3. Conclusion

In this paper we presented DataDock, a service aimed at helping organisations and individuals publish their CSV data as Linked Open Data. This service builds on top of the static web publishing features of GitHub Pages with the addition of a proxy service that provides the necessary HTTP redirects to enable GitHub Pages sites to serve Linked Data under the http://datadock.io/ domain. The service also partially implements the W3C TR for generating RDF from CSV.

Each user has their own separate Git data repository that contains the source CSV data, the CSV Metadata file that specifies the mapping to RDF and the RDF resource pages in both HTML and NQuads formats.

This paper also introduces Quince, a file-based RDF data repository specifically built to enable the use of Git for version tracking. Quince uses sorted NQuads files to store teh raw RDF data, with a dynamic file system that avoids both very large files and large numbers of very small files by iteratively splitting files as they exceed a specified file size threshold.

We have also presented a number of areas for future work including the possibility of adding a Triple Pattern Fragments interface to the DataDock repositories which would then enable distributed queries against the DataDock repositories.

Bibliography

[1] *Generating RDF from Tabular Data on the Web*. Jeremy Tandy, Ivan Herman, and Greg Kellog. World Wide Web Consortium (W3C). 17 December 2015.
https://www.w3.org/TR/csv2rdf/

[2] *RDF 1.1 N-Quads*. A line-based syntax for RDF datasets. Gavin Carothers. World Wide Web Consortium (W3C). 25 February 2014.
https://www.w3.org/TR/n-quads/

[3] *Metadata Vocabulary for Tabular Data*. Rufus Pollock, Jeni Tennison, Ivan Herman, and Greg Kellog. World Wide Web Consortium (W3C). 17 December 2015.
https://www.w3.org/TR/tabular-metadata/

[4] *Linked Data Fragments*. Ruben Verborgh.
http://linkeddatafragments.org/

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

Urban Legend or Best Practice

Teaching XSLT in The Age of Stack Overflow

Nic Gibson

<nicg@corb.as>

1. Background

Traditional approaches to teaching XSLT and other development technologies are undergoing rapid change. The rise of online training platforms and peer to peer environments such as stackoverflow.com have changed the way that developers learn technologies. In the XSLT world we are extremely lucky to have some amazing people answering questions on the Mulberry mailing list and Stack Overflow. However, when a developer asks a question on Stack Overflow or uses Google to find an existing answer, the *why* behind any particular answer is often lost.

A recent exchange on Stack Overflow (SO) led me to wonder how much of our best practice might be urban legend and to consider how XSLT and other technologies could be taught well in this online environment.

This paper investigates one of these questions and answers and considers whether ten year old questions and answers are the wisdom of the ages or myths and legends. I will consider whether answering questions online should be part of a teaching or training experience or whether it is simply outsourced problem solving. Which of these approaches leads to higher quality XSLT development (and developers)?

2. Investigation

Let us look at the following question asked on SO earlier this year[1]:

> Extract unique elements from the input XSLT
>
> For example the input is this:
>
> ```
> <root>
>   <command name="comm1">aa</command>
>   <command name="comm2">bb</command>
>   <command name="comm3">cc</command>
>   <command name="comm3">dd</command>
>   <command name="comm2">ee</command>
>   <command name="comm1">ff</command>
>   <command name="comm5">gg</command>
> </root>
> ```
>
> The desired output is this:
>
> ```
> <root>
>   <command name="comm1">aa</command>
>   <command name="comm2">bb</command>
>   <command name="comm3">cc</command>
>   <command name="comm5">gg</command>
> </root>
> ```
>
> You can see that at the output, we don't have repeating tags, the text values are not important here.

I answered this question in the way that many people seem to do on SO. That is, I dashed off a quick answer without thinking clearly about the answer. I chose an answer that was the simple possible in terms of XSLT because I hoped to provide something that didn't need explanation:

```
<xsl:template match="command">
  <xsl:if test="not(preceding-sibling::command
                    [@name = current()/@name])">
    <xsl:copy-of select="."/>
  </xsl:if>
</xsl:template>
```

I gave an XSLT 2.0 solution as well:

```
<xsl:template match="command[
                  preceding-sibling::command[
                    @name = current()/@name]]"/>
```

[1] https://stackoverflow.com/questions/42023432/extract-unique-elements-from-the-input-xslt

doi:10.14337/XMLLondon17.Gibson01

Martin Honnen pointed out that this was not a good answer and referred to Jeni Tennison's explanation of Muenchian grouping. We disagreed because I felt that a simple solution was better than a complex one in this circumstance.

In order to determine how much better the efficient approach is compared to the simple approach, I have used a dataset the matches the one given in the question above but varies from ten distinct elements to five hundred distinct elements with from 0% to 50% duplication (so a five hundred distinct value set will include randomised sets of values containing a total of five hundred to seven hundred and fifty elements). Five variants were created at each level of duplication to increase confidence that the data was actually randomised. Given a maximum number of twenty different duplications and using a step of ten unique elements, this led to around two thousand documents.

Each document was processed using xsltproc, Saxon 9.7 and MSXSL 4.0. The data suggests that, whilst different approaches obviously give differing results, they are not as significant as one might hope. Three different XSLT variants are used although xsltproc and MSXSL cannot be used with and XSLT 2.0 or 3.0 solution.

