a conference on XML

Lesser Town Campus
Prague, Czech Republic
June 16-17, 2007

ISBN 978-80-239-9540-4

Contents

General Information

Date

Saturday, June 16th, 2007
Sunday, June 17th, 2007

Location

Lesser Town Campus, Lecture Halls S5, S6
Malostranské náměstí 25, 110 00 Prague 1, Czech Republic

Speakers

Štěpán Bechynský
Geert Bormans
Erik Bruchez
Uche Ogbuji
Lars Oppermann
Oleg Parashchenko
Bryan Rasmussen
Eric van der Vlist
Norman Walsh
Mohamed Zergaoui

Organizing Comitee

Petr Cimprich, *Ginger Alliance / U-Turn Media Group*
Tomáš Kaiser, *University of West Bohemia, Pilsen*
Jaroslav Nešetřil, *Charles University, Prague*
Jirka Kosek, *xmlguru.cz / University of Economics, Prague*
James Fuller, *Webcomposite*

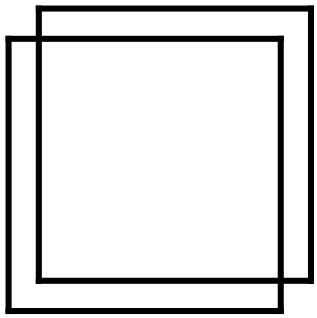
Advisory Board

Eric van der Vlist, *Dyomedea*
Jaroslav Pokorný, *Charles University, Prague*

Proceedings Typesetting

Vít Janota
Jirka Kosek

ISBN 978-80-239-9540-4



I T I

Microsoft®

Preface

This publication contains papers presented at XML Prague 2007.

XML Prague is a conference on XML for developers, markup geeks, information managers, and students. The conference focuses this year on Alternative Approaches to XML Processing.

Experts will be speaking, for the two days, about various techniques for optimal processing of XML, this includes discussion on emerging topics such as pipeline processing of XML and how to work with XML office applications.

This year, all participants are encouraged to participate either through poster session or attending the various BoF sessions that run concurrently throughout the weekend.

The conference is hosted at the Lesser Town Campus of the Faculty of Mathematics and Physics, Charles University, Prague. XML Prague 2007 is jointly organized by the Institute for Theoretical Computer Science and Ginger Alliance in the framework of their cooperation supported by project 1M0545 of the Czech Ministry of Education.

The conference joins the academic world with the world of IT professionals. The organizers hope this year's conference shall be enjoyable for both audience and speakers, providing a dynamic interchange and environment to discuss XML technologies.

This is the third year we organize this event. Information about XML Prague 2005 and 2006 was published in ITI Series 2005-254 and 2006-294 (see `http://iti.mff.cuni.cz/series/`).

Program

Saturday, June 16, 2007 – Lecture Hall S5

9:00	Processing XML With Fun, *Eric van der Vlist*
9:35	XProc: An XML Pipeline Language, *Norman Walsh*
10:35	*Coffee Break*
11:05	Applications of XML pipelines to web applications with XPL, *Erik Bruchez*
12:10	Generative XPath, *Oleg Parashchenko*
12:40	*Lunch Break*
13:55	XML Processing by Streaming, *Mohamed Zergaoui*
15:00	Python and XML, *Uche Ogbuji*
16:00	*Coffee Break*
16:30	XLinq, *Štěpán Bechynský*
17:30	Poster Sessions
17:45	*End of Day*

Sunday, June 17, 2007 – Lecture Hall S5

9:00	Beyond the simple pipeline: managing processes over time, *Geert Bormans*
9:35	DocBook, *Norman Walsh*
10:35	*Coffee Break*
11:05	Open XML Overview, *Štěpán Bechynský*
11:40	Processing OpenDocument, *Lars Oppermann*
12:10	*Lunch Break*
13:25	Leapfrogging microformats with XML, linking, and more, *Uche Ogbuji*
14:30	A Generic Transformation Architecture, *Bryan Rasmussen*
15:00	*End of Day*

Sunday, June 17, 2007 – Lecture Hall S6

9:00	PHP and XML BoF
10:35	*End of Session*

Produced by

Institute for Theoretical Computer Science
(http://iti.mff.cuni.cz/)

Ginger Alliance (http://www.gingerall.com/)

with support of

University of West Bohemia in Pilsen (http://www.zcu.cz/)

General Sponsors

Microsoft (http://www.microsoft.com/)

Sun Microsystems (http://www.sun.com/)

Co-Sponsors

U-Turn Media Group (http://www.u-turnmediagroup.com/)

OASIS (http://www.oasis-open.org/)

Main Media Partner

ROOT.cz (http://root.cz/)

Media Partners

jax*magazine* (http://jaxmag.com/)

oXygen XML editor (http://www.oxygenxml.com/)

<xml>*fr* (http://xmlfr.org/)

Webcomposite (http://www.webcomposite.com)

Processing XML With Fun

Eric van der Vlist (Dyomedea)

Processing XML With Fun

If you find XML processing dull and boring, then you are probably using last century's techniques such as the DOM and this talk is for you.

You will see during the two days of XML Prague 2007 that you have no excuse to process XML without fun and in this presentation I'll do a quick review of the most promising techniques that can save you from the DOM without loosing the power of XML: it can be seen as a road map that gives you the big picture before the following speakers lead you through more detailed areas.

The focus of the talk is on data binding APIs, programming extensions and XMP pipeline languages.

If you find out that you're spending most of your time to compensate impedance mismatches between XML trees and programming objects, you definitely need to have a closer look at data bindings libraries: this is exactly what they can do for you!

The good news is that whatever object oriented programming language you use, you should be able to find such a tool.

If your programming language is statically typed, such as Java, C# or C++, the classes matching the structure of your documents will have to be explicitly defined and they can eventually be generated from a XML schema.

If you're more lightweight and if your language is dynamically typed, you'll have the option to use a binding library that dynamically generate classes for you if they don't exist.

In both cases, using XML with your favorite programming language becomes as easy as using JSON in JavaScript!

If this isn't enough for you and if you believe that XML is so important than XML fragment should be considered as native datatypes, you'll be happy to learn that this is already happening in JavaScript and C# and that similar proposals have been made for other programming languages. If you are using one of these other languages, it's time to bug their authors so that they catch up!

The last category of tools that can liberate you from low level manipulations are XML pipeline languages: they are to XML what make or ant is to source code or what Unix pipes are to programs. You'll find them very handy as soon as you'll need to chain more than a couple of treatments on XML documents.

XProc: An XML Pipeline Language

Norman Walsh (Sun Microsystems, Inc.)

Abstract

Notes on the design and continued progress of XProc: An XML Pipeline Language *currently being developed by the* XML Processing Model Working Group *at the World Wide Web Consortium. Here we'll identify some of the use cases that XProc is designed to address, describe highlights of the current design, and discuss the state of the latest working draft.*

Introduction

One consequence of the tremendous success of XML as a document- and data-interchange format is that a great many applications now perform some sort of XML processing. This processing ranges from simply loading configuration parameters out of an XML file to complex query and transformation as part of the critical functionality of the application.

Another consequence is a great proliferation of XML-related specifications. There are now several validation technologies (DTDs, W3C XML Schemas, RELAX NG grammars, Schematron), several transformation and query technologies (XSLT 1.0, XSLT 2.0, XQuery 1.0), several presentation vocabularies (XHTML, SVG, MathML, XSL-FO), and a suite of related technologies: XInclude, XML Base, xml:id, XML HTTP Requests, Digital Signatures, Encryption, etc. The preceding list is neither complete, nor is any list you can write today likely to be complete tomorrow. The number and variety of XML specifications and technologies is likely to continue to grow for some time.

Many of these technologies can be seen as the building blocks for more complex applications. [2] describes a whole range combinations required by different applications:

- Appling a sequence of operations
- XInclude processing
- Parsing, validating, and transforming
- Aggregating several documents

- Processing single documents
- Processing multiple documents
- Processing subparts of a document
- Extracting MathML
- Styling an XML document in a browser
- Running a custom program
- Generating and applying the results of HTTP requests
- Inserting XML fragments
- Deleting XML fragments
- Wrapping and unwrapping XML fragments
- Exchanging SOAP messages
- Etc.

If you have any experience with XML, you'll recognize that these are hardly new problems. The question of how to link together pieces of XML infrastructure arose before the ink was even dry on the XML Recommendation. In fact, the problem has been tackled many times and in many different ways. The XProc.org[1] site lists more than a dozen such technologies among the Background Materials[2].

XProc is an attempt at standardization. While all of the extant solutions work in their own right, the lack of a standard mechanism means that there's very little portability and interoperability among tools. My Makefile won't work with your Ant script. Your SmallX pipeline isn't compatible with my SXPipe pipeline. Etc.

The development of XProc is motivated by a small set of design goals:

- Standardization, not design by committee
- Able to support a wide variety of components
- Prepared quickly
- Few optional features
- Relatively declarative
- Amenable to streaming

[1] `http://xproc.org/`
[2] `http://xproc.org/#background`

- "The simplest thing that will get the job done."

With that in mind, let's consider pipelines in general and XProc pipelines in particular.

What is a pipeline?

In the words of the specification, [1], an "XML Pipeline specifies a sequence of operations to be performed on a collection of XML input documents. Pipelines take zero or more XML documents as their input and produce zero or more XML documents as their output."

Conceptually, a pipeline is a sequence of steps. Each step takes the output of the preceding step, performs some operation on it, and produces a new output which will be consumed by the step that follows it.

The analogy of boxes connected by physical pipes through which water flows is actually remarkably illustrative. For example, consider the following example from the specification:

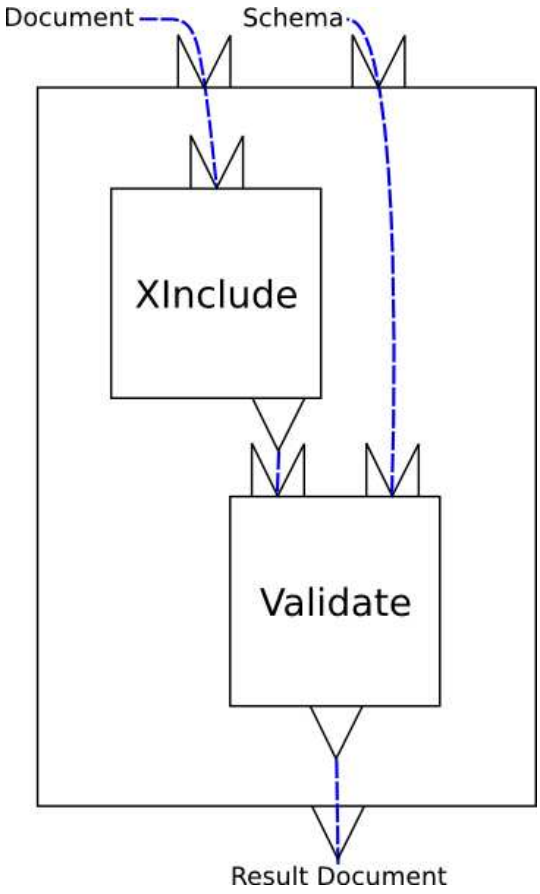


Figure 1: A simple, linear XInclude/Validate pipeline

Viewed from the outside, you have a box with two holes in the top and one in the bottom. Pour water into the top and it comes out "processed" at the bottom. Looking inside the box, we can see how the "Document" and

"Schema" openings at the top of the box are connected to the tops of other boxes and how these other boxes are connected to each other.

In this example, the XInclude and Validate boxes, called "steps" in the pipeline are atomic, they don't have any internal structure, but in general, each box can have an arbitrary number of openings, called "ports", and arbitrary internal structure.

In the context of XProc, the "water" that flows through the pipes are XML documents. Not elements or nodes, not Infosets or PSVIs or XDMs, but XML documents. XProc doesn't impose any constraints on the implementation of connections between the steps except that they be XML documents.

XProc Pipelines

With the caveat that [1] is still a Working Draft and subject to change, here is an XProc pipeline that instantiates the example shown in Figure 1:

```
<p:pipeline name="fig1" xmlns:p="http://www.w3.org/2007/03/xproc">
  <p:input port="source"/>
  <p:input port="schemaDoc" sequence="yes"/>
  <p:output port="result">
    <p:pipe step="valid" port="result"/>
  </p:output>

  <p:xinclude name="include">
    <p:input port="source">
      <p:pipe step="fig1" port="source"/>
    </p:input>
  </p:xinclude>

  <p:validate-xml-schema name="valid">
    <p:input port="source">
      <p:pipe step="include" port="result"/>
    </p:input>
    <p:input port="schema">
      <p:pipe step="fig1" port="schemaDoc"/>
    </p:input>
  </p:validate-xml-schema>
</p:pipeline>
```

Example 1: A simple, linear XInclude/Validate XProc pipeline

The steps in a pipeline can be divided into two broad classes, compound steps and atomic steps. Compound steps, like `p:pipeline` declare their inputs and outputs and contain additional steps which define their semantics. Atomic steps, like `p:xinclude` and `p:validate-xml-schema`, have explicit declarations which identify their inputs and outputs; they cannot contain additional steps.

This pipeline defines two input ports, "source" and "schemaDoc", and

one output port, “result”. The “schemaDoc” port will accept a sequence of documents, the other ports will only accept a single document. What this means is that this pipeline expects two inputs and produces one output, that’s its interface to the outside world.

The body of this pipeline consists of two steps, an XInclude step and a validate (with W3C XML Schema) step.

In order to fully understand these steps, it will be helpful to look at their declarations:

```
<p:declare-step type="p:xinclude">
     <p:input port="source" sequence="no"/>
     <p:output port="result" sequence="no"/>
</p:declare-step>

<p:declare-step type="p:validate-xml-schema">
     <p:input port="source" sequence="no"/>
     <p:input port="schema" sequence="yes"/>
     <p:output port="result" sequence="no"/>
     <p:option name="assert-valid" value="true"/>
     <p:option name="mode" value="strict"/>
</p:declare-step>
```

The XInclude declaration tells us that the XInclude step has exactly one input port, named “source”, and one output, named “result”. The validate component has two inputs, “source” and “schema”, and one output, “result”. (We’ll ignore the options for now.)

The `p:pipe` elements tell us that these steps get their input from other steps. In particular, the `p:xinclude` reads the pipeline’s “source” input and the `p:validate-xml-schema` reads the XInclude’s “result” output and the pipeline’s “schemaDoc”.

The `p:pipe` in the pipeline’s `p:output` tells us that the output of the pipeline is the output of the validate step.

The default semantics of the validate step are that it fails if the document is not valid. Therefore this pipeline either produces the XInclude-expanded, validated document or it fails.

XProc Language Constructs

The XProc language consists of six core language constructs, exposed as compound steps, and a library of atomic steps. In this section, we’ll consider each of the compound steps.

Pipelines

A `p:pipeline` is a wrapper around a user-defined set of steps. This is the root of a pipeline.

For Each

A `p:for-each` iterates over a sequence of documents. Many steps, like the XInclude and validate steps we saw earlier, do not accept a sequence of documents. If you want to apply those steps to all of the documents in a sequence, you can wrap them in a `p:for-each`.

Each output of the `p:for-each` receives the sequence of documents produced by each iteration on that port.

Viewport

A `p:viewport` operates on portions of a document. In other words, it can process each `section` of a large document. The result of the viewport is a copy of the original document with each of the processed sections replaced by the result of the processing applied by the viewport.

Consider this example:

```
<p:pipeline xmlns:p="http://www.w3.org/2007/03/xproc">
  <p:input port="source">
    <p:inline>
      <document>
        <title>Document title</title>
        <section>
          <title>First section title</title>
          <para>Some text</para>
        </section>
        <section>
          <title>Second section title</title>
          <para>Some text</para>
        </section>
      </document>
    </p:inline>
  </p:input>
  <p:output port="result"/>

  <p:viewport match="section">
    <p:output port="result"/>
    <p:xslt>
      <p:input port="stylesheet">
        <p:inline>
          <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                          xmlns="http://www.w3.org/1999/xhtml"
                          version="1.0">
```

```
            <xsl:template match="section">
              <div><xsl:apply-templates/></div>
            </xsl:template>
            <xsl:template match="title">
              <h2><xsl:apply-templates/></h2>
            </xsl:template>
            <xsl:template match="para">
              <p><xsl:apply-templates/></p>
            </xsl:template>
          </xsl:stylesheet>
        </p:inline>
      </p:input>
    </p:xslt>
  </p:viewport>
</p:pipeline>
```

Example 2: XProc Viewport

Here we use inline documents to specify the source (or the default source, in any event) and the stylesheet. We've also use default input bindings to simplify the pipeline. The result is:

```
<?xml version="1.0" encoding="UTF-8"?>
      <document>
<title>Document title</title>
<div xmlns="http://www.w3.org/1999/xhtml">
  <h2>First section title</h2>
  <p>Some text</p>
</div>
<div xmlns="http://www.w3.org/1999/xhtml">
  <h2>Second section title</h2>
  <p>Some text</p>
</div>
      </document>
```

Example 3: XProc Viewport Results

As you can see, each of the `section` elements has been independently transformed and replaced by the transformation.

Choose

The `p:choose` component allows the pipeline author to specify different pipelines (perhaps validating with one schema or another) based on the evaluation of an XPath 1.0 expression.

Try/Catch

The `p:try` and `p:catch` steps provide a level of exception handling. Any error that occurs in the `p:try` branch will be captured and the `p:catch` branch will be evaluated instead. In other words, if no error occurs in the

`p:try` branch, then that is the result of the step otherwise, the `p:catch` branch is evaluated and its output is the result of the step.