3. Testing The Concept

Given the conversation on SO, testing the hypothesis that the simpler answer is often as good as the most effective answer, it seemed appropriate to generate some experimental data. Given that different users will be working in different environments, it seemed wise to generate data using different tools. There are a limited number of easily available XSLT processors so I used the following processors:

- Saxon 9.7 EE (Java)
- Microsoft MSXSL 4
- XSLTproc 20904

I ran all tests using an Apple iMac with a quad core 2.9Ghz i7 processor and 32 GB RAM. XSLTproc and Saxon were tested under MacOS whilst Microsoft MSXSL 4 was tested on a virtual machine running Windows 7.

3.1. Test Data

In order to test the various approaches to resolving this problem, I generated a set of data (using XSLT 3 running under Saxon EE). All of the elements were variants on

```
<root>
  <command name="command1"/>
  <command name="command2"/>
</root>
```

The data was generated by generating one thousand variants on `<command/>` as above. From that set of data between ten and one thousand elements were selected with an interval of ten

For each set of unique elements, random duplicates were added (using `fn:random-number-generator()`). A range of random elements representing between one and fifty percent of the total was generated. In order to keep the number of documents to a reasonable size no more than 20 sets of duplicates spread evenly between one and fifty percent were generated:

Uniques	Duplicates	Step
10	1 to 5	1
100	2 to 50	2 or 3
1000	25 to 500	50

Each combination of unique and duplicate elements was generated five times and the elements were ordered randomly before being output to documents. The generating script is in Appendix A, *generator.xsl*.

For the purpose of analysis the mean of each of the five sets for each unique/duplicate pair was used.

3.2. Test Scripts

I used eight different test scripts. Each of these is very simple and tries to resolve the problem in a different way. Each script was run against the full output set of 16,980 files (five variants of each unique/duplicate pair). There are only three fundamental approaches:

- Predicates
- Grouping
- Distinct Values

3.2.1. xsl-1-predicate.xsl

This was the approach I suggested on SO. There are three variants of this scripte using XSLT 1.0, 2.0, and 3.0. Unfortunately, only the first could be tested using

multiple processors. It is not intended to be efficient just simple.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="command">
    <xsl:if test="not(preceding-sibling::*
                      [@name = current()/@name])">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

3.2.2. xsl-1-group.xsl

This is a more efficient stylesheet using Muenchian Grouping. This is the stylesheet that would probably have been the best solution in XSLT 1.0

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

  <xsl:key name="element-key" match="command"
           use="@name"/>

  <xsl:template match="root">
    <root>
      <xsl:for-each select="*[count(
        . | key('element-key', @name)[1]) = 1]">
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </root>
  </xsl:template>

</xsl:stylesheet>
```

3.2.3. xsl-2-predicate.xsl

This stylesheet simply moves the predicate into the `match` attribute of the template.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="command[preceding-sibling::
                command[@name = current()/@name]]"/>

</xsl:stylesheet>
```

3.3. xsl-2-group-by.xsl

This stylesheet uses `xsl-for-each-group` to generate the same result. A useful additional test would have been to use `xsl:sequence` instead of `xsl:copy-of` but that was simply another variant on the same theme.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="root">
    <root>
      <xsl:for-each-group select="command"
                          group-by="@name">
        <xsl:copy-of select="."/>
      </xsl:for-each-group>
    </root>
  </xsl:template>

</xsl:stylesheet>
```

3.3.1. xsl-2-distinct-values.xsl

This stylesheet uses the `distinct-values()` function from XPath 2.0 to resolve the issue in a different way. The preceding stylesheets all retain the elements in some way whilst this one completely rewrites them and would not

be suitable for use in any more complex 'real-world' situation.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="root">
    <root>

      <xsl:for-each
           select="distinct-values(command/@name)">
        <command name="{.}"/>
      </xsl:for-each>
    </root>
  </xsl:template>

</xsl:stylesheet>
```

3.3.2. XSLT 3.0 variants

The remaining two stylesheets were simply minor modifications to stylesheets 1 and 3. The identity transformation was replaced by:

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

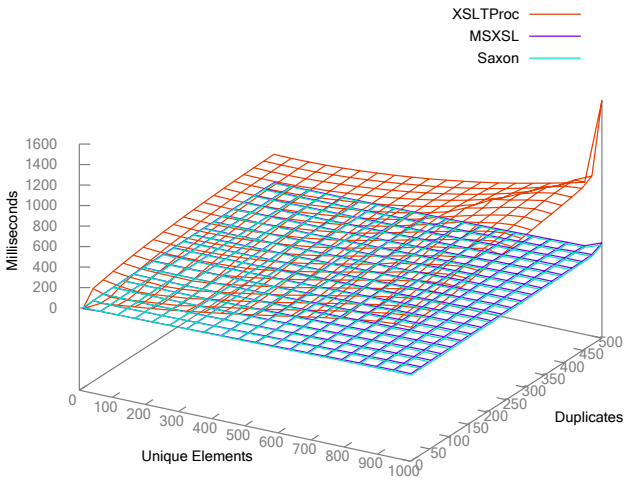
3.4. Process

Each of the XSLT engines has a command line interface and each is able to report on processing times. For each of these the time to process the input documents was ignored and only the time taken to execute the stylesheet was used.

3.5. Results

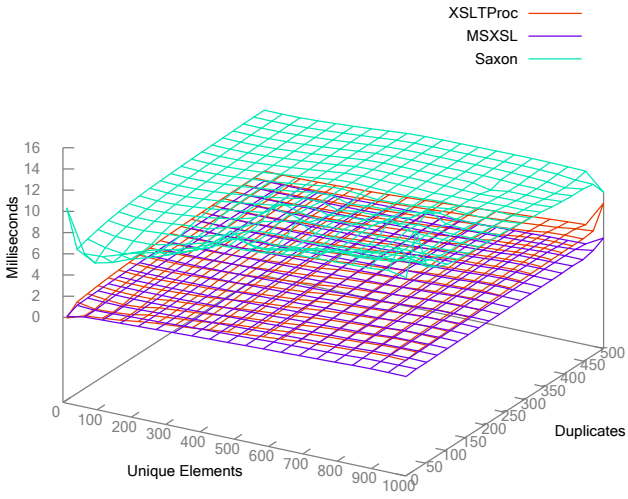
The most obvious result was that Microsoft's MSXSL processor is dramatically faster at XSLT 1.0 than the other processors. However, MSXSL is old and has not been updated in ten years. We can see from the initial results mentioned above that an optimised solution does increase performance, but that the performance benefits are not as significant as one might hope given that the maximum processing time was still under two seconds, see Figure 1, “XSLT 1.0 Predicates”.

Figure 1. XSLT 1.0 Predicates



The approach suggested by Martin Honnen provides better results by a factor of up to one thousand. The performance of MSXSL at both tests is similar and leads to suspicions about the validity of the data provided by the tool, see Figure 2, “XSLT 1.0 Groups”.

Figure 2. XSLT 1.0 Groups



The overall flatness of the graphs suggests that adding random repetitions to the groups does not have a significant effect on the results. If we graph the scripts against time using only the total number of elements to be tested we see the graph in Figure 3, “XSLT 1.0 Approaches”. The other approaches tried have been added in Figure 4, “XSLT 2.0 Approaches”.

Figure 3. XSLT 1.0 Approaches

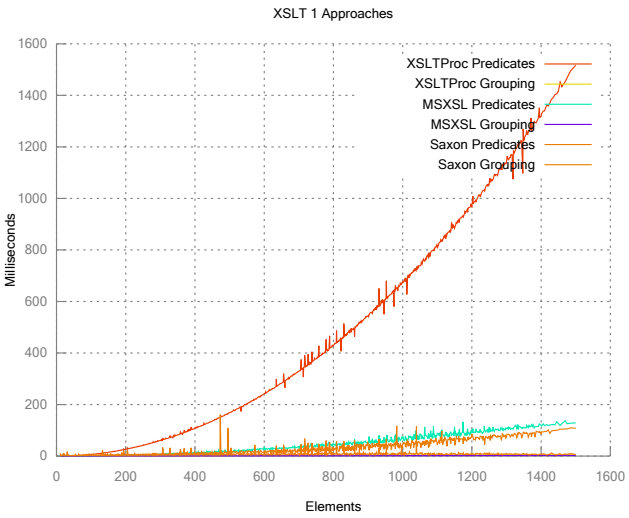
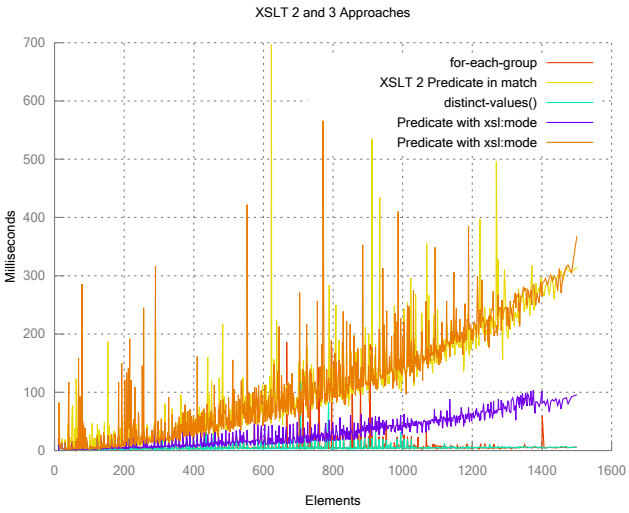


Figure 4. XSLT 2.0 Approaches



We can see that approach that uses the `distinct-values()` function is effectively as fast as the approach that uses the `xsl:for-each-group` statement and that those are both as fast as the Muenchian Grouping based approach.