Group

The `p:group` element is a generic wrapper for collecting together a set of steps. This is often done so that options and parameters can be calculated once for a set of steps.

XProc Step Library

The step library consists of almost thirty atomic steps that can be used in a pipeline. Implementors may provide additional components and may provide mechanisms that allow others to create thier own extension components.

In addition to steps for many of the standard XML technologies (XInclude, validate, XSLT, etc.), there are a variety of smaller "micro-components" that can perform simple operations such as deleting elements or attributes, inserting content, renaming namespaces, replacing strings, wrapping and unwrapping content, etc.

Although all of these simpler steps can, and often are, implemented using XSLT, the advantage of these simpler steps is that many of them can be streamed.

User-defined Libraries

One of the significant features of XProc is that a `p:pipeline` is itself a step. Pipeline authors can modularize their pipelines by decomposing complex tasks and calling their own pipeline modules.

If, for example, I often processed DocBook documents, I could create a pipeline library with a "DocBook-to-HTML" pipeline:

```
<p:pipeline-library xmlns:p="http://www.w3.org/2007/03/xproc"
    namespace="http://nwalsh.com/xproc/pipelines">

<p:pipeline name="docbook-to-html">
  <p:input port="source"/>
  <p:output port="result"/>

  <p:xinclude/>

  <p:validate-relax-ng>
    <p:input port="schema">
      <p:document href="/home/ndw/docbook.org/xml/5.0CR3/rng/docbook.rng"/>
    </p:input>
  </p:validate-relax-ng>
```

```
  <p:xslt2>
    <p:input port="stylesheet">
      <p:document href="/sourceforge/docbook/xsl2/base/html/docbook.xsl"/>
    </p:input>
  </p:xslt2>
</p:pipeline>

</p:pipeline-library>
```

Example 4: Pipeline Library

If I then wished to process a particular document, say a paper for a conference, I could import that library and use it:

```
<p:pipeline xmlns:p="http://www.w3.org/2007/03/xproc"
    xmlns:px="http://xproc.org/2007/03/xproc/ex"
    xmlns:nw="http://nwalsh.com/xproc/pipelines">
<p:input port="source">
  <p:document href="xproc.xml"/>
</p:input>

<p:import href="library.xml"/>

<nw:docbook-to-html/>

<p:store name="store">
  <p:option name="href" value="xproc.html"/>
</p:store>

<px:tidy name="tidy">
  <p:option name="href" select="/*/@href">
    <p:pipe step="store" port="result"/>
  </p:option>
</px:tidy>

</p:pipeline>
```

Example 5: XML Prague Paper Pipeline

The `px:tidy` step is an example of an extension step. It performs "Tidy" processing on an HTML document. The `p:store` step produces, as it's output, a document which includes the URI where the document was actually stored. The `href` option of the `px:tidy` step uses XPath to extract that value.

State of the art

The design of XProc has come a long way. Our Working Group charter expires in October, 2007. There is every reason to be optimistic that we'll have a completed W3C Recommendation before our charter expires.

Most questions of language design, syntax and semantics, have been settled. Some work remains on the details of the step library. That's one area where user feedback might have a significant influence on the time required to finish. Luckily, I believe we have all the components we need to satisfy the [2].

At the time of this writing, the two largest issues left undecided have to do with handling parameters and dealing with how pipeline state is exposed inside atomic components.

But, with luck and hard work, these questions will be answered before you read this.

References

1. N. Walsh, A. Milowski (eds.). *XProc: An XML PipelineLanguage.* W3C Working Draft, 5 April 2007.
   `http://www.w3.org/TR/xproc/`

2. A. Milowski (ed.). *XML Processing Model Requirements and Use Cases.* W3C Working Draft, 11 April 2006.
   `http://www.w3.org/TR/xproc-requirements`

Applications of XML pipelines to web applications with XPL

Erik Bruchez (Orbeon, Inc.)

Abstract

The XProc XML pipeline language is well on its way to be standardized at W3C. But, exactly, what are XML pipelines good for? And how do they work in practice?

In this talk, we attempt to answer these questions by presenting use cases for XML pipelines implemented with XPL, a close cousin of XProc. We show in particular how XML pipelines fill a niche in the constantly evolving web applications ecosystem. Can XML pipelines help deal with multiple web browsers? With REST services? With the plethora of syndication formats such as RSS and Atom? With Ajax? We suggest that the answer is yes in all these cases.

We also show how XML pipelines can play a particularly interesting role when used in conjunction with XForms.

The talk will feature live demonstrations using open source software.

XPL and XProc

XPL stands for *XML Pipeline Language* . It was developed by Alessandro Vernet and Erik Bruchez at Orbeon in 2002, along with an implementation which became open source in 2004 under LGPL. In 2005, the authors submitted a specification for XPL at W3C. They now participate in the XProc Working Group at W3C, which was created subsequently.

XPL is very close from XProc (as of May 2007). In particular, both languages:

- Have the same goal: performing sequences of operations on XML documents.
- Have an XML-based syntax.
- Support the exchange of XML documents between "steps" such as XSLT, XInclude, and many more.

- Support steps with multiple inputs and outputs.
- Support iterations and conditionals.

XProc possesses features over XPL, including:

- Exception handling.
- Viewports.
- Passing sequences of documents instead of just one document between steps.
- Parameters and options.
- A standard step library.

For most practical purposes, XProc can be seen as a superset of XPL. Most existing XPL pipelines can be transformed into XProc with a simple XSLT stylesheet, provided that the same steps are available on both sides. (This means that the use cases presented in this paper can be easily implemented with XProc.) Conversely, a subset of XProc can be implemented on top of XPL with a similar transformation.

XPL and Orbeon Forms

XPL is at the core of the open source Orbeon Forms platform. Orbeon Forms has evolved over time from a general-purpose XML transformation platform, to a web presentation platform, and finally to a forms solution. In that solution, XPL holds together critical parts of the platform, including:

- Putting together models and views in an MVC page architecture.
- Providing configurable post-processing of resulting documents, including XForms processing.
- Implementing an Ajax server hooking-up to the XForms engine.
- Implementing lightweight REST services that can be called from XForms submissions.

The following sections present these use cases in more details.

XML Pipelines and MVC

A Model-View-Controller (MVC) architecture is one that clearly separates data and presentation. XML pipelines can be used as *controllers* in such an architecture. Consider a pipeline provided with access to an HTTP request and response. In this pipeline:

- A "request" step extracts a portion of the URL path.
- A "choose" step contains one "when" branch per path to process.
- Each branch does the following:
  - Call a "model" sub-pipeline step in charge of obtaining the data to present.
  - Call a "view" sub-pipeline, stylesheet or XHTML+XForms step to format the data obtained by the model.
- A "serialize" step sends the result back to the client as serialized HTML.

Rather than requiring application authors to write controller pipelines, it is possible to define a higher-level description of application components. For example, a page declaration can look as follows:

```
<page id="view-account" path-info="/atm-view-account"
      model="view-account-model.xpl"
      view="view-account-view.xhtml"/>
```

The description is then "compiled" into an actual XPL pipeline (this can be done with XSLT or natively) and the pipeline is executed.

XML Pipelines, Document Types and Client Capabilities

Building on the basic MVC architecture presented above, consider the different tasks that can possibly be performed on a document before actually reaching a web client:

- A pseudo-HTML document (i.e. an XML document representing a plain HTML document in no namespace):
  - Is serialized according to the XSLT 2.0 HTML serialization rules.
  - Is accompanied by a text/html media type.
- Depending on the client, an XHTML document may be processed in different ways:
  - If the client supports XHTML (Firefox), the document:
    - Is serialized according to the XSLT 2.0 XHTML serialization rules.
    - Is accompanied by an application/xhtml+xml media type.
  - If the client doesn't support XHTML (IE), the document:

    * Is moved from the XHTML namespace to the null namespace.
    * Is serialized according to the XSLT 2.0 HTML serialization rules.
    * Is accompanied by a text/html media type.

- An XHTML document containing XForms markup may be processed in different ways:
  - If the client supports XForms, the document is sent as described above for XHTML clients.
  - If the client does not support XForms:
    * The document goes through an "XForms" step that transforms XHTML+XForms into XHTML+JavaScript+CSS.
    * The resulting XHTML document is then processed as described above.
- An XSL-FO document is converted to (for example) PDF and sent as application/pdf.
- An Atom or RSS document:
  - Is serialized according to the XSLT 2.0 XML serialization rules and accompanied with the proper media type.
  - Alternatively, an RSS document may be first converted to Atom with an "XSLT" step (or the other way around).
- An XML document embedding test or binary (Base64) data is serialized as plain text or binary.
- Any other document is simply sent to the browser according to the XSLT 2.0 XML serialization rules.

A single pipeline (which can be called an "epilogue" pipeline) can implement the above with an outer "choose" step testing on the content of the document (typically the root element of the document, but XForms for example requires to look deeper).

Similarly, an epilogue pipeline can also be used to adapt content to portal environments, which require HTML fragments instead of complete HTML documents, using an "XSLT" step to extract the content of an HTML or XHTML body before serializing it.

XML Pipelines and REST

A typical REST service receives XML data from a client browser, and returns back XML (in practice, many variations on the payload are possible,

including using JSON, HTML, or plain text formats). Given rich enough steps, pipelines can be used to implement a wide range of REST services. As an example, this is a simple service which has the task of returning a window of results in a search. It can be expressed this way with a pipeline:

- A "request" step to extract the data submitted by XForms. This can be a POSTed document, URL parameters, etc.
- A "validation" step validates the posted data.
- A "SQL" step calls up a relational database to obtain the search results.
- A "serialize" step to convert the XML result into a stream sent to an HTTP response.

Pipelines can also be used to implement adapter services. For example, assume a REST service which doesn't support XML but returns a plain text format. An adapter pipeline uses, in this order:

- A "request" step to extract the POSTed data.
- An "http" step to call the service with the extracted data.
- An "XSLT 2.0" step with regular expressions to extract the data returned by the service and format an XML response.
- A "serialize" step to convert the XML result into a stream sent by to an HTTP response.

XML Pipelines and Ajax

The server component of an Ajax application is often just a REST service, its only particularity being that it is consumed by a web browser. In some cases, Ajax servers perform more sophisticated tasks that can only be implemented natively. But even in such cases, pipelines can (you guessed it) take care of a lot of the plumbing.

As an example of this, Orbeon Forms has a built-in Ajax server which is the interface between the client-side part of the XForms engine (written in JavaScript and running in the browser) and the server-side part of the engine (written in Java and running on the server). It performs the following tasks:

- A "request" step extracts request headers and HTTP method.
- When the request is an HTTP POST:
  - A "request" step extracts the body of the message and exports it as a temporary URI.

  - A "generator" step dereferences the URI and parses it as XML.
  - The payload is validated with Relax NG.
  - An "XForms server" step receives the XML payload and processes it.
  - The response from the XForms server is validated with Relax NG.
  - A "serializer" step serializes the response from the XForms server and sends it back through HTTP as XML.
- When the request is a pseudo-Ajax submission (used for background uploads in HTML iframes):
  - A "request" step extracts request parameters.
  - An "XSLT" step formats the parameters into an XML document.
  - The payload is validated with Relax NG.
  - An "XForms server" step receives the XML payload and processes it.
  - The response from the XForms server is validated with Relax NG.
  - An "XSLT" step serializes and embed the XML response into a small XHTML document.
  - A "serializer" step serializes the response from the XForms server and sends it back through HTTP as HTML.

Pipelines can also be used to solve the lack of cross-domain support from XMLHttpRequest. A proxy pipeline can:

- Use a "request" step to obtain relevant request data.
- Use an "http" step to forward the step to an external service. Because this runs on the server, there is no cross-context issue.
- Forward the response back to the browser through HTTP.

XML Pipelines and XForms

XForms applications interact with the world mainly through a mechanism of *submissions* (expressed using the `xforms:submission` element) which allow serializing XML instance data and submit the serialization through a particular protocol. XML data can be returned by the protocol as well. There is no limit to the number of protocols and serializations supported by XForms submissions. XForms 1.1 (currently in Last Call) specifies the behavior for the http, https, file, and mailto protocols and provides different serializations including SOAP, but implementations are allowed to support additional protocols and serializations. An implementation could define serializations to and from JSON over HTTP, for example.

XForms 1.1 extends the HTTP and HTTPS support available in XForms 1.0, with the intent to fully support REST interfaces. This allows XForms applications to natively talk with RESTful services that speak XML. However, when services do not directly support REST (or SOAP) with XML, there is a clear impedance mismatch. XML pipelines can play the role of impedance adapters between XForms and non-REST services, and even between XForms and REST services where adapting protocols is desirable.

There are also tasks that are not conveniently done directly in XForms. For example, XForms does not directly support XSLT 2.0, but an XSLT 2.0 transformation can be implemented in an XML pipeline called from an XForms submission.

With XForms, i18n resources and vocabularies (used for example to fill-out selection controls) are often stored externally and loaded into XML instance data during XForms initialization. When these resources are static, they can simply be loaded off a regular web server. When they must come from a database or other locations, again there is an impedance mismatch, which is elegantly solved with pipelines.

Conclusion

The use cases described in this paper show that XML pipelines can play the role of a glue that allows putting together web applications entirely out of XML technologies. The XProc specification makes one more component of this architecture a standard. The last missing piece of the puzzle may well be a standard specification for a web application controller.

Generative XPath

Oleg Parashchenko (Saint-Petersburg State University, Russia)

Abstract

Generative XPath is an XPath 1.0 implementation, which can be adapted to different hierarchical memory structures and different programming languages. It is based on a small, easy to implement virtual machine, which is already available for different platforms, including plain C, Java and .NET.

Introduction

As a consultant in the areas of XML and technical documentation, I often face the task of data transformation. In many cases I can't use convenient tools such as XPath or XSLT (at least, not initially), rather I am limited to the scripting tools of the source platform.

In my projects, the source data is tree-like, and one of the most required of functionalities is navigating over trees. After implementing this functionality several times for different platforms, I noticed I'm writing essentially the same code again and again, with the only difference being the actual programming language and tree data type. There were two problems:

- The code is hard to write, debug and maintain. What is of few characters length in XPath, is several screens of actual code.
- Re-phrasing Greenspun's Tenth Rule of Programming: "Any sufficiently complicated tree navigation library contains an ad hoc informally-specified bug-ridden slow implementation of half of Xpath."

Obviously, it's better to use XPath than custom libraries. But what's if XPath is not available for the platform of choice? Implementing XPath correctly isn't an easy task.

Here is an alternative approach. Generative XPath is an XPath 1.0 processor that can be adapted to different hierarchical memory structures and different programming languages. Customizing Generative XPath to a specific environment is several magnitudes of order easier than implementing XPath from scratch.

The paper is organized as follows. First, I introduce a few use cases for using XPath over hierarchical data. Then the architecture of the Generative XPath approach is explained, followed by a description of the Virtual Machine (VM). The next sections are highly technical and details-oriented, they contain the specification of the interfaces between a program and the VM. The paper continues by highlighting some features of the compiled XPath code. Then we talk about correctness and performance of our system. Finally, we compare Generative XPath with related work and outline further development.

Generative XPath downloads can be found at the XSieve project page: `http://sourceforge.net/projects/xsieve/`.

Use cases

Here is a selection of a few use cases when XPath over something, which is not exactly XML, is possible.

File system

Many XPath tutorials say that the basic XPath syntax is similar to filesystem addressing. Indeed, in both cases we have relative and absolute paths, and the step separator is slash. Representing the file system as XML is a topic of several projects. For example, one of them, FSX [1], was presented by Kaspar Giger and Erik Wilde at the WWW2006 conference.

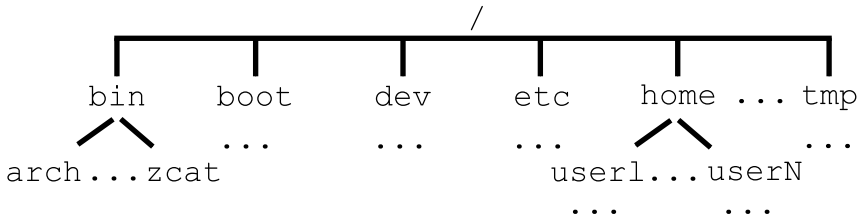


Figure 1: File system as tree

Compilers and interpreters

Compilers and interpreters deal a lot with abstract syntax trees (ASTs), which appear as the result of parsing expressions. A possible AST for the expression `1+b+2*3` is shown on the picture.

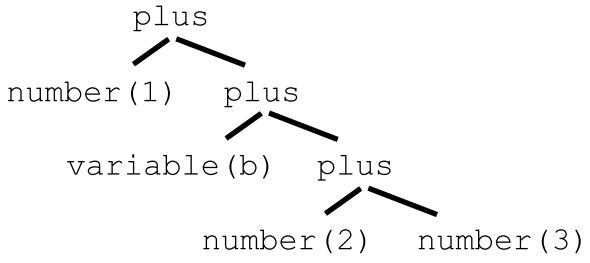


Figure 2: AST for the expression `1+b+2*3'`

Suppose that an optimizer wants to calculate expressions at the compilation stage. One of the possible reductions would be to use “mult” operations when operands are numbers. A corresponding XPath over an AST might look so:

```
//mult[count(number)=count(*)]
```

Text processors

The visual representation of a document introduces an informal structure, with such elements as titles, paragraphs or emphasis.