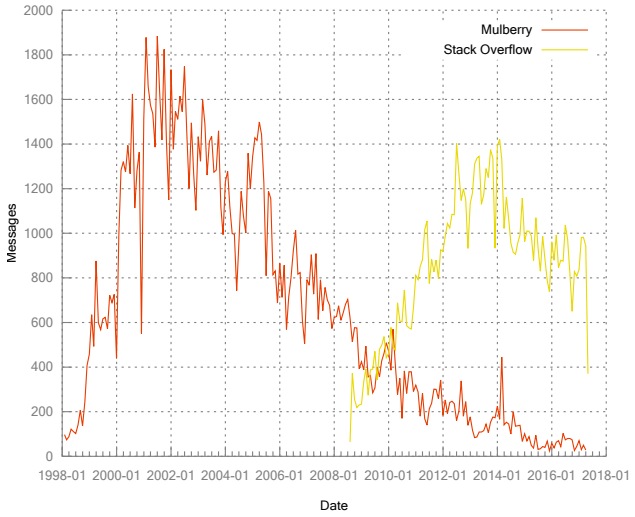
We have multiple approaches to solving this problem that work as well as each other and are all considerably superior to the accepted answer.

4. Learning and Training

The amount of XSLT training on offer via training providers has diminished dramatically over the last few years. In the UK the vast majority of training providers list schedules for XSLT courses but will even attempt to identify a trainer to run those courses unless enough people request the course. Experience suggests that this no longer happens. The largest training providers in the UK have reduced their schedules for their XSLT training to a minimum (QA training run a course once a year, Learning Tree have removed it from their schedule). Custom training for corporate customers has also seen reduced activity (Learning Tree running two on-site courses over the last two years)

Activity on mailing lists and SO shows a reduction over time. The graph in Figure 5, "Activity over time" shows the posts to the Mulberry Technologies XSLT mailing list over time.

Figure 5. Activity over time

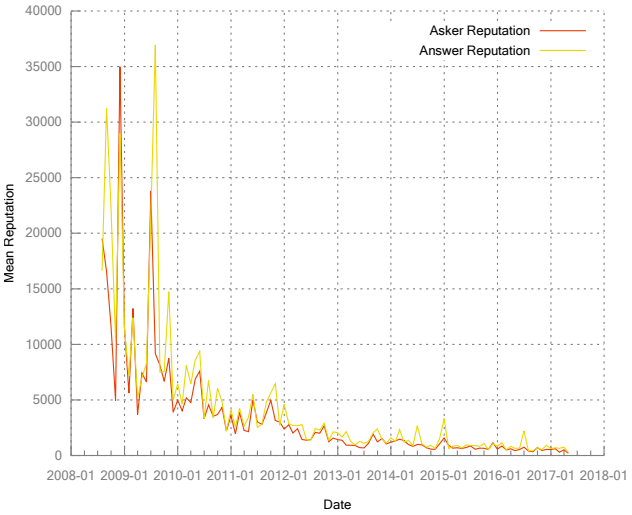


However, the amount of XSLT development taking place has not diminished in the same way (although conversation suggests some decrease). New developers are being hired but most of those do not go through traditional training and, in fact, seem most likely to use SO for their information. Whilst the XML (and XSLT) and the Web have no provided as much work for XSLT developers as could have been the case, publishing and banking have taken up much of the slack (although banking focusses more and more an XQuery and MarkLogic).

Considering the average reputation of those asking and answering messages on SO, we can see that the mean (see Figure 6, "Mean Reputation on Stack Overflow" has consistently dropped over the nine years of data available after an initial peak. This suggest that (bar a few honourable exceptions) those answering questions about XSLT are becoming less skilled (if one assumes that

reputation can translate to 'skill') over time as are those asking the questions.

Figure 6. Mean Reputation on Stack Overflow



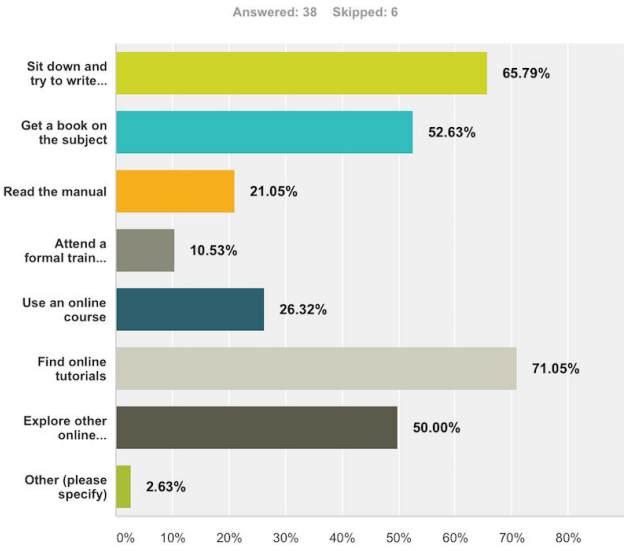
This situation has implications for those of us who either answer questions on forums or mailing lists and for those of us with a background in traditional training (or both).

4.1. Survey

In order to find out more about the way that developers and XML developers in particular go about learning and studying we ran a small survey using Survey Monkey. There were a total of forty four responses to this survey. Of those who responded the majority were aged over thirty suggesting that they are unlikely to be at the start of their careers.

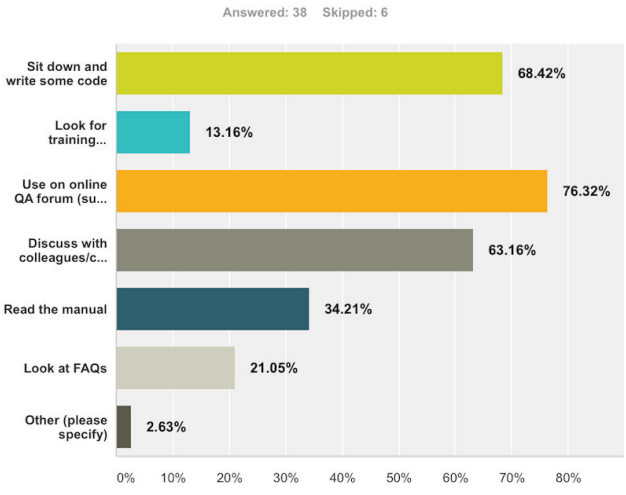
We asked how they approached learning a new skill. We allowed users to answer multiple times on this question. Only 10.5% of the 38 respondents who answered this question would approach this task by attending traditional formal training. There are many reasons why this may be the case but the important idea is that formal training is no longer the primary approach to skills learning. This may have been the case for many people at any time. However, corporate training programmes used to assume that formal training was the primary approach and this is no longer the case.

Figure 7. If you need to learn a new development skill, what would be your main approach?



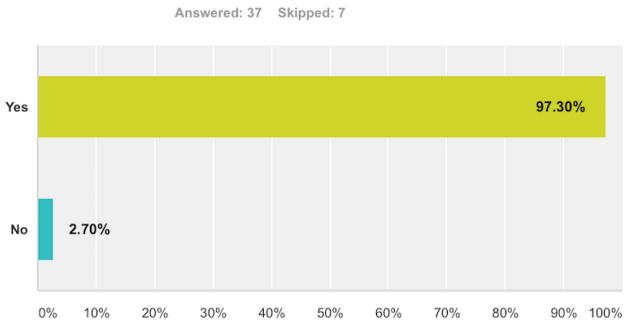
We also asked how respondents would go about solving a specific development problem, again allowing them to respond multiple times. Over three quarters of those responding identified online resources such as SO as part of their approach.

Figure 8. If you have a specific development problem, how would you approach solving it?



We asked how many respondents used sites such as SO:

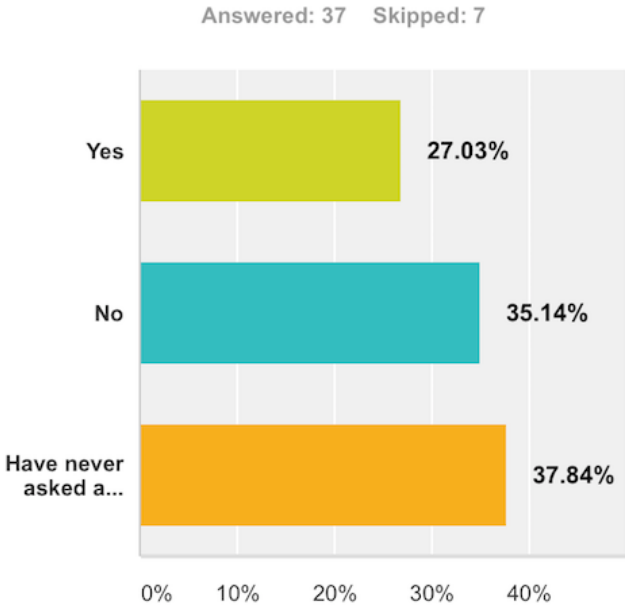
Figure 9. Do you use sites such as stackoverflow.com?



Given that formal training is not a priority (or possibility) for survey respondents and SO (and similar sites) appear to be extremely important, it seems that those people answering questions need to be aware that they are taking the role of trainer to some extent.

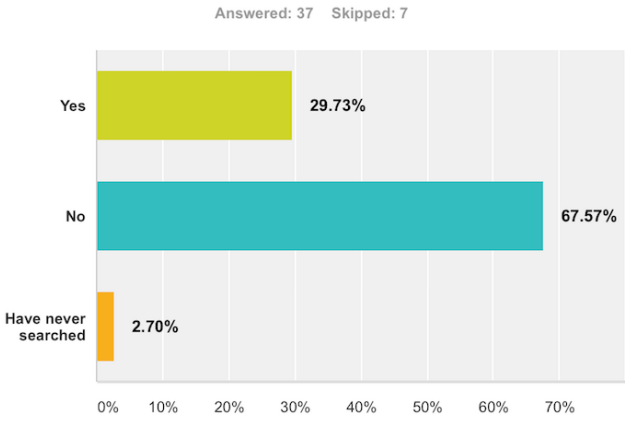
However, the survey responses seem to indicate that the identity and reliability of the individual answering the question may not be particularly important to those asking questions as more respondents felt that this was not important than did:

Figure 10. When you ask a question, does the rank of the person answering affect your view of the answer?