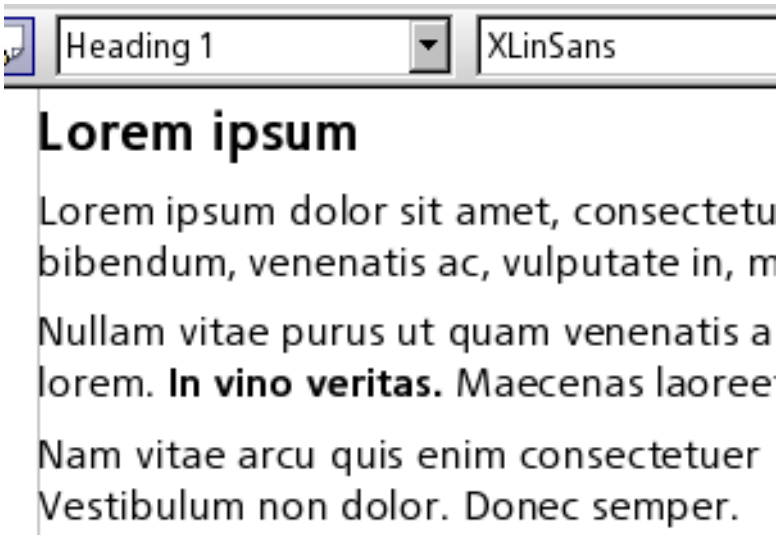


Figure 3: A sample document

The informal structure can be approximated by a tree, in which the nodes are implied by the styles and formatting overrides.

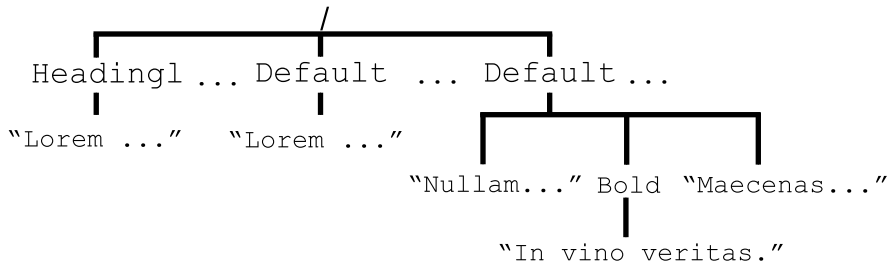


Figure 4: The tree, inferred from the formatting

Now it is easy to get, for example, all the titles of the first level:

```
//Heading1
```

Architecture

The Generative XPath approach consists of two main components:

- XPath compiler
- Runtime environment

The XPath compiler transforms XPath expressions to the executable code for the virtual machine (VM). The runtime environment helps an application to execute the compiled code. The runtime environment can be further divided into three logical layers:

- The application layer
- The VM layer
- The customization layer

The application layer is the program which needs an XPath implementation over its tree-like data structures.

The VM layer is:

- The VM itself
- Compiled XPath code
- Runtime support library

The compiled XPath code relies on the runtime support library, which is also written in the VM language.

The customization layer is an intermediate between the application layer and the VM layer, which are completely independent. In particular, the customization layer maps the application-specific trees to a form, suitable for the VM.

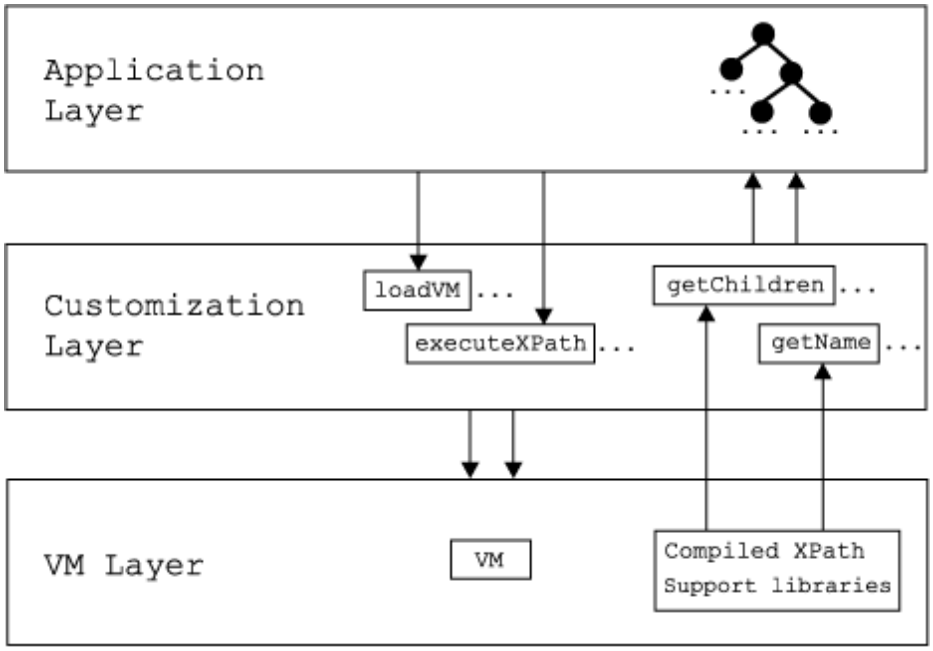


Figure 5: The elements of the Generative XPath architecture

The application layer uses the customization layer to:

- Load and initialize the VM
- Start XPath evaluation and access the result

The VM layer uses the customization layer to:

- Get a collection of nodes by a node (XPath axes)
- Compare nodes in the document order
- Query node properties, such as name or string value

The internal structure of nodes is of no interest to the VM layer, it handles the nodes as "black box" items.

To add XPath support to an application, the developer needs to:

- Find or create a suitable VM implementation
- Implement the customization layer

Virtual Machine

Each VM instruction is represented as a list, in which the first element is the name of the instruction, and the remaining elements are the instruction's arguments. Here is an example of a program to calculate factorial:

```
(define (fac n)
  (if (< n 2)
    1
    (* n (fac (- n 1)))))

(fac 1) ; Evaluates to 1
(fac 6) ; Evaluates to 720
```

This code uses the following instructions. `define` creates a new function named `fac` which takes one argument `n`. The conditional instruction `if` evaluates the first argument, which is the comparison `<`, and either returns 1, or continues recursive execution and returns the result of the multiplication.

The program for the VM is actually a program written in the programming language Scheme R5RS [2]. The VM itself is:

- A subset of Scheme R5RS
- A few extension functions
- Functions defined in the customization layer

There are a lot of Scheme implementations to choose from [3], some of them are tailored for embedding. For example, C applications can use Guile [4], and Java applications can use [5].

As Scheme is a small language, its implementation can be written from scratch with little efforts. A possible approach is explained in Marc Feeley's presentation "The 90 Minute Scheme to C compiler" [6]. Furthermore, the XPath compiler generates code which doesn't use complex features of the language:

- No continuations, dynamic-wind and related features
- Only a few simple macro definitions are used

Compiled XPath code and the support library rely on the extension functions defined in the following SRFI documents [7]:

- SRFI 8: receive: Binding to multiple values
- SRFI 13: String Library
- SRFI 42: Eager Comprehensions

With time, the extension functions will be re-implemented for the VM runtime support library, and the dependencies on SRFIs will be removed.

Interacting with the VM

This section explains how the application layer communicates with the customization layer, and the latter with the VM layer. I will give concrete code examples rather than explaining principles only. As a result, this section is highly technical.

The code is written in C for the Scheme implementation "Guile". A completely working program can be found in the Generative XPath distribution in the folder `examples/c_string`.

Booleans, numbers, strings

To exchange data with the VM, the application and customization layers need to convert data to a format which the VM can understand (hereinafter referenced with the prefix "vm-"). The exact details depend on the VM implementation.

For example, in Guile, the following functions can be used to convert boolean, number and string values to vm-values:

```
SCM gh_bool2scm(int x)
SCM gh_double2scm(double x)
SCM gh_str02scm(const char *s)
```

Having a vm-value, one gets the value using the following Guile functions and macro:

```
int gh_scm2bool(SCM obj)
double gh_scm2double(SCM obj)
SCM_STRING_CHARS(scm)
```

Lists

The Scheme list is a recursive data type, consisting of pairs. For each pair, the first element (usually referred to as `car`) is an item of the list, the second element (`cdr`) is the tail of the list. Having the `car` and the `cdr`, the list can be re-constructed using the function `cons`. The list is finished when the `cdr` is the special value "empty list".

For example, consider the list `(1 2 3)`. Its car is `1`', its `cdr` is the list `(2 3)`. The list can be re-written as `(cons 1 (cons 2 (cons 3 '())))` where `'()` is the empty list.

To create or deconstruct vm-lists, use the VM functions for `car`, `cdr` and `cons`. For example, to construct the list `(1 2 3)` in Guile, the following code is possible:

```
SCM scm = SCM_EOL; // empty list
for (int i = 3; i; i--) {
  scm = gh_cons(gh_int2scm(i), scm);
}
```

And here is an example of walking over a list. For simplicity, we suppose it consists of integers only:

```
void
print_list_of_integers(SCM list) {
  for (
    SCM scm = list;   // start with the first element
    scm != SCM_EOL;   // walk until the end of list
    scm = gh_cdr(scm) // go to the next element
  ) {
    printf("List item: %i\n", gh_scm2int(gh_car(scm)));
  }
}
```

Boxes

There is no such type as "box" is Scheme. It's our extension to represent application's tree nodes. "Box" is a Scheme value, which contains a link to the

tree node and possibly additional information, useful for the customization layer.

In other words, box is a form of tree node that can be used in Scheme code.

Creating such objects for Guile is a slightly cumbersome task. It is required to register a new type together with its garbage collection support functions. The code might look like this:

```
//
// Underlying object for boxes
//
typedef struct _vmbox {
  my_node *node; // pointer to the tree node
  ...            // more data, if required
} vmbox;

//
// Initialization of a new type, attaching the garbage
// collection functions and the destructor.
//
scm_t_bits vmbox_tag;
...

//
// Creating a box
//
SCM
make_box(my_node *node, ...extra...) {
  SCM smob;
  vmbox *box = (vmbox*)scm_gc_malloc(sizeof(vmbox), "box");
  box->node  = node;
  ...extra...
  SCM_NEWSMOB(smob, vmbox_tag, box);
  return smob;
}

//
// Unboxing a node
//
my_node *
unbox(SCM vmbox) {
  scm_assert_smob_type(vmbox_tag, vmbox);
```

```
  vmbox *box = (vmbox*)SCM_SMOB_DATA(scm_box);
  return box->node;
}
```

Details of creating and managing custom types in Guile are described in the Guile manual [8] in the section "Defining New Types (Smobs)".

Boxes-2

The application layer and the VM layer don't care how boxes are created and managed. These details are left to the customization layer. However, here are few guidelines.

The only case when the application layer can create a box from a node is when the node is the root of a tree. All other boxes are created only inside the customization layer.

The only operations on boxes, available to the application layer, are:

- unboxing (extracting the node from a box), and
- applying an XPath.

Initializing the VM layer

The VM layer is initialized with help of the customization layer. The initialization steps are as follows:

- Initialize the VM
- Set up the environment for the VM, such as include paths for libraries
- Load VM extensions, such as SRFI extensions
- Create the box type and register the interface functions
- Load the Generative XPath runtime library

Loading XPath code

The XPath compiler transforms an XPath expression into Scheme code with the free variables `c`, `k` and `n` (the context node, position and size, respectively). Before loading the code, an application should bind these variables, otherwise loading might fail.

The details are not shown here. Instead, an alternate approach is suggested:

- Convert each XPath code to a procedure
- Load all procedures at once

- Locate the procedures in the Scheme environment

Converting XPath code is a simple task, the code should be wrapped this way:

```
(define (xpathNNN c k n)
  ... original code goes here ...
)
```

The code fragment above defines a procedure with the name `xpathNNN` (it's the responsibility of the application to assign unique names) and three context parameters.

Supposing all such definitions are stored in the file `vmcode.scm`, they can all be loaded at once:

```
gh_eval_file("vmcode.scm");
```

Finally, here is how to locate the loaded procedures:

```
SCM  xpathNNN = gh_lookup("xpathNNN");
if (xpathNNN == SCM_UNDEFINED) {
  puts("XPath procedure 'xpathNNN' isn't found");
}
```

Executing loaded XPath

Suppose the variable `xpath` contains a loaded compiled XPath code, wrapped to a procedure as described in the previous section. Before calling the procedure, its arguments should be prepared.

The first argument is a context node in its box form. The application layer gets the box as the result of boxing the root node or as the result of executing another XPath expression.

The second and the third arguments are the context position and context size, respectively. Unless the application uses XPath expressions with the functions `position()` and `last()` at the top level (outside any predicate), these arguments are ignored. Otherwise, they should be vm-integers, and the application should supply the correct values.

Guile example:

```
//
// xpathNNN is an XPath procedure
// box is the context node
// We don't care about context position and size
//
ret = gh_call3(xpath, box, SCM_UNSPECIFIED, SCM_UNSPECIFIED);
```

Interpreting the result

Generative XPath always returns the result as a list. The actual type of the result is interpreted as follows:

- If the list is empty, the result is the empty node set.
- If the type of the first element of the list is boolean, number or string, then this element is the result.
- Otherwise, the result is the node set.

Example of interpreting the result in Guile:

```
if (SCM_EOL == ret) {
  puts("empty node set");
} else {
  SCM node, val = gh_car(ret);
  if (SCM_BOOLP(val)) {
    printf("boolean, %i\n", gh_scm2bool(val));
  } else if (SCM_NUMBERP(val)) {
    printf("number, %f\n", gh_scm2double(val));
  } else if (SCM_STRINGP(val)) {
    printf("string, %s\n", SCM_STRING_CHARS(val));
  } else {
    printf("nodeset:\n");
    for(; SCM_EOL != ret; ret = gh_cdr(ret)) {
      val  = gh_car(ret);
      node = unbox(val);
      print_node(node);
    }
  }
}
```

Interface to the tree

This section described the functions which return information about the application's trees. These functions are the part of the customization layer.

`box gx-ffi:root(box c)`

Input is the context node `c`, the function should return the root node of the tree. Note that the root node and the highest-level element are two different things.

```
list gx-ffi:axis-child(box c, symbol node-test,
string/symbol namespace-uri, string/symbol local-name)
```

And twelve functions with the same input and output parameters: `gx-ffi:axis-descendant`, `gx-ffi:axis-parent`, `gx-ffi:axis-ancestor`, `gx-ffi:axis-following-sibling`, `gx-ffi:axis-preceding-sibling`, `gx-ffi:axis-following`, `gx-ffi:axis-preceding`, `gx-ffi:axis-attribute`, `gx-ffi:axis-namespace`, `gx-ffi:axis-self`, `gx-ffi:axis-descendant-or-self`, `gx-ffi:axis-ancestor-or-self`.

These functions should return the corresponding XPath axes for the node `c`. The nodes in axes are sorted in the document order, except for the four axes in which the nodes are sorted in the reverse document order: `ancestor`, `ancestor-or-self`, `preceding`, `preceding-sibling`.

The symbol `node-test` defines the filter for the node type. The range of the values is: `*`, `text`, `comment`, `processing-instruction`, `node`. The semantics are identical to those defined in the XPath specification. Note that `*` doesn't mean "all the nodes", it means "all the nodes of the axis' principal type".

The strings `namespace-uri` and `local-name` are the filters for the names of the nodes. Alternatively, if these parameters are the symbol `*`, then no filtering by name is required.

```
number/boolean gx-ffi:<=>(box n1, box n2)
```

The function compares the nodes in the document order. It returns:

- `-1`, if the node `n1` comes before the node `n2`,
- `0`, if `n1` and `n2` are the same nodes,
- `1`, if `n1` comes after `n2`,
- `#f`, in case of error.

The function can return `#f` if the nodes are from different documents. But it also can return `-1` or `1`, as long as the result is consistent for all the nodes of these documents.

```
string gx-ffi:string(box n)
string gx-ffi:namespace-uri(box n)
string gx-ffi:local-name(box n)
string gx-ffi:name(box n)
```

These are the counterparts to the XPath functions `string()`, `namespace-uri()`, `local-name()` and `name()`. The special cases such as *no parameters* or *a nodeset as the parameter* are handled by Generative XPath itself. These functions have only one node as the parameter.

```
boolean      gx-ffi:lang(box c, string lang)
node/boolean gx-ffi:id(box c, string id)
```

These are the simplified counterparts to the XPath functions `lang()` and `id()`. The variable `c` is the context node. The function `gx-ffi:id()` should return either a node, or `#f` (false).

XPath compiler

The description of the XPath compiler consist of two parts:

- The tools
- Outline of the internals of the compiler.

Tools

The XPath compiler is a standalone command-line program `ast_to_code.scm` (wrapped by the shell script `run.sh`). The program gets an XPath expression as a command line argument and dumps the compiled XPath code to the standard output.

```
$ ./run.sh '//a/b'
```

The section "Loading XPath code" suggests to wrap each compiled code in a procedure and store the result in a file. The program `bulk.scm` performs exactly this task. Edit the file, adding the required XPath expressions, and run it.

There is also an improved version of the program, `bulk_escaped.scm`, which allows for compilation of XPath expressions with invalid XML names by URI-escaping them. For example, instead of writing

```
/*[name()='C:']/*[name()='Program Files']
```

we can use a more nice expression

```
/C%3A/Program%20Files
```

The compiler is written in Scheme. If an application embeds the VM as a full-featured Scheme implementation, then the application can compile XPath expression in runtime using the library function `gx:xpath-to-code-full`.

Compiler internals

The compiler parses and rewrites XPath expressions using the libraries of the project ssax-sxml [9]. In particular, the first step is to convert XPath to an

abstract syntax tree (AST) using the function `txp:sxpath->ast`. The AST tree is rewritten, bottom-up, to code using SXSLT, a Scheme counterpart of XSLT.

The code is generated as straightforwardly as possible, with no attempts at optimization. It is supposed that the generated code is later processed by an optimizer. Here is an example of the code generated for `1+2`:

```
(gx:unit
  (+ (gx:number (gx:sidoaed (gx:unit 1)))
     (gx:number (gx:sidoaed (gx:unit 2))))))
```

In our approach, each XPath step should return a sequence, therefore we wrap atomic values to lists using the function `gx:unit`. When processing `+`, the compiler doesn't look down the AST tree in search for optimization, but generates the common code. First, the arguments are sorted in the document order ("sidoaed" stands for "sort in document order and eliminate duplicates"), and than converted to numbers. The code is suboptimal, but, fortunately, easy to optimize.

The majority of the XPath core functions are implemented in the runtime support library. For the remainder, the library takes care of the special cases, defined in the XPath specification, and calls the customization layer only if required. For example, here is the code of the function `gx:string`:

```
(define (gx:string nset)
  (if (null? nset)
    ""
    (let ((val (car nset))) (cond
      ((boolean? val) (if val "true" "false"))
      ((number?  val) (cond
                ... NaN/infinity/integer cases ...
                (else (number->string val))))
      ((string?  val) val)
      (else           (gx-ffi:string val))))))
```

It checks, in order, if the argument is the empty node set, or boolean, number or string value, and acts accordingly. Otherwise, the argument is a node set, sorted in the document order, and the result is the string value of the first node, as returned by the customization layer.

Compilation of XPath step sequences is non-trivial. The syntactic form for it is called "list comprehension". The expression `step1/step2` is compiled to:

```
(list-ec
    (:list c   ... step1 code ...)
    (:list c2  ... step2 code ...)
  c2))
```

The literal interpretation is as expected: execute `step1`, bind the variable `c` to each node of the result, execute `step2` and bind the variable `c2` to each node of the result. The whole result of `list-ec` is the list of the values of `c2`.

A similar construction is used to represent filtering. The only added complexity is preprocessing the input list and annotating the nodes with the context position and size, and unpacking the annotations before evaluating the predicate.

The main feature of the generated code (and the support library) is that it is recursion-free. Only higher-order constructions, such as list comprehension and morphisms, are used. As result, it is possible to substitute all the function calls by the bodies of the functions and get one big code block. I believe it allows to perform aggressive optimization of the generated code, and it's the topic of further investigations.

Compliance and performance

Correct implementation of the XPath 1.0 standard is the main constraint for the Generative XPath project. To reach this goal, I use:

- A set of unit tests
- Testing with real world stylesheets

For the latter, I use XSieve [10], a mix of XSLT and Scheme. Initially, many XSLT constructions are converted to the XSieve form. For example,

```
<xsl:apply-templates select=@*|node() >
```

becomes

```
<s:scheme>
  (x:apply-templates (x:eval @*|node()))
</s:scheme>
```

Secondly, I extract all the `x:eval` expressions and replace them with the corresponding compiled code. As a result, most of the XPath selects are performed by Generative XPath.

Now we can run the original stylesheet, then the Generative XPath version of the stylesheet, and compare the results. They should be the same. I executed this test for DocBook stylesheets, and the test was passed successfully.

Therefore, with a high degree of confidence, Generative XPath is a compliant implementation of XPath.

Performance is still a topic of further work. The generated code is not optimized for speed, instead, it is in a form suitable for analysis by an optimizer, which is under construction.

In the worst case, in an unfair setup and measurement, Generative XPath is 30 times slower than a libxml2 implementation of XPath. However, I expect that speed is comparable in the case of fair conditions, and Generative XPath might be faster when the optimizer is implemented.

Related work

The majority of XSLT/XQuery processors somehow allow custom trees in theirs XPath engines, but this is mostly a side-effect, not intended functionality. I'm aware of a few projects, in which virtualization of XML is the main selling point:

- Jaxen [11] (Java),
- JXPath [12] (Java) from Apache Software Foundation,
- XML Virtual Garden [13] (Java) from IBM,
- `IXPathNavigable` interface [14] (.NET) from Microsoft,
- XLinq [15] (.NET) from Microsoft.

The main limitation of these projects is that they are bound to the platform of choice. For example, we can't use Jaxen if we develop in .NET. Worse, we can't use any of the mentioned tools if we are programming in plain C.

Unlike the above projects, Generative XPath is highly portable as it is based on a small, easy to implement virtual machine.

Conclusion and further work

Generative XPath is an XPath 1.0 processor, which can be adapted to different hierarchical memory structures and different programming languages. Generative XPath consists of the compiler and the runtime environment. The latter is based on a simple virtual machine, which is a subset of the programming language Scheme.

In this paper, we've explained why XPath over arbitrary tree-like structures is useful at all, described the architecture, interfaces, tools and internals of Generative XPath, and given links to the related developments.

The next step for the Generative XPath project is the development of an optimizer. The corresponding announcements and downloads will be available from the XSieve project page: `http://sourceforge.net/projects/xsieve/`.

References

1. K Giger, E Wilde. *XPath filename expansion in a Unix shell*, in Proceedings of the 15th international conference on World Wide Web, 2006. http://www2006.org/programme/files/xhtml/p95/pp095-wilde.html
2. R. Kelsey, W. Clinger, J. Rees (eds.). *Revised5 Report on the Algorithmic Language Scheme.* Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998. http://www.brics.dk/~hosc/11-1/
3. Schemers.org. *Implementations.* http://schemers.org/Implementations/
4. Free Software Foundation, Inc. *Guile (About Guile).* http://www.gnu.org/software/guile/guile.html
5. S. G. Miller. *Second Interpreter of Scheme Code.* http://sisc-scheme.org/
6. M. Feeley. *The 90 Minute Scheme to C compiler.* http://www.iro.umontreal.ca/~boucherd/mslug/meetings/20041020/minutes-en.html
7. The SRFI Editors. *Scheme Requests for Implementation.* http://srfi.schemers.org/
8. Free Software Foundation. *GNU Guile Manual.* http://www.gnu.org/software/guile/manual/
9. O. Kiselyov. *SXML Specification.* http://okmij.org/ftp/Scheme/xml.html#SXML-spec
10. O. Paraschenko. *XSieve book.* http://xsieve.sourceforge.net/
11. Apache Software Foundation. *JXPath – JXPath Home.* http://jakarta.apache.org/commons/jxpath/index.html
12. jaxen. *universal Java XPath engine – jaxen.* http://jaxen.org/
13. IBM alphaWorks. *Virtual XML Garden.* http://www.alphaworks.ibm.com/tech/virtualxml
14. Microsoft Corporation. *XPathNavigator in the .NET Framework.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconXPathNavigatorOverDifferentStores.asp
15. E. Meijer, B. Beckman. *XLINQ: XML Programming Refactored (The Return Of The Monoids)*, in Proceedings of XML 2005, November 2005. http://research.microsoft.com/~emeijer/Papers/XMLRefactored.html

XML Processing by Streaming

Mohamed Zergaoui (Innovimax)

Abstract

The first part will be to present the state of the art of XML Streaming processing by reviewing the products in place (joost, cocoon, saxon, etc.), the API available (SAX, Stax, XOM), languages (CDuce, XDuce, XJ), and the spec in progress or stalled (STX, XML Processing, XQuery update). Speaking of what is currently in preparation (i.e. an XML Streaming XG at W3C). And taking the time to present what has already been done in SGML time (Balise and Omnimark, cursor idea that can be find in Arbortext OID in ACL, etc.)

Then the goal is to present all the area where some work has still to be done and give some hints on an elaborated vision of XML Processing trough different kind of process: around constraints, normalizing, streamable path, multilayer transformation, and last but not least constraints aware streamable path. Some light will be spot on static analysis of XSLT and XQuery to detect streamable instances. What are the needed evolutions of the cursor model? What are XDuce-like languages added values?

Python and XML

Uche Ogbuji (Zepheira, LLC)

Python/XML libraries

Python is a popular language for general-purpose development, including Web development, but it has not always had the coziest relationship with XML. There is a history of often unnecessary philosophical differences between boosters of Python and of XML. Partly because of this the state of the art has been unfortunately slow to develop, despite the work of many open-source developers in the XML-SIG and elsewhere. At present there are several options for XML processing in Python. Because of Python's flexibility and the breadth of library and tool options, it can be a very productive language for XML projects.

The most popular libraries for general-purpose XML processing in Python are:

- PySAX (python.sax in the standard library)
- PyDOM (python.dom in the standard library)
- ElementTree (in the standard library since Python 2.5)
- lxml
- 4Suite+Amara

PySAX and PyDOM are standard SAX and DOM (with some minor language-specific variations to fit the language). ElementTree is a tree data structure optimized for speed, space, with an API that emphasizes simple use of Python conventions. lxml is a Python wrapper around the popular and very high performance libxml2 and libxslt2, written in C. 4Suite is a library that provides a broad range of XML processing capabilities, but at some expense in friendliness for Python developers. Amara XML toolkit is a 4Suite add-on that adds a very Python-friendly layer, as well as some additional processing facilities. 4Suite and Amara are so often used in tandem that I treat them as a joint library in this discussion.

Two key APIs to compare for XML processing are ElementTree's and Amara's. Both are designed to emphasize Python-friendly idiom, and are tree APIs. The basic species of tree API defines a standard object structure to represent the parts and pieces of XML. You can think of this as defining a class and naming convention to correspond to each type of information item in the XML Infoset spec. All elements are represented using the same classes and object reference names. All attributes are represented using the same classes and object reference names, different from that of elements, and so on for text and, if supported, processing instructions, comments, etc. As such, these can be considered approximate translations of the Infoset, and abstract model, to a Python representation. Accordingly I call this species of tree APIs Python XML Infosets, of which the foremost example is ElementTree.

Amara, on the other hand, is a data binding, a system for viewing XML documents as databases or programming language or data structures, and vice versa. There are many aspects of data bindings, including rules for converting XML into specialized Python data structures, and the reverse (marshalling and unmarshalling), using schemata to provide hints and intended data constructs to marshalling and unmarshalling systems, mapping XML data patterns to Python functions, and controlling Python data structures with native XML technologies such as XPath and XSLT patterns. A data binding essentially serves as a very pythonic API, but in this paper, the main distinction made in calling a system a data binding lies in the basics of marshalling and unmarshalling. In data bindings, the very object structure of the resulting Python data structure is set by the XML vocabulary. The object reference names come from the XML vocabulary as well. Data bindings in effect hide the XML Infoset in the API, in contrast to Python XML Infosets. They are based on the model of the XML instance, rather than of the Infoset.

Infosets and data bindings effectively lead to different idioms and conventions in XML processing, but they provide for more expressive and maintainable code than SAX and DOM.

XLinq

Štěpán Bechynský (Microsoft Corporation)

Abstract

.NET Language Integrated Query for XML Data

There are two major perspectives for thinking about and understanding XLinq. From one perspective you can think of XLinq as a member of the LINQ Project family of technologies with XLinq providing an XML Language Integrated Query capability along with a consistent query experience for objects, relational database (DLinq), and other data access technologies as they become LINQ-enabled. From a another perspective you can think of XLinq as a full feature in-memory XML programming API comparable to a modernized, redesigned Document Object Model (DOM) XML Programming API plus a few key features from XPath and XSLT.

XLinq represents a new, modernized in-memory XML Programming API. XLinq was designed to be a cleaner, modernized API, as well as fast and lightweight. XLinq uses modern language features (e.g., generics and nullable types) and diverges from the DOM programming model with a variety of innovations to simplify programming against XML. Even without Language Integrated Query capabilities XLinq represents a significant stride forward for XML programming.

Beyond the simple pipeline: managing processes over time

Geert Bormans (Independent Consultatnt)

Abstract

This paper shows an application that takes XML pipeline processing to a next level. The paper shows the first results of a project that requires version management of pipeline components. As will be shown, the approach taken in this project is generally applicable for building complex XML processing systems with a lesser risk.

Introducing the Project

A company sends printed statements to customers on a yearly basis[1]. This company has a legal obligation to archive an exact copy of each statement that was sent to customers.

The original application (see Figure 1) has a complex monolithic code-base, taking care of the following:

- Information is extracted from two relational databases, using complex SQL queries.
- Based on numbers in the query results, a lookup table and some complex calculus, some numbers are added.
- Header and footer information is added. This header and footer information changes yearly. Because of this change, the code is changed on a yearly basis.
- All this information is eventually distilled into a PS document, that gets printed and then sent to the customer.

[1]For confidentially reasons, some details of the project were changed from reality. Some irrelevant details have been removed from the total setup. The context remains practically the same though.

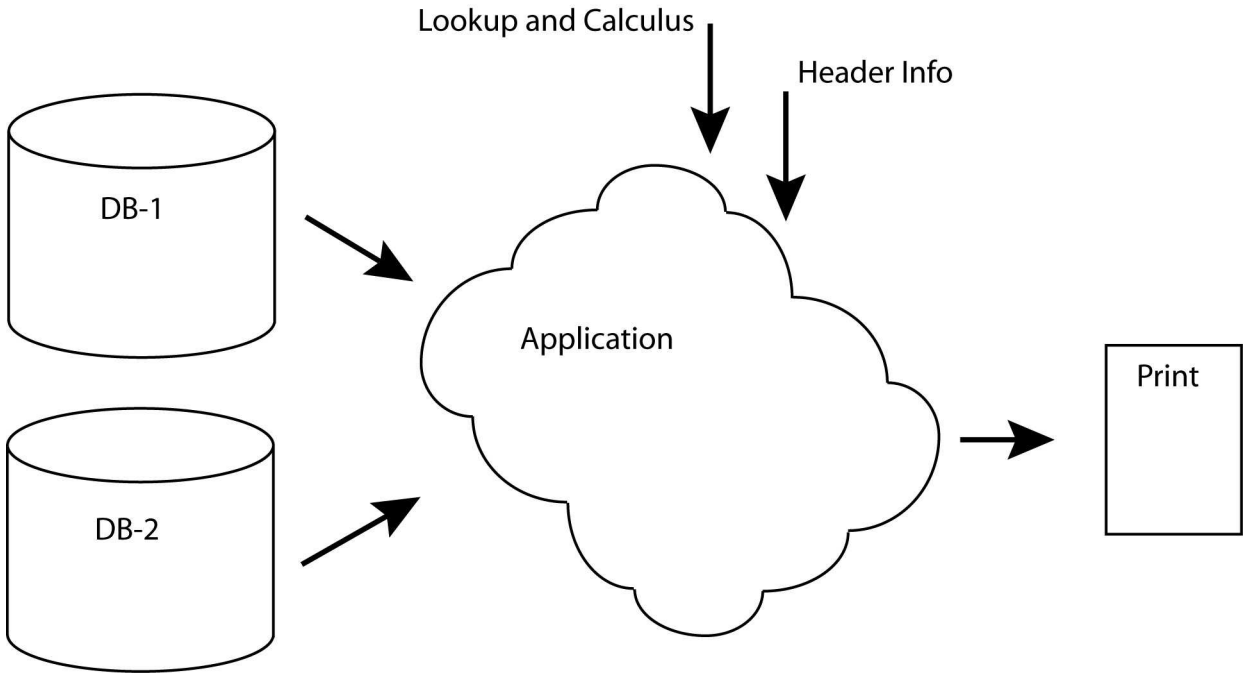


Figure 1: Schematics of the original application

This company originally planned to commit to their legal obligation, by printing all the statements to PDF and store the PDF documents in a Content Management System or an Asset Store (see Figure 2).

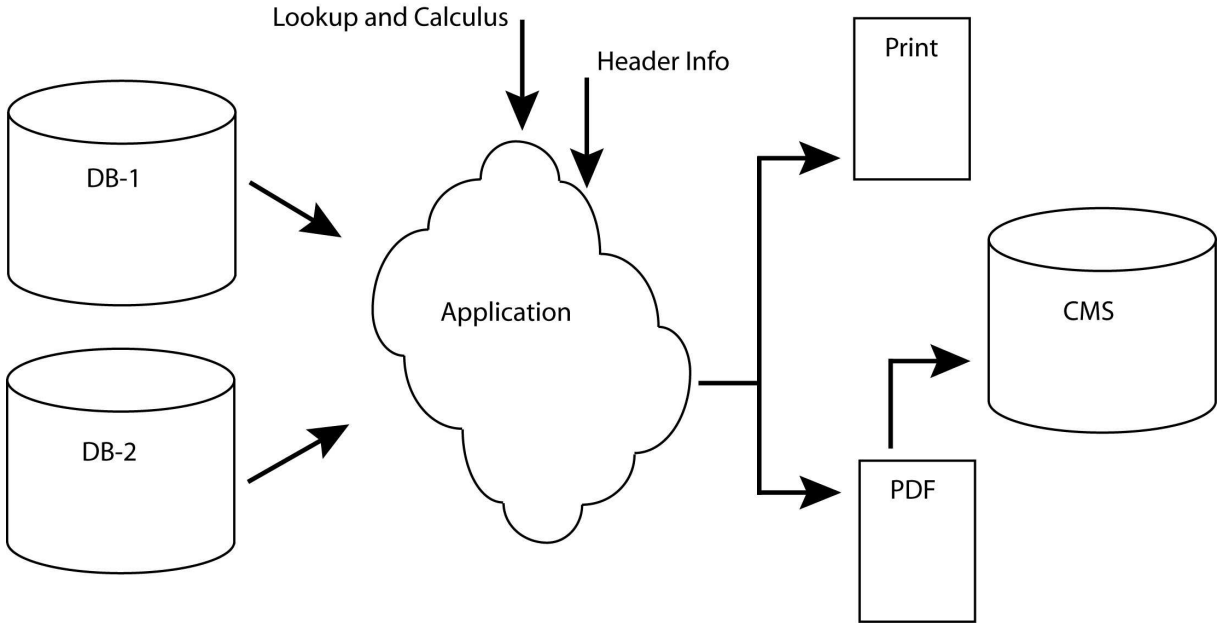


Figure 2: Schematics of the application as detailed in the original Request for Proposal

Introducing the Solution

The original Request for Proposal described not much more than an asset management system to fulfil the legal obligation to maintain a (scanned) copy of all the statements sent out.