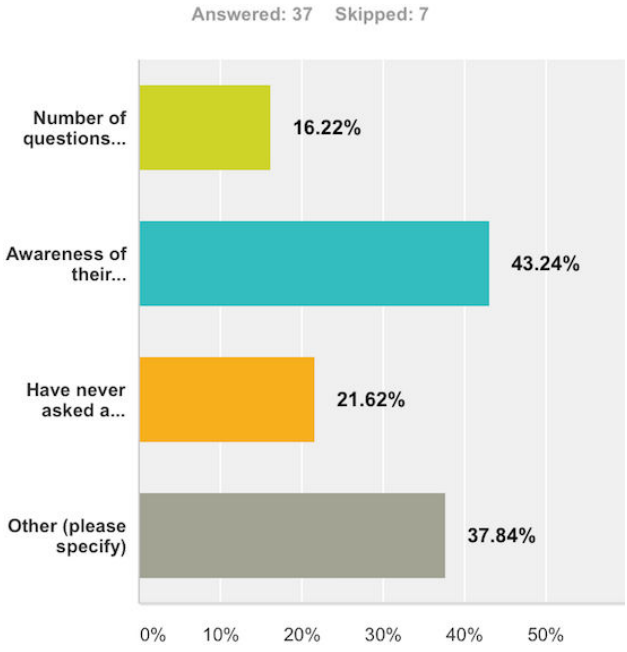
The same appears to be true for searches for answers to existing questions:

Figure 11. When you search existing answers, does the rank of person answering affect your view of the answer?



Finally, we asked how respondents judged the quality of answers:

Figure 12. How do you decide on the validity of answers to your questions?



This gave an interesting range of answers as awareness of an answerer's experience and reputation were ranked highly. The 'other' answers where given related to how well answers could be evaluated based on the questioner's own experience.

4.2. A Crisis of Authority

The training industry and the way that developer's learn have been impacted by the internet in a very similar way to the effect it has had on the traditional news media. One of the criteria for news media is *credibility*. Historically, credibility has been mostly defined by the *authority* of the media. The traditional news media has lost authority due to the easer of publishing on the internet via the web and social media (hence 'fake news').

The ease with which developers can find information on the internet has lead to a similar crisis of authority in the training industry. Credible information can be found online much more easily and (importantly) at a much lower (if not zero) cost compared to traditional technology training. The ability of an organisation to optimise their website for Google has more impact on training than the the credibility of the content on that website. This leads to sites such as W3Schools having a depressingly high level of impact.

SO (and other similar sites such as application specific forums) attempt to create a new measure of credibility based on *reliability*. SO implements this measure of reliability. using the *reputation* measure. The survey results above suggest (given the small number of respondents, it can only be a suggestion) that the ubiquity of SO is as significant as the measure of reliability.

One drawback of the reputation measure is that it does not give an indication of whether or not the person answering the question has specialist knowledge or a large degree of general knowledge (or good search engine skills).

Again, the change in the way that developers study and learn over the last ten years or so has significant implications for the way that questions should be approached.

4.3. Training in the Age of SO

If we look back to the question on SO the led to this investigation, we can see that the accepted answer (mine) was inadequate in many ways. It was not the best answer — it is inefficient at best. Additionally, I gave no explanation at all to the person asking the question.

One of the benefits of traditional training approaches over video based online training and Q&A sites is *context*. Give a three day period and a reasonable number of students, it is possible to apply training to their context. The ongoing communication means trainers can offer a contextual explanation of concepts and ensure that general concepts are understood. Q&A sites such as SO provide a simple way to provide specifci answers to specific questions and the general problem that would benefit the questioner's understanding is lost.

It is simple to find many questions on SO that demonstrate this situation including the one I give above. The drawback to this approach is that answering a specific question without contextual and conceptual information does not improve the knowledge of the person asking the question, it simply solves their immediate problem. There is an increased chance that they will ask a similar question again (or another user will) because the general problem and solution were not addressed.

XSLT is a niche programming language. If we take training and activity on mailing lists and websites as one indication of the health of a programming language's ecosystem and community, then XSLT is not thriving. According to the latest Tiobe language ranking[1], we can see that XSLT is not in the top 100 development languages. Tiobe still track it which does mean that it is in the top 150.

Many of the more popular languages are there due to trend and fashion. However, many of those languages have vibrant user communities helping to drive them forwards. The XML and XSLT community appears to be insular and hermetic to those outside it to some extent (although not to the extent of the Perl community with PerlMonks[2]). Given the popularity of SO, it can be seen by many as an indicator of the health of a language.

4.4. Applying That to the Question

Looking back at the original question on SO, we can see that my answer failed to provide any explanation of how it worked. The `current()` function is useful and often confuses neophytes. I could have used this question as a quick introduction to that concept and to the `preceding-sibling` axis but I failed to.

Martin Honnen chastised me for not mentioning Muenchian Grouping. This was deliberate because I felt that anyone asking such a simple question probably had very little XSLT knowledge. I was wary of referring to Muenchian Grouping. However, I did not explain the solution I provided in any way and this was a failure.

The fact that Jeni's blog on Muenchian Grouping is used as the prime reference is also a failing. As a

[1] https://www.tiobe.com/tiobe-index/

[2] http://perlmonks.org

community we have not created the infrastructure that younger developers expect (xslt.com is registered to a domain squatter) and are referring novice developers to personal blogs for information. In fact, the blog is clear that Jeni was referring to large datasets so this may not have been appropriate for the user's situation. This is where SO fails compared to the Mulberry mailing list — mailing lists provide an environment where asking for additional information works well. The SO comments concept does not achieve that.