Storing the scanned statements or a PDF version of the statements would require an awful amount of disk space.

Eventually they realised it would be better to archive the XML source document of the different statements, as extracted from the databases. Instead of version managing the resulting PDF documents, it was decided to version manage the publication and rendering processes with the XML, ready for execution.

The customer bought into a complete redesign of the monolithic program that is currently creating the PDF. This program has been broken up into smaller processing steps in a pipeline (Figure 3).

In this new application, the individual statement information is dumped in an XML document. This XML document is stored in a Content Management System (with some pretty simple meta data).

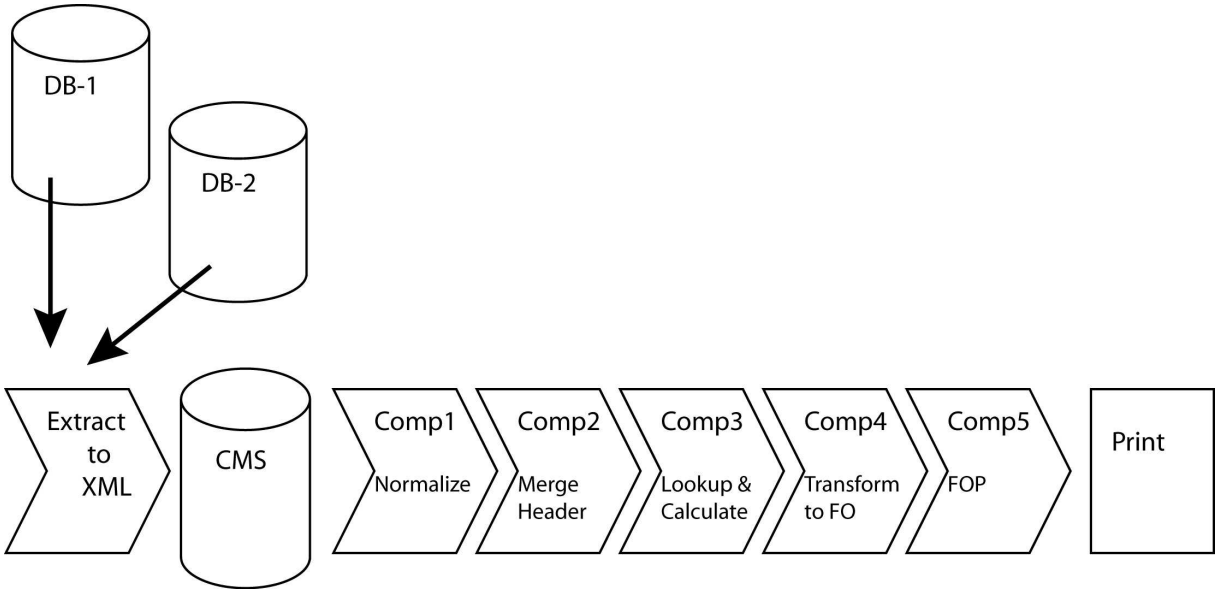


Figure 3: Pipeline replacement of monolithic application

For printing, the following sequence of processing steps (components) is activated on each XML document

- Comp1: normalization of the XML (XSLT)
- Comp2: merge the header and footer information in (XSLT)
- Comp3: find some values in a lookup table based on numbers in the document, and do some complex calculus (Python)
- Comp4: transform the resulting XML into XSL-FO (XSLT)
- Comp5: render the XSL-FO into PDF for print (FOP)

The process steps (components) in the pipeline are themselves version managed in some sort of light CMS. The underlying architecture knows how to

construct pipelines from the correct version of the components and execute them.

XML statements have metadata associated for binding them to a specific version of the PDF generation process (pipeline).

When in a new year some aspects of the PDF must be changed (e.g. the header, the layout or the structure of the statement) the component involved will be updated. This will result in a version update of the pipeline (see Figure 4).

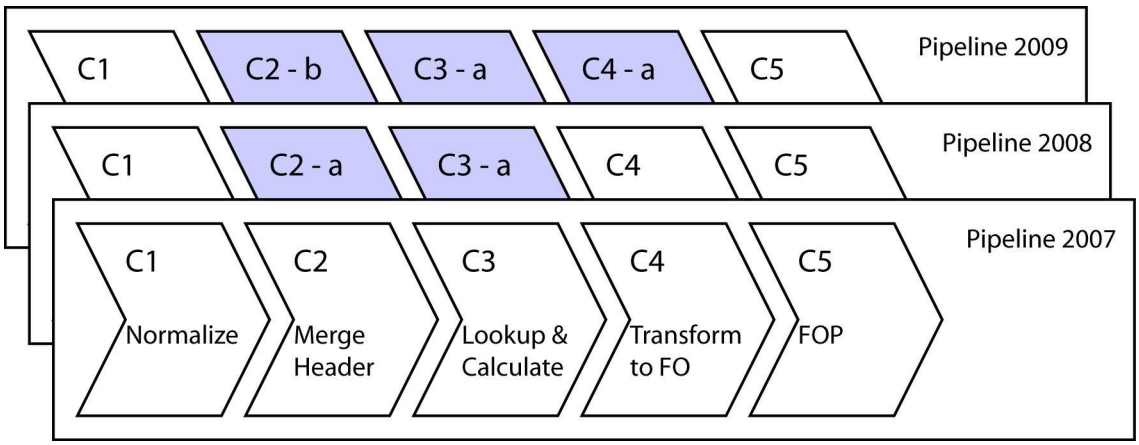


Figure 4: Different versions of the pipeline, sharing components

New statements in the CMS will automatically be tied to this new version of the pipeline. The execution engine will know which pipeline to execute on a particular statement. At any time in the future, it will be possible to instantly reconstruct the PDF that was used for the original printed statements, exactly as it was at the time of printing, even if that was quite some time ago.

What is it that is so good about Pipelines?

In the context of this paper, a pipeline or content processing pipeline is a sequence of content processing steps. In a pipeline, a complex content processing application is broken up into several more simple processes. The content or data flows from one processing step (also component) to the next. The output of one component is the input of the next.

This section summarizes some of the commonly understood advantages of pipelines:

- By breaking up the complexity of the original program into adequately chosen process steps, the total application becomes a lot simpler to develop and easier to debug and maintain.
- By having a chain of small and simple programs with a well defined input and output, it is a lot easier to assign responsibilities. Development

of components can be assigned to different people with different roles. Communication between actors can be kept to a minimum. Basically actors just have to agree how the information they can accept has to look like and what will be the structure of the information they will create.

- It is possible (depending on the underlying architecture) to develop components using the best possible technology to get this particular job done. Rather than requiring a set of JavaScript extension functions in an XSLT, one could put the JavaScript work in one component, and the XSLT in the next.
- Components can be (re)used between different pipelines (or in this case between different versions of the same pipeline) One could call this "Single Sourcing" for process components.

Challenges for building Pipeline Solutions

Pipelines for XML processing, or content processing in general, are gaining popularity. There are a number of frameworks around that help you to set up XML processing pipelines and execute them. This section summarizes some of the challenges met when developing the above described system.

- Though almost too obvious to mention, the execution of the pipeline itself is some sort of a challenge. The underlying infrastructure needs to be able to fit components together, compile a pipeline and execute it, in a robust way, with high performance.
- Intermediate serialization / de-serialization of input and output between components can put an unnecessary burden on performance if done without a reason. Passing the result from one process as the input to the next process without creating too much of timing overhead is a next interesting challenge.
- Preferably one has the freedom to develop each and every component in a technology most adequate for the work to be done. One could imagine developing some components using a scripting language such as Ruby, Python or JavaScript and other components using XSLT or XQuery. Providing this functionality, taking into account the first two requirements, forms another interesting challenge.
- The language used to describe the pipeline should be easy to learn. Ideally some sort of Graphical User Interface is available for adding or changing components to a pipeline. Constructing pipelines should be an easy job, not necessarily performed by developers. Executing the pipeline should

be fast. There should be some sort of abstraction layer that separates the execution from the pipeline "editing". That is our final challenge.

Pipeline Management

In a way, one could see pipelines as normalization or structuring of content processing. This is, in a way, very similar as to how XML has been normalizing or structuring this content itself. Much in the same way as people started maintaining their XML documents in Content Management Systems for versioning, collaboration, etc. it seems very natural to see a need for managing these pipelines in some sort of Pipeline Management System.

One of the factors that add complexity to any software system is its variation in time. This dimension of time, gave the design of the pipeline solution for this project yet another interesting challenge.

The Solutions Architecture

At the heart of the solution sits a Pipeline Component Management System (PCMS) and a Pipeline Execution Engine (Figure 5).

The processing steps in the pipelines are all defined as URI[2]-addressable resources. The PCMS pulls all these resources together at run time, constructs a pipeline and presents that to the Execution Engine for execution as requested.

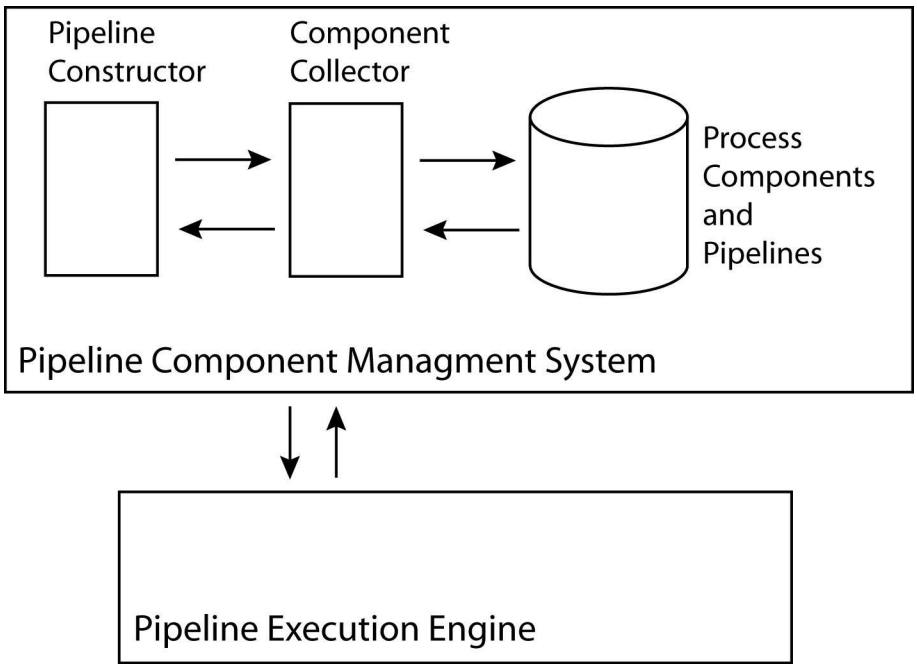


Figure 5: Schematic of the core of the projects solution

[2]Uniform Resource Identifier

The data-model of the management system allows version management on the pipeline component level. Components can be used across multiple pipelines, or versions of pipelines.

Pipelines are dynamically constructed from pulling components together at runtime.

Here is how the application deals with the different challenges as mentioned above.

- The Pipeline Execution Engine is a thin layer on top of NetKernel[3] (see below). The pipeline execution is taken care of by NetKernel. NetKernel has a smart caching mechanism, which keeps the performance up.
- NetKernel has a notion of aspects and transreptors. The result of a process is kept in the form it was created in (for XML, this could be a DOM-aspect, a SAX-aspect, a serialized string or even proprietary ones such as Saxon[4] trees). Only if the next step requires a different input aspect, the aspect will be transformed (transrepted). This helps keeping performance up between processes.
- Process components in NetKernel can be developed in various dynamic languages (including Ruby, Python, Groovy, JavaScript or dynamically compiled Java) or domain specific languages (XSLT, XQuery,...). The underlying kernel takes care of a smooth execution.
- The PCMS serves as an abstraction between pipeline development and execution.
- The PCMS also takes care of the pipeline management.

A word about NetKernel[5]

NetKernel forms an important part of the actual Pipeline Execution Engine.

NetKernel is a resource-oriented computing platform. NetKernel enables a resource-oriented approach to the construction of software systems. In a resource-oriented system, the main focus is on resources. Resources are information that a computer locates, computes, transforms, formats, and delivers. Resources are what computer users want out of their system. They are a higher level concept than code and objects as found in code-oriented and

[3] `http://www.1060research.com/netkernel/index.html`

[4] `hhttp://www.saxonica.com/products.html`

[5] Some parts are adapted from the documentation (`http://www.1060.org/community/documentation.html`)

object-oriented environments. In a resource-oriented system, code and objects are still critically important, but a NetKernel developer will spend most of his time at the resource level, leveraging objects and code that are located behind services. This way, systems could change without breaking, just like for example the World Wide Web.

Some of the following features have proven tremendously important for an efficient design of the Pipeline Execution Engine:

- NetKernel's foundation is a robust microkernel that efficiently schedules every resource request.
- NetKernel provides scripted (dynamic) languages as well as compiled languages such as Java. Each language is dynamically compiled to native byte-code and cached so that the flexibility of using a dynamic language has negligible performance impact.
- XML is an important resource type and numerous XML languages are provided with NetKernel.

Wider applicability of the solution

In general a number of factors play a part in the complexity of an XML processing pipeline:

- The complexity of the different processes
- Number of documents, document types and schemata
- Complexity of the content and the schemata involved
- Variability of processes, schemata, style sheets,...

A very important factor for complexity is the evolution of all of the above over time. Even simple XML processing applications tend to become very complex to manage if they are subject to frequent changes over time.

The architecture used in this project provides functionality for version management of pipeline components and as such for management of pipelines as a whole.

It makes sense to layer a complex XML processing application on top of an architecture as presented above. By doing that, it would be possible to make necessary changes to processes, without having to worry about breaking parts of the operational system. The pipelines of the application would be properly managed, providing a clearly maintainable system.

This can be extremely helpful when the application later requires changes to the schema, the meta-data-model, the link resolution mechanisms, etc.

Future plans

This section describes some of the future changes to the core functionality of the proposed application, in order to make it useful in a broader spectrum of content processing systems:

- Better support for non linear pipelines that include iterating, forking and conditional processing.
- A dashboard for monitoring the behaviour of the pipelines execution, including execution timings and breakpoints. This dashboard would also allow checking out intermediate results, which would be very helpful for debugging purposes.
- Import filters for XProc[6] expressed pipelines or Stylus Studio[7] designed pipelines.

Summary

This paper showed some of the first results of a project that required version management of pipeline components. The author hopes that he could also demonstrate that the approach taken in this project is generally applicable for building complex XML processing systems with a lesser risk.

[6] http://xproc.org/
[7] http://www.stylusstudio.com/videos/pipeline2/pipeline2.html

DocBook: Case Studies and Anecdotes

Norman Walsh (Sun Microsystems, Inc.)

Abstract

DocBook provides a rich markup vocabulary. Too rich for some and not rich enough for others, of course. The transition to RELAX NG for DocBook V5 has made it easier to address both these concerns.

This presentation examines some case studies and explores how schema design changes percolate through the tool chain. We'll look at a few real-world examples, exploring the markup design changes and their rationale as well as the impact on the stylesheets necessary to turn DocBook into a presentation format.

Introduction

DocBook provides a rich markup vocabulary suitable for describing many classes of technical documentation. Of course, any attempt to build a global standard is invariably an exercise in compromise.

To many users, DocBook is *too* rich. Any technical domain is likely to find that there are elements in DocBook that are only relavent to other domains and are therefore a distraction.

Conversely, some specialized domains find that the markup is too coarse; they need additional elements to provide the fine granularity of markup necessary for their particular requirements. This is especially true for domains that are on the fringes of, or plainly outside, DocBook's scope.

Customizing DocBook V5 is considerably easier than customizing any of its SGML or XML DTD predecessors. To the extent that your entire toolchain can be built around RELAX NG validation, customization is fairly straightforward. Users that still need other schema formats may find that there are additional challenges in converting RELAX NG customizations, but hopefully, over time that will become both easier and less often necessary.

For most users, customizing the schema is only the first step. After the

markup design is complete, some mechanism for transforming the markup into a format suitable for presentation must be found. Increasingly, this is the work of XSLT, transforming DocBook into either HTML or XSL-FO. (There are other options too, conversions to TeX and a variety of other backends are common.) Here we'll consider transformation from DocBook to HTML as illustratory of the process.

Case Studies

We'll look at six specific examples adding:

1. A new hierarchy root
2. HTML markup
3. Geographic metadata
4. "Database generated" trip itineraries
5. Inline markup for metadata
6. Inline markup for personal names

In each case, we'll consider the schema design changes and the rationale for making them; we'll examine the schema customizations; and we'll explore the stylesheet changes necessary to support them.

These examples are all drawn from the markup used to generate the weblog at `http://norman.walsh.name/`. This entire site is generated from a (growing) collection of DocBook essays, some RDF metadata extracted from several sources, and a set of XSLT 2.0 stylesheets.