The analysis of approaches to solving the user's problem indicates clearly that Muenchian Grouping is the ideal solution to the problem. It also shows that the dataset must be fairly large (we tested a `preceding-sibling` axis up to 1600 elements long) before the approach to the problem even starts to have a significant impact on the speed (if not efficiency) of the solution. Whilst the suggestion that Muenchian Grouping would be the best is clearly not an urban legend, it isn't a panacea either and simpler solutions would have worked.

If a novice asks a question, good teaching practice suggests that a simple solution is often the best first solution to give. More effective solutions can and often should be given but automatically reaching for complex solutions does not necessarily help the questioner. Answers which provide complex solutions without explanation enhance the reputation of the language for being impenetrable and complex. It is not unusual to encounter developers who still feel that writing an application in Java or C# that uses the DOM to manipulation content is *simpler* than writing an little XSLT.

The best answers on SO provide explanation along with answers. Too often, we fail to do that. When answering this question on SO, I omitted everything that makes it a good answer.

5. Summary

A short answer to a question on SO has led to a lot of thought and experiment. I believe that SO and its peers are valuable opportunities for the XML and XSLT community to provide ad hoc training to novice developers. Novice developers with good experience of XSLT may become journeymen and valued members. of the community.

This suggests that those members of the community who answer questions on SO (and to a lesser extent, the Mulberry mailing list) should consider that they are not just answering questions but that they are also, much of the time, teaching.

Good teachers understand the context of the question. Most of those who answer XSLT questions on SO ask questions in the form of comments. That is the maximum extent that it allows but it is essential.

It is important to give context and explanation in an answer. An answer that provides a short snippet of XPath or XSLT with no explanation might well solve a user's problem but it does not add to the sum of knowledge held in SO (and thus, easily found via search engines). If an answer has context and explanation, it becomes much more likely that another user will be able to use that answer for a similar problem and the issue of duplicate or near duplicate questions is diminished.

The online experience of a language is defined to a great extent by the infrastructure and community provided by sites such as Stack Overflow, FAQs and mailing lists. These must be treasured and properly curated in an era where minimal or unchanging online presence is interpreted as death.

A. generator.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:corbas="http://www.corbas.co.uk/ns/xsl/functions"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
  exclude-result-prefixes="xs math xd" expand-text="yes" version="3.0">
  <xd:doc scope="stylesheet">
    <xd:desc>
      <xd:p><xd:b>Created on:</xd:b> Apr 30, 2017</xd:p>
      <xd:p><xd:b>Author:</xd:b> nicg</xd:p>
      <xd:p>Generate randomised sets of elements for use in testing</xd:p>
```

```
  </xd:desc>
</xd:doc>

<xsl:param name="max-uniques" as="xs:integer" select="1000"/>
<xsl:param name="min-uniques" as="xs:integer" select="5"/>
<xsl:param name="max-dups-percentage" as="xs:integer" select="50"/>
<xsl:param name="max-dups" as="xs:integer" select="20"/>
<xsl:param name="variants" as="xs:integer" select="5"/>

<xsl:variable name="uniques" as="element(*)*">
  <xsl:for-each select="1 to $max-uniques">
    <xsl:element name="command">
      <xsl:attribute name="name" select="concat('command', .)"/>
    </xsl:element>
  </xsl:for-each>
</xsl:variable>

<xsl:template name="main">

  <xsl:for-each select="$min-uniques to $max-uniques">

    <xsl:variable name="current-uniques" select="xs:integer(.)"
      as="xs:integer"/>

    <xsl:if test="$current-uniques lt 100 or $current-uniques mod 10 eq 0">
      <xsl:variable name="current-uniques" as="xs:integer" select="."/>

      <xsl:call-template name="unique-set">
        <xsl:with-param name="unique-count" select="$current-uniques"/>
        <xsl:with-param name="generator"
          select="random-number-generator($current-uniques)"/>
      </xsl:call-template>
    </xsl:if>

  </xsl:for-each>

</xsl:template>

<xsl:template name="unique-set">

  <xsl:param name="generator" as="map(xs:string, item())"/>
  <xsl:param name="unique-count" as="xs:integer"/>

  <xsl:call-template name="build-doc">
    <xsl:with-param name="generator" select="$generator?next()"/>
    <xsl:with-param name="dup-counts"
      select="corbas:dup-counts(
                $unique-count, $max-dups-percentage, $max-dups)"/>
    <xsl:with-param name="unique-count" select="$unique-count"/>
  </xsl:call-template>

</xsl:template>
```

```
<xsl:template name="build-doc">

  <xsl:param name="generator" as="map(xs:string, item())"/>
  <xsl:param name="unique-count" as="xs:integer"/>
  <xsl:param name="dup-counts" as="xs:integer*"/>

  <xsl:call-template name="build-doc-variant">
    <xsl:with-param name="unique-count" select="$unique-count"/>
    <xsl:with-param name="generator" select="$generator?next()"/>
    <xsl:with-param name="variant-count" select="$variants"/>
    <xsl:with-param name="dup-counts" select="$dup-counts"/>