Open XML Overview

Štěpán Bechynský (Microsoft Corporation)

Abstract

Office Open XML File Formats

Office XML Formats for the 2007 Office system introduce or improve many types of solutions involving documents that developers can build. You can access the contents of an Office document in Office XML Formats by using any tool or technology capable of working with ZIP archives. The document content can then be manipulated using any standard XML processing techniques, or for parts that exist as embedded native formats, such as images, processed using any appropriate tool for that object type.

You will see basic concepts of Open Packaging Conventions (OPC), WordprocessingML and SpreadsheetML. OPC is fundamental for all documents types, WordprocessingML is used for text documents and SpreadsheetML is used for spread sheets. There are more "MLs" but you will see them just briefly.

Processing the OpenDocument Format

Lars Oppermann (Sun Microsystems Inc.)

Abstract

The OpenDocument Format (ODF) is a an open XML based file format for office productivity applications. The specification of which is maintained by the OASIS OpenDocument Technical Committee. The OASIS membership has recently ratified revision 1.1 of the specification [1] *and revision 1.0 is also available as an ISO/IEC standard* [2]*.*

The open nature of the OpenDocument specification has spurred adoption of the format by multiple applications. Furthermore, OpenDocument is well suited for being processed outside of end-user applications. This paper explains the basic structure of OpenDocument and how standard XML processing tools can be used to create applications beyond the scope of traditional office productivity applications. The benefits and limits of processing documents based on an open file format rather than based on an application specific API are explained.

Introduction

Office productivity documents play a vital role in todays business world. Documents are created by humans for humans. Information is mostly structured visually with paragraphs, font styles, tables and bullet lists. One important aspect of open storage formats for office productivity documents is the preservation of the information, so documents can be read in the future [3]. However, it is also important, that the information contained in documents can be used in automated processes and thus computer systems should be able to process documents.

Large office application packages provide automation or scripting APIs, which allow to programmatically open, access and modify documents. However, this has a number of drawbacks:

Office productivity suites are large and complex software packages. Au-

tomatic document processing is not necessarily their primary design goal. Moreover, using such application on a server is not always feasible and in some cases might not even be possible at all. Furthermore, such desktop applications are not designed with parallel processing of large quantities of documents in mind − something that is very much thought after in settings, where a high volume of information stored in documents is to be dealt with.

Additionally, using a specific application API for document processing limits its application choice. It is unlikely, that the programs written for a particular applications API will be easily translated to the API of another applications. On the other hand, directly processing documents of a standardized open format widens choice, because the standardized format ensures a level of interoperability in how information is represented. Thus, OASIS OpenDocument Format enables such an alternative approach of standalone direct document processing. Because the format specification is publicly available, documents can be processed independently of the application that is used to display or edit the documents.

ODF structure

In order to process OpenDocument Format files, the general principles and structures of the format should be understood. It is not necessary to know every aspect of the specification. Knowing the general principles and then focusing on the area that is of particular interest to solve the specific problem will suffice in most cases.

Package structure

OpenDocument Format files are composed of a hierarchical collection of XML files and other resources. These are stored in a directory structure in a zip-file [4]. The individual XML files in the collection are called streams. Streams are identified by their hierarchical name in the zip-file structure. An OpenDocument Format file may include any number of streams and resources. The following names are reserved for specific streams:

- `mimetype`, must be stored uncompressed as the first entry in the zip file structure. It contains the mime-type of the document. This allows for convenient file type detection, without actually having to open and parse the whole document.

- `META-INF/manifest.xml`, contains information about all the streams and resources that make up the document. Declares type of embedded objects. Information about signatures and encryption.

- `content.xml`, contains the actual document content and automatic styles
- `styles.xml`, style information. Styles are referenced from the content.
- `meta.xml`, metadata about the document, such as title, author and keywords.

Document Types

The main document types represented by the OpenDocument Format are text documents, spreadsheet documents, drawing documents and presentation documents. Furthermore, formula/equation documents and, charts can be represented as stand-alone documents although they are normally used as embedded objects inside the main document types. ODF further supports master/global documents and templates.

Text Documents

Text documents are the most common office productivity documents. They are among the most universal containers for ideas, knowledge and information. The use of text documents has a long tradition and is deeply rooted within our culture.

On a general abstract level, text documents can be modeled as an "ordered hierarchy of content objects (OHCO)" [5]. There are many possibilities as to the granularity and semantics of the objects in this hierarchy. When a model for a text document is designed, the choice of what will be represented is influenced by how users will interact with the documents and the types of documents that will be represented. For XML based document representations, this translates into the choice of elements and the semantics that are associated with them. The imaginable spectrum reaches from very generic concepts such as a text-portion, run or paragraph up to more abstract text objects such as headings, lists, citations or cross-references. Even domain specific concepts like a verse, or a figure-of-speech might be desirable.

In physical text documents, abstract concepts are represented with visual conventions: bold means important, vertical spacing denotes paragraph boundaries etc. Thus, when looking at the rendition of a document, appearance of individual elements becomes part of the content and meaning of the document [6]. Consequently, users that work with a text processing application want to use this visual vocabulary.

On the other hand, when a document is processed with software a more explicit and unambiguous representation of abstract concepts is desirable. The human brain is very good at extracting meaning from context and convention. Computers are generally not.

The OpenDocument Formats text document model tries to strike a balance between the above scenarios. The model uses a mix of lower and higher level abstractions that is based on the content model of HTML. This has the convenient side effect of making ODF text documents easy to understand for anyone that knows HTML and also eases conversion between OpenDocument Format and HTML.

In general, text document content consists of a body, which may contain block-level structural elements such as headings paragraphs and tables, sections and lists. These structural elements contain inline content such as text, images, drawings or fields.

Style is applied to individual elements through referencing named styles from a collection of styles that is part of the document. Thus, by interpreting the style referenced by an element a program may gain knowledge about the higher level semantics of an element without actually processing the style collection.

Styles

Styles describe the visual properties of the elements that reference them. A specific style is identified by its name. Styles in ODF support inheritance, in the sense that one style may be based upon another style. That style will inherit all properties set by the base style unless it specifies an own value for that property. The properties are based on the visual properties in CSS and XSL. This makes it easy to convert back and forth between ODF and formats that use CSS or XSL stylesheets. It also eases the task of programming with the OpenDocument Format for developers that are already familiar with those technologies.

Styles can specify a follow-on style, which is mostly relevant for editing applications. For instance, if A specifies B to be the next style, when the user edits a paragraph with style A and creates a new paragraph, that paragraph should use style B.

Furthermore, styles are separated into style families. A style family denotes the properties that can be specified in a style that belongs to a certain family. Specific content elements may only reference styles of an appropriate family. For instance, it would not make sense for a paragraph object to reference a style of the table-cell family.

Styles should not be seen as only having relevance with regards to the visual representation of a document. Semantic styles can be used to enhance the structure of a document and increase the possibilities for automatic processing significantly. Formatting in documents is used to express semantics: “this is important” or “this is example code”. If styles are used consistently to express

these semantics, programs are able to process documents more effectively.

The OpenDocument Format specification itself is an example for the use of semantic styles. The specification includes RELAX-NG fragments. When combined, these fragments form the OpenDocument Format schema. When XML elements or attributes are mentioned in the specification text, these are marked by styles. This enables a process, where it can be ensured that the element names mentioned in the text are consistent with the elements and attributes defined in the schema. Furthermore, it allows for another process, that extracts the normative schema from the specification document, where it is maintained along with the descriptive text.

Spreadsheets

Spreadsheets have been touted the "Swiss army knife of the office worker". Spreadsheet applications provide a convenient interface by which users can interact with relatively large amounts of data. They empower users to filter, sort and analyze information in an ad-hoc and intuitive fashion.

An OpenDocument Format spreadsheet document consists of a number of spreadsheets. Each spreadsheet is a grid of rows and columns − a table. Consequently, text documents and spreadsheet documents use the same content model for the representation of tables.

ODF, being an XML file format uses XML to represent the spreadsheet data model. Today, XML has become the lingua franca in which to express business information. Consequently, it is quite practical to transform back and forth between domain specific XML representations and spreadsheets in order to use a spreadsheet application to interact with the data from those domain objects. Such translations can happen independent from the actual implementation that will be used to open the resulting spreadsheet. Actually, this provides a much more general approach than using the scripting features of a specific application, which will not be compatible with those offered by any other application.

An example for using such a format based translation of data representations in order to use a spreadsheet application to handle domain specific data is VOFilter [7]. VOFilter is an XML based filter developed by the Chinese Virtual Observatory project to transform tabular data files from VOTable format into OpenDocument Format.

Processing Documents

XML Processing

For processing tasks, that perform manipulations of existing documents or that perform translations between ODF and other XML languages, tools from the XML technology stack play an important role.

XSL-Transformations can be used to process the XML streams that make up the content of an ODF file. Applications include conversion to and from other formats, extraction and import of information from other data sources as well as reformatting of existing documents. The Sun Microsystems OpenOffice.org team has used XSLT to implement a number of file format filters, including ODF to XHTML conversion and conversion to and from docbook.

XSLT is particularly useful when creating renditions of more abstract XML documents, that can then be used in an OpenDocument Format supporting office productivity suite. For instance, UBL documents can be made to display in a word processor or spreadsheet application, making the information accessible for users that lack the specialized tooling required to work directly with such documents. Spreadsheet applications and documents are a very practical way in which structured information can be authored which is then translated into specific XML dialects with XSLT.

XQuery is another attractive technology which can be used to construct OpenDocument Format content from XML repositories.

Programmatic processing

While XML tools can be used to implement many interesting and practical processing scenarios with ODF, more sophisticated and complex scenarios require a more flexible approach. In such cases it is desirable to interact with ODF in a general purpose programming language. With zip file handling and XML processing capability being a functionality offered by all major programming platforms, the foundations are readily available.

Thus, on the most elementary level, we can use the zip-file and XML functionality of our programming platform to access the individual XML streams in a document and perform specific manipulations on those XML documents. However, in that case, the level of abstraction through which we interact with the document is the XML view provided by the XML implementation of our programming platform (e.g. SAX or DOM). Thus, like with the XML tools described above, we would be interacting with the document on the XML information set level.

This type of processing is very useful, when ODF content is to be integrated

with existing systems and only a light weight wrapper is needed to make information from ODF files available to the other system. A typical example is an adapter that integrates OpenDocument Format files with a search engine.

Interactive WYSIWYG-applications like word processors allow the user to interact with the document on a level that we may call the visual information set. When writing programs, that are to work with documents, we want to interact with the structural information set [8]. This structural view is mapped on top of the XML representation of the document. However, instead of an automatic XML/OO binding it should take into account the specific semantics of the individual elements that make up a document as they are specified in the OpenDocument Format standard.

Such tasks are usually accomplished by frameworks or toolkits. One might now argue, that the use of an applications scripting facilities is the same as using a framework for the OpenDocument Format. This today however is not the case. An application specific API traditionally is tied to a specific application and that applications specific functionality. It has no direct relationship to the standardized OpenDocument Format. Rather does it represent the application specific view of the document.

A framework based directly on ODF on the other hand has a direct relationship to the standard. While current applications API are centered around the rendition, a direct ODF framework can emphasize the actual document structure as defined by the ODF specification. This allows for such frameworks to be created for different programming platforms in a consistent way. While adapting to the principles and idioms prevalent in the particular platform, the ODF specification provides the conceptual and structural integrity. Thus switching between frameworks and programming platforms is easy. Both for programmers that know one environment and want to work with an other and also in terms of porting applications between environments. This can be extended to include office productivity APIs, when they have made the transition from the current application specific APIs to those defined by the framework.

ODF Toolkit and Related Work

The OpenOffice.org community is now addressing the need for such frameworks in the newly created "odftoolkit" project
(`http://odftoolkit.openoffice.org`).
This project aims to create tools for programmers which allow them to build applications that interact with ODF on various platforms, and aims to provide OpenOffice.org as a framework itself Just like office productivity suites provide interaction in terms of the visual rendition of documents, components

from the odftoolkit project will provide interaction with the abstract structure of documents consistent with the OpenDocument Format standard.

One particular sub project of the odftoolkit effort is the "odf4j" project, which aims to bring structured ODF processing to the Java platform in a way that is both consistent with that particular platform and with the OpenDocument Format specification.

This particular framework, is based on a set of design principles, which we view as vital for any ODF processing library on any platform:

1. Access to the document is possible on multiple levels of abstraction: either direct access to package contents such as the XML streams. This allows the for easy integration with existing XML processing tools and for the creation of light-weight adapters that integrate ODF with other systems.

2. Preservation of the XML info-set, such that markup from other namespaces that was embedded in a document for some specific purpose is retained and may be handled by other participants in a workflow processing the document.

3. Adherence to patterns and idioms of the platform. In the case of Java, this includes the use of the Java collections framework and providing interface suitable to be used with the Java XML infrastructure.

While odf4j is a work-in-progress, there are similar projects that offer OpenDocument Format processing on other platforms:

AODL, which is short for "An OpenDocument Library". Provides OpenDocument Format processing for the .NET platform. It is also developed as part of the OpenOffice.org community as a sub project of the odftoolkit effort. AODL adheres to the same design principles as odf4j. However, as of this time, AODL offers a much more complete implementation of the OpenDocument Format standard.

OpenOffice-OODoc by Jean-Marie Gouarn
(`http://search.cpan.org/dist/OpenOffice-OODoc/`)
implements ODF processing for the PERL platform. This project was the first to offer structured programmatic processing for ODF, the first versions where created f or the OpenOffice.org XML format, from which the OASIS OpenDocument Format was created.

[9] gives a detailed account of the direct XML manipulations that can be applied to the OpenDocument Format.

Limitations of format based processing

Programmatic processing based on the standardized OpenDocument Format has many advantages. Most importantly, it separates office productivity document processing from office productivity applications. However, there are limitations to this approach that must be considered .

When the documents that are to be processed are created by human users with office productivity applications, there is always the danger of ambiguity. Because productivity apps use the visual rendition of the document to provide a user face to the document, things that look alike might not actually be the same thing when looking at abstract document structure. The more a user relies on intangible visual conventions in order to express semantics, the harder it will be for a program to understand the document. Depending on the actual problem, this may or may not be a problem. However, it shows that automatic processing of office productivity documents is not a means to all ends and that there are situations in which more structured formats and more restricted content entry methods are required.

Furthermore, if actual renditions of documents are to be created, it is highly questionable, whether it is feasible to implement this as part of document processing. Office applications include sophisticated pagination and layout algorithms, and subtle differences in these algorithms may have a high impact on the rendered result. It is the authors opinion, that rendition related tasks should be performed by an application that is designed for that task. In the case of OpenDocument Format processing, this would mean that a document is either rendered by the layout engine of an Office productivity application, or that a processed document is translated to formatting objects and than rendered by an appropriate engine. For ODF processing frameworks as discussed in this paper, the creation of rendition is considered out of scope.

Future Work

In order to further emphasize the advantages of direct document processing, conceptual convergence of existing solutions is an important goal. It is not yet clear how such convergence can be governed, but the odftoolkit project presents a place for interested developers to share their ideas and visions. The OpenOffice.org development community is actively participating in the odftoolkit effort in order to also offer conforming API as part of the OpenOffice.org application.

One of the biggest problem in automatic processing of documents lies in understanding the semantics of the contents on a level that goes beyond what is expressed by the explicit markup. This is limited both by design choices

made to keep the standardized vocabulary manageable and by the document interaction paradigms used in office productivity applications. The OASIS OpenDocument Format TC is addressing this concern in a metadata sub committee, which is working on integrating semantic web technologies into the OpenDocument Format.

Conclusion

We have shown how for a range of application, direct processing of OpenDocument Format files provides an attractive alternative to the traditional practice of processing documents through automating the application that is normally used to work with those documents.

Emerging ODF processing frameworks will provide a consistent programming experience because the OpenDocument Format standard provides a platform independent specification which frameworks can translate into platform specific concepts while maintaining overall integrity.

The OpenOffice.org community has created the odftoolkit project as a place for interested developers to work together on frameworks for various platforms, exchange ideas and enhance interoperability of their products based on the OpenDocument Format standard.

References

1. *OpenDocument v1.1 Specification.*
   http://docs.oasis-open.org/office/v1.1/OS/OpenDocument-v1.1.pdf

2. *ISO/IEC 26300:2006.*
   http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=43485&scopelist=PROGRAMME

3. H. M. Gladney. *Preserving Digital Information.* Springer Press, 2007. ISBN 978-3-540-37887-7. Str. 139-161.

4. *Info-ZIP Application Note 970311.* 1997.
   ftp://ftp.uu.net/pub/archiving/zip/doc/appnote-970311-iz.zip

5. S. J. DeRose, D. G. Durand, E. Mylonas, A. H. Renear. *What is text, really?.* SIGDOC Asterisk J. Comput. Doc. 3, pages 1-24, Vol. 21, 1997.

6. S. A. Selber. *The OHCO model of text: merits and concerns.* SIGDOC Asterisk J. Comput. Doc.. 3, pages 26-31, Vol. 21, 1997.

7. Ch-Z. Cui, M. Dolensky, P. Quinn, Y-H. Zhao, F. Genova. *VOFilter: Bridging Virtual Observatory and Industrial Office Applications.* Chinese Journal of Astronomy and Astrophysics. 2006.

8. J. Ziegler. *Structured data and the death of WYSIWYG.* 2005.
   http://www.conglomerate.org/docs/death_of_wysiwyg.html

9. J. D. Eisenberg. *OASIS OpenDocument Essentials.* 2005. ISBN 978-1-4116-6832-4.

Leapfrogging microformats with XML, linking, and more

Uche Ogbuji (Zepheira, LLC)

Microformats the XML way

Some microformats are straightforward, useful and unobjectionable. Others, including many of the popular ones, abuse HTML, are poorly specified, and are quite prone to confusion. When designed and applied without careful consideration, microformats can detract from the value of the structured information they seek to provide. Beyond the simplest class of microformats it is often better to avoid hackery and embrace the native structure of the Web. XML and other natural data representaton technologies such as JSON are just as viable as many of their counterparts in microformats. The main argument against these is that microformats provide graceful degradation for unsophisticated Web clients. But such graceful degradation can also be achieved through the power of linking. A Web page can still be a Web page, and not a scaffolding for a bunch of ill-fitting and ill-specified records. All it has to do is link to those records in their native format. More sophisticated browsers can be dressed up with all the AJAX gear you like, loading simple, linked XML or JSON into dynamic views while crawlers and legacy Web clients can access the structured information through user-actuated links.

Microformats work best where they add a very little bit of nuance to common constructs in a host language such as HTML, XHTML, or Atom. An example is rel-license, which allows you to express that a link is identifying the usage license for the source page's contents. The link

```
<a href="http://creativecommons.org/licenses/by/2.0/"
  rel="license">cc by 2.0</a>
```

anywhere in the page means that the page's contents are available under a Creative Commons 2.0 Attribution Required license. I've seen people abuse this microformat to assert a license for software described by a page, rather than for the page itself, but one can't really blame the microformat's developers for this. A bigger problem is that it's possible for such conventions to

clash but, for the most part, microformats ignore such problems, hoping that lightweight agreements will solve them as they arise. A microformat such as rel-license provides a convention for use of an HTML attribute designed to carry such conventions. These are what microformats designers call elemental microformats, constructed entirely within an element.

Some microformats try to tunnel specialized structure into the constructs of the host language. Microformats designers call these compound microformats. A good example is XOXO, a microformat for expressing outlines. XOXO is far more inscrutable and harder to process than almost any XML you might design to replace it. XML was, of course, designed to express complex and specialized structures for content, and it seems a step backward to use a far less expressive construct just to embed the structure within HTML. Microformats folks do this because they feel that XML is too complex, not yet ubiquitous enough and, more importantly, doesn't allow for graceful degradation, which means that microformats look like regular HTML to user agents that do not understand more advanced technologies such as XML. This is a fairly weak argument, in part because XML is supported by most user agents these days and also because sometimes a scalable design for the Web is worth such tradeoffs and inconveniences. Atom was developed as a better Web feed format despite the ubiquity of other RSS dialects. Cascading Stylesheets (CSS) is developed as a way to separate content from presentation in Web pages, despite the fact that it's easier for the lazy Web publisher to just use font and center tags, and even regardless of the hurdles browsers have placed in front of the conscientious Web developers who do try to apply CSS. Despite the heavy burden of legacy, both technologies are doing well and legacy is certainly a poor excuse for bad design in microformats. The design problems are aesthetic and functional. The aesthetic problems are simply the ghastly result of hijacking one format to serve a tenuously related purpose, as in the case of XOXO. The functional problems come from the likelihood of term clashes, and the difficulty of automating processing of microformats, because of the sloppiness of schema and semantics. Many elemental microformats reuse, for example the `link/@rel` attribute with lightweight controlled vocabularies, and there have already been problems with clashes and term confusion. Even worse, some microformats abuse such host format constructs completely.

To be fair, microformats developers have recently admitted that microformats are intended to be a hacky stop-gap in this period where XML technology on the Web is still not very reliable, and other Web technologies are in such flux. The most stable parts of the stack are still basic HTTP and HTML, and Microformats looks to cobble together what they can around the latter. Regardless of the circumspection of Microformats experts, many

users are far less so, and without the needed note of caution some very bad Web habits are beginning to proliferate under the umbrella of microformats. Certainly elemental microformats are useful, and build on classic respect for rough consensus and running code. They are like formal registries for MIME types, filename extensions, and such, but with less central control. Problems come about when microformats start to distort host formats rather than just build on them. It's almost better to use a more suited format in such cases, and the Web is full of helpful means that leave you with a healthy choice.

The Generic Transformation Architecture

Bryan Rasmussen (Danish OIOXML)

Abstract

This paper describes the implementation of a Generic Transformation Architecture, the requirements and difficulties of an XSL-T framework where it must be considered that the potential amount of inputs and the potential amount of outputs can be infinite, it is essentially a higher level guide and reference for the examples that will accompany the talk showing implementation of DocBook transforms using the architecture, examples of various document types transformed into various media provided as part of the GTA, cross-media scripting using Funx, code examples pertinent to generating media in XSL-FO, HTML, Open Office with Xforms, and how it can be used to for generation of XSL-T used to implement the GTA, using it in XML Pipelines, using it for generation of Example XML files from templates.

Introduction

A lot of this talk actually goes back to some work I did in 2002 but could not release the code for at the time because of various rights to that code, later when working for the Danish National IT agency (a subsidiary agency of the Ministry of Technology, Science, and Development), I was able to test it further through various scenarios involving Data oriented XML, as well as with such international standards as UBL.

In 2002 I was working on a product the purpose of which was to make a cross-media management tool. What is cross-media management, it is something like a content management system, providing just the level of content management needed for managing the multiple inputs multiple outputs single-sourcing promised by the XML revolution.

What this meant practically was that I had to build a system that would take theoretically infinite numbers of XML inputs, and be easily extensible to generate a theoretically infinite number of media outputs.

Before continuing I should ask and answer a question, that being:

What does the word Media mean?

Well, if I go look it up I get a concise answer from Wordnet: "transmissions that are disseminated widely to the public", if I look in Barron's business dictionary I get "Channels of communication that serve many diverse functions, such as offering a variety of entertainment with either mass or specialized appeal, communicating news and information, or displaying advertising messages.", which I think is too narrow for most people not intent on how media relate to a business purpose, and if I use it in the sense of a plural of medium and then try to find out what a medium is I find that it is something like art, we all know it when we see it.

Note that of these definitions there seems to be a bias for electronic media, what with the word transmissions and channels of communication, which can give the impression that the means of delivery of the medium is an essential part of the medium.

For the purpose of this paper I will argue that media and medium is dependent on the context of the age, the media of spoken epic poems from antiquity delivered by an aged poet is in our age the problem of how to make Grandpa shut up at the dinner table. In an industrial age a medium is that which the production of can be automated by industrial processes, and in a computerized it is that which the production of can be automated by computerized processes. Thus media can include television, radio, newspapers, books, websites, and so forth but not, in this dispensation, garrulous old men with tales of brave Ulysses.

If a medium is generated from a computer, then it is the result of a computation, and from the point of view of a computer programmer the difference between media would be, two media differ when, to efficiently deal with the generation and automation of the media, there should be two programs.

I could of course generate a printed media and a website from the same program, but it would not be the most efficient, the two solutions while requiring at a sufficiently high level an input and an output also require widely different setups and organizations of their display, differences in user input, and sundry other differences would mean that I would either do one media particularly well, and skimp on the other, or I would do both media particularly badly. This tension between different media would only increase as more media were added to the generating program.

Generation of Media with XSL-T – The Tradition

Now this statement as to the generation of a media really follows the design

of most XSL-T solutions when drawn in those nice graphs from companies that tell you they have a wonderful product and the only thing you need to do to get its XML output to your chosen media is to add a stylesheet. The following is an example (only not very nicely drawn because we all know it):

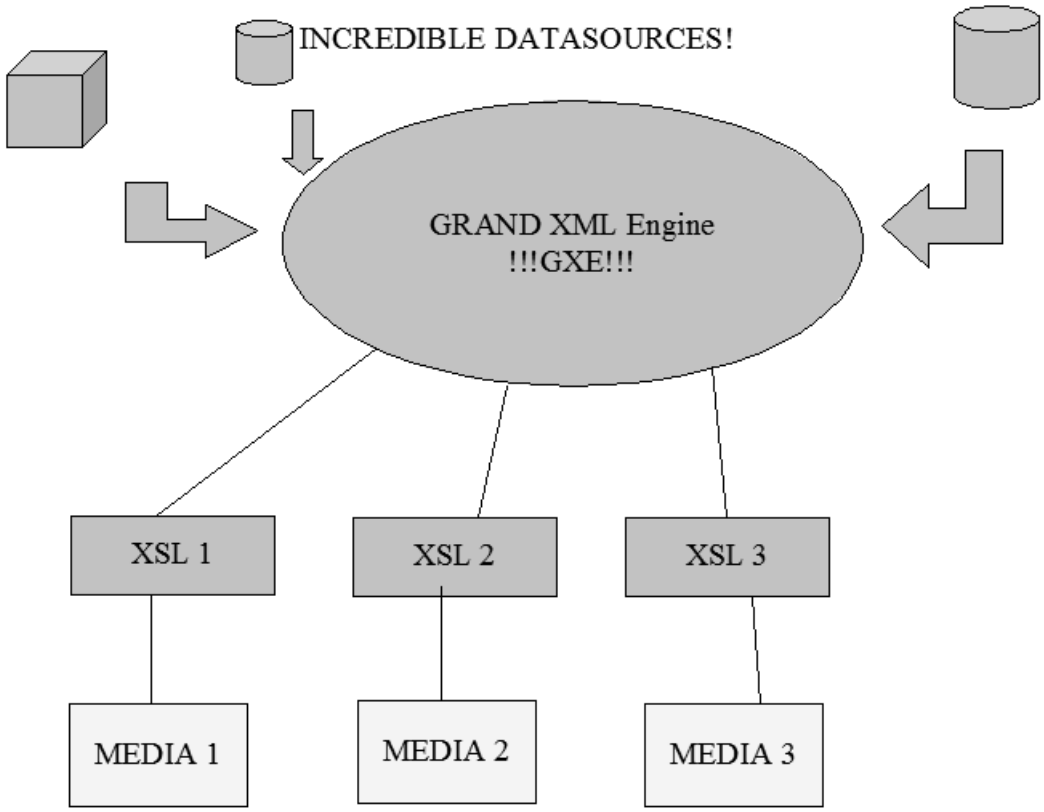


Figure 1: Generation of Media with XSL-T

Now in the world that diagram shows us those "XSL" and "MEDIA" boxes on the bottom just kind of go off into infinity because if there is one thing people know it is that slotting in a new XSLT to deal with a new media channel is as easy as falling off a log. And even easier is adding in a new XML format to your process, and rolling out twenty new stylesheets for all your media. Also did I mention maintenance would be a breeze? GXE (Grand XML Engine) made all that stuff so easy it wasn't even going to bother telling you how to do it.

Well thinking over these things at the time I pretty quickly came to the following realizations

1. Basically we were thinking about just handling the little "XSL" and "MEDIA" boxes going into infinity, and trying to make it easy for GXE to work with us.

2. It is a somewhat well-known idea that the solution for any sufficiently complex problem in the end involves the creation of a programming language (whether or not the programmer knows it), thus I have always

figured that when confronted with a sufficiently complex problem the first order of business is to figure out what the programming language has to be like, and this was a sufficiently complex problem.

3. Users should be able to configure media with a great deal more ease than they could just by slotting in XSL-Ts into GXE. But they should be able to keep from loosing the power of slotting in XSL-Ts. If the solution was significantly less powerful in the configuration of the media output then the user would be going back to slotting in XSL-Ts, people tell you they are willing to give up power for configurability but two things cause me to believe this is not so: 1. nobody has ever been called configurability-mad and 2. my spell check tells me that configurability is not even a word.

The Theory – On the structure of Media, Static and Dynamic

If a media is produced by a process of computation this means that a media is (generally speaking) in some way structured. Furthermore this means that our media are generated by the interpretation of programming or markup languages or conglomerations of the two (for various weak meanings of the word means).

In languages we often have what we can think of as the core structures of the language, and we have syntactic sugar. Everything that can be done by syntactic sugar can be done by the core structures, but in some cases it would be inefficient to do so. In the same way it seems that in a medium there are elements that are the core elements of that media, and all the various forms of syntactical sugar of the media's presentation can be built up by manipulating those core elements (and how could it be otherwise if the media itself is the representation of a program). I will give some examples of this perhaps strange statement during the course of the talk.

The most flexible and generic format for a media should be chosen as the root default output of that media when transformed. I will give some examples arguing that the most flexible is the most generic.

There are in our computerized society two main types of media, static – meaning that they have been processed through a computer in the creation but they can not respond to user input changing their state, and dynamic – meaning that they are processed through a computer and interpreted at the user's request. To say that they are static and dynamic in this context of course means in relation to the program doing the processing, if I run an XSL-T on the command line and it outputs a python script meant to run at localhost 8080 and interpret requests it may be a dynamically interpreted

language but the interaction between it and the XSL-T is static.

Static media are in many ways remnants of the pre-computerized era, and are generally analogues in some way to a Printed media, as examples PDF is in structure close to that of a book with some extended capabilities such as search thrown in and email is meant as analogous to mail, these metaphorical relations to pre-computerized media is one reason for their acceptance as static media by the public. Of course in the age of the computer this division between static and dynamic media is illusory. The thing that actually makes a media static or dynamic is the decision on how to generate it.

Dynamic media essentially contain a concept above media, which is that of an application.

The most common dynamic medium familiar to most people as a dynamic medium is of course the Web but the same aspect of dynamism and the blurring between media, interactivity, and applications can be found in telephony, although it is likely that the full realization of this will occur when telephony moves to being VOIP based.

A system for generating dynamic media must also allow for generation of applications (if it is to be considered at all serious in the target medium and not some toy), to do so efficiently would mean that the application doing the generation of the media would in effect be a framework for managing other applications in the target media.

The Requirements of a Generic Transformation Architecture

The rules are given in following list:

1. Given a multiplicity of files, there must be a way for the transforming application to find what XSL-T to apply on an XML input. Unsatisfactory solutions to this include having one XSL-T for inputs, which would obviously constrain the number of supportable inputs, keeping the information as to what stylesheets should be used for what data formats in memory which will cause scalability problems if we want to keep increasing our formats, passing the information to the application along with the XML to be transformed, this is often done in dynamic media such as the web by referencing a stylesheet to use via the querystring, this is especially unsatisfactory because it is to be expected that one moves from one type of data to another via linking, in such a situation if the stylesheet to use is passed along with the link then one will essentially need to specify in links what the data at the end is drawn from. Tying the transformation somehow to validation, because by being able to validate

something we know what it is thus we know what validation applies to it. This is a tight coupling of two disparate things and is also unsatisfactory.

2. For dynamic media one should have support for building applications in the Architecture, making the implementation essentially a framework.

3. The implementation of the various media should abstract enough away the data formats allowed and the specific rules of the output format that the media generated by the implementation can be maintained and too some extent extended by people unfamiliar with the input formats and the output formats.

4. The media generation should be able to respond to user input at the time of generation. This should be at a higher level than that required in point number three.

5. Validation should be turned off. It can be that validation will need to be handled by other applications, but the validation should not be part of the transformation step.

The Implementation

Implementation of Rule 1

The implementation of these rules is very simple, for applying stylesheets we choose the stylesheet for a particular media by taking the namespace URI of the document element, this is escaped by URI escaping rules and then escaped further by changing all characters that might cause filesystem confusion into underscores, all characters will be lower cased. The application then checks for a stylesheet that matches the following characteristics:

1. A stylesheet with the name of the escaped namespace and the baseName of the document element (also escaped) followed by an underscore and the name of the media channel the application is meant for. Thus if we have a webmedia media and a DocBook document where the document element is Chapter the first stylesheet that the application would look for would be named: `http%3A_docbook_org_ns_docbookchapter_webmedia.xsl` however if we had some documents that were older DTD based DocBook files then we may have an XSL-T named `chapter_webmedia.xsl` to handle it.

2. If the application cannot find a stylesheet matching the document element and namespace concatenated in the method outlined it should look for a

stylesheet named after the namespace followed by an underscore and the media channel, as in `http%3A_docbook_org_ns_docbook_webmedia.xsl`

3. If it cannot find a stylesheet with one of these two names it should look inside the XML document for a namespace qualified attribute on the document element, `gta:type` (where the namespace `gta` is bound to is `http://rep.oio.dk/gta`) where the name gives a specific document type name that will be used to determine the stylesheet by concatenating the escaped name with an underscore and the name of the media channel.

4. If it cannot find this it will look for an xml `processing-instruction` named `gta-type`, with the same process to determine the stylesheet to use.

5. If it cannot find this it will use a stylesheet with a name matching the concatenation of the media channel and `generic.xsl`, this must be provided for each media channel. For example for a media channel named `pdf` the name would be `pdfgeneric.xsl`. This stylesheet will attempt to determine from examining the XML design patterns used in the document what the proper layout of the document is. This stylesheet can be imported into others and via the setting of a parameter be set to output a particular presentation, for example if the natural presentation of an XML structure is that of a table the XML could use the `gta:type` attribute on the document element to assert that is a document of type `gta:table`. Structures that are well suited to tables are ones where the children of the document element all are of the same type and these elements each have a sequence of elements with the same names.

In dynamic media there can be additional functionality to override the stylesheet choosing algorithm of the application by explicitly stating what stylesheet should be used at transformation time, such as was discussed in the example of passing the name of a stylesheet in the querystring.

This method may seem theoretically unsound to some, the argument may arise that we do not really know what the purpose of the following document is, and thus to process it for display can be problematic. The answer is that the problems of display are problems of the media and human interaction with it, the processing for such purposes does not present any inherent security risks and can be undertaken without concern. But aside from this, despite it not being specified there has arisen a common habit in XML that the purpose of a document is really its namespace and document element. The passing around of XML fragments is, outside of the specialized XML community, not done that often. Of course there are exceptions to this, thus the various

possibilities of overriding the standard practice discussed. I will also discuss some other aspects of exceptions later.