  </xsl:call-template>

</xsl:template>

<xsl:template name="build-doc-variant">

  <xsl:param name="generator" as="map(xs:string, item())"/>
  <xsl:param name="unique-count" as="xs:integer"/>
  <xsl:param name="dup-counts" as="xs:integer*"/>
  <xsl:param name="variant-count" as="xs:integer"/>

  <xsl:if test="not(empty($dup-counts))">

    <xsl:choose>

      <xsl:when test="$variant-count = 0">
        <xsl:call-template name="build-doc-variant">
          <xsl:with-param name="unique-count" select="$unique-count"/>
          <xsl:with-param name="generator" select="$generator?next()"/>
          <xsl:with-param name="variant-count" select="$variants"/>
          <xsl:with-param name="dup-counts" select="tail($dup-counts)"/>
        </xsl:call-template>

      </xsl:when>

      <xsl:otherwise>

        <xsl:message>BUILD-DOC-VARIANT - uniques={$unique-count},
          dups={head($dup-counts)}, variant="{$variant-count}"</xsl:message>


        <xsl:variable name="to-write"
          select="corbas:random-sequence(
                    $unique-count, head($dup-counts), $generator)"/>

        <xsl:call-template name="write-sequence">
          <xsl:with-param name="unique-count" select="$unique-count"/>
          <xsl:with-param name="dup-count" select="head($dup-counts)"/>
          <xsl:with-param name="output-sequence" select="$to-write"/>
```

```
          <xsl:with-param name="variant" select="$variant-count"/>
        </xsl:call-template>

        <xsl:call-template name="build-doc-variant">
          <xsl:with-param name="unique-count" select="$unique-count"/>
          <xsl:with-param name="generator" select="$generator?next()"/>
          <xsl:with-param name="variant-count" select="$variant-count - 1"/>
          <xsl:with-param name="dup-counts" select="$dup-counts"/>
        </xsl:call-template>

      </xsl:otherwise>

    </xsl:choose>

  </xsl:if>

</xsl:template>

<xsl:template name="write-sequence">

  <xsl:param name="output-sequence" as="element(*)*"/>
  <xsl:param name="unique-count" as="xs:integer"/>
  <xsl:param name="dup-count" as="xs:integer"/>
  <xsl:param name="variant" as="xs:integer"/>

  <xsl:variable name="href"
    select="'output/' || format-number($unique-count, '0000') || '/' ||
    format-number($unique-count, '0000') || '-' ||
    format-number($dup-count, '0000') || '-' ||
    format-number($variant, '000') || '.xml'"/>

  <xsl:result-document href="{$href}">
    <root>
      <xsl:sequence select="$output-sequence"/>
    </root>
  </xsl:result-document>

</xsl:template>

<xsl:function name="corbas:dup-counts" as="xs:integer*">

  <xsl:param name="unique-count"/>
  <xsl:param name="max-dup-percentage"/>
  <xsl:param name="max-dups" as="xs:integer"/>

  <!-- if a simple divide up into max-dup-percentage would
       give too many values then redo -->
  <xsl:choose>
    <xsl:when
      test="$unique-count * ($max-dup-percentage div 100) gt $max-dups">
      <xsl:variable name="base"
        select="$unique-count * ($max-dup-percentage div 100)"/>
```

```
        <xsl:variable name="increment" select="$base div $max-dups"
          as="xs:double"/>
        <xsl:message>increment is <xsl:value-of select="$increment"/></xsl:message>
        <xsl:sequence
          select="distinct-values(for $n in (1 to $max-dups)
          return xs:integer(floor($n * $increment)))"/>
      </xsl:when>

      <xsl:otherwise>
        <xsl:sequence
          select="1 to xs:integer(floor($unique-count *
                  ($max-dup-percentage div 100)))"/>
      </xsl:otherwise>
    </xsl:choose>

  </xsl:function>

  <xsl:function name="corbas:dups" as="element(*)*">

    <xsl:param name="unique-count" as="xs:integer"/>
    <xsl:param name="dup-count" as="xs:integer"/>
    <xsl:param name="generator" as=" map(xs:string, item())"/>

    <xsl:sequence
      select="if ($dup-count = 0) then () else (
      subsequence($uniques, 1, $unique-count)[floor($unique-count *
                                              $generator?number) + 1],
      corbas:dups($unique-count,$dup-count - 1, $generator?next()))"/>

  </xsl:function>

  <xsl:function name="corbas:random-sequence" as="element(*)*">

    <xsl:param name="unique-count" as="xs:integer"/>
    <xsl:param name="dup-count" as="xs:integer"/>
    <xsl:param name="generator" as="map(xs:string, item())"/>

    <xsl:variable name="dups"
      select="corbas:dups($unique-count, $dup-count, $generator?next())"
      as="element(*)*"/>
    <xsl:variable name="current-uniques"
      select="subsequence($uniques, 1, $unique-count)"/>

    <xsl:sequence select="$generator?permute(($dups, $current-uniques))"/>

  </xsl:function>

</xsl:stylesheet>
```

Charles Foster (ed.)

XML London 2017
Conference Proceedings

Published by
XML London

103 High Street
Evesham
WR11 4DN
UK

This document was created by transforming original DocBook XML sources into an XHTML document which was subsequently rendered into a PDF.

1st edition

London 2017

ISBN 978-0-9926471-4-8