Implementation of Rule 2

Obviously in such a framework it is most useful if semantically rich data is the source of presentation, However in a framework of multiple formats it will be necessary to be able to move outside of the semantically rich data in order to do programmatic things not inherently related to the data one finds oneself in but rather related to the framework as a whole.

This may also be used to do arbitrary presentation, as opposed to the structured presentation that is generated from the semantically rich data.

The way this is done in the Generic Transformation Architecture is through four related methods:

1. The GTA allows a double configuration layer, there is a global configuration for the media and a configuration for the document type input. Generic documents have a generic configuration at the document type level.Each media configuration has similar concept put some things may be configurable in some media while not in others. Presentational objects that have a similar nature across media will have the same names. As an example most media solutions will need a way to output content before and after the content of the input document. This applies equally to an audio document of some sort, a printed document, a webpage and so forth. This can of course be done via the XSL-T, and in cases where doing so is a complicated affair it should be, but it can also be done via the configurations. There are defined the following configurable areas in almost every possible configuration file: universal-before, universal-after, before, after. The universal-before and after are set at the global configuration level. The before and after at the document. In some formats however we could expect further areas for configuration, for example a PDF media may need for maintainers to configure the output of the such areas as the left outer margin of an even page. This possible configuration is provided in the XSL-FO implementation but it will obviously not translate to very many other media because the concept of the left outer margin of the page being especially meaningful for all media. This normalization of the configuration method means that the current structure of large transformations such as DocBook can be improved and made more reusable.

2. The implementation of an inline templating language, such as one sees in web based templating languages like ASP, PHP and so forth. This

language can basically be executed in the input XML documents or the configurations for the media and the document types. As an example the configuration file can define functions callable across the global media solution, for example the following function definition is for an xforms in Open Office media:

```
<function name="linetotalprice">
  <fnx:if boolean="True">
    <fnx:eval>
      <fnx:greater>
        <arg><att:linenumber/></arg>
        <arg>0</arg>
      </fnx:greater>
    </fnx:eval>
    <fnx:then>
      <fnx:m-tag type="fnc:linetotalprice">
        <fnx:attribute name="linenumber">
          <fnx:subtract>
            <arg><att:linenumber/></arg>
            <arg>1</arg>
          </fnx:subtract>
        </fnx:attribute>
      </fnx:m-tag>
      <fnx:m-tag type="xe:linetotalprice"
        fnx:namespace="http://rep.oio.dk/tools/
        xformsExtension">
        <fnx:attribute name="count">
          <att:linenumber/>
        </fnx:attribute>
      </fnx:m-tag>
    </fnx:then>
  </fnx:if>
</function>
```

It could be called with the following function call: `<fnc:linetotalprice linenumber="10"/>`, the fnc namespace `http://rep.oio.dk/funx/fnc` holds user defined functions in the funx language. What it will do in this instance is write out an element `<xe:linetotalprice xmlns:xe="http://rep.oio.dk/tools/xformsExtension" count="{the current value of the recursion}"/>`

3. A simple presentation language consisting of a subset of html, and some extra capabilities for composable linking. In media that support CSS as the presentational layer of the media CSS will be used, in static media where CSS is not supported the styling of the markup elements is provided by an implementation of an XML-based version of CSS. The benefits of this however is that the Funx language can be used in the CSS XML format as well, and evaluated so that includes, parameters, and mathematical operations can be carried out at the time that styling, and provides the kind of seperation of style and content touted as the benefit of XML. This XML version of CSS can also be used to store CSS data and to generate the actual CSS for the media through a batch process, but this is not a requirement for dynamic media. This usage of the XML format to generate the CSS format is beneficial from an application perspective because in building tools to generate and work with GTA implementations the more an application becomes just a bunch of processes for manipulating and passing around XML the easier it is to make generic.

4. The passing in of a parameters nodeset to the XSL-T, this method is of course used by a number of tools on the market, making it easier to implement GTA on top of those tools. These parameters are callable by the funx language via the `param` element and the `param-exists` element. All parameters to the application should be sent in the parameters nodeset, in a dynamic media it should be possible for the user to add or subtract from the nodeset by their interaction with the media, however some parameters are not changeable by the user. These following parameters are defined as being set by the application and cannot be overridden in anyway by the user: `year`, `month`, `day`, `random`, `time`, `uniqueid`, and `securitylevel` − for applications that need to branch logic dependent on security level − note that the actual security identifier should be handled by the application, the application should just pass in what security level the user of the application has.

Implementation of Rule 3

The factors discussed in implementation of rule 2 apply here as well. By using an interpretable language that can be applied in the global configuration or document specific configuration of the media it will be possible to maintain and extend the solution for media output of multiple input formats.

This also applies to maintaining connections across media. The funx language does not fail when encountering unknown functions, thus it is possible

for one media to have a definition of a function and another to not, this is one of the useful ways of branching between the media.

Implementation of Rule 4

Response to user input is a factor achieved via the aforementioned parameters passing.

The basic diagram, for those who like these things, of an individual GTA transformation is as follows:

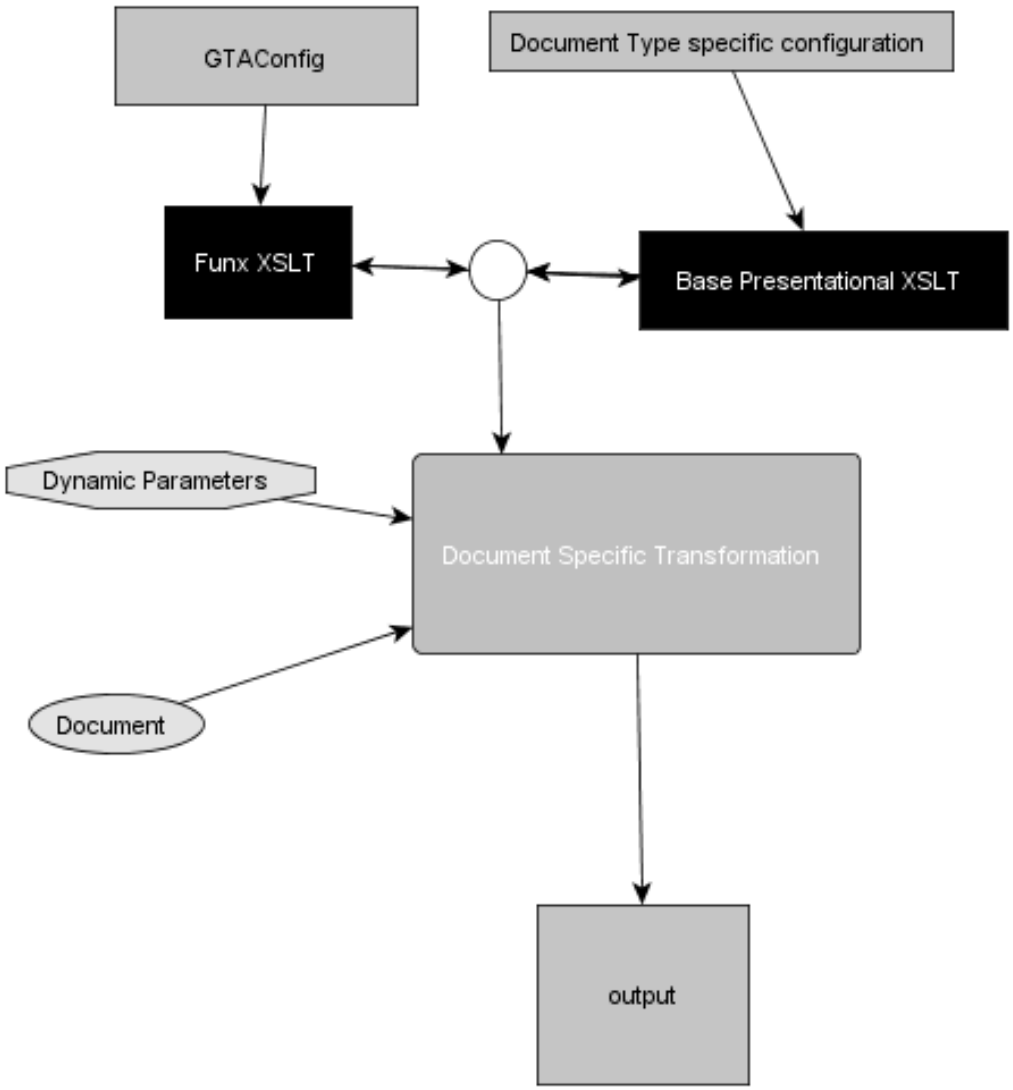


Figure 2: GTA Transformation

Variant and Dynamic Outputs

For this section I will discuss mainly XSL-FO, one of the common points of wisdom regarding transformation of data to XSL-FO is that it is suited to highly structured outputs, such as Textbooks and similar document formats. However in the GTA XSL-FO usage the output can be both structured and

unstructured, by unstructured meaning that the output of particular pages can be in the sequence of an otherwise structured document can be escaped to draw input as needed.

The examples here will be generation of PDF with XSL-FO.

Any individual PDF can be constructed of a number of documents that do not necessarily have a relation to each other. The styling of these documents is set by the user at generation time by choosing a specific CSS XML file to use for styling the specific XML input.

The layout regions of the PDF, such as the document header and footer, and whether these footers have numbers and where they are located are set by the configuration file. The generation of XSL-FO can also be escaped by writing XSL-FO dynamically via Funx, or having formatting object fragments within the configuration. It is the default behavior of any media to copy through markup of the sort that is being generated, thus HTML markup will be copied through in web media, and XSL-FO in pdf media. Finally there is the possibility of using SVG via the XSL-FO foreign object. Thus if one sends an SVG page directly to the processor, it will first wrap it in an FO page set to the styling of documents of type SVG, then wrap the SVG in a foreign object element, and then copy out all SVG, evaluating Funx expressions in the SVG.

This level of configurability of the media means that we can effectively break from the overall structure of documents if we need a particular part of the document to be structured via graphics.

Examples will involve generation of simple documents, Vcards to business cards, and use of XSL-FO, SVG, and Funx to generate specific displays.

The Funx Language

The funx language is basically an inversion of XSL-T. In the context of the framework because I believe it is useful if a framework's technologies relate to each other well enough that somebody working with the framework at a lower level can easier move up to the higher level. As such if a developer or user has worked with the funx language for a while it should be possible for them to move relatively simply to doing some xslt based configuration of the solution, finally moving up to producing straight ahead XSL-T following the rules of the GTA. I have had some indication that this theory was correct, but it may just be that coworker that was able to move up was bright and a quick study.

Thus most language constructs just implement XSL-T functionality in an XML input, for example the element

```
<fnx:add>
  <arg>4</arg>
  <arg>6</arg>
</fnx:add>
```

is just addition of the two `arg` elements.

Funx code can be contained in the input document, the global configuration for the media and the configuration for the document type.

For example the following is an xhtml document with Funx statements in it:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
  "http://www.w3.org/TR/xhtml-basic/xhtml-basic10-f.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" xmlns:f=''http://rep.oio.dk/funx''>
  <head>
    <meta http-equiv="Content-Language" content="en" />
    <title xml:lang="en">a document</title>
    <meta http-equiv="Content-Script-Type"
      content="text/javascript" />
    <link href="GTA.css" type="text/css" rel="stylesheet" />
  </head>
  <body>
    <h1 id="title" class="title">
      <f:meta name=''title''/> + <f:param name=''title''/>
    </h1>
  </body>
</html>
```

Metadata in Funx is read into a metadata object that tries to generalise the metadata for formats it knows, so that the query `<f:meta name=''Creator''/>` will, dependent on various inputs, resolve to that formats understanding of the concept Creator.

The parameters are interpreted in the way discussed, given a webmedia implementation that copies XHTML input to XHTML output and a `user` parameter that equals `howdy`, a parameter for the title power will generate the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
  "http://www.w3.org/TR/xhtml-basic/xhtml-basic10-f.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
  xml:lang="en" xmlns:f=''http://rep.oio.dk/funx''>
  <head>
    <meta http-equiv="Content-Language" content="en" />
    <title xml:lang="en">a document</title>
    <meta http-equiv="Content-Script-Type"
      content="text/javascript" />
    <link href="GTA.css" type="text/css" rel="stylesheet" />
  </head>
  <body>
    <h1 id="title" class="title">a document + howdy</h1>
  </body>
</html>
```

This is actually a very interesting aspect of the GTA, because the transformation of XHTML in the webmedia does not just copy the XHTML output but of course adds to it. Examples of this are too verbose and many for this paper to discuss directly, but will be provided as part of the presentation.

Uses of Funx Beyond the GTA

Of course Funx lends itself to usages beyond that of the GTA, such as the example of using it to evaluate expressions inside of the CSS XML format.

In this scenario the XML is a tree structure, some of which has been filled out, some of which is yet to be filled out, and some of which can be filled out by evaluating the funx expressions inside the XML in the context of the implementation.

As an example the following tree:

```
<person xmlns:fx="http://rep.oio.dk/funx"
  xmlns:fc="http://rep.oio.dk/funx/fnc"
  xmlns:att="http://rep.oio.dk/funx/att" >
  <firstName>Bryan</firstName>
  <surName>Rasmussen</surName>
  <address>
    <street>
      <fc:g-func name=''mystreetAddress''/>
    </street>
    <postalzone>
      <fc:g-func name=''mypostalzone''/>
    </postalzone>
    <city>
      <fc:g-func name=''mycity''/>
```

```
    </city>
  </address>
</person>
```

could be used to retrieve the values from the users specific Global Configuration file, if the user had these values defined.

This storing of values is I suppose a method we all use, but because Funx is a language (and a secure language since it does not provide for querying anything outside the system that we do not give it access to) this could be made a lot more useful

```
<person xmlns:fx="http://rep.oio.dk/funx"
  xmlns:fc="http://rep.oio.dk/funx/fnc"
  xmlns:att="http://rep.oio.dk/funx/att" >
  <fx:getPersonDetails/>
  <fx:getAddressDetails/>
</person>
```

which would retrieve the full person details and the full address details, or the following use of functions:

```
<person xmlns:fx="http://rep.oio.dk/funx"
  xmlns:fc="http://rep.oio.dk/funx/fnc"
  xmlns:att="http://rep.oio.dk/funx/att" >
  <fc:getPersonDetails requesterId=''1234''/>
  <fc:getAddressDetails requesterId=''1234''/>
</person>
```

which might be implemented as follows (for `getPersonDetails`):

```
<function name=''getPersonDetails''>
  <fc:if boolean=''True''>
    <fc:eval>
      <fc:m-tag name=''fc:g-func''>
        <fc:attribute name=''exists''>
          <att:requesterId/>
        </fc:attribute>
        <fc:text>TRUE</fc:text>
      </fc:m-tag>
    </fc:eval>
    <fc:then>
      <firstName>Bryan</firstName>
      <lastName>Rasmussen</lastName>
```

```
    </fc:then>
    <fc:else>
      <error>
        Sorry, you are not allowed to access this information
      </error>
    </fc:else>
  </fc:if>
</function>
```

This method was used for a Generic Xforms in Open Office implementation to add functionality not found in the implementation, in the way of a replacement for `xforms:repeat`. In the implementation it was needed to provide an expandable number of lines in a blank OIOUBL document (the danish implementation of UBL) `http://xml.coverpages.org/Denmark-OIOUBL.html` this will be provided as a fuller case study by the time of the conference

The essential solution is to use the `funx:m-tag` to dynamically make an element that is a funx function that writes out UBL InvoiceLine recursively dependent on an input parameter called `linesnumber`:

```
<fnx:m-tag type=''fnc:invoiceline''>
  <fnx:attribute name=''linesnumber''>
    <fnx:param name=''linesnumber''/>
  </fnx:attribute>
</fnx:m-tag>
```

Funx and Pipelines

This ability of Funx and the security of the language makes it interesting to run in a Pipeline setup.

This is interesting in the use of the `fnx:copy-until` function which tells the processor to copy itself out unless a certain parameter (such as parameters passed in by the application or defined in the configuration) in the current context are equal to a value defined by the `copy-until`.

Thus if we have a three step pipeline where the output of the first transformation is passed to the second and that to the third, and each step hosts its own funx implementation with its own configuration. Then the following XML:

```
<person xmlns:fx="http://rep.oio.dk/funx"
  xmlns:fc="http://rep.oio.dk/funx/fnc"
  xmlns:att="http://rep.oio.dk/funx/att" >
  <fc:copy-until gname=''personrequesterId'' value=''Allowed''>
```

```
    <fc:getPersonDetails requesterId=''1234''/>
  </fc:copy-until>
  <fc:copy-until gname=''addressrequesterid'' value=''Allowed''>
    <fc:getAddressDetails requesterId=''1234''/>
  </fc:copy-until>
</person>
```

then if in our first transformation neither `personreqesterId` or `addressreqesterid` are set in the configuration the tree will be copied and sent to the next step.

At the next step if it supports personrequesterId and the value returned is `Allowed`, the function `getPersonDetails` will be processed. If `addressrequesterid` is not supported in this step it will be copied through. And so forth.

By interspersing this with validation that makes sure no `error` elements have come out in our pipeline it becomes possible to build interesting solutions for automated filling out of XML data.

Title:	XMLPrague: a conference on XML
Published by:	Institut Teoretické Informatiky
	Malostranské Náměstí 25
	118 00 Praha 1, Czech Republic
	as the ITI Series publication no. 353
Editors:	Vít Janota, Jiří Kosek
Printed by:	Reprostředisko MFF UK
No. of pages:	104
No. of copies:	190
	First edition, Prague, Czech Republic, 2007

ISBN 978-80-239-9540-4